Diagramming Icon Data Structures*

Ralph E. Griswold

TR 84-5

April 28, 1984

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grant MCS81-01916.

Diagramming Icon Data Structures

1. Introduction

Icon, like a number of other high-level programming languages with late binding times, creates and manipulates a variety of data objects at run time [1]. These include strings, records, lists with queue and stack access methods, and tables that can be subscripted with any kind of value.

The internal representation of such objects plays a significant role in the overall implementation. The manipulation of some of these objects is comparatively sophisticated, although little of the complex underlying mechanism is visible to the user of Icon. The impact of these mechanisms on the performance of the implementation — or even their correctness — is difficult to determine. Even the designer and implementor of the language may not understand, in a broad, general sense, the processes that take place during program execution. The code itself is unsuitable for deriving such an understanding, and modifications made over a period of time may obscure the original design intentions [2-3].

This report describes a tool, called Igram, that produces diagrams of Icon's data objects. This pictorial representation makes it easy to grasp the structures that are used and to understand their interrelationships. This tool consists of a library of Icon procedures that can be called during the execution of an Icon program. Thus the user of this tool can add such calls to a program at appropriate points to produce desired diagrams. This diagrams are produced by examining the layout of memory during the execution of an Icon program. They therefore represent objects as they really are, not as they are imagined to be.

In order to interpret these diagrams, it is necessary to understand, at least in general terms, how memory is organized and referenced in the implementation of Icon. The following section covers the basics of this subject and introduces the main diagramming procedures. Subsequent sections treat special topics in more detail. The latter part of this paper describes how the procedures are implemented and how the user can write post-processors to produce diagrams for different formatting systems.

2. Icon Data Structures

Much of Icon data is composed of words, whose size depends on the architecture of the computer on which Icon is implemented. On the PDP-11, words are 16 bits long, while on the VAX-11, 32-bit "longwords" are used. The diagramming system described here runs on the VAX and, unless otherwise noted, the description here refers to that architecture. To a large extent, however, Icon data objects can be described in machine-independent terms.

2.1 Descriptors

All Icon source-language data values are represented by *descriptors*, which consist of two words: a *t-word* that generally contains flags and a type code, and a *v-word* that generally contains a data value or a pointer to one. For example, the Icon integer 37 is represented by the descriptor

Nni	**integer**
	37

The letters in the left portion of a t-word represent flag bits. **N** indicates that a descriptor is not a qualifier (explained below), **n** indicates a descriptor represents a numeric value (integer or real number), while **i** indicates the descriptor represents an integer[1]. These flags distinguish different properties of descriptors. Some

[1]On 16-bit machines, there are two kinds of integers internally — short integers, whose values fit in the v-word, and long integers, whose values are pointed to. The i flag is used for both.

flags, such as **n** are provided more for convenience than necessity. See [4] for a complete list of flags.

The name in the right portion of a t-word is a symbolic representation for a type code for the value that the descriptor represents. Type codes are small integers that range from 1 to 19; see [4] for a complete list. As mentioned above, the v-word contains the data value or a pointer to it. In the case of an integer, the value itself is in the v-word, as illustrated above.

The procedure call Descrip(x) produces a diagram of the descriptor for x. For example,

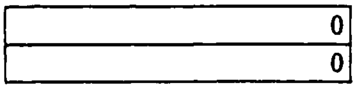
```
Descrip(37)
```

produces the diagram shown above.

The null value has a special descriptor representation in which both the t-word and the v-word are zero. For example,

```
Descrip(&null)
```

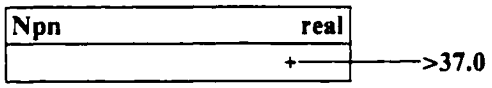
produces



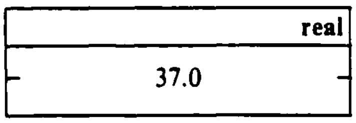
2.2 Blocks

As noted in the preceding section, the descriptors for integers and the null value are self-contained — their data values fit into v-words. All other types are represented by descriptors in which the v-word contains a pointer to the actual value. Except for strings, which are discussed in the next section, such values are blocks of words. For example, a real number is too large to fit into a v-word and therefore is stored in a block.

The first word of every block contains its type code (the same code that appears in the t-word). This type code is used by the garbage collector. For example, the descriptor for the real number 37.0 is



where the arrow indicates the v-word contains the address of a block that contains the actual value. The value 37.0 following the arrow is the Icon string image of the value (image(37.0)). The **p** flag indicates that the v-word contains a pointer instead of the actual value. The block pointed to by the v-word is



The diagrams for such blocks are produced by the procedure Struct(x). For example,

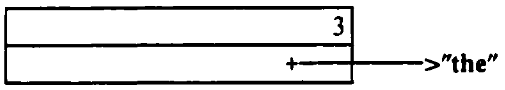
```
Struct(37.0)
```

produces the diagram of the block above.

Different types of values have different block structures. See [4] for details. Blocks for lists and tables, which are described in Sections 2.4 and 2.5, contain pointers to other blocks.

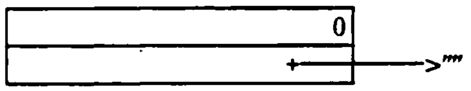
2.3 Strings

The descriptor layout for strings is specialized to allow strings to be represented compactly. The t-word for a string descriptor (*qualifier*) contains the length of the string and the v-word contains the (byte) address of the first character of the string. For example, the qualifier for the string the is diagrammed as



The arrow followed by the string enclose in quotation marks represents a byte address at which the character t occurs, followed by h and e.

