HISTORY OF THE ICON PROGRAMMING LANGUAGE

Ralph E. Griswold and Madge T. Griswold

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

ABSTRACT

The Icon programming language, which was conceived in 1977, was strongly influenced by the earlier SNOBOL languages and the subsequent SL5. This paper concentrates primarily on the early development of Icon, but also discusses subsequent versions. The motivation, design philosophy, and environmental factors that shaped Icon are emphasized in this paper.

CONTENTS

12.1 Background
12.2 Rationale for the Content of the Language
12.3 A Posteriori Evaluation
12.4 Implications for Current and Future Languages
Acknowledgments
References

12.1 BACKGROUND

12.1.1 Source and Motivation

Icon is the latest in a series of programming languages that started with the string-manipulation language SNOBOL [Farber 1964]. Evolution of the SNOBOL languages led to SNOBOL4 [Griswold 1968a], which has patterns as first-class values, and augments string-processing facilities with sophisticated data structures. Several problems are evident in SNOBOL4. The ones of most concern in subsequent research are its lack of conventional control structures and a dichotomy between conventional computation and pattern matching [Griswold 1980a].

A subsequent language, SL5 ("SNOBOL Language 5") [Griswold 1977d], integrates SNOBOL4's success/failure signalling mechanism with conventional control structures. SL5 also has a very general procedure mechanism, in which coroutines follow as a natural consequence [Hanson 1978b]. A feature
similar to SNOBOL4's pattern matching is provided in terms of scanning environments [Griswold 1976b]. Although SL5 is an improvement over SNOBOL4 in some ways, it is unsatisfactory in others. In particular, its very general procedure mechanism is overbearing in some respects, leading to unnecessarily complex programs.

A spark of insight inspired Icon: SL5's very general procedure mechanism is not necessary for pattern matching—a much simpler suspension/resumption mechanism will do.

Icon was conceived in the form of a proposed return to "old values." It was envisioned by Ralph Griswold [1977a] as

... a programming language "successor" to SNOBOL4, having the following properties:

SNOBOL4 philosophic and semantic basis
SL5 syntactic basis
SL5 features, excluding the generalized procedure mechanism.

The concept of *generators*, expressions that can produce a sequence of results by suspension and resumption, is the key to expression evaluation in Icon. Generators, in combination with goal-directed evaluation, allow pattern matching to take place in a conventional computational context [Griswold 1981a].

Icon turned out to be considerably different from what was envisioned in 1977, but that was the starting point.

12.1.2 The Name

"SNOBOL5" was the first name used for the language that was to become Icon [Griswold 1977a]. One issue in the choice of a name was the degree of connection it would imply with the SNOBOL languages. "Product identification" with the SNOBOL languages was viewed as giving the new language credibility and visibility, but it had the disadvantage of looking backward instead of forward. More significantly, the use of "SNOBOL" in the name would be misleading if the language turned out to be (as it did) very different from the SNOBOL languages.

So a new name was needed. Dave Hanson suggested "s"—a homage to "C" with its minimalistic one-character name, but in lowercase, reflecting a distaste for computer-generated text that, at the time, still was frequently printed in all uppercase. As an abstraction from "SNOBOL" and "SL5", "s" made some sense without drawing too close a connection between the old languages and the new one. But "s" wasn't a happy choice for several reasons, not least of which was its typographical difficulties in documentation (which forced the awkward quoting used here).

Over a period of months, the name "s" was disparaged and sometimes ridiculed ("ssssssss"). Other names were suggested ("irving," "bard," and "TL" ("The Language")), but nothing stuck until Madge and Ralph Griswold suggested "icon" (then with the initial lowercase).

The name "icon" is not an acronym, nor was any particular meaning attached to it, although the word "iconoclast" was immediately offered as describing the flavor of the new language.

This choice of name was made before Xerox started to use it for little screen images. It was only later that confusion arose between "Icon, the programming language" and screen icons. By then, it was too late. It didn't help that the programming language significantly predated the usage that is so common now.

Although an occasional person has mistaken Icon as a programming language for manipulating screen images, the confusion has been less than might be suspected and has never been a serious problem. Perhaps this is because so many company and product names now include the substring "icon" [Griswold 1989a].
12.1.3 Versions of Icon

In order to put the Icon programming language in perspective, it is necessary to understand that there have been eight versions of the language, spanning a period of more than 12 years. The first version of the language captured the essential ideas [Griswold 1978c]. Version 2 took some of the rough edges off the first version and added a few new features [Griswold 1979].

After completion of the second version of Icon, it became clear that there were many possibilities for further refinements and extensions. Some of the new features in subsequent versions were the consequence of research. Other new features were provided in response to requests from a growing user community. Figure 12.1 shows the chronology of Icon language versions and Table 12.1 shows the main differences between versions.

As is typical with programming languages that mature through a series of versions, Icon’s computational repertoire increased substantially over time, mostly in response to requests from users for additional facilities. The number of operations, functions, and keywords gives a rough measure

FIGURE 12.1
Version Release Dates.

<table>
<thead>
<tr>
<th>Version</th>
<th>Release Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inception</td>
<td>1976</td>
</tr>
<tr>
<td>Version 1</td>
<td>1977</td>
</tr>
<tr>
<td>Version 2</td>
<td>1978</td>
</tr>
<tr>
<td>Version 3</td>
<td>1979</td>
</tr>
<tr>
<td>Version 4</td>
<td>1980</td>
</tr>
<tr>
<td>Version 5</td>
<td>1981</td>
</tr>
<tr>
<td>Version 6</td>
<td>1982</td>
</tr>
<tr>
<td>Version 7</td>
<td>1983</td>
</tr>
<tr>
<td>Version 8</td>
<td>1984</td>
</tr>
<tr>
<td></td>
<td>1985</td>
</tr>
<tr>
<td></td>
<td>1986</td>
</tr>
<tr>
<td></td>
<td>1987</td>
</tr>
<tr>
<td></td>
<td>1988</td>
</tr>
<tr>
<td></td>
<td>1989</td>
</tr>
<tr>
<td></td>
<td>1990</td>
</tr>
<tr>
<td></td>
<td>1991</td>
</tr>
</tbody>
</table>
TABLE 12.1
Feature Changes.

