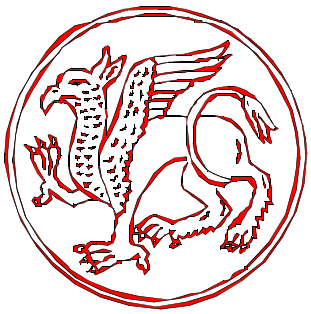
The Generator

In this issue:



"I know what I have witnessed. Now it is your turn. Prepare yourself for a journey into a world where each new step may give you a better understanding of your own reality."

VOL 2. NO 1.

FEBRUARY MMVI.

Statement of Purpose

The Generator is an international, non-for-profit journal devoted to the use of the Unicon programming language and its predecessor and subset, the Icon programming language. The Generator can be freely redistributed in its complete and unchanged form.

Printing Instructions

The Generator is designed to be printed on the both sides of the paper. It is published in Letter format; we hope our readers can manage to print it on A4 paper when needed, and welcome suggestions. In some situations, the printing options *Auto-rotate and Center* and *Fit to paper* may be appropriate.

Call for Papers

The Generator publishes a wide range of articles: papers, reviews, notes, reports etc. All articles contain some amount of the previously unpublished material; an exception is material previously published on less formal ways (preprints, mailing list and newsgroups posts etc.)
No particular article style is preferred.
Code should follow Icon and Unicon Projects' standard formatting conventions.
All submitted articles are reviewed.
The author permits unlimited publishing of the submitted article in The Generator, including any printed/bound issues, reprints, and collections.
Copyright of the articles is not transferred to The Generator.

Editorial Board

Typesetting

Typesetting is done by authors, reviewers and the publisher of The Generator.
The illustrations: Monsters of Stone, Omega Font Labs, <http://www.moorstation.org/typoasis/designers/omega/omega.htm> and DBL Corners, House of Lime, <http://www.houseoflime.com>.
The fonts: Weatherly Systems Inc. **Ramona,** Bitstream De Vinne and `Pica10` and Corel **Frankenstein.**

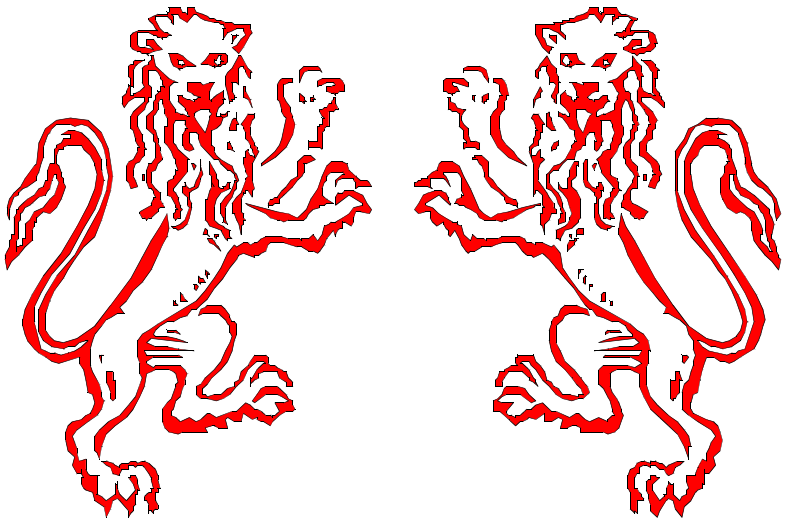
Publisher

Clinton Jeffery, Las Cruces, New Mexico USA.

Welcome to the 2nd Issue of The Generator

This is the second issue of **The Generator,** a journal devoted to the use of the Unicon programming language and its predecessor and subset, the Icon programming language. **The Generator** was founded by Kazimir Majorinc with a substantial explicit homage to a predecessor publication, **The Icon Analyst.** This issue has a new editor, and while we will try to maintain and improve the quality of the beautiful publication Kazimir established, we are sure it will suffer the loss of his time and expertise. Kazimir has provided the graphic designs and fonts that made **The Generator Vol. 1 No. 1** so special, and contributes an article to this issue.

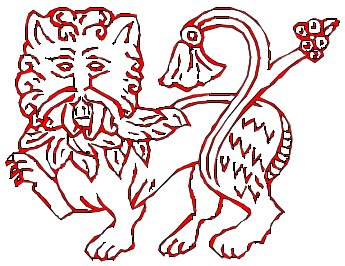
This issue has been a long time in coming, nevertheless it is numbered as Vol. 2 No. 1. The volume will reflect the year of issue, only counting years in which an issue is published. The number of issues per year will vary as needed based on the articles received and accepted for publication. The formatting of this issue has been switched to OpenOffice. The new editor, being short on time and lacking artistic skill, will likely not aspire to or attain the level of graphic beauty found in the previous issue. But, as the flagship periodical of the Unicon project, issues of the Generator may contain first-hand news and technical information that is of value to the readership, and is a primary means of presenting your Icon and Unicon-related work to our community.



State of the Unicon Project

Unicon development is proceeding at a healthy clip. In 2005, students performed important projects in the areas of: a SNOBOL-style pattern data type (Sudarshan Gaikaiwari), a simplified C-calling interface (Udaykumar Batchu), a source level debugger called udb (Eric Munson), performance improvements to the 3D graphics facilities (Omar El Khatib), a Voice-over-IP interface (Ziad Al Sharif), and a Windows PocketPC port (David Price). These projects' level of completeness and stability varies. Most of them need further testing, a code review or a follow-on project by another student or an Internet volunteer. Our ability to "deliver the goods" as working additions to the Unicon distributions is limited by students' level of experience and the full-time teaching and other obligations of the Project Lead. For some of you the most exciting project of all is the one that is still "cooking": a major update of Unicon and iconc under construction by Michael Wilder that enables compilation of Unicon programs using iconc. At present the iconc update is substantially functional but iconc's compilation model limits its scalability. Iconc does not support separate compilation, and Unicon's class libraries are large enough that iconc's type inferencer brings even 64-bit workstations with many gigabytes of main memory to their knees when compiling, say, a simple IVIB-generated program. Future work will cull unused classes and procedures prior to invoking the type inferencer in order to make this process scale reasonably; this will benefit Icon programs, not just Unicon programs.

At present Unicon development is supported directly by the U.S. National Library of Medicine and indirectly by the U.S. National Science Foundation. Currently this is paying for all the student labor we can recruit, but we still need contributions from the user community, especially in areas such as updating and maintaining ports on operating systems not used natively within the Unicon Project. New Unicon-related collaborations, contracts, projects and proposals are also very welcome.



