THE UNIVERSITY OF ARIZONA
TUCSON, ARIZONA 85721

DEPARTMENT OF COMPUTER SCIENCE

Icon Newsletter #17

Madge T. Griswold and Ralph E. Griswold

March 1, 1985

1. Icon Distribution Policy

The Icon programming language is a product of a research program in the Department of Computer Science at The University of Arizona, and its development has been supported in part by the National Science Foundation. The results of this work are freely available to anyone who requests them.

At present the only charge is for the media needed to distribute the software. This means that Icon distribution is an expense to the Department. We prefer to handle it this way to facilitate the free distribution of research results within the computing community. However, the more interest there is in Icon, the more expensive this policy becomes. At some future time, it may be necessary to add charges for the printing of documentation, handling, and shipping. But, for now, the distribution of Icon material is free.

The implementations of Icon distributed by The University of Arizona are in the public domain. There are no warranties of any kind associated with this material — it is simply freely available to interested persons. The software may be used and copied without restriction or permission. We prefer, however, that persons interested in Icon obtain material directly from us. This allows us to keep track of users and inform them of updates and new developments.

2. Implementation News

Icon for the Sun Workstation

Bill Mitchell has completed an implementation of Icon for the Sun Workstation. This implementation is known to work on Sun's Release 1.2 and should work on releases back to 0.4, but these have not been tested. This implementation is included in our "generic" distribution of UNIX Icon and may be requested using the form provided at the end of this Newsletter.

This implementation may or may not work on other systems with 68000 processors — C compiler compatibility is essential to the proper functioning of this implementation. Bill Mitchell provides the following information:

(1) C *ints* must be 32 bits.

(2) The leftmost argument to a function must be at a6@(8).

(3) The first-declared local must be at a6@(-4).

(4) Locals must be allocated in a fifo manner; locals must appear in the frame whether or not they are used.

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

(5) The instruction at a routine's entry point must be link a6,#C, where C is a constant with any value.

(6) Registers a3-a5 must be preserved across calls.

(7) Routines must return using unlk a6; rts.

The following program attempts to detect the presence of the above characteristics. Compile it on the 68000 system in question with cc -S ....

```
f(i, p, j)
int i, j; char *p;
{
        register int *p1 = 0, *p2 = 0, *p3 = 0;
        int x, y, z;

        f1(i, j, p, x, z);
}
```

The resulting .s file should look something like this:

```
        .text
        .globl  _f
_f:
        link    a6, #0              | Create call frame
        addl    #-20, sp            | Make space for locals and registers
        moveml  #0x3800, sp@        | Save registers d6 and d7
        movl    #0, a5              | p1 = 0
        movl    #0, a4              | p2 = 0
        movl    #0, a3              | p3 = 0
        movl    a6@(-12), sp@-      | Push z
        movl    a6@(-4), sp@-       | Push x
        movl    a6@(12), sp@-       | Push p
        movl    a6@(16), sp@-       | Push j
        movl    a6@(8), sp@-        | Push i
        jbsr    _f1                 | Call f1
        lea     sp@(20), sp         | Pop arguments from stack
        moveml  a6@(-20), #0x3800   | Restore a3-a5
        unlk    a6                  | Remove call frame
        rts                         | Return to caller
```

Icon for the AT&T 3B20

O. Rick Fonorow has Version 5.9 of Icon running on the AT&T 3B20. The implementation is complete, except for co-expressions, which he plans to add later. Contact him for more information about future plans for this implementation:

Mr. O. Rick Fonorow
AT&T Bell Laboratories
1100 East Warrensville Road
IW 2X-361
Naperville, IL 60655

(312) 979-7173

... ihnp4!ihu1e!orf

A preliminary version of this implementation is available from The University of Arizona. See the form at the end of this Newsletter.

Version 5.9 of Icon for VAX/VMS

Bob Goldberg has completed the implementation of Version 5.9 of for VAX/VMS. This implementation resolves the problems with input/output buffers in Version 5.8 and includes recent language extensions.

