Tools for the Measurement of Icon Programs*

Cary A. Coutant and Ralph E. Griswold

TR 79-10

May 1979

Department of Computer Science

The University of Arizona

*This work was supported by the National Science Foundation under Grant MCS75-21757.

Tools for the Measurement of Icon Programs

Cary A. Coutant and Ralph E. Griswold

1. Introduction

The DEC-10 implementation of Icon at the University of Arizona has been instrumented to provide a number of facilities for measuring the performance and behavior of Icon programs. The technical details of the instrumentation are described in Reference 1; this report is concerned with the use of the measurement facilities and postprocessing programs for displaying measurement data.

The measurement facilities fall into three essentially distinct categories:

(1) Activity of elementary language components (called tokens).

(2) Charge back of program activity to components of the Icon runtime system.

(3) Storage management information.

The first category is the one of primary interest to the Icon programmer, since it measures the source-language program. There are various kinds of measurement information and various ways of displaying it.

Interpretation of charge back information requires a considerable knowledge of the Icon system. This form of measurement is of interest primarily to the implementors of Icon and other persons concerned with Icon internals.

Storage management information falls in a middle ground between the user and implementor. While the user of Icon should ideally have no need to know about storage management (it is not a language feature), the effect of storage allocation on performance is sufficiently significant that an Icon programmer who is concerned about performance must give some attention to storage management.

2. Tokens

Each elementary language component is called a token. Tokens include function calls, operators, structure references, identifiers, and literals. For example, the following expression consists of tokens beginning at the places marked

```
sum := sum + 1
^   ^  ^   ^ ^
```

For a built-in function, there is a token for the function name. For a call of a procedure, there are tokens for the procedure name as well as for the left parenthesis preceding the argument list. For structure references there are tokens for the structure and for the left brace. Examples are

```
line := process(read(f))
^    ^  ^      ^^    ^

count[n] := 0
^    ^^  ^  ^
```

There are three kinds of data that can be obtained for the tokens in a program:

(1) Activity -- counting each time a token is evaluated.

(2) Allocation -- keeping track of the amount of storage allocated by a token (not all tokens cause allocation).

(3) Sampling -- at periodic intervals noting the token that is currently being evaluated.

Token counting gives a view of program activity without regard for the time spent evaluating each token. As such, it is useful for examining the behaviour of algorithms as reflected in program code and especially under different data loads. An example of token counting is given in Section 4.

The amount of storage that a token allocates is important, since time spent in storage management is generally a significant portion of total execution time. An example of token allocation is given in Section 4. Allocation is measured in words. In the case of strings, where allocation is actually performed in characters, the total values are truncated to the nearest word at the end of the run.

Token sampling gives an approximation to the total amount of time spent evaluating each token during program execution. If there are enough samples, this approximation may be reasonably accurate, since sampling is periodic and hence essentially independent of program execution. However, the sampling is done at 60hz, the DEC-10 clock frequency, and there must be a substantial amount of program execution to get an accurate picture of where time is spent. An example of token sampling is given in Section 4.

Storage management has a significant effect on sampling. Most of the storage management time is usually associated with reclamation (garbage collection), not allocation. On the other hand, reclamation may be caused by any allocation request, regardless of the amount of storage required. This tends to distort time distributions, since a token that triggers reclamation may be charged for the time needed to reclaim the space allocated by many other tokens. To compensate for this effect, samples that occur during reclamation are not charged directly to any token, but rather are distributed to all tokens that cause allocation in proportion to the amount of storage they allocate. This technique gives only a first approximation to accurate charge back, since storage management is a complex process [2], but it is generally within the accuracy that is obtainable with low-frequency sampling.

3. Requesting Measurement Data

In order to get measurement data, it is necessary to specify options when running Icon. In addition to the information given here, there is a summary of all options on ICN:ICON.HLP.

The options for measurement are

t	measure token activity
p	sample program counter
m	print storage management summary

These options may be used singly or in combination.

The t option causes four files to be generated during program execution. Each file has the name of the Icon program, with the following extensions:

TOK	token location data
CNT	token counting data
SMP	token sampling data
ALC	token allocation data

These files contain an entry for each token in the program and therefore are proportional in size to the size of the Icon program.

