The Icon Newsletter

No. 25 — November 1, 1987

Odds and Ends

Subscriptions to the Newsletter

We've decided to continue to distribute the *Newsletter* to all interested persons without charge. The *Newsletter* is, however, expensive to prepare, print, and mail, so we need to keep the size of the mailing list under control. Because the *Newsletter* is free and anyone can get on the subscription list, we tend to accumulate persons who have only a passing interest in Icon.

To handle this problem, we occasionally send out a renewal form with a *Newsletter* and require that this form be returned for continued subscription. The "cost" to subscribers therefore is just returning the form. The problem is, of course, that it's easy to forget to return the form — and then you never hear from us again.

From time to time we get requests like "put me on your mailing list forever". We regret that we can't accommodate such requests. We think that reaffirming your interest and address every year or so is not too much to ask in return for a free subscription.

Incidentally, if you think you've "fallen off" our mailing list, you can always write or call and ask to be reinstated.

Use of Our Mailing List

We do not sell the mailing list that we use for the *Newsletter*. Occasionally, we provide mailing labels to another organization for some purpose related to Icon.

We also prepare a printed list of our subscribers that's available free to persons who want to locate others with common interests. If you do not want us to include your name on this printed list, let us know and we'll take care of it.

Contacting the Icon Project

When you contact the Icon Project by electronic mail, be sure to include a postal mailing address in addition to your electronic one — electronic mail does not always work properly. If you don't get a reply, you can't tell if we got your message at all, if we failed to answer, or if our answer was lost in the ether. However, if our electronic mail to you bounces, we'll send a backup message via the post.

Reporting Problems

When you report a problem with running Icon, *please* give us the values of Icon's &version and &host for your system. We can't undertake to explain difficulties unless we have this information. You'll also get better response from us if you send the program and data that produced the problem. If the program and data are more than a few lines long, send them on a diskette or tape; we won't keyboard from long listings.

Implementation News

There is not much in the way of implementation news this time. Worth noting are:

The VAX/VMS Icon system was updated in June. This release uses a different strategy for dealing with VMS memory management, fixing a problem occasionally seen by programs using many files. Another fix now correctly handles icode files larger than 65,536 bytes. This new release has the same version number as the preceding one (6.3), but &version has a date of June 16, 1987. Orders filled starting June 17, 1987 contain this new release. If you are in doubt about what release you have, check the date in &version.

Icon source code for the Atari ST is now available. It requires Lattice C Version 3.03 or higher to compile.

See the order form at the end of this *Newsletter* for ordering information. Note that the two implementations described above are the only new ones since the last *Newsletter* (June, 1987).

An Extension Interpreter for Icon

David Notkin and Bill Griswold at the University of Washington provide this description of some of their recent work.

The Extension Interpreter (EI) is a set of tools that allows system builders to incorporate a procedure-based extension facility into their programs. This facility permits incremental addition of procedures to a running program, supporting applications that can be augmented by users without modifying the original program source or executable.

Extensions are written in existing languages. We currently have one EI implementation for Icon and a separate one for C. We are combining the implementations to form a multilanguage system that supports a (nearly) transparent interface between C and Icon procedures (but not between variables). Procedure calls are handled by a stub mechanism similar to that used in remote procedure-call facilities; however, all calls take place in a single address space. The stubs, which are automatically generated by an Icon program, handle not only the call itself but also type translation and the signaling of failure.

The Icon Newsletter

Madge T. Griswold and Ralph E. Griswold

Editors

The Icon Newsletter is published aperiodically, at no cost to subscribers. For inquiries and subscription information, contact:

Icon Project
Department of Computer Science
Gould-Simpson Building
The University of Arizona
Tucson, Arizona 85721
U. S. A.

(602) 621-2018

Significant enhancements were made to Icon to permit extendability and calls between C and Icon. To permit dynamic loading of Icon programs into a running Icon program, we implemented a dynamic linker for Icon programs. The implementation is similar to an Icon dynamic linker implementation by Bill Mitchell at The University of Arizona in the construction of his Ice (Icon-Emacs) editing system. Calls to newly loaded procedures are handled by the proc procedure or string invocation in Icon, which permits calling a procedure by its string value. The current implementation does not garbage collect unreachable procedure instances.

The basic approach to supporting calls from C to Icon was supplied by Andy Heron of the Government Communications Headquarters, Cheltenham, England. We have generalized Heron's approach by allowing the invocation of (1) arbitrary Icon procedures (not just main) from C and (2) arbitrary C procedures from Icon. Further, calls between C and Icon can be interleaved to an arbitrary depth. This requires making each call from Icon into C appear as a normal Icon procedure call, rather than as a call to a built-in function: the stack must be kept in a consistent state for the recursive call to the interpreter. To do this we provide an Icon built-in function callC to handle the calls appropriately. This routine takes the string name of the C routine and the parameters to the procedure. The EI supports the required mapping from a string name to a C implementation.

All modifications to Icon were made using its personalized interpreters mechanism. The dynamic linker was a bit more than a personalization, since it required changing data structures in the linker.

From Our Mail

You make up most of these questions don't you?

No, (although this one is contrived). We do, however, sometimes make composites from several similar questions.

What does "Icon" stand for?

"Icon" is neither an acronym nor a play on words. It's just a name we picked, quite a while ago. In fact, we picked the name before the word came into common use to describe those little images on the computer screen that represent objects and functions. Had we anticipated the confusion that resulted, we would have picked a different name. It's too late now.

I want to take a course at your school to learn to program in Icon. How do I apply?