Use the form at the end of this Newsletter to request this implementation.

3. Contributions from Users

The Programming Language (Pascal-Icon)

A programming language called π^* is being developed by Erick Gallesio under the direction of Olivier Lecarme of the Laboratoire d'Informatique at the University of Nice, France. Here is Gallesio's description from a recent letter:

> The programming language π (Pascal-Icon) is an attempt to embed goal-directed evaluation and generator mechanisms in Pascal. The choice of Pascal was principally motivated by the relative facility of implementing it. Our main aim is in fact to demonstrate that goal-directed evaluation and generators are not necessarily inefficient, and that they are fully compatible with Pascal-like languages (including Algol 68, Ada, ...). Therefore, this experience should serve as a framework for embedding these features into other imperative programming languages (or, perhaps, for defining new ones). Briefly stated, the syntax of π is an extension of Pascal syntax and the expression evaluation is as close as possible to Icon; a similar mechanism is also provided for allowing "goal-directed execution" of some statements. As in Icon, and unlike Cg, there is no special syntactic construct for initiating backtracking (i.e., backtracking is fully automatic).
>
> An implementation for this language is in progress on a VAX 780 running under UNIX. This implementation is intended to be portable, and is based on the P4 implementation of Pascal (with some major changes to the intermediate language). In a first step, only an interpreted version is anticipated. All the components of the π system are, or will be, written in Pascal.

Generators in Smalltalk

Tim Budd, at the University of Arizona, described his Smalltalk implementation in Newsletter #15. Here is more from him on that subject:

> Object oriented languages, such as Smalltalk, provide a rather easy and natural means of implementing generators. In Smalltalk, everything is an *object*. An object consists of a small amount of local memory (accessible only to the object) and a list of messages to which the object will respond. Associated with each message are a set of actions that will be initiated upon receipt of the given message. Because each object maintains its own memory, there is no need to manipulate the process stack in an extraordinary fashion, as is required in Icon or Cg.
>
> We can define a *generator* to be any object that responds to the messages first and next. Associated with each generator is a sequence of values, although how the sequence is constructed or what the values represent need not be made precise. In response to first, the object returns the first element in its sequence. In response to next it returns each subsequent item in turn, or the special value nil to indicate no new element is available. There is no need to define how each element is to be produced; they can simply be an enumeration of values kept in memory, as, for example, from an array, or they may be constructed as needed, as with a list of random numbers.
>
> Assuming no more than this simple protocol, it is possible to define techniques for combining and manipulating arbitrary generators. An experimental Smalltalk system, called "Little Smalltalk", as been constructed for experimenting with these and other related issues. The generator paradigm just described has been used throughout Little Smalltalk in the representation and manipulation of collections of objects. Work on defining pattern matching and other goal-directed techniques using generators in Smalltalk is being carried out.
>
> The Little Smalltalk system has recently been made available for distribution. Currently it works on the VAX 780 with Berkeley 4.2 bsd UNIX, PDP 11/70 and 11/44 with Version 7 UNIX, the Ridge with ROS 3.0, and the

---

* Editor's footnote: This π is different from the π (Production Icon) described by Steve Wampler in Newsletter #15.

HP 9000 with HP-UX. For more information on Little Smalltalk, contact:

Timothy A. Budd
Little Smalltalk Distribution
The Department of Computer Science
The University of Arizona
Tucson, Arizona 85721

Logicon: An Integration of Prolog into Icon

Guy Lapalme and Suzanne Chapleau, at the University of Montreal, discuss their incorporation of Prolog into Icon in the abstract that follows. The complete paper appears at the end of this Newsletter.

> We describe the coupling of logic programming with Icon. We show that Icon and Prolog have many similarities and that their integration is feasible and desirable, since the weaknesses of one can be compensated by the strengths of the other. In our case, a Prolog interpreter was written as an Icon procedure that can be linked and called by an Icon program. This interpreter deals with all Icon data types and can be called in the context of the goal-directed evaluation of Icon. We give an example showing the power of this symbiosis between these two languages where a Prolog call in Icon is a generator and an Icon call in a Prolog clause is an evaluable predicate.

