Personalized Interpreters for Icon*

Ralph E. Griswold

Robert K. McConeghy

William H. Mitchell

TR 84-14

August 21, 1984

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grants MCS81-01916 and DCR-8401831.

Personalized Interpreters for Icon

1. Introduction

Despite the fact that the Icon programming language has a large repertoire of functions and operations for string and list manipulation, as well as for more conventional computations [1], users frequently need to extend that repertoire. While many extensions can be written as procedures that build on the existing repertoire, there are some kinds of extensions for which this approach is unacceptably inefficient, inconvenient, or simply impractical.

Icon itself is written primarily in C [2] and its built-in functions are written as corresponding C functions. Thus the natural way to extend Icon's computational repertoire is to add new C functions to it.

The Icon system is organized so that this is comparatively easy to do. Adding a new function does not require changes to the Icon translator, since all functions have a common syntactic form. An entry must be made in a table that is used by the linker and the run-time system in order to identify built-in functions and connect references to them to the code itself.

The problem arises in incorporating the C code in the Icon run-time system. Prior to Version 5.9 of Icon, there were two separate but similar implementations of Icon: a compiler [3] and an interpreter [4]. The primary difference between the two systems is that the linker for the compiler generates assembly-language code, while the linker for the interpreter generates code that is ready to be interpreted. The interpreter uses a preconstructed run-time system, so that the assembly and loading phases of the compiler implementation is not needed.

The loading phase in the compiler is quite slow, so that when the compiler implementation of Icon is used, there is a substantial delay before getting into execution. This is a significant problem during program debugging. Furthermore, a compiled Icon program runs only slightly faster than an interpreted Icon program. This is due in large part to the fact that most programs spend only a small percentage of their time in code generated for the program itself; most of the time is spent executing code in the run-time system, which is essentially the same in the two implementations.

The primary advantage of the compiler is that it is possible to add new functions during the loading phase. In order to communicate the names of new functions to the linker, it is necessary to include "external" declarations in the Icon source programs that use these functions. There is no way to do this in the interpreter implementation, since the run-time system is preconstructed, rather than being built when the source-language program is processed.

One disadvantage of the external function approach is that every source program that uses an external function must contain a declaration for that function. In addition to the necessity for having to remember these declarations, external functions are, by their nature, not logically part of Icon proper. This results in problems of documentation and distribution of such functions to other users.

An alternative method of adding new functions to either the compiler or the interpreter implementation of Icon is to add the corresponding C functions to the Icon system itself and to rebuild the entire system. This approach is impractical for many applications. If the extensions are not of general interest, it is inappropriate to include them in the public version of Icon. On the other hand, Icon is a large and complicated system, and having many private versions may create serious problems of maintenance and disk usage. Furthermore, rebuilding the Icon system is slow, cumbersome, and comparatively complicated. This approach therefore is impractical in a situation such as a class in which students implement their own versions of an extension.

To remedy these problems, a mechanism for building "personalized interpreters" has been added to Version 5.9 of Icon. This mechanism allows a user to add C functions and to build a corresponding interpreter quickly, easily, and without the necessity to have a copy of the source code for the entire Icon system.

To construct a personalized interpreter, the user must perform a one-time set up that copies relevant source files to a directory specified by the user and builds the nucleus of a run-time system. Once this is done, the user can add and modify C functions and include them in the personalized run-time system with little effort.

Since the linker must know the names of built-in functions, a personalized linker is constructed. In order to run Icon programs with the personalized run-time system, a personalized command processor, which knows the location of the personalized linker and run-time system, is provided also.

The modifications that can be made to Icon via a personalized interpreter essentially are limited to the run-time system: the addition of new functions, modifications to existing functions and operations, and modifications and additions to support routines. There is no provision for changing the syntax of Icon, incorporating new operators, keyword, or control structures.

2. Building and Using a Personalized Interpreter

2.1 Setting Up a Personalized Interpreter System

To set up a personalized interpreter, a new directory should be created solely for the use of the interpreter; otherwise files may be accidentally destroyed by the set-up process. For the purpose of example, suppose this directory is named myicon. The set-up process consists of

```
mkdir myicon
cd myicon
icon-pi
```

Note that icon-pi must be run in the area in which the personalized interpreter is to be built. The location of icon-pi may vary from site to site [5].