The p option causes the program counter to be sampled periodically. A file is generated with the name of the Icon program and the extension MON. There is an entry for each sample; consequently the size of this file is proportional to the execution time of the Icon program.

The m option causes a summary of storage management activity to be written to the standard error output file (normally the user's terminal) at the completion of program execution.

An Icon run consists of three steps: compilation, linking, and execution. These steps may be performed separately to allow for the creation of an executable core image, among other things. See the Icon HELP file for details. If an Icon run is broken down into steps, care must be taken in the use of the measurement options. Since the t option affects both the code generated by the Icon translator and runtime measurement, it must be specified during the compilation and linking phases in order to obtain token activity information during the execution phase. The p and m options may be specified in the linking phase, even if they are not specified in the compilation phase, since they only affect execution.

4. Postprocessing Programs

There are several programs for postprocessing Icon measurement data to obtain displays that allow program performance to be viewed in different ways. All of the postprocessing programs are available on the structures CSC: and ICN: and may be accessed using the extended run command facility for users with appropriate ppn attributes.

All the postprocessing programs prompt for information from the user. The name of the Icon program for which measurement data is to be processed is requested first. This name, which must be given without the extension ICN, is used by the programs to locate the files needed for processing. All programs assume that the necessary files with standard extensions exist as described in Section 3. Furthermore, all postprocessing programs write output to files with the same name as the source program, but with specific extensions, depending on processing options. The output file names are given in the following sections and are summarized in the Appendix.

4.1 Token Displays

There are several postprocessing programs for displaying various aspects of token activity in different ways.

TOKEN

The TOKEN program provides a simple display of token activity by printing the activity beneath the token in a listing of the program. TOKEN allows selection of the desired kind of activity by prompting for

```
data (c, s, a):
```

where c stands for token counts, s stands for token samples, and a stands for token allocation. The desired display is obtained by typing the appropriate character followed by a carriage return.

The extension of the output file depends on the kind of activity selected: CPR, SPR, or APR, respectively.

Figures 1 through 3 show portions of typical output for the three kinds of activity. Note that the leftmost digit of each value is aligned under the leftmost character of the token. Values are written on successive lines where there is inadequate space between tokens to place the values on the same line.

```
while t := pop(stk) do            # pop values and process
2207  2107    2207  2107
        2207   2107

   if terminal(t) then sentence := t || sentence else setup(t)
   2107        2107           1212             895  895  895
      2107      1212             1212                     100
                   1212            1212
                                      895
```

Figure 1 -- An Example of Token Counting

```
while t := pop(stk) do            # pop values and process
38    14   11  37   21
        32

   if terminal(t) then sentence := t || sentence else setup(t)
   13 30       58  16   4         3  16   15         3    16   17
                13                     864                      6
```

Figure 2 -- An Example of Token Sampling

```
        repeat {
           =">"                       # get past > if necessary
           88

           slist[j+] :=
                  61 57

              if ="<" then nterm tab(find(">")) | break
                 241       45    18  9    60

                 else tab(find("<"))
                      104 52   5

           }
```

Figure 3 -- An Example of Token Allocation

AVERAGE

While the TOKEN program displays the total values of token counts, samples, and allocation, it is sometimes useful to know the average activity per token activation. The AVERAGE program does this for samples and allocation. It prompts in a manner similar to TOKEN:

```
data(s, a):
```

The extension of the output file is SVR or AVR according to the option selected.