<table>
<thead>
<tr>
<th>Version</th>
<th>Features</th>
<th>Functions</th>
<th>Operators</th>
<th>Keywords</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The original language [Griswold 1978c].</td>
<td>45</td>
<td>27</td>
<td>15</td>
<td>87</td>
</tr>
<tr>
<td>2</td>
<td><strong>Added</strong>: assignment to matching functions and reversible assignment [Griswold 1979].</td>
<td>47</td>
<td>33</td>
<td>19</td>
<td>99</td>
</tr>
<tr>
<td>3</td>
<td><strong>Added</strong>: string transformations, augmented assignment, and external functions. <strong>Removed</strong>: post-fix operators [Coutant 1980a].</td>
<td>51</td>
<td>45</td>
<td>19</td>
<td>115</td>
</tr>
<tr>
<td>4</td>
<td><strong>Added</strong>: mutual evaluation, repeated alternation, and limitation. <strong>Changed</strong>: stack and queue access to lists [Coutant 1981a].</td>
<td>44</td>
<td>54</td>
<td>20</td>
<td>118</td>
</tr>
<tr>
<td>5</td>
<td><strong>Added</strong>: co-expressions. <strong>Removed</strong>: string transformation and assignment to matching functions [Wampler 1983b; Coutant 1981b].</td>
<td>43</td>
<td>70</td>
<td>20</td>
<td>133</td>
</tr>
<tr>
<td>6</td>
<td><strong>Added</strong>: link declarations, set data type, new sorting options, programmer-defined control operations, and string invocation. <strong>Removed</strong>: external functions [Griswold 1983c, 1986b].</td>
<td>48</td>
<td>76</td>
<td>35</td>
<td>159</td>
</tr>
<tr>
<td>7</td>
<td><strong>Added</strong>: functions to access operating-system facilities, bit functions, variable-length argument lists, error conversion, error trace back, and access to storage management information [Griswold 1988a].</td>
<td>65</td>
<td>76</td>
<td>35</td>
<td>176</td>
</tr>
<tr>
<td>8</td>
<td><strong>Added</strong>: mathematical functions, keyboard functions, arbitrary-precision integer arithmetic, list invocation, external call interface, and instrumentation of storage management [Griswold 1990b].</td>
<td>89</td>
<td>76</td>
<td>36</td>
<td>201</td>
</tr>
</tbody>
</table>

of the size of Icon's computational repertoire. (The measure is only rough, since some operations, functions, and keywords have more computational functionality than others.) Table 1 shows the size of Icon's computational repertoire for the different versions of the language. Note that the repertoire more than doubled between the first version and the last.

12.1.4 Project Organization and Staffing

The Icon programming language was designed and implemented in the Department of Computer Science at The University of Arizona. It was funded primarily by grants from the National Science Foundation, with supplementary support from the University of Arizona.

From an organizational point of view, Icon actually was a by-product of research whose primary goal was the design of new linguistic facilities for non-numeric computation. Icon brought together new ideas from this research, in the context of a complete programming language, allowing the ideas to be evaluated and refined in a real computational context.

The initial design team consisted of faculty members Ralph E. Griswold and David R. Hanson, and Ph.D. student John T. “Tim” Korb. Walter J. Hansen, also a Ph.D. student, provided support. Cary
A. Coutant and Stephen B. Wampler, Ph.D. students in Computer Science, played major roles in the design and implementation of Versions 3 and 4. As the Icon project developed, several other graduate students participated in significant ways. These included David Gudeman, Clinton L. Jeffery, William H. Mitchell, Kelvin Nilsen, Janalee O’Bagy, and Kenneth Walker.

In the later stages of the development, support also was provided by the laboratory staff of the Department of Computer Science. Gregg M. Townsend contributed to both design and implementation, while Sandra L. Miller and Phillip Kaslo assisted with various aspects of the implementation. Many persons outside the University of Arizona participated in a variety of ways. This participation was informal, out of personal interest, and on a voluntary basis. The most significant outside contributions were made by Robert J. Alexander, Alan Beale, Mark B. Emmer, Owen R. Fonorow, Robert E. Goldberg, Andrew P. Heron, Robert McConeghy, Jerry Nowlin, John Polstra, Christopher Smith, and Cheyenne Wills.

The backgrounds and technical experience of the participants in the development of Icon were varied. Ralph Griswold had many years of experience in the design and implementation of programming languages, starting at Bell Telephone Laboratories in 1962. Dave Hanson had worked at Western Electric Engineering Research, prior to his doctoral work, which was completed at the University of Arizona in 1976 with a dissertation related to SL5. Gregg Townsend received his master’s degree in computer science at the University of Arizona and also had considerable professional experience in programming. The graduate students who participated had varying backgrounds.

As in most academic environments, the organization of the design team was somewhat informal. As time passed, some individuals left the project and others joined it. In particular, students participated during their degree programs. Ralph Griswold and Dave Hanson led the project at the beginning. Ralph Griswold continued to lead the project in later versions.

Unlike the development of many programming languages, the design and implementation of Icon were not distinctly separated. Instead, the major participants viewed themselves as designers, implementors, and users of Icon. Design and implementation were, in fact, strongly coupled with results from experimental implementations affecting the design. Almost all of the persons who contributed to the design of Icon also contributed to the implementation.

The initial implementation of Icon was done by Dave Hanson and Tim Korb, with assistance from Walt Hansen. A second and entirely different implementation was done by Cary Coutant and Steve Wampler. This implementation was later refined by Bill Mitchell, Ralph Griswold, and others. As the implementation matured, many of the persons mentioned above contributed in various ways.

12.1.5 Costs and Schedules

Since Icon was a by-product of a research program, the primary goal of which was the design of new programming language features and not the production of a new programming language per se, it is not possible to determine accurately the costs involved.

The funding, provided largely by grants from the National Science Foundation, strongly influenced both the nature of the work and the time frame under which it was undertaken. There were no fixed schedules, nor was the scope of the project ever clearly specified. Many developments were the consequence of new research ideas and feedback from users.

For similar reasons, it is not possible to determine accurately the manpower expended on the design and implementation of Icon. All participants divided their time, as is customary in academic environments. Averaged over the period of Icon development, one faculty member and two graduate students devoted about half their time to Icon. A number of persons, outside the University of Arizona,
devoted significant amounts of time on an irregular basis, but the total amount of their contribution is impossible to determine.

Since language design and implementation were closely coupled, it is not possible to assign specific portions of the effort. In the early part of the project, effort on design dominated implementation. As the language matured, the opposite was true. On the whole, probably three or four times as much effort was devoted to implementation as was devoted to design.

Other related activities—documentation, distribution, and support—accounted for a considerable amount of the total effort. The effort expended on these activities was irregular and primarily associated with different releases of Icon. In recent years, these supporting activities have dominated the work, as implementations have proliferated and an increasingly large user community has made greater demands.

Because of the “co-mingling” of more fundamental research, and the specific activities related to programming-language design and implementation, it is not possible to determine the “cost” of producing Icon with any degree of accuracy. Perhaps $100,000 was expended before the first working version was distributed to the public. Perhaps $600,000 has been expended in the development, implementation, and distribution of subsequent versions.

12.1.6 The Influence of Other Programming Languages

The developers of Icon had extensive experience with other programming languages. Ralph Griswold’s early work on SNOBOL had been influenced by SCL [Lee 1962], COMIT [Yngve 1958], and IPL-V [Newell 1965]. By the time work on Icon was initiated, he had been influenced by BCPL [Richards 1969], Bliss [Wulf 1971], EL1 [Wegbreit 1970], APL [Iverson 1962], and C precursors [Johnson 1973]. And, of course, he had a working knowledge of the more widely known programming languages. Dave Hanson had considerable experience with SNOBOL4, a broad knowledge of most other programming languages, and had been a principal in the design of SL5. Graduate-student knowledge of programming languages came primarily from their academic contact and work with their colleagues.

Since Icon developed over several years, other newly developing programming languages, such as Prolog [Cohen 1985] and C [Ritchie 1978] became known to the persons working on Icon.