Our department is in a state university. It offers graduate degrees in computer science and some undergraduate courses to support other degree programs. It does not teach how to program in Icon *per se*, although there are courses that use Icon. You can't just go to the university and take a course; you must be admitted as a student. Basically, we don't provide what you want.

My dog ate my copy of the last Icon Newsletter. Can you send me another one (Newsletter, that is)?

Sure. We hope your dog is no worse for the experience.

Can I order free Icon documents by electronic mail?

Yes, you can order free material any way you like, but please provide us with a postal mailing address, since we do not distribute documents electronically.

I know it's expensive to publish the Newsletter. While I understand why you'd prefer not to charge for subscriptions, why don't you ask for voluntary contributions?

We'd prefer not to solicit contributions or to have our subscribers feel some kind of implied obligation. That's also the reason we don't distribute Icon as shareware. However, we do get occasional contributions, which are both very welcome and helpful. Corporate contributions are particularly welcome.

Is there an Icon Users' Group?

Not that we know of, aside from the loose affiliation provided by subscribers to this *Newsletter*. We frequently are asked how to contact persons interested in Icon in a particular area. If you are interested in forming a regional or local group, send us your name; we'll connect interested persons on a geographical basis. (If you've asked before, please ask again — we did not keep a list of interested persons.)

Is it okay for me to put a copy of Icon on our local BBS?

Yes. As we've said many times, our implementations of Icon are in the public domain and may be copied freely. We encourage others to distribute Icon in any convenient way.

We want to run Icon on our VAX cluster under VMS. Do we have to get a copy of Icon for each of our VAXes, or can we just copy it from one to another?

You can just copy it; there are no restrictions on making copies of Icon.

Can I copy and sell Icon diskettes?

Yes. Icon's public-domain status allows you to make copies for whatever purpose you want. There's nothing to prevent you from selling public-domain software, but its "market value" is obviously limited. You should make it clear that you are selling public-domain software to avoid misleading potential buyers.

I read that Icon is in the public domain. Please send me my free copy.

"Public domain" does not mean "free". There are expenses involved in preparing material for distribution: media, copying, printing documentation, packaging, and shipping.

Can I make copies of the Icon Newsletter for my friends?

Starting with No. 24, the *Icon Newsletter* is copyrighted. Our written permission is required to make copies. On the other hand, we'll be happy to send copies to your friends if you give us their names and addresses. That way, they'll get copes of future *Newsletter* as well.

I don't see why you can't distribute the Icon Newsletter electronically. It's prepared on a computer, isn't it?

Yes, it's prepared on a computer. Previous versions used Troff. The present one was prepared using Xerox Ventura Publisher, a desktop publishing system. That doesn't mean it's feasible to distribute it through electronic mail or on a diskette. The printing language is PostScript, so you'd need to have a PostScript printer, such as an Apple LaserWriter. The last *Newsletter* was about 250KB of PostScript — too large to make electronic transfer attractive.

Is there an implementation of Icon for the Apollo Workstation?

Not as far as we know. Several persons are working on one.

I'd prefer to get MS-DOS Icon on 3.5" diskettes instead of the 5.25" diskettes you distribute.

While there certainly will be more MS-DOS computers with 3.5" diskette drives in the future, a large majority of existing MS-DOS computers only have 5.25" drives. We provide Icon as a service to the computing community, not as a commercial, profit-making venture. It would be expensive and difficult for us to provide 3.5" diskettes that only a minority of persons need.

I want to get the source code for Icon for my PC, but I don't know whether to get the MS-DOS source distribution or the porting source distribution. What's the difference?

The basic source code for Icon is the same for all computers. It uses conditional compilation to select system-specific code. However, there also are several system-specific files that tailor the basic source code for a specific computer. The MS-DOS source distribution contains files specific to MS-DOS in addition to the basic source code. Get the MS-DOS source distribution if you are going to compile Icon on an MS-DOS computer. The porting-source distribution is intended for use on systems for which there is not yet an implementation of Icon.

I want to get Icon for my MS-DOS computer, but I don't know whether to get the LMM or SMM implementation.

There is no simple answer to your question. The SMM implementation is smaller and faster than the LMM one, but the amount of memory available for Icon data in the SMM is very limited. SMM Icon is fine for programs that process data transiently (such as reformatting a file, line by line), but it may run out of space for a program that keeps a lot of data in memory (such as tables of names and addresses). We recommend the LMM for persons who intend to use Icon seriously. You really should have at least 350KB of RAM available to Icon to run the LMM version satisfactorily, although it can be made to run with less.

I have a generic MS-DOS computer (not nearly IBM-compatible). Will Icon run on it?

MS-DOS Icon does not require IBM hardware compatibility. If your computer runs MS-DOS 2.0 or higher, it should run Icon. We know of no verified exceptions.

Why don't you provide makefiles in Microsoft C format with your MS-DOS source distribution?

We include a public-domain make program with our MS-DOS source distribution. This make program is more powerful than the one Microsoft provides, and it works for all C compilers (or any other make application). Thus, we need only one format for all makefiles. If you want to produce Icon makefiles in Microsoft format, we'll be glad to include them in future source-code distributions.

Does MS-DOS Icon work under Turbo C?

Not yet. We've gotten it to compile, making the changes necessary to support the ANSI C draft standard. However, it doesn't run. We haven't had time to look for the cause of the problem yet.

Icon BBS at Arizona

We have changed the software on our electronic bulletin board to the public domain RBBS program. This has resulted in several new capabilities and a variety of changes to the user interface.

The new BBS software offers a wider selection of file transfer protocols. You may select from Xmodem, Xmodem(CRC), windowed Kermit (which includes unwindowed Kermit), and simple ASCII transfers. Another feature of the new BBS system is that it provides electronic mail exchange between users, including some members of the Icon Project.