Note in particular that the t-word of a qualifier does not contain any flags and has the string length in place of a type code. Any descriptor that is not a qualifier contains the N flag. The v-word of the empty string is nonzero, so that the empty string can be distinguished from the null value. The descriptor for the empty string is diagrammed as



2.4 Lists

The data structures for a list are somewhat complicated, since the size of a list can increase or decrease as the result of queue or stack access. The v-word of a list descriptor points to a *list header block* that contains the usual type word, the size of the list (the number of elements in it), and two descriptors that point to the beginning and end, respectively, of a doubly-linked list of one or more *list element blocks.*

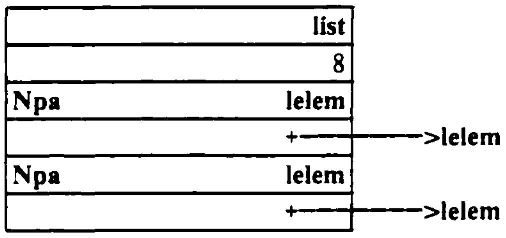
For example, if the list a is produced by

```
a := [1, 2, 3, 4, 5, 6, 7, 8]
```

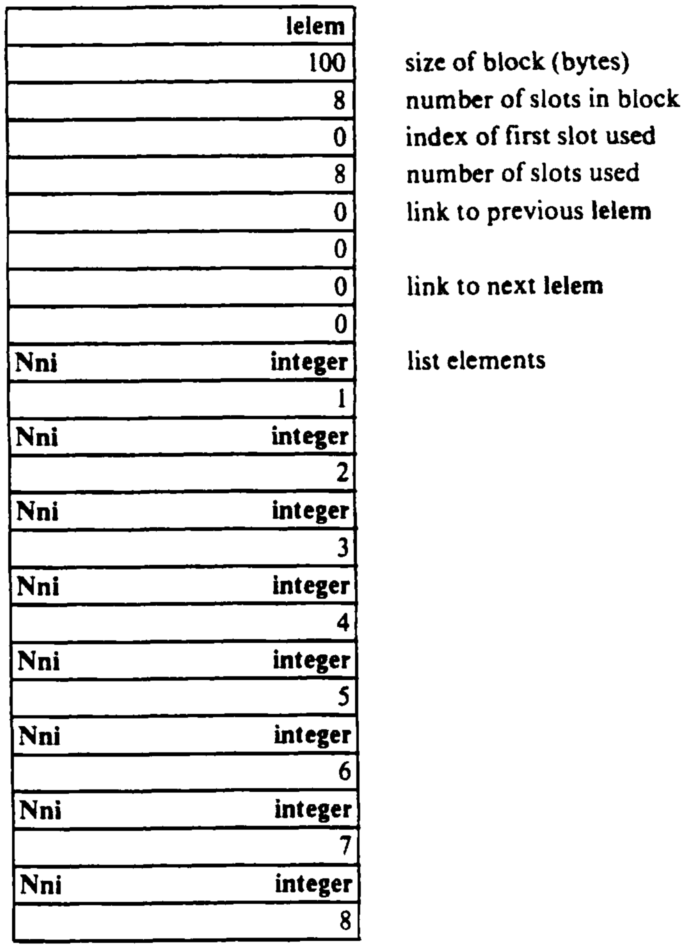
then

```
Struct(a)
```

produces the diagram



A list element block contains a type word (**lelem**), the size of the block in bytes, three words that describe the location of the elements in the block, two descriptors for links to other list element blocks, and descriptor slots for the elements in the block. For the example above, there is one list element block, which has the form

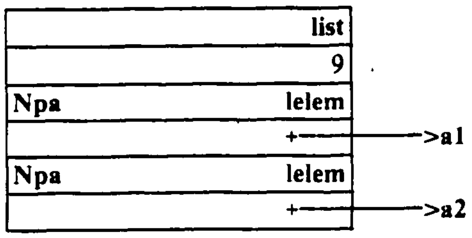


The link descriptors are zero (null descriptors), indicating that there is no previous or next list element block. The descriptor slots in a list element block are managed as a circular queue, allowing list elements to be removed and added conveniently. See [4].

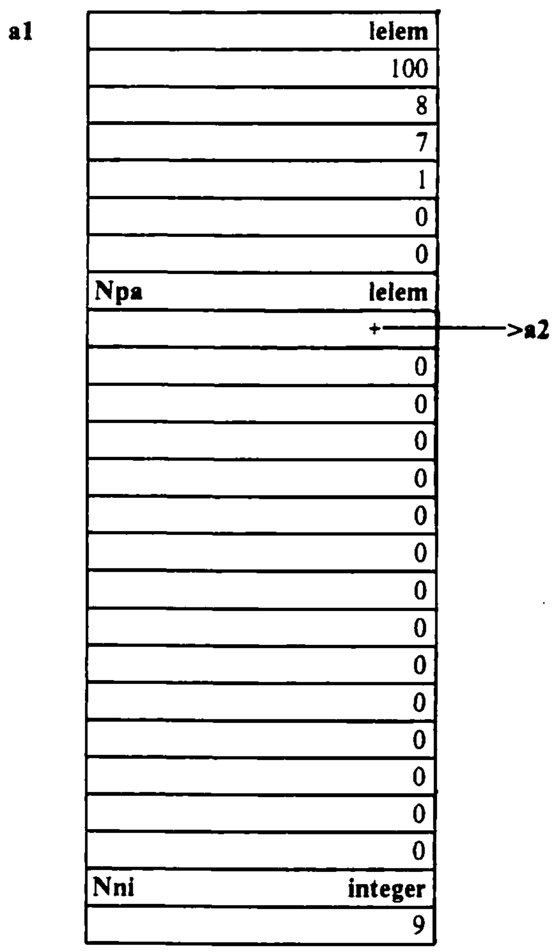
A list element block is added when an element is added to a list and there is no available slot for it. For example,