While knowledge of other programming languages inevitably influenced the design of Icon, there was a conscious attempt not to be influenced unduly by other programming languages, but instead to do something distinctly different. Clearly, SNOBOL4 and the subsequent SL5 had profound influence and were shaping forces in the design of Icon. Nevertheless, there was a deliberate attempt not to copy from other languages or to develop refined versions of their features. A philosophy of language design, more than the nature of existing languages, was the driving force.

12.1.7 Design Philosophy

The design of Icon was driven by a philosophy of what a programming language should be. Although this philosophical basis was never formalized, it was generally agreed that Icon should be easy to use. This goal was based on the view that programming is largely a human activity and that human beings are more important than computers.

Closely related to this view was the desire to develop fundamentally new linguistic mechanisms, instead of refining and improving the existing ones. This inevitably meant Icon should be distinctly different from other programming languages, as the SNOBOL languages were. Frequent references
were made to the work being “orthogonal to the mainstream.” These references were not entirely in
jest.

12.1.8 Intended Use

Icon was not designed for any specific application area, although non-numeric computation (the
manipulation of strings and structures) was its primary focus and SNOBOL4’s widespread use of
computing in the Humanities was suggestive for Icon [Griswold 1982a]. Instead, Icon was viewed as
a tool that would be suitable to certain programming contexts. There were two main contexts of
interest, influenced largely by the designers’ experience: (1) situations in which quick-and-easy
programming solutions were needed; and (2) very complex research applications that strained the
capabilities of other programming languages. It is interesting that, although these two contexts are so
different, they did not produce tension in language design.

Although Icon is now widely used for an enormous range of applications, many of which do not
occur in the contexts originally envisioned, it remains the case that many uses of Icon are either
“one-shot,” quick-and-dirty applications, or very sophisticated ones. This sometimes gives Icon a
rather schizoid appearance to potential users—they may hear that Icon is good for simple tasks and
then see powerful and sophisticated applications written in Icon. It is partly for this reason, also, that
Icon often is compared to programming languages such as Awk [Aho 1988], REXX [Cowlishaw
1985], and Perl [Wall 1991], even though Icon is very different from them.

As expected, Icon has come into widespread use for computing in the Humanities, supplanting
SNOBOL4 in many cases, although SNOBOL4 still retains many staunch supporters.

Just as Icon was not designed with a specific application area in mind, neither was it designed for
any particular programmer profile. It was expected that the persons who used Icon would have some
prior programming experience with a procedural language (C is the current referent).

No specific compromises were made in the design of Icon to cater to particular kinds of
programmers or specific programming and application experience. It was realized, from the begin-
ning, that trying to design a language that would be significantly different from other languages would
lead to some problems for experienced programmers. The need to “unlearn” concepts from other
programming languages was foreseen. On the other hand, the designers of Icon did not attempt to
make it particularly accessible to programming novices or to make it suitable as a first programming
language.

12.1.9 Implementation

The initial development of Icon took place in a mainframe environment. The envisioned language
features clearly required extensive resources and no concessions in design were made so that Icon
could run on smaller computers.

The language design probably was influenced, even if unconsciously, by the physical environment
in which its first implementation was done—a DEC-10 running in a time-sharing environment. The
primary mode of operation was interactive text entry and editing, but not interactive programming
per se. Thus, Icon programs could be run in batch mode as well.

Portability was a central concern in the implementation [Hanson 1980]. Considerable thought was
given to the language in which Icon would be implemented. Since portability was a major issue,
assembly language was ruled out. Languages commonly used at that time for the implementation of
other programming languages, especially C, BCPL, and Minimal [Dewar 1977], were given serious
consideration but rejected, partly for technical reasons but mainly for concern about their availability and support, as well as lack of local experience in their use.

The possibility of designing an implementation language was also considered but rejected on the basis of experience with the implementation of SL5, which was done this way [Korb 1977]. This approach was viewed as too much work, especially since the implementation language would have to be transported, also.

The decision finally was made to use FORTRAN with the Ratfor preprocessor [Hanson 1978a] to support better program structure. The choice of FORTRAN was not embraced with enthusiasm by all members of the project. FORTRAN was not viewed as particularly suitable as an implementation language. It was almost a question of proving such an implementation was possible.

The Ratfor implementation of Icon generated FORTRAN code, which then was compiled and linked with a run-time system written in Ratfor. This implementation of Icon proved to be quite portable, although large and somewhat inefficient. However, it required a robust FORTRAN compiler and considerable computational resources. Consequently, this implementation of Icon turned out to be limited to large computers—the CDC 6000, the DEC-10, the IBM 370, and the VAX. Ironically, newer FORTRAN compilers for smaller machines proved a barrier to this implementation of Icon. Their more rigorous error checking often rejected FORTRAN code produced by the Icon compiler that, nonetheless, corresponded to de facto FORTRAN IV.

Initially the only computer facilities available to the Icon project were the DEC-10 and CDC 6600 in the Computer Center at the University of Arizona. The Department of Computer Science acquired its first computer, a PDP-11/70, in 1979 and began to run UNIX. This local computer, albeit limited in resources, was very appealing for program development, and in the same year the C implementation of Icon was started on the department’s computer [Coutant 1980b].

This implementation effort was necessarily very concerned about memory requirements, since the PDP-11/70’s combined I and D spaces allowed only 128K bytes of memory to be addressed. At this point, portability was discarded as an implementation goal in favor of obtaining a workable and easily accessible implementation of Icon for local use. This change in emphasis was greeted with considerable relief, since much of the effort expended in the Ratfor implementation had been the consequence of portability concerns.

Although the C implementation of Icon on the PDP-11 was not designed with portability in mind, the implementation was based on a virtual machine that had instructions suitable for the implementation of Icon language features [Coutant 1980b; Wampler 1983c; Griswold 1986a]. An Icon program was translated into virtual machine instructions, which then were interpreted, interfacing a run-time system written in C. This conceptual framework eventually proved important for C implementations on other platforms and became the basis for tools for measuring the performance and behavior of Icon programs [Coutant 1983].

It is worth noting that, at the time the C implementation of Icon was started, there was no way of knowing that C would become available on most computers, running under most operating systems.

The C implementation was successful, despite the limited address space available. By the time it was running, implementations of C had begun to proliferate, and other persons expressed an interest in adapting the C implementation of Icon to their platforms. This began the process of making the C implementation of Icon portable. It is a story in itself, involving not just Icon but the maturation of C compilers and the evolving ANSI C Standard [American 1990].

At the time of this writing, the C implementation of Icon runs on nearly 60 different UNIX platforms, as well as the Acorn Archimedes, the Amiga, the Atari ST, CMS, the Macintosh, MS-DOS, MVS, OS/2, OS-9, and VMS. It is particularly ironic that an implementation that disregarded portability as a goal has, in fact, become highly portable.
12.1.10 Documentation

Documentation was a major consideration from the initial design of Icon. In fact, while Dave Hanson and Tim Korb took the major responsibility for the first implementation effort, Ralph Griswold was the official amanuensis for the project.

Since Icon was developed in a research environment within an academic organization, the initial documentation was primarily for internal use, first by the designers themselves, and then by other local users [Griswold 1977b, 1977c, 1978a, 1978b]. As the early design progressed to a working implementation, more attention was given to user manuals [Griswold 1978c].