Memoization

Kazimir Majorinc

Humans are known for their extensive use of different kinds of memories to improve the efficiency of their computations. For example, few people ever compute the square root of the number 2; even those that do, only do so once or twice. Instead, people remember its approximate value and use it whenever needed. On the other hand, computers are typically used in a heavily repetitive, redundant fashion: the same square root is computed daily by many computers, sometimes possibly with significant cost of the processor time.

Donald Michie[1] from Edinburgh was perhaps the first who described[2] attempts for reduction of such redundancies in the late sixties, using the metaphor of a *memo,* a short note written as a reminder. **D. Michie** proposed that most recently used results of some computation are stored on the top of the stack. When similar computation is needed again, program can search through stack for a previously computed result before it attempts to actually compute it. Later, hash tables with their logarithmic access time were recognized as the most useful data structure for memoization of large amounts of the data. The main idea and implementation of memoization is simple enough that it is probably widely used ad hoc, without reflection. However, support for memoization has been relatively recently introduced in programming languages. For example, **Marty Hall** developed libraries for memoization in Common Lisp[3,4 and 5] in the early **1990's** and together with **Paul McNamee** for C++[6] in late **1990's**. **Mark Jason Dominus**[7] from Philadelphia implemented a memoization module for Perl in the late **1990's** and early **2000s**. Recently, memoization seems to be discussed in contexts of many other programming languages, including Java and Python. Particularly radical step is undertaken by **Jeff Kingston** from Sidney. In

---

1 http://www.aiai.ed.ac.uk/~dm/dm.html

2 **Donald Michie**, *Memo functions and machine learning* Nature, vol. 218, April 6, **1968**. **pp**. **19-22**.

3 http://www-cgi.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/lisp/code/ext/memoize/announce.txt

4 **Marty Hall**, **J**. **Paul McNamee**, *Improving the Performance of AI Software: Payoffs and Pitfalls in Using Automatic Memoization,* Proceedings of Sixth International Symposium on Artificial Intelligence, Monterrey, Mexico, September **1993**., also available on http://www.gia.ist.utl.pt/cadeiras/tp/aulas/Monterrey-Memoization.pdf

5 **James Mayfeld**, **Tim Finin**, **Marty Hall**, *Using Automatic Memoization as a Software Enginering Tool in Real-World AI Systems,* Proceedings of 11th Conference on Artificial Intelligence for Applications, February 20 - 22, **1995**, Los Angeles, also available on http://www.cs.umbc.edu/~mayfield/pubs/caia95-memoization.ps

6 **Marty Hall**, **J**. **Paul McNamee**, *Developing a Tool for Memoizing Functions in C++,* ACM SIGPLAN Notices, August **1998**, pp. 17-22. Also available on http://apl.jhu.edu/~paulmac/publications/c++-toolbox-memoization.ps .

7 **Mark Jason Dominus** maintains web site at http://perl.plover.com/ .His article *Bricolage: Memoization,* The Perl Journal, Issue #13 Vol. 4 No. 1 (Spring **1999**) is available also on his site at http://perl.plover.com/Memoize/ . The article is reprinted in the book **Jon Orwant** (ed.), Computer Science and Perl Programming, O'Reilley, 2003, available at http://www.oreilly.com/catalog/tpj1/chapter/ch20.pdf .

his programming language Nonpareil[1] all functions are memoized by default. If memoization of some function is not wanted, it has to be explicitly turned off.

The most frequently used example for memoization is the function that computes Fibonacci numbers[2]. That function appears to be the most naturally implemented as a recursive procedure.

```
procedure r_fib(n)
  return if n<=2 then 1 else r_fib(n-1)+r_fib(n-2)
end
```

Unfortunately, simple test as **every write**(*r_fib*(1 **to** 40)) suffices to demonstrate that this implementation is as inefficient as it is simple and elegant. Although recursive procedures are generally slow, it is not the main reason for inefficiency of *r_fib*: another standard example for recursive procedure, factorials, works much faster. Instead, the problem is in not so obvious redundant computations. For some n, *r_fib* (n) calls *r_fib* (n-1) and *r_fib* (n-2), where *r_fib* (n-1) again calls *r_fib* (n-2) and *r_fib* (n-3). Two calls of *r_fib* (n-2) are computed independently, hence, computation of *r_fib* (n) requires roughly two to three times more time than computation of *r_fib* (n-2). Hence, time required for a computation of *r_fib* exponentially depends on its argument. More exactly, but not of importance on this place, it can be demonstrated by induction that running time of the procedure call *r_fib* (n) linearly depends of the value of *r_fib* (n) itself.

Non-recursive implementation of same function can eliminate some redundancy.

```
procedure nr_fib(n)
  a:=b:=1
  every 3 to n do  ( a+:=b ):=:b
  return b
end
```

However, even here, some increase of the complexity can be observed. Note that this implementation requires time that linearly depends on n, i.e. it still consumes relatively a lot of processing time. Hence, there are enough motives to investigate whether memoization can be an acceptable alternative.

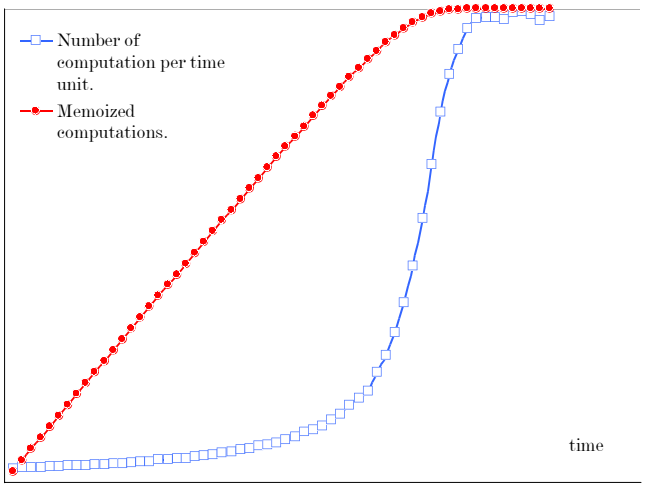
It is frequently hard to predict whether computation of some function can benefit from memoization. It depends on many factors, including, but not limited to the efficiency of the implementation of a memoization, time required for the computation of the function without memoization, probability that function is already called with same arguments, size and type of the *memoized* data and expected running time of the whole program. For some functions, like addition, the computation can be faster than search for a previous result or an excessive amount of memory might be needed for effective memoization. On the other hand, for very slow functions, there is no upper limit on the possible speed-up that can be achieved.