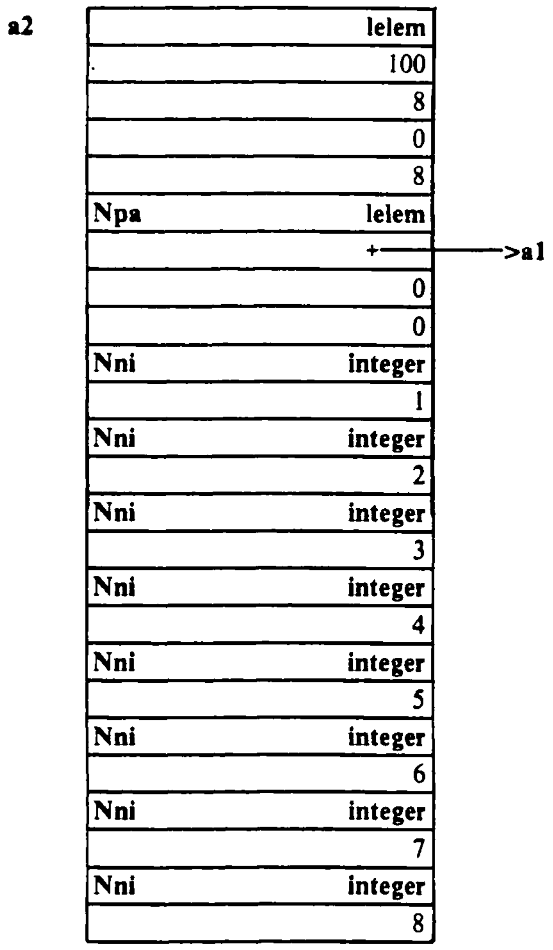
```
push(a, 9)
```

applied to the list given above causes a list element block to be added to the structure above, so that the list header block becomes



where the list element blocks are labeled **a1** and **a2**.


a1
lelem
100
8
7
1
0
0
Npa
lelem
+
>a2
0
0
0
0
0
0
0
0
0
0
0
0
0
0
Nni
integer
9



Note that the new list element block contains extra slots for the addition of new elements.

List element blocks have an internal data type. The descriptors that point to them resemble descriptors for source-language data objects in every way, except that the type code **lelem** refers to a type that is only used internally and is not visible at the source level. In order to be able to diagram list element blocks, it is necessary to be able to treat them as Icon data objects. Four procedures are provided for this purpose:

Flelem(a)	first list element block for a
Llelem(a)	last list element block for a
Nlelem(le)	next list element block for le
Plelem(le)	previous list element block for le

where le indicates a list element block. All of these procedures return the null value if there is no corresponding list element block. For example,

```
le := Flelem(a)
while \le do {
   Struct(le)
   le := Nlelem(le)
   }
```

diagrams the list element blocks for the list a.

2.5 Tables

Tables are implemented in Icon using hashing with chaining [4]. The v-word for a table descriptor points to a *table header block* that contains the type word, the size of the table (the number of elements in it), a descriptor for the table's default value, and 14 *bin descriptors* that serve as pointers to singly-linked lists of *table element blocks*.

For example, if t is a table produced by

```
t := table(0)
```

then

```
Struct(t)
```

produces

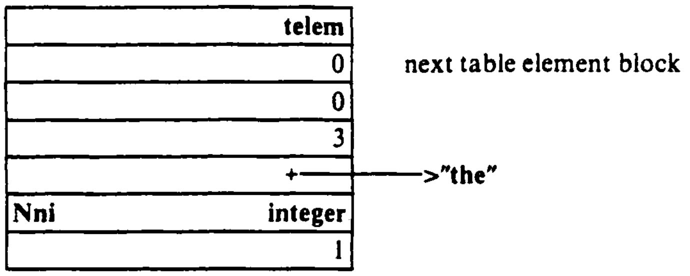
	table	
	0	size of table
Nni	integer	default assigned value
	0	
	0	bin 1
	0	
	0	
	0	
	0	
	0	
	0	
	0	
	0	
	0	
	0	
	0	
	0	
	0	
	0	
	0	
	0	
	0	
	0	
	0	
	0	
	0	
	0	
	0	
	0	
	0	
	0	bin 14
	0	

Note that the bins initially contain null descriptors, indicating that they are empty.

Each element that is added to a table is contained in a table element block that consists of the usual type word, a descriptor pointing to the next table element block, and descriptors for the entry value and assigned value. For example,

```
t["the"] +:= 1
```

produces the table element block



The bin that an element goes in is determined by a hash function of the entry value. As more elements are added to the table, more table element blocks are constructed and linked into the bins.

Like list element blocks, table element blocks correspond to an internal data type (**telem**). In order to diagram table element blocks, two procedures are provided:

Tbin(t, i)	first table element block in the ith bin of t
Ntelem(te)	next table element block after te

where te indicates a table element block. These procedures return the null value if there is no corresponding table element block. For example,

```
te := Tbin(t, 1)
while \te do {
   Struct(te)
   te := Ntelem(te)
   }
```

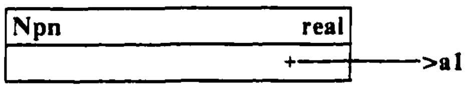
diagrams the table element blocks in the first bin of t.

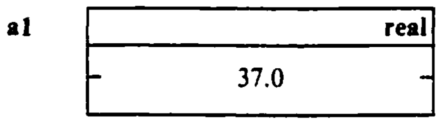
3. Spanned Diagramming

In some situations, it is useful to be able to diagram all the structures that are accessible, via pointers, from a given descriptor or block. The procedures Descrip and Struct have an optional second argument for this purpose: if it is nonnull, all structures accessible from that point are diagrammed. This is referred to as "spanned diagramming". In spanned diagramming, structures are assigned identifying addresses, **a1**, **a2**, ... An example is

```
Descrip(37.0, 1)
```

which produces





Any nonnull value can be used for the second argument to specify spanned diagramming. For example,

```
Descrip(37.0, "span")
```

produces the same diagram as the one above.

In the case that there is more than one pointer to a structure, the structure is diagrammed only once. In spanned diagramming, structures are queued and diagrammed in first-in, first-out order.

4. Artificial Diagrams

Igram, by its nature, cannot diagram variables or trapped-variable blocks [4] because arguments in procedure calls are dereferenced automatically and variables vanish before they can be processed. Similarly, there is no way to construct a descriptor or block with arbitrary contents.