For sampling, the average values are adjusted to correspond to milliseconds of residency. These averages may be very inaccurate because of the coarseness of sampling. For allocation, the average number of words per token is shown. Figures 4 and 5 illustrate typical output from AVERAGE.

```
while t := pop(stk) do               # pop values and process
6.33  0.10 0.08      0.16
        0.25   0.27

   if terminal(t) then sentence := t || sentence else setup(t)
   0.10        0.45     0.05     0.04 11.8         0.05 0.29 0.31
      0.23      0.10                 0.22 0.20                0.11
                   0.22
```

Figure 4 -- An Example of Average Time from Sampling

```
       repeat {
          =">"                       # get past > if necessary
          0.78

          slist[j+] :=
                 0.54
                    0.93

             if ="<" then nterm tab(find(">")) | break
                2.15     5.00  2.00     1.00
                                    0.15

                else tab(find("<"))
                     2.00     0.09
                         1.00

          }
```

Figure 5 -- An Example of Average Allocation

TOKENG

It is typical for the values of token activity to vary by many orders of magnitude in a single program. As a result, it is frequently difficult to compare values, determine high points, or locate unaccessed code, especially in large programs.

To overcome these problems, TOKENG provides an alternative form of display to TOKEN. In TOKENG, values are represented by "logarithmic" bar graphs.

In these graphs, an integer value is represented as a repetition of characters whose length is proportional to the value, as in conventional bar graphs, but in which there is a separate segment for each power of ten. To distinguish segments for different powers of ten, the digit for the power is used as the repeated character in the segment. Dashes are used to align corresponding segments for different values. For example, the integer 376 is represented by

000000----1111111---222

while the integer 62895 is represented by

00000-----111111111-22222222--33--------444444

Note that this representation is precise, while allowing for values varying by many orders of magnitude to be in limited space. The largest value in a sequence of such bars is the longest one and approximations to actual values can be obtained by examining only the rightmost segments. Note that the integer value 1 is represented by the segment 0 and the integer value 0 is represented by the null segment.

TOKENG provides the same options as TOKEN. An example for token counting is shown in Figure 6. Note that the tokens are listed vertically to allow easier comparison of the values. TOKENG also provides a summary of token activity by procedure and for the entire program. Figure 7 shows typical summary information for token allocation.

```
      while t := pop(stk) do          # pop values and process
--------------------------------------------------------------------
while          --------------------2
t              0000000-------------22--------33
:=             0000000-------------2---------33
pop            0000000-------------22--------33
(
stk            0000000-------------22--------33
do             0000000-------------2---------33
--------------------------------------------------------------------
         if terminal(t) then sentence := t || sentence else setup(t)
--------------------------------------------------------------------
if             0000000-------------2---------33
terminal       0000000-------------2---------33
(              0000000-------------2---------33
t              0000000-------------2---------33
then           00---------1---------22--------3
sentence       00---------1---------22--------3
:=             00---------1---------22--------3
t              00---------1---------22--------3
||             00---------1---------22--------3
sentence       00---------1---------22--------3
else           00000-----111111111-22222222
setup          00000-----111111111-22222222
(              00000-----111111111-22222222
t              00000-----111111111-22222222
```

Figure 6 -- An Example of Graphical Display of Token Counting

```
define        000000000-111-------2---------33
generate      0000----------------222222222-----------4
main          0---------1111111---222222
setup         000-------11111111--222-------3
terminal

total         0000000---111111111-----------33333-----4
```

Figure 7 -- An Example of Token Allocation Summary Information

5. Program Counter Displays

Program counter information relates program acitvity to specific modules in the runtime system. Interpretation of this information requires considerable knowledge of Icon internals and generally is not useful to the Icon programmer.

PROFILE

The PROFILE program provides a display of the periodic sampling of the program counter arrayed against the runtime system modules and their entry points. The output is written to a file with the extension PRF. Successive columns show the octal core location, the module name, the entry point, the number of samples, and a percentage based on the total number of samples for the run. Figure 8 shows a section of typical output from PROFILE. The values in main correspond to samples in the subroutine corresponding to the Icon program itself.

```
---------------------------------------------------------
        main                85        5.56%
---------------------------------------------------------
400010  xfer                22        1.42% module total
---------------------------------------------------------
400746  alcint              21        1.36% module total
400747          alcint      21        1.36%
---------------------------------------------------------
401037  alcsql              9         0.58% module total
401040          alcsql      9         0.58%
---------------------------------------------------------
401125  alcstr              110       7.12% module total
---------------------------------------------------------
                            .
                            .
                            .
---------------------------------------------------------
413111  10mac               29        1.87% module total
413137          syserr      1         0.06%
413145          tstb        1         0.06%
413156          setb        27        1.74%
---------------------------------------------------------
413276  10apr               8         0.51% module total
413276          apr         8         0.51%
---------------------------------------------------------
```