The shell script icon-pi constructs three subdirectories: h, std, and pi. The subdirectory h contains header files that are needed in C routines. The subdirectory std contains the portions of the Icon system that are needed to build a personalized interpreter. The subdirectory pi contains a Makefile for building a personalized interpreter and also is the place where source code for new C functions normally resides. Thus work on the personalized interpreter is done in myicon/pi.

The Makefile that is constructed by icon-pi contains two definitions to facilitate building personalized interpreters:

OBJS	a list of object modules that are to be added to or replaced in the run-time system. OBJS initially is empty.
LIB	a list of library options that are used when the run-time system is built. LIB initially is empty.

See the listing of a generic version of this Makefile in Appendix A.

2.2 Building a Personalized Interpreter

Performing a *make* in myicon/pi creates three files in myicon:

picont	command processor
pilink	linker
piconx	run–time system

A link to picont also is constructed in myicon/pi so that the new personalized interpreter can be tested in the directory in which it is made.

The file picont normally is built only on the first *make*. The file pilink is built on the first *make* and is rebuilt whenever the repertoire of built-in functions is changed. The file piconx is rebuilt whenever the source code in the run-time system is changed.

The user of the personalized interpreter uses picont in the same fashion that the standard icont is used [4]. (Note that the accidental use of icont in place of picont may produce mysterious results.) In turn, picont translates a source program using the standard Icon translator and links it using pilink. The resulting icode file uses piconx.

The relocation bits and symbol tables in picont, pilink, and piconx can be removed by

```
make Strip
```

in myicon/pi. This reduces the sizes of these files substantially but may interfere with debugging.

If a *make* is performed in myicon/pi before any run-time files are added or modified, the resulting personalized interpreter is identical to the standard one. Such a *make* can be performed to verify that the personalized interpreter system is performing properly.

Note that a personalized interpreter inherits the parameters and configuration of the locally installed version of Icon in v5g, including optional language extensions [6]. The file myicon/h/config.h contains configuration information. The definitions in this file should not be changed.

2.3 Adding a New Function

To add a new function to the personalized interpreter, it is first necessary to provide the C code, adhering to the conventions and data structures used throughout Icon. See [2]. Some examples of C functions taken from the Icon program library [7] are included in Appendix B of this report. The source code for these functions is contained in v5g/pifunc, where v5g is the root of the Icon system. The location of v5g varies from site to site [5]. The directory v5g/functions contains the source code for the standard built-in functions, which also can be used as models for new ones.

Suppose that getenv from the Icon program library is to be added to a personalized interpreter. The source code can be obtained by

```
cp v5g/pifuncs/getenv.c myicon/pi
```

(Note that the actual paths will be different, depending on the local hierarchy.)

Three things now need to be done to incorporate this function in the personalized interpreter:

1. Add a line consisting of

   ```
   PDEF(getenv)
   ```

   to myicon/h/pdef.h in proper alphabetical order. This causes the linker and the run-time system to know about the new function.
2. Add getenv.o to the definition of OBJS in myicon/pi/Makefile. This causes getenv.c to bbe compiled and the resulting object file to be loaded with the run-time system when a *make* is performed.
3. Perform a *make* in myicon/pi. The result is new versions of pilink and piconx in myicon.

The function getenv now can be used like any other built-in function.

More than one function can be included in a single source file. See math.c in Appendix B. Note that math.c uses the math library. To add this module to the run-time system of a personalized interpreter, PDEF entries should be made for each function in math.c, math.o should be added to OBJS, and -lm should be added to LIB in the Makefile.

2.4 Modifying the Existing Run-Time System

The use of personalized interpreters is not limited to the addition of new functions. Any module in the standard run-time system can be modified as well. The run-time system is divided into five parts:

v5g/functions	built-in functions
v5g/operators	built-in operators
v5g/rt	run-time support routines
v5g/lib	routines called by the interpreter
v5g/iconx	the interpreter and start-up routines

For example, storage allocation routines are contained in v5g/rt/alc.c.