Nonetheless, it is sometimes useful to be able to produce diagrams corresponding to variables, trapped-variable blocks, or even hypothetical structures. Two procedures are provided for such purposes:

Adescrip(tword, vword, label)	diagram an artificial descriptor
Astruct(words, label)	diagram an artificial block

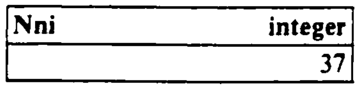
In Adescrip, tword is a two-element list whose first element is the left (flag) part of the t-word and whose second element is the right (type code) part of the t-word. The value of vword is the v-word of the descriptor. If it is a record of the form

```
Pwd(val)
```

this indicates that the v-word contains a *pointer* to val. For example,

```
Adescrip(["Nni", "integer"], 37)
```

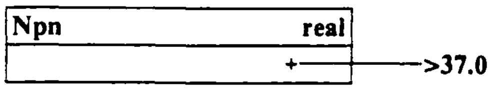
produces



while

```
Adescrip(["Npn","real"], Pwd("37.0"))
```

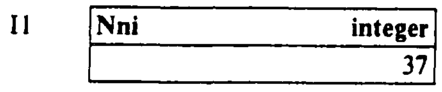
produces



The argument label is optional and provides a label at the left of the t-word. For example,

```
Adescrip(["Nni","integer"],37,"l1")
```

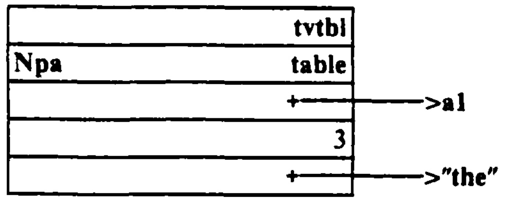
produces



The procedure Astruct is similar in form to Adescrip, except that its first argument is a list, each element of which specifies a word in the block that is diagrammed. For example,

```
Astruct(["tvtbl", ["Npa", "table"], Pwd("a1"), 3, Pwd("the")])
```

produces



5. Using Igram

Igram is run by

```
Igram [ options ] file
```

where *file* is the name of an Icon program (ending in .icn).

Igram translates, links, and executes *file*. The diagramming procedures described in the preceding sections are included automatically, along with support and layout procedures. The output of Igram is in a form suitable as input to a formatting program. Igram runs on the VAX-11 under UNIX[1]. Diagrams correspond to Version 5.8 of Icon as implemented on 32-bit computers.

The layout procedures to be used are determined by the following options:

−n	*Nroff* processing
−t	*Troff* processing
−i	*Itroff* processing

The default option is −n. The macro package −ms is assumed for all these formatting programs [5]. For the −t and −i options, the *Tbl* preprocessor is required for formatting. For example, Igram might be used as

```
Igram −t chap1.icn | tbl | troff −ms
```

The diagrams in this report were prepared using the −t option.

In addition, the option −l*file* causes the user-specified layout procedures given in *file* to be used for layout. See Section 7. Any other options are passed on to Icon. See [6].

Any Icon program can be run under Igram. Care should be taken to assure that output written by the program does not interfere with output produced by diagramming procedures. All global identifiers used in the procedures included by Igram end in underscores. User programs should not use such identifiers. Igram also uses the following external procedures from the Icon program library [7]: Word1, Word2, Indir, and Descr. User programs must not contain procedures with these names.

Diagnostic messages produced by Igram are designed to be self-explanatory.

[1]UNIX is a trademark of Bell Laboratories.

6. Implementation

Words of memory are accessed by the diagramming procedures using *Iscope* functions in the Icon program library [7]. These functions return the contents of words at specified addresses and treat them as Icon integers. For example, Word1(x) returns the word of memory at location x.

When a descriptor is processed, its two words are extracted and stored in the fields of a record. Since these fields contain integers, they can be examined for flag, type code, pointers, and so forth.

Igram has implicit knowledge of some aspects of Icon data structures. For example, in the case of a list header block, Igram knows that the first word of the block contains a type code, the second word contains the size of the list, and the next four words constitute two descriptors. Such knowledge of Icon structures is necessary, since it cannot be derived from the data itself. In particular, Igram knows where descriptors occur in blocks — there is nothing in a descriptor itself that designates it as such.

On the other hand, Igram avoids the use of implicit knowledge where possible and uses the data itself, even if its structure is known. For example, the first word in a list header block is formatted from data that is extracted from memory, even though it is known that it contains the code **list**. Similarly, the t-words of the two descriptors in a list header block are diagrammed from the data that is extracted from memory. This approach assures that, so far as possible, diagrams represent Icon data structures as they really exist, rather than as they are assumed to be.

The procedures used by Igram are divided into three groups:

- diagramming procedures
- support procedures
- layout procedures

The diagramming procedures are those described in Sections 2 through 5. These procedures, in turn, call support procedures and layout procedures. Listings of the diagramming procedures and support procedures are given in Appendices A and B, respectively.

The support procedures do most of the work in diagramming and it is in them that the implicit knowledge of Icon data structures resides. There is a procedure to analyze each type of Icon descriptor and block. Descriptors are broken down into t- and v-word integers and most of the processing is done on these integers. For example, the type of a descriptor is determined by examining the low-order five bits of the t-word, not by using the function type(x). This is done partly to avoid the unnecessary use of implicit knowledge and partly because type(x) cannot be applied to the internal types lelem and telem.

It should be noted that the handling of v-words as integers makes Igram vulnerable to garbage collection, which may change a v-word pointing to a relocatable object without changing the corresponding integer that Igram uses. This can be a problem only during the evaluation of a diagramming procedure. However, such procedures allocate storage in order to produce diagrams, so it is possible for a malfunction to occur. This has not been observed in practice, however. The likely symptom would be an obviously erroneous diagram or program error termination.

Layout procedures do not contain implicit knowledge of Icon data structures, except that they assume that words contain 32 bits. The layout procedures for *Nroff* are comparatively simple and amount to the provision of vertical and horizontal lines for boxes and the arrangement of data within these boxes. See Appendix C. The layout procedures for *Troff* and *Itroff* are more complicated, since *Tbl* commands must be provided. See Appendix D.