The bulletin board, which is used for both the Icon and SNOBOL4 Projects, is available from 5:00 p.m. to 8:00 a.m. Mountain Standard Time on weekdays and all day on weekends. The phone number is (602) 621-2283. We use a 2400-baud Hayes modem. The bulletin board prompts for your first and last names, and the city and state you are calling from. The system is set up to reject certain names that it recognizes as bogus, so it is important to enter your real name to these prompts. After getting on, running the bulletin board is simply a matter of walking through the system's menus.

For greatest flexibility in using the file downloading capabilities once you're logged on, we suggest you set your serial port for 8-bit characters, no parity, and 1 stop bit.

Ordering Icon Books Outside the United States

We've discovered that readers abroad have had some difficulty in ordering the Icon books. Here are addresses and other means of contact to use when placing an order. Please give them to your booksellers. It is our understanding that you may also place an order directly with either Prentice-Hall or Princeton University Press at these addresses.

PRENTICE-HALL

For *The Icon Programming Language*, Prentice-Hall, 1983:

U.S. Export Sales Office

Simon & Schuster International
International Customer Service Group
200 Old Tappan Road
Old Tappan, New Jersey 07675, U.S.A.
Tel: (201) 767-4990
Telex: 990348
Fax: (201) 767-5625

Carribean and South America

Same as U.S. Export Sales Office.

Mexico and Central America

Prentice-Hall Hispanoamerica, S.A.
Apartado 126 de Naucalpan
Estado de Mexico, MEXICO
Tel: (905) 358-8400
Telex: 3172379

United Kingdom, Europe, Africa, and Middle East

Prentice-Hall International (UK) Limited
66 Wood Lane End
Hemel Hempstead, Herts.
HP2 4RG ENGLAND
Tel: (442) 58531
Telex: 82445

Fax: (442) 212485

India

Prentice-Hall of India Private Ltd.
M-97 Connaught Circus
New Delhi 110001, INDIA
Tel: 352590
Telex: 31-61808
Cable: PRENHALL NEW DELHI

Japan

Prentice-Hall of Japan, Inc.
Jochi Kojimachi Bldg. 3F
1-25, Kojimachi 6-chome
Chiyoda-ku Tokyo 10, JAPAN
Tel: (03) 238-1050
Telex: 650-295-8590

Southeast Asia

Simon & Schuster (Asia) Pte. Ltd.
24 Pasir Panjang Road
#04-31 PSA Multi-Storey Complex
SINGAPORE 0511
Tel: 2789611
Telex: RS 37270
Fax: 2734400

Australia and New Zealand

Simon & Schuster (Australia) Pty. Ltd.
P.O. Box 151
7 Grosvenor Place
Brookvale, N.S.W. 2100, AUSTRALIA
Tel: (02) 939-1333
Telex: PHASYD AA 74010
Fax: (02) 938-6826

Simon & Schuster (Australia) Pty. Ltd.
4A/6 Riddell Parade
Esternwick, Vic. 3185, AUSTRALIA

PRINCETON UNIVERSITY PRESS

For *The Implementation of the Icon Programming Language*, Princeton University Press, 1986:

Canada

Book Center (wholesaler)
1140 Beaulac St.
Montreal, Quebec
H4R 1R8 CANADA

Cariad Ltd.
89 Isabella Street, Suite 1103
Toronto, Ontario
M4Y 1NB, CANADA

United Kingdom

Princeton University Press
15A Epsom Road
Guildford, Surrey
GU1 3JT ENGLAND
Tel. (483) 68364

Princeton Export Department
41 William St.
Princeton, NJ 08540 U.S.A.

Mexico, Central and South America, and the Carribean, including Puerto Rico

EDIREP
5500 Ridge Oak Dr.
Austin, TX 78731 U.S.A.

Australia and New Zealand

Cambridge University Press
10 Stamford Road
Oakleigh
Melbourne, Victoria 3166, AUSTRALIA

India

UBS Publishers' Distributors Pvt. Ltd.
Delhi, Bombay, and Bangalore INDIA

Oxford University Press
Bombay, Calcutta, Delhi, Madras INDIA

Japan

United Publishers Services Ltd.
Kenkyu-sha Bldg., 9
Kanda Surugadai 2-chome Chiyoda-ku
Tokyo, JAPAN

South Africa

Oxford University Press
P.O. Box 1141
Cape Town, 8000, SOUTH AFRICA

All Other Countries

Princeton Export Department
3175 Princeton Pike
Lawrenceville, NJ 08648, U.S.A.

If you have any difficulty getting books from these sources, please let us know.

A Brief History of Icon

Did you ever wonder where Icon came from? To really understand what motivated it and why it's the way it is, you'd first have to know the origins of SNOBOL4, another high-level programming language that emphasizes processing strings and structures. That's a long story. If you're interested, see "History of the SNOBOL Languages" in *History of Programming Languages*, Academic Press, New York, 1981; or "The Road to SNOBOL4" in Vol. 1, No. 3 (1987) of *A SNOBOL's Chance*, Catspaw, Inc., Salida, Colorado.

To summarize this early history very briefly, the first SNOBOL language originated at Bell Labs in 1962

because a few of us there needed a tool for manipulating symbolic material, such as equations. SNOBOL only had one data type, the string, and supported a limited set of pattern-matching operations. While this language was primitive in some respects, it was sophisticated in others. For example, it managed string storage automatically so that the user did not have to worry about how long strings were. At the time, a programming language for string processing was something of a novelty, and it attracted a lot of interest. This led to the development of subsequent languages, culminating in 1968 with SNOBOL4.