---

[1] http://www.it.usyd.edu.au/~jeff/nonpareil/

[2] The term function in Icon literature is sometimes used as synonymous for the built in procedure. This text cannot be consistent with that tradition, since mathematical notion of the function is needed. It is assumed that procedure or operator is a function if its result depends only on submitted arguments and it does not produce side effects.

Memoized functions are initially, when most of the *function calls* (in this context: function and submitted arguments) were not computed already, slower than non-memoized versions: time is spent on unsuccessful search, computation of the results like in the non-memoized version and storing of the results in some data structure for further use. However, as the probability that the result of the procedure call for some argument is already computed increases, the memoized version becomes faster. The following graph roughly depicts the relation between speed of the memoized function and running time of the program for a very simple memoization task, when both time required for computation of the function without memoization and time required for access to previously stored results are constant and arguments of the function are randomly chosen.



Many successful practical applications of memoization have been reported. For example, **D. Michie** wrote that his colleague from Edinburgh, **Robin Popplestone**[1] increased the speed of his programs by a factor of 10-20. Our program Finder[2] is 2-3 times faster if some of the performed computations are memoized. **Marty Hall**[3] and **J. Paul McNamee**[4] reported[5] speed up of a few Lisp programs between 15 and 600 times. Perhaps the most impressive speed up is reported by Belgian physicist **Peter Van Eynde**[6] in his Ask Uncle Peter[7] column: he reduced the running time of the quantum mechanics Lisp program from a few millions of years to five minutes.

---

1 http://www-robotics.cs.umass.edu/~pop/

2 http: //chem.pmf.hr/~kazimir/Finder.html

3 http: //apl.jhu.edu/~hall/

4 http: //apl.jhu.edu/~paulmac/

5 **Marty Hall**, **J.** and **Paul McNamee** *Improving Software Performance with Automatic Memoization*, Johns Hopkins Applied Physics Laboratory Technical Digest, Volume 18, Number 2 (**1997**),

6 http://people.debian.org/~pvaneynd/

7 http://www.cliki.net/Ask Uncle Peter

Ad hoc implementation and use of memoization is surprisingly simple, especially in sufficiently high level programming languages like Unicon or Icon, with good support for the hash tables. Arguments of the memoized function calls can be used as keys, and results as values.

```
procedure im_fib (n)
  static T
  initial { T:=table(); T[1]:=(T[2]:=1) }
  return \T[n] | (T[n]:= im_fib (n-1)+ im_fib (n-2) )
end
```

Achieved speed-up is significant, even impressive. The reader can compare running times of the *r_fib*(35) and *im_fib*(35). For simplicity, let us suppose that the time required for access to $T[i]$ does not increase significantly when the size of T increases. As already noted, the running time of *im_fib* (*n*) is linearly related with *n*, just like *nr_fib*(*n*) is. However, after the first execution of *im_fib* (*n*), results of *im_fib* (1), ..., *im_fib* (*n*) are memoized and in further calls, running time of *im_fib*(*m*) is constant for $m \leq n$ and it linearly depends on m-n, if $m > n$. Hence, it can be expected that *im_fib* has in practice significantly better running time than *nr_fib*(*n*) and especially *r_fib*(*n*).

Adding memoization to a procedure seems to be largely routine work. Hence, it can be isolated and centralized in the program. One table might be enough to store information on all function calls in the programs as keys and results of the respective calls as values. The keys of the table should be uniquely determined by function call: for example, concatenation of the name of the function and values of the arguments can be used. If arguments are not simple values, i.e. strings or numbers, they can be encoded into strings; using, for example, **Robert J. Alexander's** procedures *ximage* or *xencode* or **Ralph Griswold's** procedure *encode* from Icon Program Library. Such encoding is slow, so it can be optional.

Procedures and methods in Unicon and Icon are more general than functions in a majority of other programming languages: some of them, for some combinations of arguments, generate many, even potentially infinite number of results. For a particular function call, generated results can be stored in the list that grows as needed. Furthermore, a function can fail to generate a result. Unfortunately, failure itself is not a first class value, hence some encoding is necessary to memoize that information. For example, [1,x] and [0,&null] can code the information that function call returned x or failed, respectively. For more efficient use of memory, records can be used instead of list.

There is another problem with memoization which can be, perhaps, best described by example. Let us suppose that the first time a function is called with a given combination of arguments, it generates 100 results and all of them are memoized. These results can be used whenever 100 or less results are requested from the same function called with the same arguments. However, if some expression requests a 101st result, the memoized function needs to be called again to generate the first 100 results and only after that the 101st result can be generated, memoized for a further use and returned to the caller. This behavior spends all time that is potentially spared by memoization of the first 100 results, plus it costs some time on its own. Fortunately, it can be solved using coexpressions. For each function call a special coexpression is created, used to generate the results, and stored in the table together with all results it generated. If additional results are required, the coexpression is re-activated to produce new results. If the coexpression fails, the list of the results is complete and coexpression can be safely deleted.

All collected data on function calls can be saved on a hard disk and loaded again in subsequent program executions, allowing initially slow programs to run faster in each subsequent execution. Again,

R. Griswold's procedures *encode* and *decode* or **R. Alexander's** *xencode* and *xdecode* from IPL can be used for encoding of the memo table members into strings. The interested reader can find extensive discussion on *xencode* and *xdecode* procedures in **The Icon Analyst** column *From the Library*[1]. Saving and loading of such data is relatively slow, but it typically needs to be done only on the beginning and possibly end of the program execution. Table that contains memoized data can be easily deleted. However, special procedure for that purpose can still provide some convenience.

