Jcon: A Java-Based Implementation of Icon

Gregg M. Townsend, The University of Arizona
Todd A. Proebsting, Microsoft Research

Icon Project Document 286 (IPD286a)
Department of Computer Science
The University of Arizona
September 10, 1999
http://www.cs.arizona.edu/icon/docs/ipd286.htm

This document collects the separate Web pages that serve as documentation for Jcon. The sections are:

Introduction

Jcon (pronounced *JAY-konn*) is a new Java-based implementation of the Icon programming language (Griswold and Griswold, 1996). The Jcon translator, written in Icon, generates Java class files that execute in conjunction with a run-time system written in Java. Jcon runs on Unix systems.

Jcon is an essentially complete implementation of Icon, omitting only a few things such as `chdir()` that cannot be done in Java. Co-expressions, large integers, and pipes are provided, and a preprocessor is included. String invocation is supported. Tracing, error recovery, and debugging functions are all included, although for performance reasons they are disabled by default. There are a few minor deficiencies due to Java limitations.

Jcon includes almost all of Icon's standard graphics facilities. Wide lines and textured drawing are lacking; these should be easier to implement after version 1.2 of Java becomes universally available. A few other features, notably mutable colors, are also unavailable. Details are provided in the Graphics section.

The `jcont` script functions similarly to `icont`. The end result of compilation is an executable file, which is produced by embedding a Zip archive of Java classes (a "Jar file") inside a shell script. Separate compilation is also supported, with `.zip` files substituting for `.u1/.u2` file pairs.

Implementation

The Jcon translator is a 10,000-line Icon program that produces either Java class files or *ucode* (`.u1/.u2`) files. This is somewhat smaller than the 13,000 lines of C code that form the translator in Version 9 of Icon, but the two totals are not directly comparable because the two programs have different capabilities. The common function of generating ucode files from Icon source code accounts for about 3,500 lines in Jcon and about 9,000 lines in Version 9.

The Jcon runtime system comprises 18,000 lines of Java code. The corresponding portion of Version 9, counting only the Unix code, is over 50,000 lines of C code.

The `jcont` script that directs compilation and linking is a 400-line Korn shell script. While this works well for Unix, it is the single largest impediment to a non-Unix port. We believe that writing a replacement program would be the easiest route to a Windows or Macintosh port.

Jcon includes an automated test suite containing 8,000 lines of Icon programs. These are drawn from the standard Icon test suite, other existing Icon programs, and 3,000 lines of new tests. Jcon also includes additional manually-run graphics tests and a small collection of graphics demonstration programs.

Acknowledgements

Denise Todd contributed to the Jcon translator. Bob Alexander wrote the preprocessor. Ralph Griswold offered useful advice and helped test early versions. Some small portions of the runtime system are derived from Version 9 of Icon. The GIF encoder is from Jef Poskanzer (www.acme.com).

Usage

Jcont translates Icon source files into `.zip` files, and links `.zip` files to make an executable program. *Jcont* is similar to *icont* in concept and behavior; think of a `.zip` file as analogous to a `.u1/.u2` pair generated by *icont*.

Normally, *jcont* produces a directly executable Korn Shell script containing an embedded Zip file. Running the script executes the compiled Icon program, and arguments can be passed as usual.

Synopsis

`jcont` [*options*] *file*... [`-x` [*arguments*]]

File Arguments

file`.icn`	Icon source file to compile and link

file`.zip`	Previously compiled file to link (Must be local: *jcont* does not search $IPATH for *file*`.zip`)
file`.class`	Java bytecode file (for dynamic loading) to include in output file

Options

`-b`	generate backwards-compatible `.u1` and `.u2` files; implies `-c`
`-c`	compile only: don't produce an executable
`-E`	preprocess only
`-f d`	enable debugging features (tracing, error conversion)
`-h`	print a usage message and exit
`-j`	produce a `.jar` file instead of an executable script
`-o` *exe*	specify output file for executable
`-r`	include a copy of the run-time system in the executable
`-s`	silent: suppress commentary
`-t`	compile with `-f d` (to allow tracing) and initialize `&trace` to -1
`-u`	list undeclared variables
`-x`	execute after compilation (appears *after* file arguments)

The following options are mainly for debugging Jcon:

`-d`	debug *jcont*: use `./jtmp` for temporary files, and do not delete them when finished
`-J`	use Jcon-built versions of *jtran* and *jlink*, if available
`-P`	pessimize: don't optimize the generated Java code
`-S`	also generate a `.jvm` file listing the generated Java code
`-v`	verbose: echo shell commands and trace linking steps

Environment Variables

The following environment variables affect compilation and linking:

`LPATH`	Search path for `$include` directives
`IPATH`	Search path for `link` directives

The following environment variables affect execution:

`CLASSPATH`	Search path for Java libraries required at execution time. This is not usually needed unless the libraries have moved.
`JXOPTS`	Options passed to the Java interpreter that runs the program. Depending on the implementation, JXOPTS can select options such as profiling or run-time

	compilation to machine code.
`TRACE`	Initial value of the Icon keyword `&trace`. If set, it overrides compilation with `-t`.

If a Jcon program runs out of memory, Java's limits can be increased by setting JXOPTS. For example, `setenv JXOPTS '-mx50m'` increases the maximum space to 50 megabytes. See the Java documentation for other Java options.

Jar file output

If `-j` is passed to *jcont*, the final output file is given a `.jar` extension and the header is omitted to facilitate manipulation of the file as an archive. Execution is accomplished by naming the file in the CLASSPATH, along with the run-time library, and then running java:

```
jcont -j myprog.icn
setenv CLASSPATH myprog.jar:/myhome/jcon/bin/jcon.zip
java myprog
```

If the `-r` flag is also used, a copy of the run-time library is included in the the generated `jar` file. This provides a self-contained package that can be executed on any Java implementation, possibly even one of different architecture.

Caveats

java must be in the search path, and also *javap* if `-S` is used.

The `-S` option does not work for files containing `link` directives.

A compiled `.zip` file cannot be renamed: The file name must match the original `.icn` name. (However, executables can be renamed.)

Language Differences

The core Icon language is defined by *The Icon Programming Language, Third Edition* (Griswold and Griswold, 1996). This section documents differences with respect to that book and to the reference implementation, Version 9 of Icon. Differences related to graphics are described in the Graphics section.

Characters and Strings

Like Version 9 of Icon, Jcon uses an 8-bit superset of ASCII. Jcon does not use Java's Unicode character set.

Conversion of real numbers to strings produces more digits than Version 9.

Files

Standard error output (`&errout`) is always unbuffered.

The standard files `&input`, `&output`, and `&errout` cannot be accessed randomly using `seek()` and `where()`.

Processes run by `system(s)` or `open(s,"p")` do not inherit `&input`, `&output`, and `&errout`. Except for the case of `open(s,"wp")`, where it is provided by the program, `&input` is always empty. The two output files, `&output` and `&errout`, are copied from the process after it terminates.

Other Data Types

Random selection from sets and tables differs from Version 9, even with the same random seed.

Jcon is not always consistent with Version 9 when it encounters large integers in unsupported contexts such as subscripting.

Keywords

`&features` includes `"Java"`. A corresponding preprocessor symbol `_JAVA` is predefined.

`&time` produces elapsed wall-clock time, not CPU time, due to Java limitations.

`&allocated`, `&collections`, `&storage`, and `&regions` produce only zero values.

Built-in Functions

The functions `chdir()`, `getch()`, `getche()`, and `kbhit()` are not implemented.

The functions `getenv()` requires the presence of the utility `env` in the shell search path.

The implementation of `loadfunc()` is described in the Dynamic Loading section.

Linking

`link` directives must give a simple name, not a path.

Under Jcon, any reference to a procedure renders it *invocable* (callable by string invocation). In version 9, only procedures reachable from `main()` are made invocable by default.

Debugging

Most debugging features require compilation with the `-f d` switch. Programs compiled with default options cannot be traced, cannot use error conversion (`&error`), and produce an abridged traceback if an error occurs.

Only global variables are available to `variable()`, `display()`, and `&dump`.

Graphics

Icon’s graphics facilities are defined by *Graphics Programming in Icon* (Griswold, Jeffery, and Townsend, 1998). This section documents differences from that specification, supplementing Appendix N ("Platform Dependencies") of the book.

Jcon’s graphics implementation is nearly complete. The biggest omissions are wide lines, textured drawing, and mutable colors. Java 1.2 promises to make the first two more feasible. Mutable colors will remain out of reach.