As Icon came into widespread use, the demand for documentation led to books on the language [Griswold 1983a] and its implementation [Griswold 1986a]. The language book was revised in 1990 to correspond to Version 8 of Icon [Griswold 1990a, 1990b].

Starting in 1978, many aspects of Icon were documented in the technical report series, published by the Department of Computer Science at the University of Arizona. To date, there have been over 100 such technical reports. As use of Icon spread and the amount of documentation related to it grew, it became necessary to develop a system to keep track of less formal and more specialized documentation, such as user manuals for specific platforms. At the same time, it became important to be able to identify communications concerning Icon. The Icon Project became a semi-official entity and a series of numbered Icon Project Documents (IPDs) was started in 1986. More recently, technical documentation about Icon, too specialized for general distribution as technical reports, has also appeared in Icon Project Documents. To date, there have been over 200 IPDs.

The Icon Newsletter was started in 1978 [Griswold 1978d] to provide information to interested parties outside the University of Arizona. This newsletter was essentially a redirection of an earlier SL5 newsletter [Griswold 1976a], which in turn was an offshoot of The SNOBOL4 Information Bulletin [Griswold 1968b], which started in 1968. Thus, there was a ready community of interested persons when Icon came on the scene, many of whom had prior knowledge of SNOBOL4 and SL5.

The Icon Newsletter initially provided both news of topical interest and technical information about Icon. In 1990, a second newsletter, The Icon Analyst [Griswold 1990e], was started to cover material of a more technical nature, while The Icon Newsletter took on a more topical role.

12.2 RATIONALE FOR THE CONTENT OF THE LANGUAGE

12.2.1 Environmental Factors

12.2.1.1 Program Size

Little thought was given to the potential size of Icon programs during the initial design phases. Because of the programming contexts envisioned, it was expected that many Icon programs would be small and that larger ones usually would be written by one person. No specific attention was given to very large programs written by teams of programmers and no language facilities were included specifically to deal with the problems of writing such programs.

The lack of facilities to support the development of large programs did not pose significant problems in the early use of Icon. As Icon became more widely known, however, programmers started to use it for larger projects, where the lack of program structuring facilities began to pose problems.

This is just one of several cases where extensive use exposes limitations that were not foreseen. In fact, use of a programming language tends to produce requirements for facilities that even may be
inappropriate at the time of its initial design. Conversely, if a programming language does not have significant use, such considerations are irrelevant.

### 12.2.1.2 Program Libraries

Little thought was given to program libraries initially, but the need for separate compilation of program modules was recognized and supported, starting with Version 3 of Icon.

More attention was given to program libraries as the community of users increased and wanted to share program material. An Icon program library was established by the Icon project in 1983 [Griswold 1983e] and was placed on a subscription basis in an expanded form in 1990 [Griswold 1990f].

As of this writing, the Icon program library contains 132 complete programs and 152 packages of procedures.

### 12.2.1.3 Portability

Portability was a significant concern in the design of the Icon programming language, as well as its implementation, as mentioned earlier.

There are several views about portability, as it concerns language features. At one extreme, a language can be viewed as portable only if it provides ready access to all the features of any platform on which it runs. This is a tall order for platforms that have libraries of hundreds, if not thousands, of graphical and audio routines.

At the other extreme, a language can be viewed as portable if any program written in it runs on any platform on which the language is implemented. This is more likely to be the case if the language supports few, if any, platform-specific facilities. On the other hand, lack of support for platform-dependent facilities limits what can be done with the language on some platforms.

The initial design of Icon tended toward the latter view, providing only simple, sequential input and output, for example. As use of Icon increased on different platforms, users needed more capabilities for input, output, and file manipulation, as well as access to operating-system capabilities from within Icon programs.

The C implementation of Icon attempted to deal with these problems largely by passing the buck to C, casting its system operations in terms of those of C. For example, the system() function was added to allow commands to be executed in a shell.

At the time the C implementation of Icon was started, C was only readily available on UNIX platforms. As C implementations proliferated, its "UNIXisms" brought an alien flavor to other platforms. This problem was dealt with, to some extent, in the ANSI C Standard [American 1990].

The last version of Icon retains the C flavor in its interface to operating-system capabilities, but looks less like a "UNIX language" now than it did before C came into widespread use on many platforms.

As anticipated at the beginning of the language design, extensions have been added to the language to meet the needs of those using different platforms. Of course, such extensions render programs that use them nonportable on platforms that do not have them.

One approach that was considered to minimize these problems, was the design of platform-independent facilities to subsume platform-specific ones. The problem with this approach is that a particular facility cannot be cast in the "native" way for all platforms. Graphics and window management are particularly difficult in this respect.
Nonetheless, portability is not a concern of most users, who would rather have a facility available in the way that is natural on their platforms. The designers of Icon decided in favor of the majority of users, leaving platform-specific facilities to be provided in platform-specific ways.

12.2.1.4 Efficiency

Speed of execution was never a significant factor in the design of Icon; rather the opposite. Part of the reason for largely disregarding efficiency was the philosophical goal of designing a programming language that would be easy to use and minimize programming time and effort. This goal is in essential conflict with execution speed. Another reason for largely disregarding efficiency was the view that such concerns inhibited creativity and the invention of new linguistic mechanisms during the design process. Past experience had shown that features whose initial implementation was slow could be implemented efficiently once they were better understood [Dewar 1977, 1985]. In this sense, many efficiency considerations were deferred with the expectation that clever implementation techniques would be developed to overcome otherwise slow execution speed.

It is virtually impossible, however, for language designers who also are programmers to entirely ignore considerations related to efficiency—these considerations enter design decisions unconsciously, if not consciously. On a few occasions, efficiency was a specific factor in language design. One of these concerned matching expressions in string scanning. Starting with Version 2, strings could be assigned to matching functions to change the value of the scanned subject. This capability significantly slowed string scanning, even when the capability was not used. Assignment to matching functions was removed in Version 5, partly because of efficiency considerations but also because it was not widely used and because changing the subject, especially its length, during string scanning, tended to be confusing and conceptually ill-formed.

12.2.1.5 Programming Ease

Ease of programming was a major concern in the design of Icon. This was a natural consequence of the philosophical position that human beings were more valuable than computers. Particular attention was given to expressiveness—the ease and brevity with which computations could be formulated.

Another consideration was avoiding aspects of programming languages that are designed either to assist in generating good object code or to prevent bad programming. The intention in the design of Icon was to make it easy to write good programs, rather than difficult to write bad ones. The lack of a compile-time type system in Icon is a reflection of these views.

12.2.1.6 Character Set

Since string manipulation was a major component of Icon, the choice of a character set was given careful consideration. The possibility of using an internal character set different from that of host computers on which Icon ran, was considered. In particular, a very large character set offered the possibility of representing internally the large number of characters needed in problem domains such as linguistics and typesetting. This idea was rejected, however, because of the lack of peripheral devices to represent such characters and potential implementation problems.