In the course from the original SNOBOL to SNOBOL4, we developed a philosophy of programming language design and implementation. We wanted ease of programming and high-level features, so we created useful linguistic facilities without much concern for how well they matched traditional computer architectures. In the spirit of making programming easier, we emphasized run-time flexibility, freedom from arbitrary constraints, and a general attitude that the burden should be on the implementation rather than on the user. We didn't pay much attention to the (then) conventional wisdom about what programming languages should be like. In many respects, the SNOBOL languages ran orthogonal to the mainstream of programming-language design — and deliberately. It's interesting to note that the design and implementation of SNOBOL4 and PL/I were contemporaneous.

SNOBOL4 is a rather remarkable language. In addition to strings, it has many other data types, including arrays, tables, and patterns. It has unusual run-time flexibility; functions and operators can be redefined at will during execution, and it is even possible to compile new code during execution, so that a program can modify itself.

One important aspect of the context for the SNOBOL languages was the absence of the constraints of design committees, corporate objectives, or commercial concern. The SNOBOL languages were really the product of a small group, working largely independently. Although the persons involved changed from time to time, there was a thread of continuity that started in 1962 and still continues.

In 1971, the project moved to The University of Arizona, where it took on more of the character of a research project with, however, the same philosophical objectives.

SNOBOL4, which is still in widespread use today, carries the marks of its early origins. Despite its powerful features, its syntax is awkward and antique, and it lacks many amenities that developed in other programming languages. More fundamentally, there are conceptual problems in its pattern-matching facilities, resulting in a language that is really a combination of two: one with conventional expression evaluation and another with searching and backtracking control structures.

We continued to try to find a better framework for the best features of SNOBOL4 in combination with the development of new linguistic facilities for handling a broader range of problems. This led to a new language, SL5 (SNOBOL Language 5) in 1975.

SL5 was notable for a sophisticated procedure mechanism that allowed the components of procedure invocation to be treated as separate language operations — the creation of an environment, the binding of arguments, and activation. Coroutines followed naturally from this mechanism, and patterns were cast in terms of environments for scanning procedures. SL5 also refined operations on structures and included a repertoire of basic string-processing operations that were lacking in SNOBOL4.

SL5 was a full-blown programming language and was completely implemented. It might have gone on to become a rival to SNOBOL4, except for the discovery of a unifying view of traditional and pattern-matching expression evaluation that led to the design of Icon.

Next time: — "The Early Days of Icon"

The Icon Project (continued)

Last time, we started to describe the Icon Project, which turned out to be difficult to do. That's probably because it's not a well-defined entity. If you've read that article, you've undoubtedly noticed that it's impersonal: the Icon Project is described in terms of "it" rather than "we".

That doesn't mean that there are no persons involved; in fact that's what the Icon Project is all about. However, since the Icon Project has no formal organization and no paid employees, we sometimes have a hard time deciding who is and who isn't part of it. Some persons devote a major part of their time to the project, but most contribute only a small amount of time. Some persons affiliated with the project are here at The University of Arizona, and some are scat-

tered over the world. Some affiliations are ongoing and some are transient.

In trying to describe the Icon project, it became clear that there was no structural dividing line and also that we risked leaving out persons who feel very much a part of the Icon Project. At the same time, we risked including some persons who might prefer anonymity. So we've decided rather arbitrarily to say a bit about only those persons who are at The University of Arizona.

The office staff of the Department of Computer Science handles most of the administrative aspects of Icon distribution. Beth Stair is the person you're most likely to get if you call the Icon Project. She handles most orders, answers nontechnical questions, takes care of accounts, maintains our address list, and so on. She's been called "the Icon lady", a term she doesn't like ("I'm not ready for that!"). Jana Zeff helps Beth, assembles orders, and prepares mailings. Fabiola Cardenas takes care of copying Icon documents.

The laboratory staff of the Department of Computer Science prepares Icon distribution material. John Luiten, the lab manager, sees that supplies are stocked, sets up procedures, and supervises other members of the lab staff. Bala Vasireddi copies tapes and diskettes. The laboratory staff also provides programming support for Icon itself. Bill Mitchell and Gregg Townsend, software specialists, help with debugging and contribute to the design and implementation of new features. Gregg handles the VMS version of Icon.

Four research associates, working on their PhDs, contribute to Icon language design and implementation in addition to their individual research (see *Icon Newsletter* No. 24). Janalee O'Bagy also handles most technical questions about Icon. She's the one you're most likely to hear from if you send electronic mail to the Icon Project. Ken Walker presently is doing most of the implementation on the next version of Icon. Kelvin Nilsen is sysop for our BBS and also provides support tools for Icon. Dave Gudeman handles source version control and provides code improvements.

Madge Griswold is co-editor of the *Icon Newsletter* and co-author of the books on Icon. She handles the production of documentation and provides administrative support to the Icon Project.

Ralph Griswold directs the Icon Project. He also handles most written correspondence and a good part of the electronic mail. From time to time he does just about everything from language design and implementation to sweeping the floor. He prefers implementation.

Language Corner

We're starting a new regular feature with this *Newsletter*— the Language Corner. It's devoted to discussion of features of the Icon programming language. The difference between the Language Corner and the Programming Corner is mainly that the former deals with aspects of the language while the latter deals with how to use it. Of course, the distinction is not always clear, and we're certainly not going to worry if there's some overlap.

We're starting the language corner with a subject that we've found to be troublesome to some Icon programmers: the concept of failure in expression evaluation.