Graphics Attributes

All attributes listed in the graphics book are implemented, but some cannot be changed successfully. The `canvas` attribute can be set to `normal` or `hidden` but not to `maximal` or `iconic`. Changes to the following attributes are ignored: `linewidth`, `linestyle`, `fillstyle`, `pattern`, `display`, `iconpos`, `iconlabel`, `iconimage`.

Because textured drawing is not available, the `Pattern()` procedure always fails.

Each graphics context is associated with a particular canvas. Thus `Couple()` always fails and `Clone()` accepts only a single window argument.

Fonts

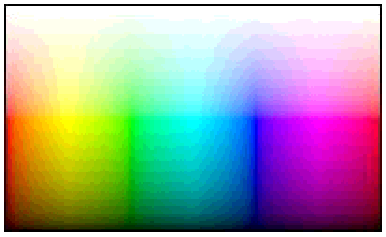
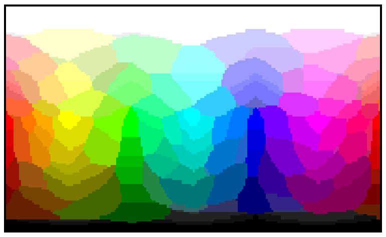
Icon generic family names, and all font characteristics, are case-insensitive; depending on the Java implementation, other font family names may be case-sensitive. Because Java never rejects any non-empty font name, `Font(s)` always succeeds for any well-formed specification.

The generic family names `serif`, `sans`, and `typewriter` work well, including bold and italic variants. The appearance of the generic family `mono` varies by platform, and bold and italic characteristics are not always effective.

The default font name `fixed` is mapped to `mono,bold,12`.

Colors

There is no inherent limit to the number of different colors that can be in use simultaneously. On 8-bit displays, Java approximates colors by selection from a limited palette that can vary from one run to another. The first image below (from Plate 8.2 of the graphics book) shows the surface of color space as rendered on a full-color display. The second image is an example of how Java renders it on an 8-bit display.

For named colors, variants (such as `light` and `dark`) of the unsaturated hues (`brown`, `pink`, and `violet`) appear less saturated than in Version 9 of Icon. Unlike Version 9, variants of `black` remain black rather than producing shades of gray.

The default value of the `gamma` attribute is 1.5.

Mutable colors are not available; `NewColor()` and `Color()` always fail. (In Version 9 of Icon, mutable colors work on 8-bit X displays.)

Images

`ReadImage()` loads images encoded in GIF or JPEG format. If the `ReadImage()` call specifies a color palette, it is ignored; Java dithers the image if necessary for display. Neither `ReadImage()` nor `DrawImage()` ever returns an integer indicating a color shortage.

The `image` attribute can be set at any time, not just initially, to load a GIF or JPEG file. The `image` attribute is readable and returns the filename of the most recently loaded image.

`WriteImage()` attempts to write a JPEG file if the specified filename ends in `".jpg"` or `".jpeg"` (case insensitive). JPEG writing fails for Java versions earlier than 1.2beta4. For all other filenames, a GIF file is written; if the area being written contains more than 256 colors, automatic quantization occurs.

`WriteImage(W,s,x,y,w,h,q)` accepts a final "quality" parameter not present in Version 9 of Icon. Its allowable range is 0.00 to 1.00, with a default value of 0.75. The quality value is passed directly to Java when writing a JPEG image. Its *useful* range is perhaps 0.10 to 0.90: Smaller values produce images dominated by artifacts, and larger values increase file size without producing any visible improvement.

If a quality value of less than 0.90 is specified for a GIF image, the maximum number of output colors is reduced to produce a smaller file for color-rich images.

Image reading and writing ignores the `gamma` attribute.

`Pixel()` clips the specified region by the window bounds before generating values. This differs from Version 9 of Icon, which generates the background color for out-of-bounds pixels. (The graphics book is silent on this point.)

Events

Java distinguishes between ALT and META keys; in Jcon, either key sets the `&meta` keyword.

The `&meta` keyword is never set in conjunction with mouse events; for these events, Java uses the ALT and META flags to encode "which button", so they are unavailable to indicate key states.

Numeric codes for the "outboard" keys such as F1 and HOME differ from those of X and are similar to those of Windows; see the copy of `keysyms.icn` distributed with Jcon.