7. User-Supplied Layout Procedures

Layout procedures are primarily "device drivers". For example, layout procedures could be produced to drive graphics hardware and generate cinematic displays of Icon structures.

There are 10 layout procedures that are called by diagramming and support procedures:

InitL_()	Initialize the layout. This procedure is called just once and before any other layout procedure. It typically assigns values to global identifiers that are used in preparing output.
Beginbl_(x)	Begin layout of a block of type x. This typically involves some initialization, such as outputting a block header.
Endbl_()	End layout of a block.
Begindr_()	Begin layout of a descriptor.
Enddr_()	End layout of a descriptor.
Bword_(x)	Layout a word of 32 bits, which are given in x.
Fcword_(f,c)	Layout a word with the flags f in the high-order part and the code c in the low-order part.
Iword_(x)	Layout x as an integer word
Pword_(x)	Layout a word pointing to x.
Rblock_(x)	Layout two words containing the real value x.

To avoid possible collisions with identifiers in user programs, all global identifiers used by layout procedures should end with underscores. Care also should be taken not to collide with identifiers used by the support procedures. See Appendix B.

A library of user-supplied layout procedures is linked in by the Igram command processor and must be available in ucode format. For example, if the user-supplied layout procedures are in graphics.icn, they should be translated by

```
icont -c graphics.icn
```

which produces the ucode files graphics.u1 and graphics.u2. These files are used when Igram is run by

```
Igram -l graphics
```

8. Conclusions

Igram was conceived as a tool to support documentation of Icon's internal structures. It has been used extensively for that purpose in a graduate-level course on the implementation of Icon. The purpose of Igram was not simply to make it easier to produce diagrams, but also to assure, as far as possible, that the diagrams represented the actual data structures without omissions and errors due to preconceptions.

In fact, the actual diagrams turned out to be somewhat different, in some cases, from what was expected, indicating how easy it is to make mistakes and overlook details. A number of errors in hand-produced documentation were discovered as a result [4].

Since Igram makes it easy to produce diagrams, it has led to a better understanding of the amount and type of data produced by the execution of Icon programs. If diagrams had been produced on the basis of analysis of the source code for the implementation, instead of being produced by executing programs, many details might have been overlooked. In fact, such details *were* persistently overlooked by students who studied the source code before seeing the diagrams.

An example occurred in determining what happens when a list element block becomes empty due to, say, pop(a). Many students failed to observe an important detail: a list element block is not removed from the doubly-link list when it *becomes* empty. Instead, an empty list element block is only removed when there is an attempt to remove an element from it. The retention of an empty list element block serves the important function of prevent thrashing in case of a repeated pop/push combination at a list element block boundary. This aspect of the implementation is immediately obvious, on the other hand, when Igram is used with a program that removes elements from a list.

Not surprisingly, Igram also was responsible for detecting errors in the implementation of Icon. For example, during the coding of the support procedures for Igram, it became obvious that there was a discrepancy between the use of descriptors in most cases to point to procedure blocks and the use of a single word to point to the procedure block for the record constructor. See Record_(x) in Appendix B.

Descriptors are used to assure the relocation of blocks during garbage collection [4]. It happens that procedure blocks normally are not relocatable, so this discrepancy did not cause Icon programs to malfunction. Interestingly, however, there is a specially tailored version of Icon that uses relocatable procedure blocks. By coincidence, the discrepancy noted above was discovered at the same time that the specially tailored version of Icon was malfunctioning because of it.

A related bug was discovered in the size of the procedure block for the record constructor, where lgram output showed that the actual value was incorrect. See Procedure_(x) in Appendix B. If lgram had used implicit knowledge for this value, rather than the data as it existed in memory, this bug would not have been discovered.

Although lgram is primarily a pedagogical tool, it also can be used to analyze the performance of the implementation. For example, the following procedure, which does no diagramming, provides a histogram of table bin usage and can be used to provide empirical evidence concerning the effectiveness of the hashing function:

```
procedure Thist(t)
   local i, j, x
   every i := 1 to 14 do {
      x := Tbin(t, i)
      j := 1
      while x := Ntelem(\x) do j +:= 1
      write(right(i, 3), ": "i, repl("x", j))
      }
   return
end
```

A program such as lgram is not limited to Icon, of course. It can be adapted to a variety of programming languages, as has been demonstrated for SNOBOL4 [2] and SL5 [3].

References

1. Griswold, Ralph E. and Madge T. Griswold. *The Icon Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey. 1983.

2. Griswold, Ralph E. "A Portable Diagnostic Facility for SNOBOL4", *Software — Practice and Experience*, Vol. 5, No. 1 (January-March 1975), pp. 93-104.

3. Griswold, Ralph E. "Linguistic Extension of Abstract Machine Modelling to Aid Software Development", *Software — Practice and Experience*, Vol. 10, No. 1 (January 1980), pp. 1-9.

4. Griswold, Ralph E., William H. Mitchell, and Stephen B. Wampler. *The C Implementation of Icon; A Tour Through Version 5*, Technical Report TR 83-11a, Department of Computer Science, The University of Arizona. 1983.

5. Lesk, M. E. *Typing Documents on the UNIX System: Using the -ms Macros with Troff and Nroff*, Bell Laboratories, Murray Hill, New Jersey. 1978.

6. Griswold, Ralph E. and William H. Mitchell. *Icont(1)*, manual page for *UNIX Programmer's Manual*, Department of Computer Science, The University of Arizona. 1983.

7. Griswold, Ralph E. *The Icon Program Library*, Technical Report TR 83-6a, Department of Computer Science, The University of Arizona. 1983.

Acknowledgements

Dave Hanson, Bill Mitchell, and Steve Wampler provided a number of helpful suggestions concerning the diagramming facility and the presentation of the material in this report.