Failure

The term *failure* often confuses persons when they start to learn Icon and the confusion sometimes persists in experienced Icon programmers. This confusion may interfere with their ability to use the full range of possibilities that Icon's expression evaluation offers.

The use of "failure" originated in SNOBOL4, where an expression, as in Icon, can produce a result or fail to produce a result. This makes it natural to say an expression such as i < j either succeeds or fails.

That's fine as far as it goes. Icon gets into trouble with this terminology because of generators — that is, because an expression may not only produce one result but many.

The mechanism whereby an expression can produce more than one result is described in terms of *suspension* and *resumption*. An expression that can produce a result, and may be able to produce another, suspends with that result. Subsequently, if the surrounding context needs another result, it resumes the expression, and so on. The trouble is that the expression may not be able to produce another result (most expressions can only generate a finite number of results). That is, to make the source of the confusion clear, it may fail to produce a result when it's resumed.

Thus, an expression that produces many results may then "fail". However, it also succeeded for every result that it produced, so it's silly to say that it failed. The confusion is between using the word "fail" for an expression that produces no results at all and using the

same word when it produces one or more results and then no more.

The problem, of course, is all in the use of language. For example, in

```
every i := 1 to 10 do f(i)
```

the expression 1 to 10 produces 10 results and, although it then fails to produce another one when resumed, you'd probably not think of it as failing. It would be foolish to say that 1 to 10 fails. On the other hand, in

```
if find(s1,s2) = i then write(i)
```

you might be inclined to say find(s1,s2) failed if it didn't produce a value equal to i, even if it produced 10 values not equal to i. That would, of course, be confusing.

The confusion is compounded by the use of the expression fail to indicate that no result is produced by calling (or resuming) a procedure. The standard format of a programmer-defined generator is:

```
procedure gen()
   . . .
   suspend expr1 | expr2 | ...
   fail
end
```

The fail at the end does not mean that gen() fails to produce a result; instead, it means that it eventually fails to produce a result after being resumed several times. Here, it's a bit clearer if you leave the fail out and let flowing off the end of the procedure body take care of it.

Part of the problem with all this is that the English language doesn't come equipped with all the words that are needed to succinctly describe the concepts that have been developed in programming languages. This is, of course, a consequence of the fact that programming languages contain new concepts. To describe these concepts, it's necessary to invent new terms, use cumbersome phrases, or try to use existing words in ways that are suggestive of the concepts. The use of "failure" to describe an aspect of expression evaluation just has not worked out well. Unfortunately, it is ingrained in the literature and the vocabulary of many programmers and it cannot simply be expunged in favor of better terminology (if we could think of any). On the other hand, it's impractically cumbersome to say "an expression does not produce any result when it is evaluated" or "an expression does not produce a result when it is resumed". Something like "an expression blonks" or "an expression bleeks" is not very appealing either. If you have a good solution to this problem, we'd like to know about it.

In the absence of a good solution to this problem of terminology, keep in mind the distinction between the two cases. Where the context does not make the difference clear, try to think something like "its evaluation fails" or "its resumption fails" — still cumbersome, but better than getting it wrong.

Incidentally, Icon's terminological heritage from SNOBOL4 has led to another misunderstanding: that expressions *signal* success or failure. The problem with this view is not with the idea of a signal — in the implementation, it's just that — it's with the accompanying assumption that the evaluation (resumption) of an expression produces two things, a result and a signal, and that if the signal is "failure", the result is discarded. This assumption probably comes about because most programmers are familiar with programming languages in which the evaluation of an expression always produces a result. They can't quite accept the fact that an expression may not produce a result and figure that the idea of not producing a result is some kind of trick to hide the fact that a result is discarded. It isn't a trick. When an Icon expression fails (watch it!), it's because it has no result to produce.

On the other hand, thinking of expressions as signaling doesn't lead to any essential contradictions. If you must think of it this way, all we can say is that you'd be better off if you could think of it as it really is. The unnecessary concept of a signal just gets in the way and increases the possibility of confusion. (The next thing you know, you'll be wanting to test this non-existent signal, store it as a value, or whatever.)

Programming Corner

Pattern Words

In the last *Newsletter*, we posed the problem of writing a procedure patwords(s) that returns the pattern word for s. (A pattern word is obtained by replacing all occurrences of the first letter of a word by A, the next different letter by B, and so on.)

Experienced Icon programmers naturally turn to map(s1,s2,s3) when there is a hint of character substitution or rearrangement in the air. The problem above is a natural for map — the difficulty is finding the unique characters on which to base a substitution.

Most solutions we received were based on removing duplicate characters in the word by "convention-

al" string processing, followed by a straightforward use of map. Here's one that's based on a submission by Gregg Townsend:

```
procedure patword(s)          # Solution 1
  static letters

  initial letters := string(&lcase)

  out := ""
  every c := !s do
    if not find(c,out) then out ||:= c
  return map(s, out, letters [1+:*out])
end
```

The static identifier is used to avoid cset-to-string conversion every time the function is called. This makes a noticeable difference, as does the use of find instead of upto — the latter requires a string-to-cset conversion in the loop.

Ardent Icon programmers try to find a way to do it entirely with map — both because of the challenge and because of the knowledge that map operates on all characters of its arguments in each call, avoiding loops over the characters at the source level.