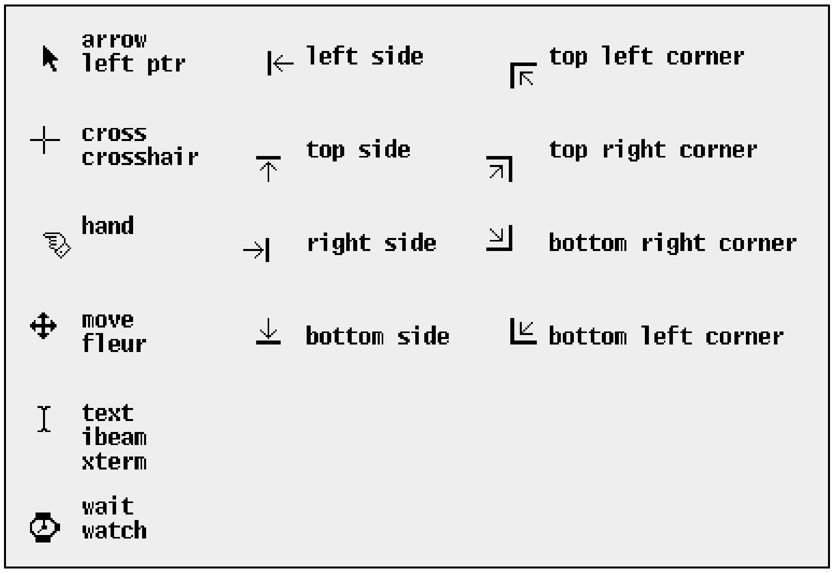
When a window is closed by the user, a `&resize` event is produced with associated `&x` and `&y` values of (-1, -1).

Cursors and Pointers

The visible text cursor, when enabled, is a solid, nonblinking underbar.

The mouse position can be read using the `pointerx` and `pointery` attributes (or `pointerrow` and `pointercol`), but it cannot be set by the program. Attempts to alter these attribute values are ignored.

Acceptable values for the `pointer` attribute are illustrated in the following figure. For compatibility with other systems, some pointers have multiple names.



Deprecated Features

Version 9 of Icon retains compatibility support for some old graphics features that are no longer documented. Jcon omits most such support. Two examples are support for `very light` and `very dark` colors and the acceptance of bi-level images expressed in decimal notation.

Dynamic Loading

The capabilities of Icon can be extended through the use of dynamic loading at execution time. The built-in function `loadfunc(libname,procname)` loads the compiled Icon or Java procedure `procname` from the Zip archive `libname` and returns a procedure value. This value can be called just like any other procedure.

If `libname` is null, the procedure must be a Java procedure, and it is loaded from the same file as the current executable. (Icon procedures linked into the executable are not dynamically loadable.)

While the library archive can be built ahead of time, it is also possible for the running program to generate code and then build it by calling `system()` with the appropriate commands.

In Version 9 of Icon, `loadfunc()` loads procedures written in C. Jcon, in contrast, loads precompiled Java or Icon procedures. Although the Icon interface is similar, it is not possible to load the same procedure with both systems.

Preparing Icon Procedures

An Icon procedure is prepared for dynamic loading by compiling it with a command such as `jcont -c file.icn`. This produces a `file.zip` archive suitable for use with `loadfunc()`.

When a Zip file of Icon procedures is first referenced by `loadfunc()`, all the globals and procedures in the file are linked before the requested procedure is returned. Subsequent `loadfunc()` calls can access other procedures from the file, but the file is not relinked when this is done.

A dynamically loaded Icon procedure can reference globals and procedures defined in the original program, its own source file, and any other files loaded before its own file is first linked. Unreferenced procedures must be declared `invocable` if they are to be referenced by subsequently loaded procedures.

Preparing Java Procedures

Java code can be used to provide programs with additional capabilities not expressible in Icon. Compilation of Java code produces `.class` files which are then bundled up by the `jar` utility to produce libraries for dynamic loading.

Construction of Java procedures requires not only a knowledge of Java but also some understanding of Jcon’s run-time system. A tutorial on that subject is far beyond the scope of this discussion. It is hoped that the key points presented here, combined with inspection of the examples and the Jcon source code, will provide enough of a foothold to allow at least the construction of simple procedures. An understanding of Java is assumed in what follows.

Run-time system basics

The Jcon run-time system is contained in the source directory named `jcon` and forms a Java package of that name. A file containing a loadable procedure declares `import jcon.*;` to gain access to the `jcon` namespace.