Unfortunately, coexpressions cannot be saved into files, which results in performance penalties. If memoized data are loaded from file, and more results are required from memoized function than there are memoized results, coexpression need to be reconstructed and re-activated until it produces first result not memoized before.

```
link codeobj
global memo_table, memo_encode
record memo_type(succeeded, result)
record memo3_type( L, c, is_coexpression_updated )
procedure memo3(p, rest[ ])
  /memo_table:=table()
  /memo_encode:=encode

  procedure_name := if type(p)=="string" then p
                            else { s1:=image(p); s1[find(" ",s1)+1:0] }

  code_of_procedure_call:=procedure_name
  every code_of_procedure_call||:=(";" || memo_encode( !rest ))

  if /(result:=memo_table[code_of_procedure_call] ) then {
    c:=create( procedure_name ! rest )
    result:=memo3_type( [], c, 1 )
    memo_table[code_of_procedure_call]:=result
    }

  #Following part suspends results stored in previous executions.

  every x:=!(result.L) do {
    if x.succeeded=0  then {
      # Memo3 fails from table.
      fail
      }
    else  { #write("Memo3 suspends ",x.result, "  from table.");
      suspend x.result
      }
    }

  #Following part updates coexpressions if it lags behind list
  #of the results. It can be very time consuming - but it is typically performed
```

[1] **Icon Analyst** 34, February **1996**, pp. 9-12, http://www.cs.arizona.edu/icon/analyst/backiss/IA34.pdf

```
   # only once in the program, if memo table is loaded from file, but additional
   # results are required.

   if result.is_coexpression_updated=0 then {
      #write("It is found that there is not enough elements in memo-table. ")
      #write( "Coexpression need to be updated, i.e. activated ",*result.L," times.")

      (result.c):=create( procedure_name ! rest )
      every !result.L do {
         @(result.c)
         #write("Coexpression activated.")
         }
      result.is_coexpression_updated:=1
      }

   #activate coexpression, store results for future use and susped them
   while y:=@(result.c) do {
      put(result.L, memo_type(1,y))
      #write("Memo3 computes, stores into memo-table and suspends ",y,".")
      suspend y
      }

   # Coexpression failed. Information on failure is memoized.
   # Coexpression can be deleted.

   put(result.L, memo_type(0, ))
   result.c:=&null
  #write("Memo3 cannot suspend anything, it stores information on that and fails.")
end

procedure memo_save(s)
   /memo_table:=table()
   #write("\n====memo_save(",s,")")
   if not (f := open(s, "w")) then fail
   every x:=key(memo_table)  do {
      write(f, x);
      write(f, encode(memo_table[x]) )
      #write("-----------")
      #write("memo_table[\"",x,"\"]=",ximage(memo_table[x]), " saved.")
      }
   close(f)
   return 1
end

procedure memo_load(s)
   #write("\n====memo_load(",s,")")
   memo_table:=table()
```

```
  if not (f:=open(s, "r")) then fail
  while x:=read(f) do {
    y:=decode(read(f))
    if type(y)=="memo3_type" then y.is_coexpression_updated:=0
    #write("----------")
    #write("memo_table[\"",x,"\"]=",ximage(y)," loaded.")
    memo_table[x]:=y
    }
  close(f)
  return 1
end

procedure memo_delete()
  memo_table:=&null
  return 1
end
```

Some lines of the code used for inspection of the procedure are only commented out, not deleted. The reader can easily introduce them back, if such inspection is of his interest.

The procedure call *f*(expr1,...,exprn) can be replaced with *memo3*("f", expr1,...,exprn) or *memo3*(*f*, expr1,...,exprn) if *f* is a function i.e. its result depends only on its arguments, and it does not produce side effects. If the memoized procedure is recursive, as for example *r_fib* is, such a replacement should be done inside the procedure code as well.

```
procedure m_fib(n)
  return if n<=2 then 1 else memo3(m_fib,n-1)+memo3(m_fib,n-2)
end
```

One can note that the procedure for support of the memoization is named *memo3*. Really, two other procedures for the same purpose are made; they are less general but also less resource demanding versions of *memo3*. The procedure *memo* does not memoize generators, while *memo2* does, but it does not use the described coexpression trick. All three memo procedures can be used in the same program, and results can be stored in the single memo_table, as long as the same memo procedure is used consistently for each procedure call.

```
procedure memo(p, rest[ ])
  /memo_table:=table()
  /memo_encode:=encode

  procedure_name := if type(p)=="string" then p
                            else { s1:=image(p); s1[find(" ",s1)+1:0] }

  code_of_procedure_call:=procedure_name
  every code_of_procedure_call||:=(";" || memo_encode( !rest ))

  if \(x:=memo_table[code_of_procedure_call]) then {
    if x.succeeded=0 then fail else return x.result
    }
```

```
  if x :=( procedure_name ! rest )
    then { memo_table[code_of_procedure_call]:=memo_type(1, x ); return x }
    else { memo_table[code_of_procedure_call]:=memo_type(0, ); fail }
end

procedure memo2(p, rest[ ])
  /memo_table:=table()
  /memo_encode:=encode

  procedure_name := if type(p)=="string" then p
                            else { s1:=image(p); s1[find(" ",s1)+1:0] }

  code_of_procedure_call:=procedure_name
  every code_of_procedure_call||:=(";" || memo_encode( !rest ))

  /memo_table[code_of_procedure_call]:=[]
  L:=memo_table[code_of_procedure_call]
  cardL:=*L

  every x:=!L do { if x.succeeded=0 then fail else suspend x.result }
  i :=0
  every y:=( procedure_name! rest ) do
    if ( i+:=1) >cardL then { put(L,memo_type(1,y)); suspend y }

  put(L,memo_type(0, ))
end
```

