Personalized Interpreters for Version 7 of Icon*

Ralph E. Griswold

TR 88-7

January 22, 1988

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grant DCR-8502015.

Personalized Interpreters for Version 7 of 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 [1,2], 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 C 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 new functions 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.

One method of adding new functions to 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 expensive. This approach therefore may be 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'' is included in UNIX* implementations 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 also is constructed. In order to run Icon programs in a self-contained personalized run-time system, personalized versions of the translator, pitran, and the command processor, picont, are 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 setup process. For the purpose of example, suppose this directory is named myicon. The setup consists of

---
*UNIX is a trademark of AT&T Bell Laboratories.

```
mkdir myicon
cd myicon
icon_pi
```

Note that icon_pi must be run from the area in which the personalized interpreter is to be built. The location of icon_pi may vary from site to site.

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 machine-independent 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, but the math library is loaded as a normal part of building the run-time system.

See the listing of the generic version of this Makefile in the appendix.

2.2 Building a Personalized Interpreter

Performing a

```
make pi
```

in myicon/pi creates four files in myicon:

picont	command processor
pilink	linker
piconx	run−time system
piconx.hdr	header file for linker output

Links to these files also are 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 as a result of modifications to h/fdefs.h. 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. (Note that the accidental use of icont in place of picont may produce mysterious results.) In turn, picont translates a source program using pitran and links it using pilink. The resulting icode file uses piconx.

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

```
make Stripx
```

in myicon/pi. This reduces the size of this file 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.

2.3 Version Numbering

The Icon run-time system checks an identifying version number to be sure the linker and run-time system versions correspond. The version number is the string defined for IVersion in myicon/h/version.h following the construction of a personalized interpreter as described in Section 2.1.

In order to assure that files produced by pilink can only be run by the current versions of piconx, the value of IVersion should be changed whenever a change is made to a personalized interpreter. It is not important what the definition of IVersion is, so long as it is a short string that is different from previous ones.

2.4 Adding New Functions

To add new functions to the personalized interpreter, it is first necessary to provide the C code for them, adhering to the conventions and data structures used throughout Icon [3]. The directory src/iconx in the Version 7 Icon hierarchy [2] contains the source code for the standard functions, which can be used as models for new ones.

Suppose that a function setenv is to be added to a personalized interpreter.

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

1. Provide the code to the function (in, say, setenv.c).
2. Add a line consisting of

   ```
   FncDef(setenv)
   ```

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

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

More than one function can be included in a single source file. To incorporate these functions in a personalized interpreter, an FncDef entry needs to be added for each function, but only the file need be added to the OBJS list.

2.5 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.

To modify an existing portion of the Icon run-time system, copy the source code file from v7/src/iconx 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. 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 Madge T. Griswold. *The Icon Programming Language*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey. 1983.
2. Griswold, Ralph E., Gregg M. Townsend, and Kenneth Walker. *Version 7 of Icon*, Technical Report TR 88-5, Department of Computer Science, The University of Arizona. 1988.
3. Griswold, Ralph E. and Madge T. Griswold. *The Implementation of the Icon Programming Language*. Princeton University Press, Princeton, New Jersey. 1987.

Appendix — Makefile for Personalized Interpreters

The ‘‘generic’’ Makefile for personalized interpreters follows. A copy, with the value of Dir filled in and appropriate definitions for the flags, is placed in myicon/pi when icon_pi is run.

```
SHELL=/bin/sh
CFLAGS= −O
LDFLAGS=
XCFLAGS= −O
XLDFLAGS= −N
Dir=‘pwd‘
Sdir=‘pwd‘/../pi

RHDRS=          ../h/rt.h ../h/config.h ../h/define.h ../h/cpuconf.h ../h/memsize.h
LHDRS=          ../std/ilink.h $(RHDRS)
#
#   To add or replace object files, add their names to the OBJS list below.
#   For example, to add nfncs.o and iolib.o, use:
#
#                   OBJS=nfncs.o iolib.o            # this is a sample line
#
#   For each object file added to OBJS, add a dependency line to reflect files
#   that are depended on.  In general, new functions depend on $(RHDRS).
#   For example, if nfncs.c contains new functions, use
#
#                   nfncs.o:        $(RHDRS)
#
#   Such additions to this Makefile should go at the end.

OBJS=
LIB=

RTOBJS=../std/imain.o ../std/rconv.o ../std/idata.o $(OBJS)

Pi:             picont piconx pilink piconx.hdr

picont:         ../std/icont.c ../h/config.h ../h/define.h ../h/paths.h
                rm −f ../picont picont
                $(CC) $(CFLAGS) −o picont −DItran="\"$(Dir)/pitran\""\
                        −DIconx="\"$(Dir)/piconx\"" \
                        −DIlink="\"$(Dir)/pilink\""   ../std/icont.c
                strip picont
                ln picont ../picont

pilink:         ../std/linklib ../std/lsym.o ../std/ilink.o ../std/lcode.o ../std/lmem.o
                rm −f ../pilink pilink
                $(CC) $(LDFLAGS) −o pilink ../std/lsym.o ../std/ilink.o\
                         ../std/lcode.o ../std/lmem.o ../std/linklib
                strip pilink
                ln pilink ../pilink

../std/ixhdr.o: ../h/config.h ../h/define.h
                cd ../std;      $(CC) −c $(XCFLAGS) −DIconx="\"$(Sdir)/piconx\"" ixhdr.c
```

```
piconx.hdr:     ../std/ixhdr.o
                rm –f ../piconx.hdr piconx.hdr
                $(CC) $(XLDFLAGS) ../std/ixhdr.o –o piconx.hdr
                strip piconx.hdr
                ln piconx.hdr ../piconx.hdr

piconx:         ../std/rtlib $(RTOBJS)
                rm –f ../piconx piconx
                $(CC) $(LDFLAGS) –o piconx $(RTOBJS) ../std/rtlib $(LIB) –lm
                ln piconx ../piconx


../std/idata.o:                 $(RHDRS) ../h/fdefs.h fdefs.h
                cd ../std;      $(CC) –c –DPI $(CFLAGS) idata.c

../std/imain.o:                 $(RHDRS) ../h/header.h ../h/version.h ../std/gc.h \
                                        ../h/opdefs.h
                cd ../std;      $(CC) –c $(CFLAGS) imain.c

../std/rconv.o:                 $(RHDRS) ../h/fdefs.h fdefs.h
                cd ../std;      $(CC) –c –DPI $(CFLAGS) rconv.c

../std/lsym.o:  $(LHDRS) ../h/fdefs.h fdefs.h
                cd ../std;      $(CC) –c –DPI $(CFLAGS) lsym.c

../std/lmem.o:  $(LHDRS)
                cd ../std;      $(CC) –c –DPI $(CFLAGS) lmem.c

../std/ilink.o:                 $(LHDRS) ../h/header.h ../h/paths.h
                cd ../std;      $(CC) –c $(CFLAGS) –DHeader="\"$(Sdir)/piconx.hdr\""   ilink.c

../std/lcode.o:                 $(LHDRS) ../h/header.h ../h/paths.h ../std/opcode.h \
                                        ../h/keyword.h ../h/opdefs.h ../h/version.h
                cd ../std;      $(CC) –c $(CFLAGS) lcode.c

Stripx:         piconx
                strip piconx
```