Here is such a solution, based on a submission by Ken Walker:

```
procedure patword(s)          # Solution 2
  local numbering, orderS, orderset, patlbls
  static labels, revnum

  initial {
    labels := &lcase || &lcase
    revnum := reverse(&cset)
    }

  # 1: Map each character of s into another character, such that
  #   the new characters are in increasing order left to right (note
  #   that the map function chooses the rightmost character of its
  #   second argument so things must be reversed).
  # 2: Map each of these new characters into contiguous letters.

  (numbering := revnum [1 : *s + 1]) | stop("word too long")
  orderS := map(s, reverse(s), numbering)
  orderset := string(cset(orderS))
  (patlbls := labels [1 : *orderset + 1]) |
    stop("too many characters")
  return map(orderS, orderset, patlbls)
end
```

Yet another all-map solution is:

```
procedure patword(s)          # Solution 3 (anonymous)
  static backwards, letters

  initial {
    backwards := reverse(&ascii)
    letters := string(&lcase)
    }

  z := reverse(s)
    z := map(z,z,backwards [1:*z + 1])
    cz := cset(z)
    return reverse(map(z,cz,map(cz,cz,letters [1:*cz + 1])))
end
```

We'll leave you to figure this one out.

The concept of "good style" is more controversial in Icon than in many other programming languages. We personally prefer the first solution for clarity and the second for cleverness. Timing is more objective. Solution 2 is fastest. Here are comparative timings from processing 10,000 words from the word list from Webster's 2nd. The results are normalized to Solution 2:

Solution 1	1.15
Solution 2	1.00
Solution 3	1.51

It's worth noting that the timing is very sensitive to the method used. Solutions that use table-lookup instead of mapping typically are three to five times slower than Solution 2. Even putting the cset-to-string conversion in the body of the procedure in Solution 1 adds about 20% to its running time.

Environment Variables

The last *Newsletter* also asked for a procedure getenv(s) for UNIX that returns the value of the environment variable s if it's set but fails otherwise.

The solutions we got varied somewhat, depending on the flavor of UNIX involved. The approach we liked best, used by Mike Beede and Dave Hanson, is to read in all of the environment variables the first time getenv is called. Here's a combination of their submissions for use with Berkeley UNIX:

```
procedure main()
  while s := read() do
    write(getenv(s))
end

procedure getenv(s)
  local pipe, line
  static environment
  initial {
    environment := table()
    pipe := open("printenv","pr")
    while line := read(pipe) do
      line ? environment [tab(upto('='))] := (move(1),tab(0))
    close(pipe)
    }
  return \environment [s]
end
```

Benchmarking Icon Expressions

In the last *Newsletter*, we talked a little about efficient programming in Icon. There are several problems in writing efficient programs in Icon. One problem is that Icon often provides many different ways of accomplishing a task, and it's often difficult to tell which is the most efficient. Another problem is that many of Icon's features do not have any direct counterpart in the architecture of the computers on which Icon runs. You can guess how a feature like table-lookup is implemented, but you may guess wrong. Even if you know, you may be mistaken about its speed. In other cases, features may be implemented in ways that you'd not expect. Even if you are an expert on the implementation, you may be surprised by how relatively fast or slow some operations are. Even those of us who did the implementation frequently are wrong about speed.

In theory, if you knew enough about the implementation, you could come up with analytic results — formulas, bounds, and so forth. In practice, this approach has limited usefulness because of the complexity of the problems and the sensitivity of the operations to the kinds of data on which they operate. An alternative is an empirical approach: measurement of different kinds of expressions or small program segments. Such measurements can help answer questions, pinpoint potential problems, and suggest the most efficient approach to a particular problem.

Benchmarking expression evaluation isn't difficult. We have developed a few tools for translating expressions of interest into programs that time their evaluation in loops and report the results in a convenient format. These tools are described in IPD18, which is available for the asking.

To see how helpful benchmarking can be, consider the subscripting of a list: a[i]. This expression is simple enough, and in a more conventional programming language you probably would have a good idea of how it's implemented. But Icon supports stack and queue access as well as positional access, so you might guess that positional access is not as simple as it appears.

In fact, how fast positional access is depends on how the list is constructed. This is a case where benchmarking is useful. Consider two 1,000-element lists, a1 and a2 constructed as follows:

```
a1 := list(1000)
a2 := list [ ]
every 1 to 1000 do put(a2,&null)
```

Benchmarking shows the following comparative times for referencing the first and last elements of the two lists:

a1[1]	0.1084
a1[1000]	0.1084
a2[1]	0.1084
a2[1000]	0.2914

The time for referencing the first and last elements of a1 is the same, as might be expected. But why should it take longer to access the last element of a2? Both lists have the same size and the same values, but because of the way lists are implemented in Icon, they do not have the same structure. The first consists of a single block of 1,000 elements, while the second consists of a doubly-linked list of smaller blocks. The extra time to access the last element of a2 is a consequence of chaining through the blocks to get to the last one. (See the Icon implementation book for details.)

Questions: How long do you think a1[1001] and a2[1001] take? Suppose you build a list by adding elements in the fashion above but later need to access the elements by position? Is there anything you can do to eliminate the referencing overhead that results from the piecemeal construction of the list?

Next time we'll give some more results from benchmarking and extend it to measure how much storage various Icon expressions allocate.