The procedures above are tested to some extent. One of the tests is provided here, as an example of different possibilities for memoization.

```
procedure m_fib(n)
  return if n<=2 then 1 else memo(m_fib,n-1)+memo(m_fib,n-2)
end

procedure m2_fib(n)
  return if n<=2 then 1 else memo2(m2_fib,n-1)+memo2(m2_fib,n-2)
end

procedure m3_fib(n)
  return if n<=2 then 1 else memo3(m3_fib,n-1)+memo3(m3_fib,n-2)
end

procedure g_fib() # suspend first n Fibonacci numbers, non-recursive
  suspend (a:=1)|(b:=1)
  repeat {( a+:=b ):=:b; suspend b }
```

```
end

procedure demo1()
   write("\n---------- Test of the correctness: fib(20) should be  6765. ----------\n")

   j := 20
   write("Recursive, not memoized implementation: ",r_fib(j))
   write("Non-recursive, not memoized implementation: ",nr_fib(j))
   write("Recursive implementation with integrated memoization: ",im_fib(j))
   write("Generator without memoization: ",(every x:=g_fib()\j) | x)

   every memo_encode:=![1, encode] & tested_case:=!["",2,3] do {
      write(repl("•",40))

      comment:="Memo"||tested_case||"-ized"
      tested_procedure:=proc("m"||tested_case||"_fib")
      file_name:="memofile"||tested_case||".txt"

      memo_delete() & collect()
      write(comment, " recursive implementation (encode: ", image(memo_encode),
            "): ",tested_procedure(j))
      memo_save(file_name)

      memo_delete() & collect()
      memo_load(file_name)
      write(comment, " populated recursive implementation (encode: ",  image(memo_encode),
            "): ",tested_procedure(j))
      memo_save(file_name)

      tested_procedure:=proc("memo"||tested_case)

      memo_delete() & collect()
      write(comment,"  generator (encode: ", image(memo_encode),"): ",
            (every x:=tested_procedure(g_fib)\j) | x)
      memo_save(file_name)
      #
       memo_delete() & collect()
      memo_load(file_name)
      write(comment," populated generator (encode: ", image(memo_encode),"): ",
            (every x:=tested_procedure(g_fib)\j) | x)
      }
end

procedure demo2()

   write("\n---------- Test of the speed/ms. ----------\n")
```

```
   m:=1000

   write("Recursive, not memoized implementation is too slow to be compared with others.")
   t:=&time;  every k:=1 to m do nr_fib(?k);
   write("Non-recursive, not memoized implementation: ",&time-t)
   t:=&time; every k:=1 to m do im_fib(?k);
   write("Recursive implementation with integrated memoization: ",&time-t)
   t:=&time; every k:=1 to m do (every x:=g_fib()\?k)|x;
   write("Generator: ", &time-t)

   every memo_encode:=! [1, encode] & tested_case:=!["",2,3] do {
     write(repl("•",40) )
     comment:="Memo"||tested_case||"-ized"
     tested_procedure:=proc("m"||tested_case||"_fib")
     file_name:="memofile"||tested_case||".txt"

     memo_delete() &  collect()
     t := &time
     every k:=1 to m do tested_procedure(?k);
     write(comment, " recursive implementation (encode: ", image(memo_encode),"): ", &time-t)
     tested_procedure(m)
     memo_save(file_name)

     memo_delete() &  collect()
     memo_load(file_name)
     t := &time
     every k:=1 to   m do tested_procedure(?k)
     write(comment, " populated recursive implementation (encode: ", image(memo_encode),
           "): ", &time-t)

     tested_procedure:=proc("memo"||tested_case)

     memo_delete() & collect()
     t := &time
     every k:=1 to  m do every tested_procedure(g_fib)\?k;
     write(comment, " generator (encode: ", image(memo_encode),"): ", &time-t)
     memo_save(file_name)

     memo_delete() &  collect()
     memo_load(file_name)
     t := &time
     every k:=1 to   m do every tested_procedure(g_fib)\?k;
     write(comment, " populated generator (encode: ", image(memo_encode),"): ", &time-t)
     memo_save(file_name)
     }
end
```

```
procedure main()
  demo1()
  demo2()
end
```

The test is performed on PC computer with PIII processor working at the rate of 2 GHz, under Unicon 10 and Microsoft Windows. All tests, except attempt of application of procedure memo for memoization of generators gave correct results. Running times vary; however, all memoized versions are faster than recursive-non memoized version *r_fib*, and some of the memoized versions were significantly faster than non-recursive non-memoized function *nr_fib*.

It should be noted that the presented test is by no means representative. Even a slight change of the tested code can cause significantly different and hardly predictable results. For example, the reader can try to replace every occurrence of "*?k*" in *demo2* with simple "*k*" and execute the program again.

```
---------- Test of the correctness: fib(20) should be  6765. ----------

Recursive, not memoized implementation: 6765
Non-recursive, not memoized implementation: 6765
Recursive implementation with integrated memoization: 6765
Generator without memoization: 6765
••••••••••••••••••••••••••••••••••••••••
Memo-ized recursive implementation (encode: 1): 6765
Memo-ized populated recursive implementation (encode: 1): 6765
Memo-ized  generator (encode: 1): 1
Memo-ized populated generator (encode: 1): 1
••••••••••••••••••••••••••••••••••••••••
Memo2-ized recursive implementation (encode: 1): 6765
Memo2-ized populated recursive implementation (encode: 1): 6765
Memo2-ized  generator (encode: 1): 6765
Memo2-ized populated generator (encode: 1): 6765
••••••••••••••••••••••••••••••••••••••••
Memo3-ized recursive implementation (encode: 1): 6765
Memo3-ized populated recursive implementation (encode: 1): 6765
Memo3-ized  generator (encode: 1): 6765
Memo3-ized populated generator (encode: 1): 6765
••••••••••••••••••••••••••••••••••••••••
Memo-ized recursive implementation (encode: procedure encode): 6765
Memo-ized populated recursive implementation (encode: procedure encode): 6765
Memo-ized  generator (encode: procedure encode): 1
Memo-ized populated generator (encode: procedure encode): 1
••••••••••••••••••••••••••••••••••••••••
Memo2-ized recursive implementation (encode: procedure encode): 6765
Memo2-ized populated recursive implementation (encode: procedure encode): 6765
Memo2-ized  generator (encode: procedure encode): 6765
Memo2-ized populated generator (encode: procedure encode): 6765
```

••

Memo3-ized recursive implementation (encode: procedure encode): 6765
Memo3-ized populated recursive implementation (encode: procedure encode): 6765
Memo3-ized generator (encode: procedure encode): 6765
Memo3-ized populated generator (encode: procedure encode): 6765

---------- Test of the speed/ms. ----------

Recursive, not memoized implementation is too slow to be compared with others.
Non-recursive, not memoized implementation: 341
Recursive implementation with integrated memoization: 0
Generator: 431

••

Memo-ized recursive implementation (encode: 1): 60
Memo-ized populated recursive implementation (encode: 1): 20
Memo-ized generator (encode: 1): 10
Memo-ized populated generator (encode: 1): 10

••

Memo2-ized recursive implementation (encode: 1): 80
Memo2-ized populated recursive implementation (encode: 1): 30
Memo2-ized generator (encode: 1): 401
Memo2-ized populated generator (encode: 1): 340

••

Memo3-ized recursive implementation (encode: 1): 802
Memo3-ized populated recursive implementation (encode: 1): 30
Memo3-ized generator (encode: 1): 361
Memo3-ized populated generator (encode: 1): 351

••

Memo-ized recursive implementation (encode: procedure encode): 150
Memo-ized populated recursive implementation (encode: procedure encode): 61
Memo-ized generator (encode: procedure encode): 10
Memo-ized populated generator (encode: procedure encode): 10