An Application for Linguistic Analysis

Persons often ask what Icon is used for. One application that Icon shares with SNOBOL4 is linguistic analysis. An example is Glenn David Blank's research described in his doctoral thesis *Lexicalized Metaphors: A Cognitive Model in the Framework of Register Vector Semantics*, completed at the University of Wisconsin-Madison in 1984 under the direction of Peter Schreiber. The abstract reads as follows:

> Lexicalized (alias 'dead' or 'frozen' or 'familiar') metaphors are those that receive predictable interpretations, wholly from the context of predication. The likes of *precious seconds*, *angry letter*, *happy home*, *reviving a romance*, *grasping a theory* are virtually as familiar and sensible as 'literal' expressions, yet on closer examination seem to involve violations of predicative constraints. Their predictability may be accounted for by either of two strategies: 1) full lexicalization, where the figurative sense is entered as part of literal meaning; and 2) sense extension, where it is derived from literal meaning by a semantic rule sensitive to local predication. An experiment, designed to detect lexical access times for systematic metaphors vs. literal and anomalous controls, shows that two processing strategies do seem to be involved. Sense extension implies a stage model, but the second stage may be obligatory upon failure of the first, and hence an *automatic* process.
>
> A computational model of lexicalized metaphors is described and demonstrated in the framework of Register Vector semantics. This is a componential approach that insists upon the standardization of feature vectors and lexical structure. Standardization eliminates the complexity of symbol-manipulation, and paves the way for a compact, finite semantics with fast, predictable operations. It also allows for the formalization of data structures that apply to the lexicon as a whole — directories for efficient semantic research, as well as the sense extension rules. Thus natural language semantics, including lexicalized metaphors, may be processed by a computational model in real, linear time.

4. Use of Icon in Computer Science Courses

In Newsletter #15, we reported on the use of Icon in Computer Science classes at CMU. Here is a report from Thomas Christopher at Illinois Institute of Technology:

> Last semester I provided a number of compiler writing tools in and for Icon for my compiler writing class. More students got more of their compilers done than ever before. Now I've reorganized the class around the tools. Students should be generating code before the middle of the semester for a minimal subset of Pascal, and then add more constructs every week to the end of the semester. We have started doing all our compilers this way — prototype in Icon, get a subset going first, and augment the already working compiler.

Students in the undergraduate programming languages course got a large dose of Icon last semester. Several came to ask me if they could use it in the compiler writing course this semester, so they seem to have become fans.

5. Programming Corner

Old Business

A number of readers objected that the following two expressions from the last Newsletter are not equivalent:

```
if not (1 <= i <= *s) then ...

if *s < i < 1 then ...
```

Actually, no claim of equivalence was made. As an exercise, determine in what situations the two expressions *are* equivalent.

New Business

Since this Newsletter is full of other material, the Programming Corner is abbreviated. Here is one new problem to work on, however.

Anagramming:

There is an easy way in Icon, given a string **s**, to produce another string that contains the characters of **s** in alphabetical order with duplicate characters removed:

```
string(cset(s))
```

Problem: Write a procedure **anagram(s)** that produces a string consisting of the characters in **s** in alphabetical order, but without the deletion of duplicates. For example,

```
anagram("hello")
```

should produce ehllo.

Experiment with different techniques to try to find the fastest method.

Logicon: an Integration of Prolog into Icon

Guy Lapalme and Suzanne Chapleau

Département d'informatique et de recherche opérationelle
Université de Montréal
C.P. 6128, Succ. A
Montréal, Québec, Canada H3C 3J7

1. Integration of Prolog into Icon

The proof of a Prolog request and the execution of an Icon expression have the following similarities:

- Prolog solves one request at a time and Icon evaluates one expression at a time.
- A Prolog request can refer to other requests, each of which is capable of giving more than one result. An Icon expression can be decomposed into many sub-expressions, each of which can possibly generate more than one result.
- A Prolog system can give all solutions of a request by doing a depth-first search with backtrack on the different alternatives. Icon uses a similar strategy to generate the sequence of results.
- A Prolog request and an Icon expression terminate by failing when their sequence of results is exhausted.