An eight-bit character set was chosen because it naturally fits most computer architectures and because it provides enough different characters for most applications. It is important to know that Icon allows all characters in the underlying character set to occur in strings, including the null character, control characters, and other characters with no external graphic representations.
Ralph E. Griswold & Madge T. Griswold

The question of which eight-bit character set to use was more difficult. The problem involved language design, implementation, and the interface between programs and the systems on which they run. In the discussion of character sets, Dave Hanson [1977] retorted “ASCII rules the world! What’s an EBCDIC?” The ASCII character set was eventually selected for the first version of Icon.

For the C implementation of Icon, which initially disregarded portability, the choice of a character set was easy—it obviously was ASCII. As portability of the C implementation became a concern, the issue of the character set for use on EBCDIC-based platforms arose. The first edition of the Icon language book [Griswold 1983a, p. 234] stated that the underlying internal character set for Icon is ASCII, regardless of the native character set of the platform on which Icon is implemented. This statement was based largely on the incorrect assumption that implementations in C for EBCDIC-based computers would most naturally use ASCII internally.

Several design issues arose regarding character sets, even for character sets of the same size:

1. Availability of characters used to represent programs on peripheral devices such as keyboards and printers;
2. Differences in graphics;
3. Collating (sorting) sequences.

When the first EBCDIC implementation of Icon was started, one of the obvious problems was that Icon, from ASCII heritage, used braces, square brackets, and other such characters to represent important syntactic structures in programs, while many EBCDIC peripherals did not support these characters. These problems were solved by providing alternative characters and digraphs. For portability, these constructions are supported on ASCII-based Icon implementations as well.

Collating sequence is a somewhat more troublesome problem, since upper- and lowercase letters and digits appear in quite different relative positions in the ASCII and EBCDIC collating sequences.

There are two obvious choices—using the native collating sequence or translating to a standard internal character set on input and output. In the former case, string comparison and sorting are done in the native manner for the platform. In the latter case, string comparison and sorting are the same for all platforms, but “unnatural” on EBCDIC platforms. This, again, is a trade-off between what best suits most users (results that are the same as for other programs on the local platform) and portability (results that are the same for all platforms).

There are, in fact, two EBCDIC implementations of Icon, both based on the implementation of Icon that originally supported ASCII. One of these takes the former view of collating sequences [Beale 1989] and one takes the latter [Schiller 1989].

12.2.1.7 Standardization

Standardization was not an issue in the design of Icon. It was felt that considerations of standardization would stifle innovation in a programming language in which innovation was a prime concern.

On the other hand, the availability of the highly portable public-domain implementation discouraged other essentially different implementations that would have been a potential source of major dialectic differences, providing a kind of de facto standardization of the central part of Icon.

It is easy to make extensions to Icon, especially additions to its function repertoire, and there are many of these. Some are platform-specific and in general use, while others have been done by individuals for their own use.
12.2.2 Functionality

The primary concern in designing the functionality of Icon was a repertoire of features for processing nonnumerical data—strings and structures. At the same time, it was expected that Icon would be a general-purpose programming language and, therefore, would have a large repertoire of conventional operations. As shown in Table 12.1, both aspects of the functionality of Icon increased over time, with new versions. Because of the growing user community, it became impractical in later versions of Icon to remove or substantially change functionality.

Except for the expansion of the function repertoire, the most notable changes in functionality occurred in Version 4 of Icon, with new control structures [Coutant 1981a]; in Version 5 of Icon, with co-expressions [Coutant 1981b]; in Version 6 of Icon, with sets [Griswold 1986b]; and in Version 8, with arbitrary-precision integer arithmetic [Griswold 1990a].

The addition of sets shows how capricious major additions to the functionality of a programming language can be. The first version of Icon provided tables with associative look up as one of the major data structures carried forward from SNOBOL4. Sets, which are in some sense more natural to programming, came about much later as a class project for a graduate “language internals” course on the implementation of Icon. Since tables already existed, most of the mechanism for implementing sets was available and most students were able to complete the project satisfactorily. The results of one of the best projects were added to Version 6 of Icon.

If tables had not existed, adding sets to Icon would have been impractical in the context of the course, and they probably would not have been added to Icon. Yet, sets had never been considered as a language feature for Icon before this class.

12.2.3 The Design Process

The design of Icon was dominated by the goals mentioned earlier: the development of new features for string and list processing, innovation rather than refinement, and ease of use. Because of the importance placed on these goals, the design of Icon was not done by systematically applying any particular principles or methodologies. Instead, the design process emphasized informality and encouraged “brainstorming.” The designers met frequently, often on a daily basis, to discuss issues and work out the details of language design. Much of the communication was via electronic mail. This electronic mail was informal, often brash or funny, and sometimes zany. Occasionally it even bordered on lunacy.

A central, even dominating, aspect of early Icon language design was experimentation. Ideas for language features were implemented almost at once, using modified versions of SL5 and high-level interpreters designed for quickly testing new ideas.

Although Icon made a significant break with SL5 and SNOBOL4, these earlier languages had considerable influence on the design of Icon, sometimes suggesting that things be done differently and sometimes suggesting that they be done in similar ways. Thus, the experience with SNOBOL4 and SL5 had a major impact on Icon.

There were two areas of major concern: (1) pattern matching and control structures; and (2) data structures. The design of features for these two areas did not interact significantly. In fact, attempts at integrating the two failed. Sometimes work proceeded more on one and sometimes more on the other.

Although the initial design of Icon was not governed by any specific design methodologies, later work was influenced by the abstract characterization of sequences [Wampler 1983a] and formal semantic models [Gudeman 1991].
12.2.4 Distribution

The distribution of Icon was a major factor in its acceptance. Since Icon was developed in an academic setting, the normal commercial forms of distribution were unavailable and inappropriate.

One question was the legal status of Icon. In the tradition of the SNOBOL languages [Griswold 1981b, pp. 612–614] and SL5, Icon was placed in the public domain without any of the usual restrictions (such as proprietary notices, copyrights that permit unlimited copying, and licensing requirements). Since Icon originated in an academic environment and was funded primarily by grants from the National Science Foundation, public-domain status, without any qualifications, was viewed as appropriate and desirable. It was in the best interests of the computing community and it would contribute to Icon's acceptance. The public-domain status applies to source code as well as to executable files, and no attempt was made to prevent individuals from modifying the source code for their own purposes.

Early versions of Icon were distributed on magnetic tape, at no charge to recipients. In some cases, persons interested in Icon were asked to provide a magnetic tape on which Icon was then written. In the case of persons likely to make a contribution to the design or implementation of Icon, tapes usually were provided free of charge. Similarly, documentation was provided at no cost during the early years of Icon.

As Icon matured and attracted a larger group of interested persons, it became impractical for the Icon project to underwrite the costs of media, printing, and shipping—what can be given freely to 100 persons is impractically expensive for 1,000.

Gradually, a system was developed to provide Icon on various magnetic media for a nominal charge. The charge was somewhat more than actual costs. The net revenue was used to purchase software and hardware needed to support Icon, to underwrite the (free) Icon Newsletter, and so forth.

In a time of rapid evolution of computer systems, distribution formats became an increasing problem. While magnetic tapes in a few formats sufficed in the early days of Icon, later there was a need for various floppy disk formats and data cartridges. While it was not feasible to supply Icon for every conceivable media format and packaging, many packages were eventually developed for its distribution. As of this writing, the Icon Project provides Icon for 13 different platforms, in 12 different formats, on magnetic tape, data cartridges, and three kinds of floppy disks.