Appendix A — Diagramming Procedures

```
record Pwd(val)

procedure Adescrip(tword, vword, label)         # diagram artificial descriptor
   addr_ := \label | ""
   Begindr_()
   Fcword_(tword[1], tword[2])
   addr_ := ""
   if type(vword) == "Pwd" then Pword_(vword.val)
   else Iword_(vword)
   Enddr_()
end

procedure Astruct(words, label)                 # diagram artificial structure
   local word
   addr_ := \label | ""
   Beginbl_()
   every word := !words do {
      case type(word) of {
        "list":  Fcword_(word[1], word[2])
        "Pwd":  Pword_(word.val)
        default:  Iword_(word)
        }
      addr_ := ""
      }
   Endbl_()
end


procedure Descrip(x, u)                         # diagram descriptor
   local y, p
   span_ := u
   y := D_(Word1(x), Word2(x))                  # representation as two integers
   if p := \dtable_[Type_(y)]                   # get diagramming procedure
   then p(y, x) else stop("unknown type")
   Struct(x,\u)
   span_ := &null
end

procedure Flelem(a)                             # get first list block
   local x
   x := D_(Word1(a), Word2(a))
   if Type_(x) ~== "list" then stop("invalid argument to Flelem")
   return Descr(var_, x.v + wsize_ * 2)
end

procedure Llelem(a)                             # get last list block
   local x
   x := D_(Word1(a), Word2(a))
   if Type_(x) ~== "list" then stop("invalid argument to Llelem")
   return Descr(var_, x.v + wsize_ * 4)
end

procedure Nlelem(x)                             # get next list block
   local y
   y := D_(Word1(x), Word2(x))
   if Type_(y) ~== "lelem" then stop("invalid argument to Nlelem")
   return Descr(var_, y.v + wsize_ * 7)
end
```

```
procedure Ntelem(x)                          # get next table element
   local y
   y := D_(Word1(x), Word2(x))
   if Type_(y) ~== "telem" then stop("invalid argument to Ntelem")
   return Descr(var_, y.v + wsize_)
end

procedure Plelem(x)                          # get previous list block
   local y
   y := D_(Word1(x), Word2(x))
   if Type_(y) ~== "lelem" then stop("invalid argument to Plelem")
   return Descr(var_, y.v + wsize_ * 5)
end

procedure Struct(x, u)                       # diagram structure
   local y, z, p
   span_ := u                                # set global to enable derivative processing
   put(llist_, x)                            # queue structure
   done_ := table()                          # table of structures already processed
   while y := get(llist_) do {               # next value to diagram
      z := D_(Word1(y), Word2(y))            # representation as two integers
      if \done_[z.v] then next               # skip if already processed
      done_[z.v] := 1                        # mark as processed
      if  p := \stable_[Type_(z)]            # get diagramming procedure
      then p(z) else stop("non-structure type in Struct")
      }
   span_ := &null
end

procedure Tbin(t, i)                         # get bin of table
   local x
   if not(1 <= integer(i) <= tsize_) then stop("invalid bin in Tbin")
   x := D_(Word1(t), Word2(t))
   if Type_(x) ~== "table" then stop("invalid type to Tbin")
   return Descr(var_, x.v + 2 0* wsize_ * (i + 1))
end
```