To modify an existing portion of the Icon run-time system, copy the source code file from the standard system to myicon/pi. (Source code for a few run-time routines is placed in myicon/std when a personalized interpreter is set up. Check this directory first and use that file, if appropriate, rather than making another copy in myicon/pi.) When a source-code file in myicon/pi has been modified, place it in the OBJS list just like a new file and perform a *make*. Note that an entire module must be replaced, even if a change is made to only one routine. Any module that is replaced must contain all the global variables in the original module to prevent *ld(1)* from also loading the original module. There is no way to delete routines from the run-time system.

The directory myicon/h contains header files that are included in various source-code files. For example, error message text for a new run-time error can be provided by adding it to myicon/h/err.h. The file myicon/h/rt.h contains declarations and definitions that are used throughout the run-time system. This is where the declaration for the structure of a new type of data object would be placed.

Care must be taken when modifying header files not to make changes that would produce inconsistencies between previously compiled components of the Icon run-time system and new ones.

References

1. Griswold, Ralph E. and Griswold, Madge T. *The Icon Programming Language*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey. 1983.
2. 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.
3. Griswold, Ralph E. and William H. Mitchell. *ICONC(1)*, manual page for *UNIX Programmer's Manual*, Department of Computer Science, The University of Arizona. July 1983.
4. Griswold, Ralph E. and William H. Mitchell. *ICONT(1)*, manual page for *UNIX Programmer's Manual*, Department of Computer Science, The University of Arizona. August 1984.
5. Griswold, Ralph E. and William H. Mitchell. *Installation and Maintenance Guide for Version 5.9 of Icon*. Technical Report TR 84-13, Department of Computer Science, The University of Arizona, Tucson, Arizona. August 1984.
6. Griswold, Ralph E., Robert K. McConeghy, and William H. Mitchell. *Extensions to Version 5 of the Icon Programming Language*. Technical Report TR 84-10a, Department of Computer Science, The University of Arizona. August 1984.
7. Griswold, Ralph E. *The Icon Program Library*, Technical Report TR 84-12, Department of Computer Science, The University of Arizona. August 1984.

Appendix A — Makefile for Personalized Interpreters

The "generic" Makefile for personalized interpreters follows. The values of PATH and DIR are filled in when Pimake is run.

```
CFLAGS=
LDFLAGS=
LIB=
iroot=PATH
V5GBIN=$(iroot)/bin
DIR=

#
# To add or replace object files, add their names to the OBJS list below.
#  For example, to add foo.o and bar.o, use:
#
#              OBJS=foo.o bar.o            (this is a sample line)
#
# For each object file added to OBJS, add a dependency line to reflect files
#  that are depended on.  For example, if foo.c includes rt.h
#  which is located in the h directory use
#
#              foo.o:          ../h/rt.h
#

OBJS=

PIOBJS=../std/init.o ../std/strprc.o
RTOBJS=$(PIOBJS) $(OBJS)

Pi:            ../picont ../piconx ../pilink

               rm -f ../picont picont
               cc -o ../picont -DIntBin="\"$(DIR)\"" -DIconx="\"$(DIR)/piconx\"" \
                         -DIconxEnv="\"ICONX=$(DIR)/piconx\"" \
                         -DILINK="\"$(DIR)/pilink\"" \
                         -DITRAN="\"$(V5GBIN)/itran\"" -DFORK=QFORK \
                         ../std/icont.c
               ln ../picont

               cc $(LDFLAGS) -X -o ../pilink ../std/builtin.o ../std/linklib

               cc $(LDFLAGS) -X -o ../piconx -e start -u start $(RTOBJS) ../std/rtlib $(LIB)

               cd ../std;     cc -c init.c

               cd ../std;     cc -c builtin.c

               cd ../std;     cc -c strprc.c

Strip:         ../picont ../piconx ../pilink
               strip ../picont ../piconx ../pilink
```