Figure 8 -- An Example of a Program Counter Profile

CHARGE

A more detailed chargeback of program activity to the runtime system is provided by CHARGE, which breaks down samples for each token according to the runtime module in which the sample occurred. Figure 9 shows an example of typical output from CHARGE. Samples allocated to storage regeneration are tallied to regen (which includes all regeneration routines), as discussed in Section 2.

```
      while t := pop(stk) do              # pop values and process
--------------------------------------------------------------------
while        2   main    100%
:=           5   xasg    60%     main    40%
pop          1   main    100%
stk          4   type    100%
do           5   main    60%     xmark   40%
--------------------------------------------------------------------
         if terminal(t) then sentence := t || sentence else setup(t)
--------------------------------------------------------------------
termina      3   xderef  66%     xcproc  33%
(            8   xinvok  75%     xmark   12%     main    12%
t            2   xderef  100%
then         1   xdrive  100%
sentenc      1   main    100%
:=           1   xasg    100%
t            3   xderef  100%
||         158   regen   72%     ldc     8%      mvc     8%
                 stc     6%      xcat    1%      alcstr  0%
                 zabump  0%
else         1   main    100%
setup        6   type    83%     xglobl  16%
(            3   xinvok  100%
t            2   main    50%     xderef  50%
--------------------------------------------------------------------
```

Figure 9 -- Example of Token Chargeback to the Runtime System

6. Storage Management Summaries

The m option causes a summary of storage management activity to be written to the standard error output file upon normal program termination. A typical summary is shown in Figure 10.

CPU time: 15300 ms

	String	Qual.	Int	Heap
Allocations	1234	1500	3521	220
Elements alloc.	35102	1500	2048	3150
Regenerations	107	108	21	0
Elements recov.	32596	1372	1951	0
Regen. time	3367	2471	423	0
Expansions	1	0	0	0
Expan. time	62	0	0	0

Figure 10 -- A Storage Management Summary

The four columns correspond to the four regions from which storage is allocated. In each case, the values given are in "elements", not words. For strings, an element is a character (there are four characters per word), for qualifiers, it is the number of qualifiers (there are two words per qualifier), for integers, it is the number of integers (there is one word per integer), and for the heap, it is the number of words.

Regenerations (garbage collections) occur on a per-region basis, as is indicated, although a regeneration in one region may trigger regenerations in others if insufficient space is reclaimed.

A region is expanded if there is not enough space in the region to meet an allocation request after regeneration.

Storage management in Icon is a complex process. See Reference 2 for a complete description.

References

1. Cary A. Coutant and Ralph E. Griswold. Instrumenting Icon for Performance Measurement. Technical Report TR 79-9, Department of Computer Science, The University of Arizona, Tucson, Arizona. May, 1979.

2. David R. Hanson. A Portable Storage Management System for the Icon Programming Language. Technical Report TR 78-16a, Department of Computer Science, The University of Arizona, Tucson, Arizona. February, 1979.

Appendix -- Summary of Postprocessing Programs and Options

The time required for postprocessing measurement data depends on the kind of processing, the number of tokens in the program (t), and the time spent in program execution, which determines the number of samples (s). The programs below that are superscripted by t require time approximately proportional to t and are generally inexpensive to run. The programs superscripted by s are dominated by the length of program execution and are generally expensive to run. The CHARGE program is particularly expensive.

program	options	input files	output files
TOKEN^t		ICN, TOK	
	c	CNT	CPR
	s	SMP	SPR
	a	ALC	APR
AVERAG^t		ICN, TOK, CNT	
	s	SMP	SVR
	a	ALC	AVR
TOKENG^t		ICN, TOK	
	c	CNT	CBR
	s	SMP	SBR
	a	ALC	ABR
PROFILE^s		MON	PRF
CHARGE^s		ICN, TOK, MON, ALC	TPR