••

Memo2-ized recursive implementation (encode: procedure encode): 250
Memo2-ized populated recursive implementation (encode: procedure encode): 100
Memo2-ized generator (encode: procedure encode): 421
Memo2-ized populated generator (encode: procedure encode): 340

••

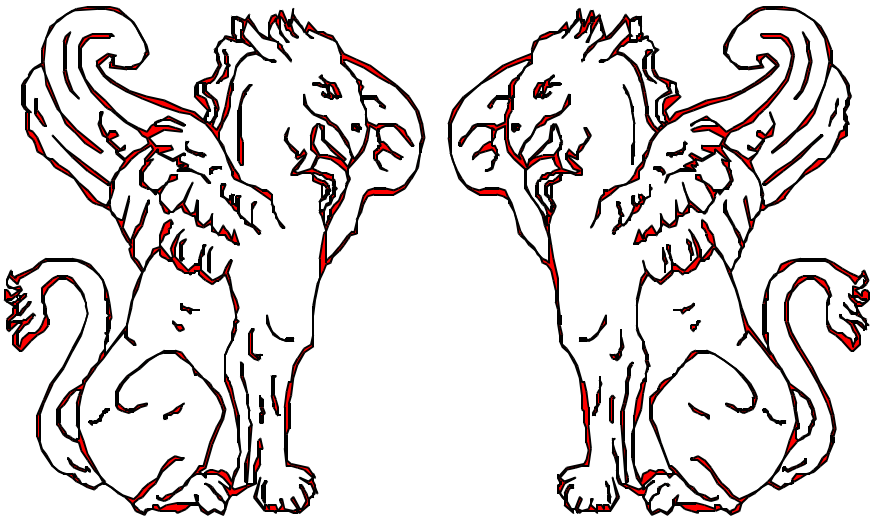
Memo3-ized recursive implementation (encode: procedure encode): 942
Memo3-ized populated recursive implementation (encode: procedure encode): 120
Memo3-ized generator (encode: procedure encode): 361
Memo3-ized populated generator (encode: procedure encode): 350

The procedures for support of memoization presented here have obvious limitations. As already noted, because coexpressions cannot be saved, we did not succeed to eliminate redundancy completely, if generators are memoized across the program execution sessions. Furthermore, due to multi-paradigmatic, "postmodern" nature of the Icon and even more, Unicon, there are many special cases that

require additional attention to be successfully memoized. For example, important string scanning procedures depend on values of &**subject** and &**pos**. In object oriented programming, results of the procedure members typically depend not only on the supplied arguments, but also on the values of data members. There are other special cases, as well.

Theoretically[1], performances of the tables can be improved if expressions like **member**(T, x) and **insert**(T, x, y) are used instead of $\backslash T[x]$ and $T[x]{:=}y$ respectively. Our experience, although far from extensive, however, does not suggest that improvement is significant in this context.

Also, improvements of the memoization technique unrelated to the programming language used are possible. As already noted, some function can not be effectively memoized because there are too many possible combinations of the arguments, especially functions accepting real numbers as arguments. **D. Mitchie** proposed that in such cases, result of the procedure call might be approximated using memoized data on similar procedure calls; it can be achieved through redefinition of the relation "equal to". Similar ideas seem to be recently researched by **Ron Perry**[2] from Cambridge, USA, under the name of *continuous memoization*. For more ambitious memoization projects, essentially limited space might allow only *selective memoization*, similar to the human approach. A good starting point for further reading appears to be a web site of **Umat A. Acar**[3] from Pittsburg.



[1] Programming tips, **The Icon Analyst,** Vol 1 No 1, August **1991**, p. 6,

[2] http://www.merl.com/projects/memoization/

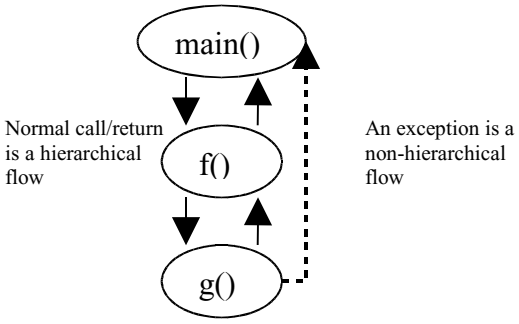
[3] http://www-2.cs.cmu.edu/~umut/

Fun with Co-expressions, part two

Steve Wampler

The first paper in this series looked at the ability to capture result sequences in co-expressions. There is a second, unrelated role that co-expressions play - the ability to function as coroutines.

Each coroutine maintains its own execution environment and resumes execution from the point at which execution last left it. While this sounds a lot like how Icon/Unicon generators behave, the fundamental difference is that the resumption of execution can come from any other coroutine - it need not follow the hierarchical flow inherent in both backtracking and expression evaluation in traditional languages. In essence, coroutines allow the thread of control in a program to follow non-hierarchical patterns:



An example of a non-hierarchical execution flow pattern that is found in a number of languages is the handling of exceptional conditions. Examples of this flow pattern range from the primitive long-jump feature in C to the try-throw-catch clause of Java. Programmers in Icon/Unicon have not had access to a built-in language feature with similar capability - the innate flexibility of goal-directed evaluation, coupled with success-failure semantics of expression evaluation greatly reduces the need for exception handling. This is especially true with the small size of many Icon programs.

With the introduction of classes and objects in Unicon, however, it has become easier to build larger programs. And object-oriented programming seems to lend itself to complex control flows with execution weaving in and out of many objects. An exception mechanism may well have its uses in this new class of Unicon programs.

It turns out that it is possible to implement an exception handling mechanism in Unicon by taking advantage of the coroutine aspect of co-expressions. By coupling co-expressions with objects a remarkably simple implementation can be provided.

Generally speaking, two things are needed to implement exception handling:

- a representation of an exception
- a mechanism for throwing and then catching these exceptions

For convenience both of these aspects are implemented in an `Exceptions` package.

Representing an exception is easy - a record would suffice but a more natural representation is an object that encapsulates actions associated with the exception. Using a class to implement an exception also allows new types of exceptions to be built quickly and easily through subclassing.

Traditionally, exceptions include a message to associate detailed information about the nature of the exception. It is also common to provide each exception with a stack-trace to help locate the source of the exception and at least some portion of the execution flow that lead to that source. Other fields are, of course, possible.