In 1987, Icon material was made available via FTP for network transfer, and an electronic bulletin board (RBBS) was established to allow persons to download Icon material via telephone. In addition, Icon is available on many other electronic bulletin boards and conferencing systems.

Consequently, Icon is widely available for a nominal cost. Combined with its public-domain status, this makes Icon accessible to most interested persons. This has contributed, substantially, both to its usefulness and its acceptance in the computing community.

While there is no way to determine the number of persons who use Icon, the size of the Icon-user community can be estimated from the fact that over 13,000 copies of Icon have been distributed by the Icon Project on magnetic media and more than 30,000 copies of Icon-related files are downloaded via FTP annually.

Feedback from users indicates a large user population in the academic community, primarily in computer science and the humanities. Icon also is used extensively in companies that produce software, as well as in industrial and governmental research laboratories. Perhaps the largest community of users consists of individuals using Icon on personal computers for diverse applications.
12.3 A POSTERIORI EVALUATION

12.3.1 Meeting of Objectives

It is not surprising that the design of a language that stressed innovation and language characteristics over specific applications, users, or markets, turned out to be somewhat different from what was originally envisioned.

12.3.1.1 Icon and SNOBOL4

Icon is, to a certain degree, a successor to SNOBOL4, as was originally anticipated. Icon is appropriate for the same kinds of applications as SNOBOL4. In fact, Icon is much the same kind of clientele as SNOBOL4. Icon, however, strongly resembles SNOBOL4 only in its types and their handling: strings as atomic data objects, structures with pointer semantics, associative look up in tables, and so forth.

Icon has a much larger repertoire of low-level string-processing operations than SNOBOL4. String scanning in Icon, unlike pattern matching in SNOBOL4, is integrated into the rest of the language. String scanning is, however, lower-level and more imperative in nature than pattern matching in SNOBOL4.

Icon has a syntax typical of modern imperative programming languages, making Icon programs much different in appearance from SNOBOL4 programs. Icon also lacks some of the most powerful features of SNOBOL4—run-time compilation and operator redefinition.

The most significant fundamental difference between Icon and SNOBOL4 is in expression evaluation. Icon’s generators and goal-directed evaluation have a profound effect on programming techniques [Griswold 1981a, 1982b, 1988c]. They not only remove the dichotomy between conventional computation and pattern matching found in SNOBOL4 [Griswold 1980a], thus allowing conventional computation during pattern matching, but they also greatly increase the expressiveness of conventional computation. As a consequence, the approaches to programming in Icon are much different from those in SNOBOL4— which is what makes the two languages so different in practice.

12.3.1.2 Icon and SL5

While SNOBOL4 still is generally available and in widespread use, SL5 is a dead language. Consequently, the differences and similarities between SL5 and Icon are less significant than those between SNOBOL4 and Icon.

As originally envisioned, Icon has a syntax that is similar to that of SL5. And, as planned, it lacks SL5’s very general procedure mechanism. Much of Icon’s repertoire of operations and functions consists of a refinement to SL5’s. Again, the big difference between Icon and SL5 lies in expression evaluation.

12.3.2 An Evaluation of Icon

An evaluation of Icon can be divided into three general categories: pleasant surprises, results as expected, and disappointments.
12.3.2.1 *Pleasant Surprises*

As mentioned earlier, prior research, as well as the insight that sparked the design of Icon, revolved around issues of expression evaluation and, in particular, around ways of providing conventional computational facilities during pattern matching.

Generators and goal-directed evaluation solved these problems, but they did much more. It was quite a while before it was apparent that the converse was true: generators and goal-directed evaluation greatly enrich all kinds of computation. In fact, pattern matching (cast as string scanning) is an almost trivial by-product of a general expression-evaluation mechanism that supports generators and goal-directed evaluation.

12.3.2.2 *Results as Expected*

Icon's handling of types and values, as well as its data structures, satisfies the original design goals. They provide substantial, but not revolutionary, advances over those of SNOBOL4 and SL5 [Griswold 1989b].

Coupled with better control structures than those of SNOBOL4, Icon's data-structuring capabilities have proven to be popular for rapid prototyping [Fonorow 1988; Fraser 1989]—an application that was not anticipated or considered in the design of Icon. In fact, in such applications Icon often is used in a style that can be characterized as "C with automatic storage management." The higher-level features of Icon provide programming leverage that allows prototypes to be built quickly, and changed easily, while retaining much of the program structure and organization of C.

The choice of an expression-based syntax for Icon was given careful consideration and chosen for the generality it provided. The results of this decision were largely as expected, with an expression-based syntax making the identification of syntax errors more difficult. An expression-based syntax, on the other hand, allowed more expressiveness, albeit with a potential loss of clarity [Griswold 1992].

12.3.2.3 *Disappointments*

String scanning is one portion of Icon that did not achieve the design objectives. By integrating backtracking control structures with conventional ones, many of the problems with pattern matching in SNOBOL4 were solved, but at a considerable expense. The higher-level, declarative nature of patterns and their status as first-class values is lost in Icon. String analysis in Icon is much more imperative than pattern matching in SNOBOL4. The programmer has more flexibility, but must deal with more detail at a lower level. Furthermore, the increased flexibility in string scanning brings with it problems in program structure. String scanning expressions often are poorly organized. While there are disciplines to overcome these problems [Griswold 1980c, 1983d, 1990d], they are somewhat demanding and are rarely used.

It is worth noting that the somewhat unsatisfactory nature of string scanning in Icon was recognized during the early development of the language. Several attempts were made to solve the problem, but they were unsuccessful. At the time that the first edition of the Icon language book was being written, it was recognized that the book, by its existence, would inevitably limit future language changes. The book was deferred for a while in hope that some improvement could be made to string scanning, but eventually the effort was abandoned with the recognition that the problems with string scanning would not be solved in the context of Icon.
12.3.3 Contributions

A programming language makes contributions in many ways, ranging from concepts, to specific features, to providing a useful tool. The most important conceptual contribution in Icon is its method of expression evaluation with generators and goal-directed evaluation. This method of expression evaluation allows searching, in pattern matching and other applications, to be integrated with conventional computation. As a result, many computational tasks can be expressed more easily and concisely in Icon than in other programming languages.

It is difficult to quantify ease of use in a particular programming language. It is a subjective and somewhat personal matter, but it is nonetheless important. Icon programmers who also program in C often contrast the two languages. For programming tasks of moderate size, an Icon program often is about one tenth the size of a corresponding C program and usually can be done in about one tenth the time. For larger tasks, the difference usually is less, with a factor of one half to one third being more typical.

Another, even more subjective, factor is psychological. Programmers often characterize Icon as a "fun language." Programming is, after all, largely a human activity. If the programming process is enjoyable, programmers are more motivated and productive.

One measure of the success of a language like Icon is whether it allows programmers to do things they otherwise would not attempt. This has been repeatedly reported by Icon programmers.