Icon values of the various types are instances of classes such as *vInteger*, *vString*, and *vList*. Most of these classes implement a factory method such as `vReal.New(3.14159)` for constructing new instances.

All are subclasses of a parent class *vValue*. Note that Icon strings are not implemented by the Java *String* class but instead by code in the vString class.

The class *vDescriptor* is the superclass of most run-time classes. It encompasses vValue as well as other objects that represent things such as subscripted strings. The `vDescriptor.java` source file lists a large number of methods that operate on vDescriptors and vValues. A vDescriptor `d` can be dereferenced to produce a vValue either by calling `d.deref()` or by calling an operation that implicitly dereferences it, such as `d.Negate()`.

Procedures in Java

In Java, an Icon procedure is a subclass of *vProc* that defines a public `Call()` method that returns a vDescriptor object.

A procedure that expects two arguments extends the class *vProc2* (which extends vProc) and defines a `Call` method that accepts two vDescriptor arguments and returns a vDescriptor result. More generally, a procedure expecting *n* arguments, for 0 <= *n* <= 9, extends vProc*n* and declares *n* vDescriptor arguments.

A procedure that expects more than nine arguments is written by extending the class *vProcN* and declaring a `Call` method that accepts an array of vDescriptors as its single argument. vProcN can also be used for any other procedure when an array of arguments is more convenient than using a fixed argument list.

The arguments passed to the `Call` method are not dereferenced. In the Jcon implementation, this is the responsibility of the called procedure. Often this is done by using the vDescriptor arguments in operations that implicitly dereference them.

The `Call` method returns a Java null to fail or a vDescriptor, usually a vValue, to succeed. (An *Icon* null value is produced by calling `vNull.New()` and returning the result.) Suspension will be covered in the next subsection.

Here is a procedure that accepts three arguments, coerces them to integer, and returns the sum:

```
import jcon.*;

public class sum3 extends vProc3 {

    public vDescriptor Call(vDescriptor a, vDescriptor b, vDescriptor c) {
        vInteger i = a.mkInteger();
        vInteger j = b.mkInteger();
        vInteger k = c.mkInteger();
        return vInteger.New(i.value + j.value + k.value);
    }

}
```

This procedure could be used as follows:

```
procedure main()
   local sum3
```

```
       sum3 := loadfunc("sum3.zip", "sum3")
       write(sum3(5, 8, 11))
    end
```

With the source code in `sum3.java` and `sumtest.icn`, the shell commands would be something like this:

```
    jcont -u sumtest.icn
    setenv CLASSPATH /myhome/jcon/bin/jcon.zip
    javac sum3.java
    jar cf sum3.zip sum3.class
    ./sumtest
```

Many examples of procedures can be found in the `jcon/f*.java` files in the Jcon distribution. These files implement Icon's built-in functions.

Suspension

A procedure suspends by returning an instance of class *vClosure*. This is another subclass of vDescriptor, so the declaration of the `Call` method does not change. The vClosure object encapsulates two key items:

- a `retval` field containing the value being suspended
- a `Resume()` method for generating subsequent values

In general, any procedure that suspends requires its own subclass of vClosure to implement its particular `Resume()` method. Java's "inner classes" are useful for this.

The `Resume()` method takes no arguments and returns a vDescriptor. It is called to produce the next value when the suspended procedure is resumed. `Resume()` can do one of four things:

- fail, by returning a Java null
- suspend, by returning a vDescriptor object
- suspend, by setting `this.retval` and returning itself ("`return this;`")
- abort, by calling `iRuntime.Error()`

It is not possible for `Resume()` to "return" in the Icon sense. It must instead suspend a value and then fail upon later resumption.

Here is an example of a procedure that generates the factors of an integer. To avoid a special case, even the first value is produced by calling the `Resume()` method.

```
    import jcon.*;

    public class factors extends vProc1 {

        public vDescriptor Call(vDescriptor a) {
            final long arg = a.mkInteger().value;
            return new vClosure() {
                long n = 0;
                public vDescriptor Resume() {
                    while (++n <= arg) {
                        if (arg % n == 0) {
```

```
                        retval = vInteger.New(n);
                        return this;
                    }
                }
                return null; /*FAIL*/
            }
        }.Resume();
    }

}
```

The vClosure object is created, called, and returned by the large `return` expression,

```
return new vClosure() { ... }.Resume();
```

which encompasses the entire definition of the anonymous subclass of vClosure.