The following is an initial implementation of an exception. More functionality will be added later:

```
import Utils
class Exception : Class (message, location)
    initially()
        location := Utils::buildStackTrace(2)
end
```

This implementation extends the `Class` class found in the *unilib* library at **http://tapestry.tucson.az.us/unicon**. The `Class` class provides some useful operations for all subclasses, but the implementation of `Exception` could be done without subclassing `Class`. (It just might be a bit more work.). Similarly, the `Utils` package from the same site is used to provide support, such as the procedure `buildStackTrace`() used to construct a stack trace from the point of `Exception` creation.

Traditionally, getter and setter methods are used to provide controlled access to class fields (for example, a null field can be managed within a getter). The following getters are added here:

```
    method getMessage()
        return "Exception: " || (\message | "no message")
    end

    method getLocation()
        return (\location | "no stack trace")
    end
```

That is certainly easy enough, though there is more code that could be added to make an Exception easier to work with.

Subclassing `Exception` is easy. Probably the most that might be done is to redefine getMessage:

```
class NumException : Exception()
    method getMessage()
        return "NumException: " || (\message | "no message")
    end
end
```

In many cases all that would be needed is to obtain a new name through subclassing:

```
class IOException : Exception()
end
```

The bulk of the work is in throwing and catching exceptions. This is where co-expressions can help.

The awkward part of implementing the throwing of an exception is that you don't want to do so by unwinding the call stack from the point at which the exception is thrown to the point at which it is caught and handled. Besides being slow, this would also require all the intervening code to be involved in this unwinding, much as it must be to unwind a deeply nested expression failure. It would be much better (and in fact, a requirement for effective use of exceptions) to transfer control directly to the point at which the handling of the exception is performed.

If the point of handling the expression was in a different co-expression from the code throwing the exception, then this could be done by transferring control between the two co-expressions. All that's needed is a standard, uniform method of managing the interaction between these two co-expressions. Ideally, this method would hide the role played by co-expressions from the programmer. After all, their interest is in using exceptions, *not* co-expressions.

The approach given here loosely follows the general structure of Java's exception handling:

1. the code that may end up throwing an exception is encapsulated in a try-clause
2. somewhere in that code an exception may be thrown, possibly from deeply nested in the evaluation
3. the code following the try-clause may choose to catch the exception

Of course, there are some significant differences as well:

1. there is no compile-time checking of the try-throw-catch structure
2. because the entire implementation is done using existing Unicon language features, programmers have more flexibility in how they handle the catching of exceptions.
3. the try-clause produces either the thrown exception (if any) or the standard outcome of evaluating the try-clause code
4. finally, the implementation given here allows exceptions to throw themselves, more in keeping with some object-oriented philosophies

The class `Try` implements the exception handling mechanism. Its outer layer is:

```
class Try : Class (sources, exceptions, lastException)
    ...
    initially()
        sources := []    # stack of 'try' clauses
        exceptions := [] # stack of thrown exceptions
        Try := aTry
        return aTry(self)
end

procedure aTry(c)
    static ego := if Try == aTry then c else Try()
    return ego
end
```

The procedure `aTry`, coupled with the `initially` clause in `Try`, imposes the singleton design pattern onto `Try`. At most one instance of the `Try` object exists, no matter how often the programmer thinks they are calling the constructor. The `lastException` field is used internally to 'capture' the last thrown exception and make it available to testing when catching the exception later.

The three aspects of the exception handling structure are implemented by three methods appropriately named: **call**(coexpression), **throw**(exception), and **catch**(exceptionName). Each is discussed in turn.

The `call` method is responsible for establishing the environment in which code that may throw an exception is executed. It creates a new co-expression for evaluating this code and pushes onto a stack the current co-expression. It also pushes a place holder for the thrown exception. The use of this place holder makes it simpler to perform clean up actions since control may return to the current co-expression by either throwing an exception or by normal expression evaluation if no exception ends up being thrown. The new co-expression is then evaluated and execution in the old co-expression (where this code is) waits for a result. This result may be an exception or the normal outcome (a *result* or *failure*). It doesn't matter what the result is - some simple clean up is all that's needed in any case.

The `call` method is implemented as a PDCO (programmer-defined control operation) which simply means that any expression used as an argument at the point of call is automatically converted into a co-expression. PDCOs are covered in more detail in both the Icon and Unicon books.

```
method call(L)      # L is a list of co-expressions,
                    #    only the first matters.
    local result

    push(sources, &current)     # remember this try clause
    push(exceptions, &null)     # push an exception place holder

    if result := @L[1] then {   # evaluates the try-clause code
        pop(sources)            # clean up on return
        lastException := pop(exceptions) | &null
        return result
        }
    else {                      # try-clause failed
        pop(sources)            # clean up on failure
        lastException := pop(exceptions) | &null
        fail
        }

end
```

The `throw` method simply remembers the exception and passes it to the top co-expression on the sources stack. Additional code terminates the program if an exception is thrown outside of a try-clause.

```
method throw(exception)
    if *sources = 0 then {  # no try-clause!
        stop("Exception thrown outside of Try call: ",
             exception.getMessage(),":\n",exception.getLocation())
        }
    exceptions[1] := exception  # replace place holder
```

```
        exception @ sources[1]
    end
```

The `catch` method is particularly straightforward, it simply checks the name of the last exception against its argument. As a convenience, a missing argument defaults to the name of the `Exception` class above. An existing package `Utils` found on the *unilib* site given earlier makes this easy:

```
    method catch(exceptionName)
        /exceptionName := "Exceptions::Exception"
        return isException(lastException, exceptionName)
    end

    method isException(x, exceptionName)
        /exceptionName := "Exceptions::Exception"
        return Utils::instanceof(x, exceptionName)
    end
```

These methods fail if no exception was thrown (`lastException` will then be null) or if the last exception is not a subclass of the named exception. Otherwise they return the exception.