12.3.3.1 Other Implementations and Dialects

The ready availability of the highly portable interpreter for Icon [Griswold 1986a, 1990c] eliminated the usual motivation for most alternative implementations of programming languages—the need for the language on a platform for which there is no implementation. Instead, other implementations of Icon have focused mainly on two issues: efficiency and conceptual models for the implementation of generators and goal-directed evaluation.

Thomas Christopher [1985] addressed both issues in an implementation for the VAX-11. This implementation was brought to the point where the design could be evaluated, but it was not completed. Andrew Freeman [1985] proposed a stackless implementation model for generators and expressions. David Gudeman [1986] produced a prototype implementation of a subset of Icon in T, using continuation semantics as an implementation model. Mark Bailey [1990] implemented a similar idea in Scheme.

O'Bagy [1987] designed a conceptual model for implementing generators and goal-directed evaluation in which recursion was a central aspect of interpretation. She subsequently cast this model in the context of compilation and designed a framework for optimizing the generated code for expression evaluation in Icon [O'Bagy 1988].

The first commercial implementation of Icon, based on the implementation done at the University of Arizona, was done for the Macintosh [Prolcon 1989]. This implementation supports the standard Macintosh visual interface and includes a number of language extensions. A second commercial implementation was done for UNIX 386/486 platforms [ISI 1991]. It also includes several language extensions.

Recently Ken Walker developed techniques for type inference and liveness analysis in Icon and combined these with O'Bagy's expression-evaluation model in a production-quality optimizing compiler for Icon [Walker 1991].
Other implementation work related to Icon includes Kelvin Nilsen’s on-the-fly garbage collection algorithm for producing constant-time execution speed [Nilson 1988b], and his related work on Conicon [Nilson 1988a]. Kelvin Nilsen and John Martinek also designed a temporary variable model for the virtual machine that underlies the Icon interpreter [Martinek 1989a, 1989b, 1989c].

12.3.4 Mistakes

The original design of Icon, as mentioned earlier, stressed elimination of unnecessary “baggage.” As a result, defaults were provided so that programmers would not have to specify the typical or obvious usage. In most cases, these defaults work well, but the default scoping for local identifiers proved to be misguided. If an identifier is not declared otherwise, its scope is taken to be local. While this is a reasonable choice for programs that are compiled from a single file, it can introduce mysterious errors in the case of the separate compilation of a program from several files: A global declaration in one file may change the scope of an otherwise undeclared identifier in another. In retrospect, it would have been better to require scope declarations for all identifiers.

It also probably was a mistake not to consider facilities to aid in the development of very large Icon programs. This problem is addressed in a commercial version of Icon [ISI 1991].

12.3.5 Problems

Programmers have had problems with several aspects of Icon. In most cases, these problems are not mistakes in the design of Icon, but rather a consequence of its difference from most other programming languages and a necessary by-product of its valuable characteristics.

Programmers who have learned programming languages, like Basic and Pascal, before learning Icon, sometimes have difficulty with Icon’s concepts of success and failure. They tend to feel that success and failure are merely hidden Boolean true and false values. The consequence of this misconception can be difficulty in learning Icon and using it properly.

Icon’s pointer semantics for structures, also, sometimes cause problems. Problems with unintentional aliasing in this regard are well known. More subtle problems occur when a structure is the default value for a table and is not replicated when used for different new table keys. Despite these problems, the usefulness of pointer semantics in efficiently handling large and complex data structures, as well as the ability to produce program data structures that are isomorphic to abstract structures in the problem domain, outweigh the problems.

As mentioned earlier, programmers have trouble using string scanning in a well-structured manner. There also is a tension between Icon’s low-level string operations and its higher-level string scanning. The two linguistic levels clash, but programmers nonetheless often mix them; in fact the language provides no alternative to this in some cases. SNOBOL4 programmers often have more difficulty with string scanning than other programmers. SNOBOL4 programmers, in particular, miss the ability to compose complex patterns out of simpler ones.

SNOBOL4 programmers also often lament Icon’s lack of SNOBOL4’s CODE and EVAL functions that allow strings to be compiled and executed during program execution.

The lack of a conversational, interactive interpreter for Icon, also, is frequently cited as a deficiency. This problem is being addressed by Blanchard [1991].
12.4 IMPLICATIONS FOR CURRENT AND FUTURE LANGUAGES

12.4.1 Direct Influence

Several programming languages have been inspired, at least in part, by various aspects of Icon. Generators have had the greatest influence. They have been incorporated in dialects of several existing languages: Cg: for C [Budd 1982], Little Smalltalk [Budd 1987], and Pi for Pascal [Gallesio 1986]. Generators also appear in the multi-paradigm languages Leda [Budd 1992], and G [Placer 1991], and in the "hacker's language" SPLASH [Abrahams 1989].

The relationship between generators and sequences of values is evident in programming languages that manipulate "streams," such as those formulated by Nakata and Sassa [Nakata 1991]. At a higher level, Seque developed the idea of generators to the level of first-class values [Griswold 1988b].

Several other programming languages have been based on Icon or strongly influenced by it in more general ways. EZ [Fraser 1983, 1985] adapted several aspects of Icon in a language that subsumes facilities normally provided by operating systems. Rebus [Griswold 1985] recast SNOBOL4 with a syntax derived from Icon. Logicon [Lapalme 1986] is a hybrid formed by combining many of the features of Icon and Prolog. CommSpeak [Nilsen 1986], a programming language designed for real-time string-processing applications, also contains several features of Icon. CommSpeak evolved into Conicon [Nilsen 1988a], which augments Icon with a stream data type [Nilsen 1990a] and control structures for handling concurrency [Nilsen 1990b]. Walker [1989] added patterns to Icon as first-class data objects. Idol [Jeffery 1990] adds object-oriented facilities on top of Icon.

12.4.2 Indirect Influence

Icon's indirect influence on other programming languages, current and future ones, can be viewed in terms of the specific linguistic characteristics of Icon or in more general terms.

In the areas of specific linguistic characteristics, Icon's expression-evaluation mechanism is certainly its most influential feature. In addition to the inclusion of generators in the languages mentioned previously, Icon's control structures have influenced work such as Leichter [1984], Magma2 [Turini 1984], and Grandi [1986]. The view of strings and string operations used in Icon also has influenced work on string handling [Hansen 1992], and Icon's use of character sets has led to new programming techniques [Griswold 1980b].

The philosophy and character of Icon may have more far-reaching, if less identifiable, influences. In many respects, Icon runs contrary to the current conventional wisdom about programming-language design. It favors the programmer over computational efficiency, it attempts to make it easy to write good programs rather than imposing barriers to bad programming practices, and it is fun to use. Icon is a reminder that programming need not be excessively encumbered with restrictions in order to produce good programs.

The philosophy and mechanism of the distribution of Icon certainly are influential. In a time when commercial considerations influence programming languages in major ways, there are significant social and legal concerns and an essential tension between creativity, access, and proprietary rights. The proliferation of various pseudo-legalisms, such as shareware and freeware, reflects conflicts between access, intellectual property rights, and computation.