Primes

Andrew Appel's "what does it do?" submission in the last *Newsletter* provoked this response from Bill Griswold:

Here is a sequence of modifications of the prime program in the last *Newsletter*. The first program is the original submission. The next three are modifications that attempt to restrict the range of iteration for checking if the current number is composite. Although the latter programs look more cumbersome (a *lot* of co-expression creation and invocation), they are actually competitive in speed for large numbers of primes. This is probably due to the fact that the sieve is more discriminate than the simpler programs. (You can also contrast these programs with the sieve in sample programs distributed with UNIX Icon systems, which isn't lazy. It is *much* faster than any of these).

```
procedure main()
  every write((i := 2) | (|i := i + 1) & (not(i = (2 to i) * (2 to i))) & i)
end
```

```
procedure main()
  every write(i := seq(2) & (not(i = (2 to i) * (2 to i))) & i)
end

procedure main()
  every write(i := seq(2) & (not(i = (l := (2 to i^0.5)) * (l to i))) & i)
end

procedure main()
  every write(i := seq(2) & (not (i = (k := 2 to i/2)*(i/k))) & i)
end
```

This program produces the primes lazily using the basic sieve technique and co-expressions. Nested creation of co-expressions eats space and causes (co-expression?) stack overflow. This is a modified version of a program by Robert Henry, which is a translation of a Miranda program.

```
procedure main(arglist)
  every write(sieve(create seq(2)))
end

procedure modcheck(s, p)
  repeat if ((x := @s) % p) ~= 0 then suspend x
end

procedure sieve(e)
  suspend (p := @e)
  every suspend sieve(create modcheck(e, p))
end
```

Here's a modified version of the sieve program. Elimination of tail recursion in the sieve lets the program run longer than the more straightforward version of this program.

```
procedure main()
  every write(sieve(create seq(2)))
end

# Check that every value in s is relatively prime to p
procedure modcheck(s, p)
  while x := @s do if (x % p) ~= 0 then suspend x
end

# Sieve of Eratothsenes, sans tail recursion
procedure sieve(e)
  repeat {
    suspend p := @e
    e := create modcheck(e, p)
    }
end
```

And here's the one from the distributed sample programs:

```
# This program illustrates the use of sets in implementing the
# classical sieve algorithm for computing prime numbers.

procedure main(arglist)
  local limit, s, i
  limit := arglist [1] | 100
  s := set( [])
  every insert(s,1 to limit)
  every member(s,i := 2 to limit) do
    every delete(s,i + i to limit by i)
  primes := sort(s)
  write("There are ",*primes," primes in the first ",limit," integers.")
  write("The primes are:")
  every write(right(!primes,*limit + 1))
end
```

Queens Never Die

The non-attacking n-queens problem continues to fascinate persons who are interested in program structure. Here's a recent contribution by Paul Abrahams, based on an earlier program by Steve Wampler:

```
global n, solution

procedure main(args)
  local i
  n := args [1] | 8                 # 8 queens by default
  if not(0 < integer(n)) then stop("usage  [ n ]")
  solution := list(n)               # ... and a list of column solutions
  write(n,"–Queens:")
  every show(q(1))                  # show the result of placing queens
                                    # in cols 1 – n in all possible ways
end

# q(c) – place queens in columns c through n in all possible ways.
# Suspend with a list of row positions for columns c through n
#
procedure q(c)
  local r
  static up, down, rows
  initial {
    up := list(2*n–1,0)
    down := list(2*n–1,0)
    rows := list(n,0)
    }

  every (r := 1 to n,if 0 = rows [r] = up [n+(r–c)] = down [r+c–1]
    then rows [r] <– up [n+(r–c)] <– down [r+c–1] <– 1) do
      suspend {
        if c = n then  [r] else  [r] ||| q(c + 1)
        }
end
```

```
# Show the solution on a chess board. The argument is a list of
# row positions for columns 1 through n.
#
procedure show(solution)
  static count, line, border
  initial {
    count := 0
    line := repl("|  ",n) || "|"
    border := repl("+--",n) || "+"
    }
  write("solution: ", count+:=1)
  write("  ", border)
  every line [4*(!solution - 1) + 3] <- "Q" do {
    write("  ", line)
    write("  ", border)
    }
  write()
end
```

Icon Electronic Clip-Art "Contest"

Now that we're using a desktop publishing system to produce the *Newsletter*, we can provide a more varied and visually interesting format. We've tried to resist the temptation to "tart-up" the *Newsletter* with a lot of gimmicks, but there are some things we can do that we've not attempted yet.

For example, we could include a logo for the Icon programming language — except that we don't have one.

So we're soliciting electronic clip art for a possible Icon logo, as well as other art related to Icon. We'll publish the best entries we receive in the next *Newsletter* and award a $50 credit at the "Icon Store", good for either program material or publications, to the entry we judge to be the best. We'll also award $15 credits to any other entries we decide to publish.

Entries are not limited to proposed logos — anything related to Icon is okay. We're looking for artistic merit and subject interest.

The usual contest rules apply: All entries must be original and free of copyright or other restrictions. All entries become the property of the Icon Project. The decision to publish entries and to award prizes is solely up to the editors of the *Icon Newsletter*, and all decisions are final.

All entries must be submitted in both printed and machine-readable form in one of the following formats: MacPaint, Macintosh PICT, PC Paintbrush (but *not* Paint), GEM Draw, GEM Paint, DFX (Drawing Interchange Format), or Encapsulated PostScript. Be sure to tell us what program you used and the format.

We can read MS-DOS format diskettes, 3.5" as well as 5.25" DD and HD, and also Macintosh diskettes.

The deadline for entries is January 15, 1988. We will consider later entries for inclusion in a subsequent issue of the *Newsletter*.

(We have a feeling we're going to regret this adventure, but it seems like a fun idea, so we're giving it a whirl.)

New Documents

Two new technical reports related to Icon are now available:

- IPD18, *Benchmarking Icon Expressions:* This report describes some simple tools that can be used to time the evaluation of individual Icon expressions. The tools are written in Icon and program listings are included.

- IPD41, *Tabulating Expression Activity in Icon:* This report describes a system for counting the number of times each expression in an Icon program is evaluated, produces a result, fails, and is resumed. The results are summarized in a program listing in which the counts appear below the corresponding expressions. The tabulation system uses an Icon variant translator and a few simple Icon programs. Examples and program listings are included.