Appendix B — Support Procedures

```
link "/usr/icon/ibin/duops"                        # user diagramming procedures
link "/usr/icon/ilib/bitops"                       # bit operations

global span_, llist_, var_, done_, symtab_, ptrswitch_, btype_, addr_
global wsize_, bsize_, tsize_, cwords_, stable_, dtable_, figsym_

record D_(t, v)                                    # Icon descriptor as t-/v-words

procedure Block_(x, t)                             # diagram descriptor pointing to block
   if \span_ then put(llist_, t)
   t := Symbol_(x.t, x.v, t)
   Begindr_()
   Descr_(x.t, x.v, Image_(t), t)
   Enddr_()
end

procedure Cset_(x)                                 # diagram image of a cset
   Head_(x, "cset")
   every Bword_(bitstring(Offset_(x, 1 to cwords_)))
   Endbl_()
end

procedure Descr_(w1, w2, t, s)                     # diagram a descriptor
   if Tword_(w1) | (\t == "string") then Pword_(s) else Word_(w2)
end

procedure Eblock_(x)                               # diagram co-expression block
   local i
   Head_(x, "eblock")
   Word_(i := Offset_(x, 1))                       #size of block
   Pword_("entry point")
   every Word_(Offset_(x, 3 | 4))
   i /:= wsize_
   every Writedescr_(x, 5 to i - 1 by 2)           # arguments and locals
   Endbl_()
end

procedure Estack_(x)                               # diagram co-expression stack
   Head_(x, "estack")
   Writedescr_(x, 1)
   Pword_("stack base")
   Pword_("stack pointer")
   Pword_("address pointer")
   Pword_("Icon/C boundary")
   Word_(Offset_(x, 7))                            # "size" of co-expression
   Writedescr_(x, 8)
   every Word_(Offset_(x, 10 to 25))               # only a few words; block is huge
   Endbl_()
end

procedure File_(x)                                 # diagram file block
   Head_(x, "file")
   Pword_("file descriptor")
   Word_(Offset_(x, 2))
   Writedescr_(x, 3)
   Endbl_()
end
```

```
procedure Head_(x, l)                             # diagram head of block
   local t
   if \span_ then addr_ := Symbol_(x.t, x.v, x)      # addr_ labels first descriptor
      else addr_ := ""
   Beginbl_(l)
   lword_(btype_[Offset_(x, 0)])
   addr_ := ""
end

procedure Image_(x)                               # augmented image to handle internal types
   return (Type_(D_(Word1(x), Word2(x))) == ("lelem" | "telem" | "eblock")) |
     image(x)
end

procedure Init_()                                 # initialize diagramming package
   local x
   InitL_()                                       # initialize layout procedures
   addr_ := ""
   bsize_ := 32                                   # number of bits per word on VAX
   wsize_ := 4                                    # word size in addressing units on VAX
   tsize_ := 14                                   # number of hash bins
   cwords_ := *&cset / bsize_                     # number of words in a cset block
   var_ := Word1(x, 1)                            # internal representation of a variable
   llist_ := []
   figsym_ := "Nvtpniam"
   symtab_ := table()                             # symbolic addresses for structures
   btype_ := list(19)                             # names of the block types
   btype_[1] :=  "integer"                        # integer
   btype_[2] :=  "longint"                        # long integer for 16-bit machines
   btype_[3] :=  "real"                           # real block
   btype_[4] :=  "cset"                           # cset block
   btype_[5] :=  "file"                           # file block
   btype_[6] :=  "procedure"                      # procdure block
   btype_[7] :=  "list"                           # list header block
   btype_[8] :=  "table"                          # table header block
   btype_[9] :=  "record"                         # record block
   btype_[10] := "telem"                          # table element block
   btype_[11] := "lelem"                          # list element block
   btype_[12] := "tvsubs"                         # substring trapped variable
   btype_[13] := "junk"                           # junk block — not used
   btype_[14] := "tvtbl"                          # table element trapped variable
   btype_[15] := "tvpos"                          # &pos trapped variable
   btype_[16] := "tvrand"                         # &random trapped variable
   btype_[17] := "tvtrac"                         # &trace trapped variable
   btype_[18] := "estack"                         # co-expression stack block
   btype_[19] := "eblock"                         # co-expression heap block
   stable_ := table()                             # type names/procedures for structures
   stable_["eblock"] := Eblock_
   stable_["estack"] := Estack_
   stable_["cset"] := Cset_
   stable_["file"] := File_
   stable_["list"] := List_
   stable_["lelem"] := Lelem_
   stable_["procedure"] := Procedure_
   stable_["real"] := Real_
   stable_["table"] := Table_
   stable_["telem"] := Telem_
   stable_["record"] := Record_
   dtable_ := table(Block_)                       # type names/procedures for descriptors
   dtable_["eblock"] := Eblock_
   dtable_["integer"] := Nblock_
   dtable_["null"] := Nblock_
   dtable_["string"] := String_
end
```

```
procedure Lelem_(x)                              # diagram list block
   local i
   Head_(x, "lelem")
   every Word_(Offset_(x, 1) | (i := Offset_(x, 2)) | Offset_(x, 3 | 4))
   every Writedescr_(x, 5 + (0 to (i + 1)) * 2)
   Endbl_()
end

procedure List_(x)                               # diagram list header block
   Head_(x, "list")
   Word_(Offset_(x, 1))
   every Writedescr_(x, 2 | 4)
   Endbl_()
end

procedure Nblock_(x)                             # diagram integer
   Begindr_()
   Descr_(x.t, x.v, "integer")
   Enddr_()
end

procedure Offset_(x, i)
   return Indir(x.v + i * wsize_)
end

procedure Procedure_(x)                          # diagram procedure block
   local i
   Head_(x, "procedure")
   i := Offset_(x, 1)

#  The following expression would make the size for the record constructor
#  what it should be:
#
#     if i < 36 then i := 36
#
#  Note that no procedure block can be less that 36 bytes long.

   Word_(i)
   Pword_("entry point")                         # next word is entry point
   every Word_(Offset_(x, 3 to 6))
   i /:= wsize_
   every Writedescr_(x, 7 to i - 1 by 2)
   Endbl_()
end

procedure Real_(x)                               # diagram real block
   Head_(x, "real")
   Rblock_(Descr(x.t, x.v))
   Endbl_()
end

procedure Record_(x)                             # diagram record block
   local i, p, y, s
   Head_(x, "record")
   Word_(i := Offset_(x, 1))
   i /:= wsize_
   p := Offset_(x, 2)

#  Note that the next word that points to a procedure should, instead, be
#  a descriptor.  In order to get to the block that is pointed to, it is
#  necessary here to construct a descriptor with the appropriate address.
```

```
   y := Descr(Word1(main), p)
   s := Symbol_(-1, p, y)                      # fake a pointer
   Pword_(s)
   \span_ & put(llist_, y)
   every Writedescr_(x, 3 to i - 1 by 2)
   EndbL_()
end

procedure String_(x)                           # diagram string descriptor
   Begindr_()
   Descr_(x.t, x.v, "string", Image_(Descr(x.t, x.v)))
   Enddr_()
end

procedure Symbol_(t, v, d)                     # get representation for pointer
   static i
   initial i := 0
   if /span_ then return Image_(d)
   else if t >= 0 then return ""               # only negative t-words have v-word pointers
   return \symtab_[v] | (symtab_[v] := ("a" || (i +:= 1)))
end

procedure Table_(x)                            # diagram table block
   Head_(x, "table")
   Word_(Offset_(x, 1))
   every Writedescr_(x, 2 * (1 to tsize_ + 1))
   EndbL_()
end

procedure Tcode_(i)                            # get internal descriptor type code
   i %:= 32
   if i < 0 then i +:= 32
   return i
end

procedure Telem_(x)                            # diagram table element
   Head_(x, "telem")
   every Writedescr_(x, 1 | 3 | 5)
   EndbL_()
end

procedure Tword_(x)                            # format t-word of descriptor
   local b, code, c, flags, i
   b := bitstring(x)
   flags := ""                                 # get symbolic representation for flags
   every i := upto('1', b[1+:8]) do flags ||:= flgsym_[i]
   c := code := Tcode_(x)                      # extract type code from word
   if x < 0 then c := btype_[c]
   Fcword_(flags, c)                           # user procedure for formatting t-word
   return (x < 0 & code > 1)                   # signal if it indicates a v-word pointer
end


procedure Type_(x)                             # determine internal type of object
   if x.t = x.v = 0 then return "null"         # null descriptor
   if x.t >= 0 then return "string"            # nonnegative type is string
   return btype_[Tcode_(x.t)]
end

procedure Word_(x, t, s)                       # format a word
   if x > 10000 then Pword_()                  # guess at pointer
   else Iword_(x)                              # user procedure for formatting integer
end
```

```
procedure Writedescr_(x, offset)                      # diagram a descriptor
   local w1, w2, d, s, t
   s := ""
   t := ""
   d := Descr(var_, x.v + offset * wsize_)            # form real descriptor
   w1 := Offset_(x, offset)
   w2 := Offset_(x, offset + 1)
   if w1 = w2 = 0 then &null
   else if w1 >= 0 then {                             # string descriptor
      s := Image_(d)
      t := "string"
      }
   else {
      if Tcode_(w1) ~= 1 then \span_ & put(llist_, d)
      s := Symbol_(w1, w2, d)
      }
   Descr_(w1, w2, t, s)
end
```