Appendix B — C Functions from the Icon Program Library

getenv.c:

```
/*
#               GETENV(3.icon)
#
#               Get contents of environment variables
#
#               Stephen B. Wampler
#
#               Last modified 8/19/84
#
*/
#include "../h/rt.h"

/*
 * getenv(s) - return contents of environment variable s
 */

Xgetenv(nargs, arg1, arg0)
int nargs;
struct descrip arg1, arg0;
   {
   register char *p;
   register int len;
   char sbuf[MAXSTRING];
   extern char *getenv();
   extern char *alcstr();

   DeRef(arg1)

   if (!QUAL(arg1))                          /* check legality of argument */
      runerr(103, &arg1);
   if (STRLEN(arg1) <= 0 || STRLEN(arg1) >= MAXSTRING)
      runerr(401, &arg1);
   qtos(&arg1, sbuf);                        /* convert argument to C-style string */

   if ((p = getenv(sbuf)) != NULL) {         /* get environment variable */
      len = strlen(p);
      sneed(len);
      STRLEN(arg0) = len;
      STRLOC(arg0) = alcstr(p, len);
      }
   else                                      /* fail if variable not in environment */
      fail();
   }

Procblock(getenv, -1)
```

math.c:

```
/*
#               MATH(3.icon)
#
#               Miscellaneous math functions
#
#               Ralph E. Griswold
#
#               Last modified 8/19/84
#
*/

#include "../h/rt.h"
#include <errno.h>

int errno;
/*
 * exp(x), x in radians
 */
Xexp(nargs, arg1, arg0)
int nargs;
struct descrip arg1, arg0;
   {
   int t;
   double y;
   union numeric r;
   double exp();

   if ((t = cvreal(&arg1, &r)) == NULL) runerr(102, &arg1);
   y = exp(r.real);
   if (errno == ERANGE) runerr(252, NULL);
   mkreal(y, &arg0);
   }
Procblock(exp, 1)

/*
 * log(x), x in radians
 */
Xlog(nargs, arg1, arg0)
int nargs;
struct descrip arg1, arg0;
   {
   int t;
   double y;
   union numeric r;
   double log();

   if ((t = cvreal(&arg1, &r)) == NULL) runerr(102, &arg1);
   y = log(r.real);
   if (errno == EDOM) runerr(251, NULL);
   mkreal(y, &arg0);
   }
Procblock(log, 1)
```

```
/*
 * log10(x), x in radians
 */
Xlog10(nargs, arg1, arg0)
int nargs;
struct descrip arg1, arg0;
   {
   int t;
   double y;
   union numeric r;
   double log10();

   if ((t = cvreal(&arg1, &r)) == NULL) runerr(102, &arg1);
   y = log10(r.real);
   if (errno == EDOM) runerr(251, NULL);
   mkreal(y, &arg0);
   }
Procblock(log10, 1)

/*
 * sqrt(x), x in radians
 */
Xsqrt(nargs, arg1, arg0)
int nargs;
struct descrip arg1, arg0;
   {
   int t;
   double y;
   union numeric r;
   double sqrt();

   if ((t = cvreal(&arg1, &r)) == NULL) runerr(102, &arg1);
   y = sqrt(r.real);
   if (errno == EDOM) runerr(251, NULL);
   mkreal(y, &arg0);
   }
Procblock(sqrt, 1)
```

seek.c:

```
/*
#               SEEK(3.icon)
#
#               Seek to position in stream
#
#               Stephen B. Wampler
#
#               Last modified 8/19/84
#
*/

#include "../h/rt.h"

/*
 * seek(file, offset, start) - seek to offset byte from start in file.
 */

Xseek(nargs, arg3, arg2, arg1, arg0)
int nargs;
struct descrip arg3, arg2, arg1, arg0;
   {
   long l1, l2;
   int status;
   FILE *fd;
   long ftell();

   DeRef(arg1)
   if (arg1.type != D_FILE)
      runerr(106);

   defint(&arg2, &l1, 0);
   defshort(&arg3, 0);

   fd = BLKLOC(arg1)->file.fd;

   if ((BLKLOC(arg1)->file.status == 0) ||
       (fseek(fd, l1, arg3.value.integr) == -1))
      fail();
   mkint(ftell(fd), &arg0);
   }

Procblock(seek, 3)
```