It is very important that the `retval` field be set before returning a vClosure object; a null `retval` is illegal.

Performance

Programs built by Jcon typically run somewhat slower than when built and run by Version 9 of Icon. We use the ratio of execution times as our basic benchmark measurement. The result depends on many things, but a factor of two or three is typical with a good Java system.

We have measured execution times for the standard Icon benchmarks and for three long-running additional applications. The standard benchmark programs were taken from Icon v9.3.1 and run unmodified, but some data files and command options were changed to make them run longer. The benchmark programs are as follows:

concord	produces a text concordance (word index)
deal	deals bridge hands
ipxref	cross-references Icon programs
queens	places non-attacking queens on chessboard
rsg	generates random sentences
tgrlink	optimizes vectors for drawing street maps
geddump	dumps a genealogical data base
jtran	translates Icon into Java class files

We did most of our performance tuning on a Silicon Graphics Indigo2 running SGI Java 3.1 (JDK 1.1.5). This is a good Java implementation running on a fast machine. It uses just-in-time (JIT) compilation to convert JVM code to machine code as needed. There is a minimum one-second startup cost for every execution, which we attribute to initialization and JIT compilation; this cost is included in the measurements below but is not the dominating factor.

Here are execution time ratios measured on several platforms:

platform		concord	deal	ipxref	queens	rsg	tgrlink	geddump	jtran
SGI Irix 6.2	JDK 1.1.6	3.6	9.2	2.7	1.5	3.2	3.2	2.2	3.4
Sun Solaris 2.6	JDK 1.1.6	5.2	6.1	4.3	3.4	5.1	4.1	3.5	4.6
Sun Solaris 2.6	JDK 1.2beta4	2.7	3.7	1.9	1.5	3.0	1.8	1.9	2.6
IBM AIX 4.1.5	JDK 1.1.4	6.6	10.0	4.4	2.2	6.2	3.5	3.6	N/A
Digital Unix 4.0B	Fast JVM b1	6.3	10.1	8.7	3.4	6.6	3.6	6.7	5.4

Release Notes for Version 2.1

Version 2.1 of Jcon includes minor feature additions, documentation edits, and bug fixes. Changes include the following:

- A directory can be read by opening it as a file.
- JPEG images can be written under Java 2 implementations.
- Java class files can be bundled with a Jcon program for easier dynamic loading.
- The run-time system can be bundled with a Jcon program to make it completely self-contained.
- Large integers now work with `to-by`, `seq()`, limitation (`e1 \ e2`), and exponentiation (`e1 ^ e2`)
- A zero increment value is diagnosed by `seq()`
- `&host` no longer spawns a shell invocation of `uname` (but see *Problems*, below)
- The run-time package name has been changed from `rts` to `jcon`. All programs must be recompiled.

Tested Platforms

Jcon has been successfully tested on:

Sun Sparc / Solaris 2.6 (SunOS 5.6) / Sun Java 1.3beta0
SGI Indigo2 / Irix 6.5 / SGI Java 3.1.1 (JDK 1.1.6N)
IBM RS6000 / AIX 4.3 / IBM Java 1.1.6-19990401
DEC Alpha / Digital Unix 4.0D / Digital Java 1.1.6-2
Intel / Linux 2.2.5 (Red Hat 6.0) / IBM JDK 1.1.6-990814

We would be interested to learn of either successes or failures on other platforms.

Known problems (nongraphical)

- Some JIT compilers fail to execute Jcon programs correctly. Disabling the JIT compiler produces correct execution.
- The `-S` option of `jcont` does not work for programs that link other files.

- `&clock` and `&dateline` may be off by one hour (Java bug #4059431).
- On Solaris systems, `&host` may return "localhost" (Java bug #4073539).
- Extremely large procedures (thousands of lines long) can generate code that is too large for Java to handle.

Known graphics problems

Some problems are universal:

- When an obscured part of a window is exposed, it may not be repaired (redrawn) until the program pauses to await an event.
- `WOpen("image=file.gif")` does not load an image if presented with a multi-part (animated) GIF image.