Single copies of these reports are available, free of charge. To get copies, simply list the report numbers as given above on the order form at the end of this *Newsletter* and write "free" in the price column. There is no charge for shipping.

Upcoming in the Newsletter

The following topics are scheduled for inclusion in the next *Newsletter* in addition to the regular features:

- The second in the series of articles on the history of Icon.

- A discussion of what is involved in adding new functions to Icon (delayed from this issue).

- Contributions from readers (if we get any).

Ordering Icon Material

Shipping Information: The prices listed on the order form at the end of this *Newsletter* include handling and shipping in the United States, Canada, and Mexico. Shipment to other countries is made by air mail only, for which there are additional charges as follows: $5 per diskette package, $10 per tape or cartridge package, and $10 per documentation package. UPS and express delivery are available at cost upon request.

Payment: Payment should accompany orders and be made by check or money order. Credit card orders cannot be accepted. Remittance *must* be in U.S. dollars, payable to The University of Arizona. There is a $10 service charge for a check written on a bank without a branch in the United States. Organizations that are unable to pre-pay orders may send purchase orders, but there is a $5 charge for processing such orders.

What's Available

Icon program material falls into four categories: UNIX, VMS, personal computer, and porting.

The UNIX package contains source code, the Icon program library, documentation in printed and machine-readable form, test programs, and related software — everything there is. It can be configured for most UNIX systems. The documentation includes installation instructions, an overview of the language, and operating instructions. It does not include either of the Icon books. Program material is provided on magnetic tape or cartridge.

The VMS package contains everything the UNIX implementation contains except UNIX configuration information and UNIX-specific software. However, the UNIX and VMS systems are configured differently, and neither will run on the other system. The VMS package is distributed only on magnetic tape.

Icon for personal computers is distributed on diskettes. Because of the limited space that is available on diskettes, in most cases there are separate packages for the different components: executable files, source code, and the Icon program library. Each package contains printed documentation that is needed for installation and use.

Icon for porting is distributed on MS-DOS format diskettes. There are two versions, one with a flat file system and one with a hierarchical file system. Both versions are available in either plain ASCII format or compressed ARC format.

There are two documentation packages that contain more than is provided with the program packages: one for the language itself and one for the implementation. These documentation packages contain the language and implementation books, respectively, together with supplementary material.

When ordering, use the codes given in parentheses at the ends of the descriptions that follow.

Program Material

Note: The only program material that has been updated since the last *Newsletter* (June 13, 1987) is marked by the symbol ☛.

UNIX Icon: Tapes are $25; both *cpio* format (UT-C) and *tar* format (UT-T) are available. Specify 1600 or 6250 bpi. Cartridges are $40 (DC 300 XL/P, raw mode only); specify *cpio* (UC-C) or *tar* format (UC-T).

☛ **VMS Icon:** Tapes are $25; specify 1600 or 6250 bpi (VT).

Icon for Personal Computers:

Amiga Icon executables: one 2S/DD 3.5" diskette, $15 (AME).

Atari Icon executables: one 1S/DD 3.5" diskette, $15 (ATE).

☛ Atari Icon source: one 2S/DD 3.5" diskette, $20 (ATS).

Macintosh (MPW) Icon executables: one 1S/DD 3.5" diskette, $15 (ME).

Macintosh (MPW) Icon source: one 2S/DD 3.5" diskette, $15 (MS).

MS-DOS SMM Icon executables: one 2S/DD 5.25" diskette, $15 (DE-S).

MS-DOS LMM Icon executables: two 2S/DD 5.25" diskettes, $20 (DE-L).

MS-DOS Icon source and test programs: two 2S/DD 5.25" diskettes, $25 (DS).

MS-DOS Icon program library: one 2S/DD 5.25" diskette, $15 (DL).

UNIX PC Icon executables and program library: one 2S/DD 5.25" diskette, $20 (UPEL).

XENIX Icon SMM executables: one 2S/DD 5.25" diskette, $15 (XE-S).

XENIX Icon LMM executables: one 2S/DD 5.25" diskette, $15 (XE-L).

XENIX Icon source and test programs: five 2S/DD 5.25" diskettes, $40 (XS).

XENIX Icon program library: one 2S/DD 5.25" diskette, $15 (XL).

Icon for Porting:

Flat file system, ASCII format: four 2S/DD 5.25" diskettes, $35 (PF-A).

Flat file system, ARC format: two 2S/DD 5.25" diskettes, $25 (PF-K).

Hierarchical file system, ASCII format: four 2S/DD 5.25" diskettes, $35 (PH-A).

Hierarchical file system, ARC format: two 2S/DD 5.25" diskettes, $25 (PH-K).

Documentation

Language documentation package: $29 (LD).

Implementation documentation package: $40 (ID).

Back issues of the *Newsletter*: $.50 each for single issues (specify numbers), $6.00 for a complete set (#1-24) (NL). There is no charge for overseas shipment of single back issues, but there is a $5.00 charge for the complete set.

Order Form

Icon Project • Department of Computer Science • Gould-Simpson Building • The University of Arizona • Tucson, AZ 85721 USA

Ordering information: (602) 621-2018

name _____

address _____

_____

city _____ state _____ zipcode _____

(country) _____ telephone _____

☐ check if this is a new address

qty.	code	description	price	total

subtotal	
sales tax (Arizona residents*)	
extra shipping charges	
purchase-order processing	
other charges	
total	

Make checks payable to The University of Arizona

*The sales tax for residents of the city of Tucson is 7%. It is 5% for all other residents of Arizona.