Appendix C — Layout Procedures for *Nroff*

```
global lid_, ptr_, blk_, indent_, iwidth_

procedure BeginbL_()                              # begin a block
   write(".DS")
   wlid_()
end

procedure Begindr_()
   write(".DS")
   wlid_()
end

procedure Bword_(x)                               # format word of bits
   write(indent_, "|", x, "|")
   wlid_()
end

procedure EndbL_()                                # end a block
   write(".DE")
end

procedure Enddr_()                                # end a descriptor
   write(".DE")
end

procedure Fcword_(f, c)                           # format t—word
   write(left(addr_, iwidth_), "|", left(f, 22), right(c, 10), "|")
   wlid_()
end

procedure InitL_()                                # initialize layout data
   lid_ := "+" || repl("-", 32) || "+"            # lid on word
   ptr_ := "|" || repl(" ", 28) || "+——|———>"     # pointer
   blk_ := "|" || repl(" ", 32) || "|"            # blank space
   iwidth_ := 5                                   # indentation for blocks
   indent_ := repl(" ", iwidth_)                  # offset for indentation
end

procedure Iword_(x)                               # format integer
   write(left(addr_, iwidth_), "|", right(x, 32), "|")   # allow for possible address
   wlid_()
end

procedure Pword_(x)                               # format pointer
   write(indent_, ptr_, x)
   wlid_()
end

procedure Rblock_(x)                              # format real number
   write(indent_, blk_)
   write(indent_, "+", center(x, 32), "+")
   write(indent_, blk_)
   wlid_()
end

procedure wlid_()                                 # write a lid
   write(indent_, lid_)
end
```

Appendix D — Layout Procedures for *Tbl*/*Troff*

```
global blocktype_, cswitch_, band_, plus_, point_, rest_, chead_, bhead_, rhead_

procedure BeginbL_(x)                             # begin a block
   write(".KS")                                   # need a keep
   write(".ft R")
   write(".sp 1")
   write(".TS")                                   # for Tbl
   case x of {                                    # formatting depends on type of block
      "cset":  {
         write(chead_)
         cswitch_ := 1                            # set switch for cset formatting
         }
      "real":  write(rhead_)
      default:  write(bhead_)
      }
   blocktype_ := x
end

procedure Begindr_()
   write(".KS")
   write(".sp 1")
   write(".ft R")
   write(".TS")
   write(bhead_)
   blocktype_ := &null
end

procedure Bword_(x)                               # format a word of bits
   if \cswitch_ then {
      write(".T&")                                # for cset, change form
      write(band_)
      cswitch_ := &null                           # turn off cset header processing
      }
   write("!\\s8", x, "\\s0")                      # write the word
   write("L_")
end

procedure EndbL_()                                # end a block
   write(".TE")                                   # end the table
   write(".sp 1")
   write(".KE")                                   # end the keep
end

procedure Enddr_()                                # end a descriptor
   write(".TE")
   write(".sp 1")
   write(".KE")
end

procedure Fcword_(f, c)                           # format a t-word
   if not integer(c) then c := "\\fB" || c || "\\fR"
   if f ~== "" then f := "\\fB" || f || "\\fR"
   write(addr_, "! ", f, "!", c, " ")
   write("L_L_")
   return
end
```

```
procedure InitL_()
   local width, hwidth, rmarg, dash
   width := 1.8
   hwidth := 6                                  # indentation of block from label in ens
   rmarg := 20                                  # ens from right border of block
   dash := "\\(mi"                              # use math minus for lines with arrows
   bhead_ := "center tab(!) ;\nL0w(" || hwidth || "n) | L0w(" || width / 2.0 ||
      "i)R0w(" || width / 2.0 || "i)e | L0w(" || rmarg || "n) .\nLL"
   rhead_ := "center tab(!) ;\nL0w(" || hwidth || "n) | L0w(" || width / 3.0 ||
      "i)C0w(" || width / 3.0 || "i)eR0w(" ||
      width / 3.0 || "i)e | L0w(" || rmarg || "n) .\nLLL"
   chead_ := "center tab(!) ;\nL0w(" || hwidth || "n) | R0w(" || width ||
      "i)e | L0w" || rmarg || " .\nL"
                                                # tbl continuation for bit strings
   band_ := "L0w(" || hwidth || "n) | C0w(" || width || "i)e | L0w" || rmarg || " ."
                                                # arrow components
   plus_ := "+" || repl(dash, 3)
   point_ := repl(dash, 4) || ">\\fB"
   rest_ := "\\fR"
   return
end

procedure Iword_(x)                             # format an integer
   if not integer(x) then x := "\\fB" || x || "\\fR"
   if integer(x) < 0 then x := "\\(mi" || abs(x)      # math minus
   if addr_ ~== "" then addr_ := "\\fB" || addr_ || "\\fR"
   case blocktype_ of {
      "cset":  {                                # cset header is only one column
         write(addr_, "!", x, " ")
         write("L")
         }
      "real":  {                                # real header is three columns
         write(addr_, "!!!", x, " ")
         write("LLL")
         }
      default:  {                               # others are two columns
         write(addr_, "!!", x, " ")
          write("LL")
          }
      }
end

procedure Pword_(x)                             # format a pointer
   write("!!    ", plus_, "!", point_, x, rest_)        # construct an arrow
   write("LL")
end

procedure Rblock_(x)                            # format a real number
   write("!\\L   !\\d", x, "\\u!   \\L")
   write()
   write("LLL")
end
```