Other problems are seen only on some platforms and are attributed to Java bugs:

- Fonts can be poor when one vendor's Java system displays on another vendor's X server.
- Different fonts may be written to the window and to and its backup image, leading to bizarre effects from `CopyArea()` and window repair.
- On one older system, nothing appears in a window until it is dragged to a different location.
- The initial window size may not be as specified:
  - It is one pixel too wide, or
  - It is always the default size, or
  - Its height is correct but its width is the default.
- Intermittently, `drawop=reverse` draws the wrong color.
- GIF images have a yellowish tint.
- The `CopyArea()` section of the `gpxtest` program shows a minor glitch.

Installation

Building Jcon is simple if the requisite software is available. Automated tests are provided to validate the build.

Requirements

Jcon is written in Icon, Java, and Korn Shell.

Unix with the Korn shell

Building and running Jcon requires various Unix utilities including the Korn Shell. Any system that has `/bin/ksh` will probably work. Most commercial Unix systems supply `ksh`, and the `ksh` clone on Linux also works.

Installed Software

Jcon requires a recent version of Icon and version 1.1 of Java. Prebuilt binaries of these are available for many platforms. Installation of Java may require system privileges, and proper functioning of some Java systems may require installation of OS patches.

For software see:

Icon 9.3:	http://www.cs.arizona.edu/icon/v93u.htm
Java 1.1:	http://www.javasoft.com/products/jdk/1.1/

Search Path

Executables of `icont`, `javac`, `java`, and `jar` must be in the search path to build the Jcon system. Only `java` is needed to run `jcont` or the executables it produces.

Building Jcon

Ensure that your search path is correct, as described above. Set your current directory to the top level of the Jcon distribution, and type `make`. There are no configuration options.

When the build completes, the `bin` directory contains everything needed to run Jcon. It can be used in place or copied elsewhere. The `html` directory, which contains documentation, is also worth keeping. The rest of the Jcon distribution can be deleted if no longer wanted.

After building Jcon using `icont`, it is possible use Jcon to rebuild itself. This optional step is accomplished by typing `make jj`; it enables the use of `jcont -J`.

Testing Jcon

The Jcon test suite includes new tests written for Jcon, old tests from the Icon v9.3 distribution, and a few Icon applications. The tests are run by typing `make test`. A successful run produces the names of the tests, one at a time, as they run.

In addition to the automated tests, Jcon comes with additional tests that can be run manually. The `demo` directory contains graphics programs that are interesting as demonstrations. The `gtest` directory contains other graphics programs that are less interesting and require visual comparison with the displays produced by Version 9 of Icon. The `expt` directory contains a few additional nongraphical tests and experiments.

Directory Structure

The subdirectories within the Jcon tree are as follows:

`bin`	target for all build products; also includes `jcont` script
`tran`	source code for the translator
`jcon`	source code for the run-time system

`html`	files for building documentation pages
`test`	automated test collection for validating Jcon
`gtest`	manually run test collection for graphics
`demo`	demonstration programs utilizing graphics
`expt`	other tests and experiments to be run manually
`bmark`	benchmark collection

Each directory contains a `README` file with further information.

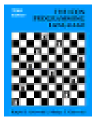
Contact Information

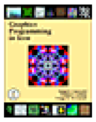
The Jcon home page is located at http://www.cs.arizona.edu/icon/jcon/. Go there for the lastest versions of the implementation and documentation.

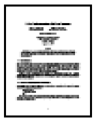
Please send all Jcon-related mail to jcon@cs.arizona.edu. Using this alias helps us log the mail and respond more promptly.

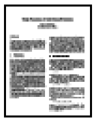
The home page for the Icon language is located at http://www.cs.arizona.edu/icon/.

References


Ralph E. Griswold and Madge T. Griswold
The Icon Programming Language, Third Edition
Peer-to-Peer Communications, 1997
ISBN 1-57398-001-3


Ralph E. Griswold, Clinton L. Jeffery, and Gregg M. Townsend
Graphics Programming in Icon
Peer-to-Peer Communications, 1998
ISBN 1-57398-009-9


Todd A. Proebsting and Gregg M. Townsend
A New Implementation of the Icon Language
Technical Report 99-13 (PostScript, PDF)
Department of Computer Science, The University of Arizona


Todd A. Proebsting
Simple Translation of Goal-Directed Evaluation
Proceedings of the 1997 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI).
Las Vegas, 1997 (PostScript, PDF)