Icon is relatively unusual in being entirely in the public domain (except for commercial implementations) and in being very accessible. The evident contribution that unrestricted and ready access has made to the acceptance of Icon will surely have some influence on the complicated and difficult issues related to software distribution.
ACKNOWLEDGMENTS

As noted throughout this paper, many persons contributed to the design and implementation of the Icon programming language. The persons cited are by no means all of those who participated in one way or another in the development of Icon—there are literally hundreds of others.

The authors would like to express their appreciation to Dave Hanson for many helpful suggestions on the presentation of material in this paper.

REFERENCES

PAPER: HISTORY OF THE ICON PROGRAMMING LANGUAGE


ICON SESSION 619
RA LPH E. G RISWOLD & M ADGE T. G RISWOLD


OLIN QUIDY (Jackson Laboratory): Is there any truth to the rumor that the name “Icon” was derived from a regional pronunciation of the phase “I can, I con, Icon”?

RALPH GRISWOLD: The name Icon didn’t come from anywhere actually. I’ve been quoted earlier as saying finding a good name for programming languages is a lot harder than designing a good programming language. And I seem to have some evidence of that. The name doesn’t stand for anything; it seemed nice at the time. I suppose “iconoclast” came to mind, but that really wasn’t where it came from. Incidentally, the use of “icon” for a little image on the screen, wasn’t in general use. If we had known that, we wouldn’t have picked it, because it has caused some confusion. Having picked the name, people have made various attempts to integrate it into a presumptive history of the origin of the name. But, no, there is nothing to that rumor.

CHRISTIAN HORN (Turtwangen): Had you some Prolog features in mind when designing Icon, or do you regard the generate-test paradigm, for structuring algorithms using backtracking techniques, as a universal multi-purpose language feature?

GRISWOLD: There are really two questions. Let me address the first one, did we have Prolog features in mind. No, at the time we started designing Icon, which was somewhat before the implementation that came out in 1978, we were not aware of Prolog, nor was Prolog aware of Icon. I remember being on a panel about languages, giving short talks at an ACM Conference. A person got up and described Prolog, I got up and described Icon, and we looked at each other. Someone in the audience stood up and said “What’s the difference?” And neither of us had heard about the other’s efforts. There is quite a bit in common, actually, between Icon and Prolog, though they are cast very differently. Down inside, at some more fundamental level, there is a great deal in common. I think that it is basically something that was an idea that was going to happen. Our background came from pattern matching and SNOBOL4, and we generalized that kind of search and backtrack capability; that dates all the way back to 1968, and that was where we were coming from. But our language is in, of course, an imperative context, which gives the programmer a considerable amount of control over what actually goes on. It is possible to limit generation and control structures in various ways. The two kinds of things are different; but no, the two languages didn’t even know about each other, initially. So it was, I think, a historical accident that two somewhat similar features, at least to the core, happened at approximately the same time.

JAY CONNE (Consultant): You spoke of Icon combining data structures with string manipulation, added to the other functions. Do you have any published guidelines for maintaining good structure?

GRISWOLD: The question concerns the quality of programs, the way the programs are structured. In Icon, they use string scanning, and that is the only context in which my remark was made. The problem really is that it allows too much freedom. The pattern-matching facilities of SNOBOL4 limited you, pretty much, to string analysis and generation of alternatives and goal-directed evaluation. Icon string analysis lets you do arithmetic, I/O, and everything else right there in the middle of string analysis. Yes, we do have published guidelines, which really constitute a program methodology on
how one should write string processing using string scanning in Icon. We encourage people to follow those, but it is very much a learned kind of a thing. It took us a while to figure out how to write well-structured string scanning; what to do, what not to do. It doesn’t come naturally from the language, and that’s where I consider the result wasn’t as successful as we had hoped.

RONALD FISHER (No affiliation given): You also designed, as far as this person remembers, Rebus. Was that a precursor of, or a parallel development to, Icon?

GRISWOLD: Rebus was an attempt to provide some of the nice modern syntax in Icon for SNOBOL4 users. It followed Icon. One of the things we discovered, when we designed Icon, was there were a lot of SNOBOL4 fans out there and they really didn’t want to give up SNOBOL4. But they were dealing with a language whose only control structure, beside the procedure call, was the conditional "goto," and they had to fabricate loops with labels and gotos. So Rebus was an attempt to give SNOBOL4 an Icon-like syntax with ordinary looping control structures, and things like that. It was done with a pre-processor that translated Rebus into SNOBOL4. It was an experimental implementation, although we don’t support it anymore. But it definitely followed Icon, because we used Icon syntax to try to recast the very old language semantics of SNOBOL4 in a more modern and acceptable package.

DANIEL SOLOMON (University of Manitoba): Do you have a new language in the works, so that you can present it at HOPL III?

GRISWOLD: No!

BIOGRAPHY OF RALPH E. GRISWOLD

Ralph E. Griswold received his Ph.D. in Electrical Engineering from Stanford University in 1962. His doctoral work was in the area of switching theory and his dissertation dealt with iterative switching networks. Dr. Griswold also holds a B.S. in Physics with honors and great distinction and an M.S. in Electrical Engineering, both from Stanford University.

In 1962 Dr. Griswold joined the staff of Bell Laboratories, where he worked in the Programming Research Department on symbolic computation and the design and implementation of high-level languages for nonnumerical applications.

In 1967 Dr. Griswold was appointed supervisor of the Computer Languages Research Group and in 1969 was appointed head of the Programming Research and Development Department. Groups under his supervision were engaged in programming language research, minicomputer interfaces, and the development of large application programs for the Bell System. His personal research continued in the areas of program portability, programming language design and implementation, software engineering, and computer-based document preparation systems.

In 1971, Dr. Griswold left Bell Laboratories to start the Department of Computer Science at the University of Arizona, where he served as department head until 1981. At the University of Arizona, he has continued to conduct and direct research with emphasis on programming languages, nonnumeric data processing, program portability, and programming methodology. In 1990 he was appointed Regents' Professor of Computer Science.

Dr. Griswold is author, or co-author, of several programming languages, including SNOBOL, SL5, and Icon. He is the author, or co-author, of six books on programming languages and programming language implementation.
BIOGRAPHY OF MADGE T. GRISWOLD

Madge T. Griswold received a BA in History and Journalism from Syracuse University in 1962 and an MA in history from the University of Arizona in 1974.

Ms. Griswold joined Bell Telephone Laboratories at Holmdel, N.J. in 1962 as a technical editor. From 1962 to 1968 she served as an editor for Bell Laboratories technical documentation, including book production. In addition, she was responsible for maintenance and programming modifications for early document-preparation systems.

In 1968, Ms. Griswold became a member of the Programming Research and Development Department at Bell Laboratories, Holmdel. She served as editorial and technical advisor, and documentation writer for document preparation systems. She also was a member of the research group investigating the interface of document-preparation systems with computerized typesetting systems.

Since 1971, Ms. Griswold has been a consultant and free-lance writer specializing in computing applications and computer-based publication techniques. She has served as consultant to the Department of Computer Science at the University of Arizona for computer-assisted document preparation, computer-assisted typesetting, and desktop publishing. She has also served as editorial consultant for book creation and production. She is coauthor of four books on computer programming languages and is president of The Bright Forest Company, Tucson, Arizona, which specializes in advanced software tools and book publishing.