The integration of Prolog and Icon is done on these grounds of similarity, but the "personality" of each one is kept. A Prolog request in Icon acts as a generator, and Icon code in a Prolog request is used as an evaluable predicate that has the same attributes as another Prolog goal.

One major difference between these languages is the way results are generated. In Prolog, results are given back as the final values of the variables (which is why Prolog interactive systems often show the final values of the variables after a request has succeeded). On the other hand, in Icon the result is the final value of the expression. Assignments made to variables have no direct consequences on the final result of an expression.

We have implemented a Prolog interpreter as an Icon generator that deals with the partially defined variables that can be completed during a proof. However, these effects are undone automatically by backtracking. The variables get values by unification with a data base of facts that has to be maintained. Chapleau [1] shows how this can be implemented. We only present here how to insert Prolog code into an Icon program as a generator (we call it the external interface) and how to put Icon code in Prolog to behave as a goal (internal interface). All this is done without any preprocessing whatsoever. This is "standard" Icon code calling precompiled procedures and using predefined Icon record types.

1.1 Prolog Term Representation

For constant terms, the following Icon types can be used: real, integer, string, cset, table, files, procedure, and co-expression. Lists will be dealt with later. The last five types are not usually found in Prolog, so we might ask what are the consequences. In pure Prolog, the only operation is the unification that we chose to implement as the Icon equality test (already defined for all Icon types). The types of the values only matter in the evaluable predicates that are defined outside of Prolog anyway, so this has no consequence for the Prolog interpreter.

One of the main originalities of Prolog is the logical variable what can only be defined once but which can have still undefined parts that can be filled in later. To represent them, the following Icon record has been predefined:

```
record logical(value)
```

A free variable has &null as the value of its component. Thus, the Prolog interpreter procedure can differentiate between variables and constants by testing the type of the value of the variable.

The complex terms of Prolog are composed of a functor with a fixed number of components. These correspond exactly to the record structure of Icon. The only implication for the Prolog programmer is that he has to declare all functors he will use in his program.

The only Icon type remaining is the list. It can be defined easily using a binary functor that links the head and the tail of the list. This notation is quite cumbersome, and the implementation of this very useful construct can be much more efficient with a linear structure instead of a tree one. Moreover, Icon has a very extensive set of predefined operators and functions operating on Icon lists. As a result, we chose to represent the Prolog lists as Icon lists. This is very useful and convenient for the programmer (the Icon list notation is almost the same as the DEC10-style Prolog lists). This has the unfortunate consequence that the unification algorithm will be more complicated, but this is hidden from the programmer.

1.2 External Interface

To enter facts or relations in the Prolog data base, we define the following Icon procedure:

```
assert(head, goals)
```

and, to execute a request, the following Icon procedure is used.

```
prolog(request)
```

Entering Relations

For each relation to be entered in the Prolog data base, the user must

- Declare a record type for the relation.
- Declare a logical variable used in the relation.
- Call the assert procedure.

For example, to enter the following classical Prolog append procedure,

```
append([], L, L).
append([Car|Cdr], C, [Car|L ]) :- append (Cdr, C, L).
```

the user declares the following:

```
record append(f1, f2, f3)
```

and executes the following assignments:

```
l := logical()
car := logical()
cdr := logical()
c := logical()
```

These global variables will be used for getting the results of the Prolog expression in Icon. Finally,

```
assert(append ([], l, l))
assert(append ([car, cdr], c, [car,l]), append (cdr, c, l))
```

will save the append definition in the Prolog database of facts. The Icon procedure assert checks the syntax of the clause and keeps an internal structure binding the same variable occurrences within a clause but keeping the ones between two clauses separate (i.e., in our example, variable l is used in both clauses but internally is not the same one). The final results of a Prolog goal are given back in the global variables named in the clause but with all substitutions made so that a programmer does not have access to the internal structures of the interpreter. As the "prolog" procedure behaves like an ordinary Icon generator, all control mechanisms can be applied: for example,

```
every prolog(append (a, b, [do, re, mi[ ]]) do { ... }
```

will execute the statements in braces for each possible binding of a and b such that their concatenation gives

the list [do, re, mi, [ ]]. (The list has to terminate by [ ].) We can even use the "prolog" generator in a co-expression.

```
Conc := create prolog (append (a, b [do, re, mi[ ]]))
```

and call it with @Conc. In this way, we can even have more than one Prolog execution going on in parallel!

Internal Interface

This part deals with the integration of Icon code within a Prolog clause so that the Icon code behaves as an evaluable predicate. Ideally, we would like to be able to only put Icon code in place of another Prolog goal. For example,

```
assert(append (x, y, z), z.value := x.value ||| y.value)
```

supposing that x and y were already instantiated to lists. Unfortunately, in this case the expression will be evaluated before giving its result to assert. If we use

```
assert(append(x, y ,z), create z.value := x.value ||| y.value)
```

we will have captured the expression, but unfortunately the bindings between the logical variables and the ones of the co-expression will not be available to the **assert** procedure. The bindings will have to be given by the programmer using an intermediary call like the following:

```
assert(append (x, y, z), icon(create z.value := x.value ||| y.value, [x, y, z]))
```

Any expression can appear in such a request, even a generator.

2. Uses of Logicon

This scheme has been implemented using the VAX/VMS Icon interpreter. Chapleau [1] shows how this scheme can simplify the expression of the now-classical, Master-Mind Prolog Program. The main interaction loop is done in Icon, but the heart of the algorithm is done with one prolog Clause, which can be used to generate tries, test tries, and tabulate the scores.

For now, the only drawback is the efficiency of the scheme, but we think that by modifying the interpreter itself, we could speed things up considerably.

Bibliography

[1] Chapleau, S. (1984). "Integration du language Prolog au language Icon, Mémoire de Maîtrise, Document de travail no 156, Département d'informatique et de recherche opérationelle, Université de Montréal.

[2] Lapalme, G. Chapleau, S. (1984). "Logicon: an integration of Prolog into Icon", Publication no 516, Département d'informatique et de recherche opérationelle, Université de Montréal.

Version 5.9 UNIX Icon Distribution Request

Note: This system can be configured for PDP-11s with separate I and D spaces, VAX-11s, and Sun Workstations.

Contact Information:

name _____

address _____

_____

_____

_____

telephone _____

electronic mail address _____

computer _____

operating system _____

All tapes are written in 9-track *tar* format. Specify the preferred tape recording density:

□ 1600 bpi □ 800 bpi

Return this form to:

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

Enclose a magnetic tape (at least 600') *or* a check for $20 payable to The University of Arizona.

Request for Version 5.9 of Icon for the AT&T 3B20

Contact Information:

name _____

address _____

_____

_____

_____

telephone _____

electronic mail address _____

computer _____

operating system _____

All tapes are written in 9-track *cpio* format. Specify preferred tape recording density:

□ 1600 bpi □ 800 bpi

Return this form to:

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

Enclose a magnetic tape (at least 600') *or* a check for $20 payable to The University of Arizona.

Request for Version 5.9 of Icon for VAX/VMS

Contact Information:

name _____

address _____

_____

_____

_____

telephone _____

electronic mail address _____

computer _____

operating system _____

All tapes are written in 9-track *tar* format. Specify preferred tape recording density:

☐ 1600 bpi ☐ 800 bpi

Return this form to:

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

Enclose a magnetic tape (at least 600') *or* a check for $20 payable to The University of Arizona.