There is one final implementation point. The design called for exceptions to be able to throw themselves, so a `throw` method is added to the `Exception` class. Code to compute the stack trace is also added to this `throw` method since the location of the throw is more informative than the location of the creation of the `Exception` object:

```
    method throw(aMessage)
        message := \aMessage
        location := Utils::buildStackTrace(2)
        Try().throw(self)
    end
```

Now that there is an implementation of an exception handling mechanism, how might it be used? The following toy program gives a simple example. The program takes each of its arguments in turn and performs a simple action on it. Somewhere during the evaluation of this action, however, an exception may be thrown (there are two different types of the exceptions possible: the 'standard' `Exception` and an exception of type `'NumException'`. The main procedure catches both types of exceptions separately in case different actions need to be taken. In this simple example, the same action is performed on each type.

```
    import Exceptions

    class NumException : Exception()
        method getMessage()
            return "NumException: " || (\message | "no message")
        end
    end

    procedure main(args)

        every i := !args do {
            case Try().call{ f(i) } of {  # Note use of PDCO!

                Try().catch("NumException"): {
                        x := Try().getException()
                        write(x.getMessage(), ":\n", x.getLocation())
```

```
                        }

                Try().catch(): {
                        x := Try().getException()
                        write(x.getMessage(), "\n", x.getLocation())
                        }

                }
            }
    end

    procedure f(i)
        write("x is '",g(i),"'.")
    end

    procedure g(i)
        if not numeric(i) then {
            NumException().throw("'"||i||"'")
            }
        if i = 3 then {
            Exception().throw("bad value of "||i)
            }
        return i
    end
```

When run with the arguments `1 2 3 a 5`, the following output results (note: The lack of line numbers and file names in the stack traces is a known deficiency in the current implementation of `Utils::buildStackTrace`):

```
x is '1'.
x is '2'.
Exception: bad value of 3:
    procedure g
    procedure f
    procedure main

NumException: 'a':
    procedure g
    procedure f
    procedure main

x is '5'.
```

There are a few interesting points about the above code, particularly for programmers familiar with exception handling in Java.

First, it is important to note that the use of a PDCO encapsulates the expression given as the argument to `Try().call` in a co-expression. This has the effect of constraining side effects that result from the evaluation of that expression. In particular, changes to local variables made within that expression are not visible upon completion of the `Try().call` method invocation. However, the fact that the outcome of the try clause is either an expression or the outcome of evaluating the encapsulated expression means that it is possible to migrate information from the try clause back to the local environment.

A Unicon **case** statement is used to handle the different catch-clauses. This is not necessary but provides a convenient syntax. The above code would work equally well if the **case** expression were replaced with:

```
if Try().call{ f(i) } then {
    if Try().catch("NumException") then {
        ...
        }
    else if Try().catch() then {
        ...
        }
    }
```

Since the outcome of invoking the `Try.call` method can be either an exception or the outcome of the encapsulated expression, it is possible to defer the catch-clause handling until later, possibly performing some clean-up actions in the interim:

```
if r := Try().call{ f(1) } then {
    ...           # perform some cleanup
    if Try().isException(r, "NumException") then {
        ...
        }
    else if Try().isException(r, "Exception") then {
        ...
        }
    }
```

Here, `isException` is used in place of `catch` because the cleanup code may contain an embedded try-clause of its own.

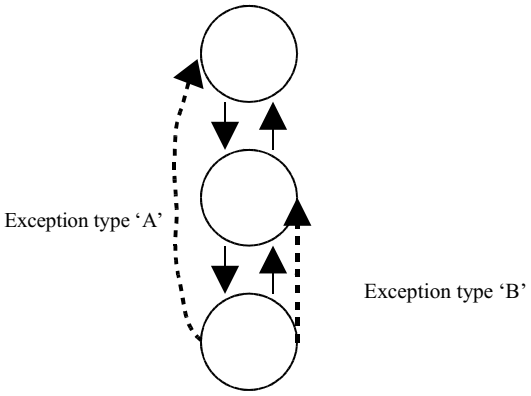
Finally, although the invocation of the `call` PDCO shown above embeds a function call, *any* Unicon expression may appear as the argument to `call`, including a compound expression:

```
case Try().call{{     # inner braces form compound expr.
        k:= f(i)
        x := process(k+i)
        write("Result is: ", x)
        }} of {

    Try().catch("NumException"): {
        x := Try().getException()
        write(x.getMessage(), ": ", x.getLocation())
        }

    Try().catch(): {
        x := Try().getException()
        write(x.getMessage(), ": ", x.getLocation())
        }
    }
```

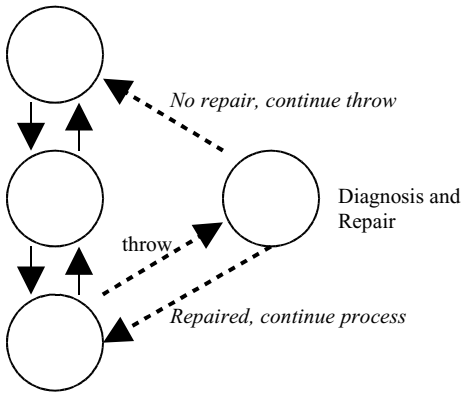
The latest version of the `Exceptions` package can be found at http://tapestry.tucson.az.us/unicon in the file **Exceptions.icn**. Similarly, a sample program using Exceptions can be found at the same site in the file **ExceptionTest.icn**.

Of course, this isn't the only possible implementation of an exception mechanism and, since the implementation in entirely written in Unicon, it is easy to experiment with other possibilities. For

example, a simple change would be to have the `Try().call` PDCO accept a second argument that is a list of the types of exceptions to be caught at this point in the code. Then Try().throw could consult these lists for each available catch point and jump directly to the proper one:



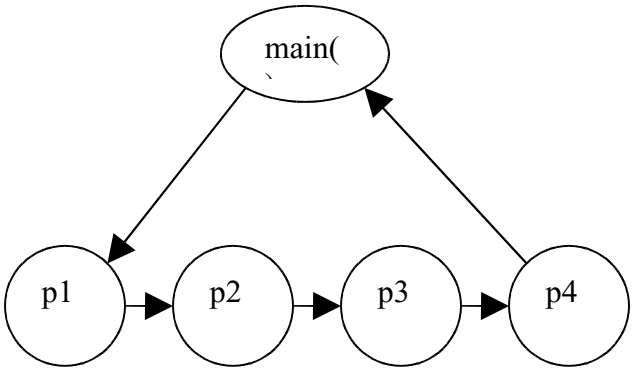
A more ambitious implementation might route all exceptions to a special 'diagnosis and repair' module that would attempt to repair the problem. If successful, this module could return control *back* to the point of the exception throw!



Lurking beneath the exception handling example is another interesting use of co-expressions as coroutines. The Try class mechanism is actually a general-purpose *call-throw-catch* mechanism that has been modified to support exceptions. It would not take much to turn it back into the general case.

What makes this interesting is that suddenly there is an entirely different expression evaluation mechanism available than the traditional hierarchical call/return sequence. While the exception handling support was layered on top of a traditional hierarchical system, a general call-throw-catch system can be used to implement a *pipeline processing* system. In a pipeline system each component takes input from the preceding component, manipulates this input in some fashion and then passes it on to the next component. There is no need for a component to wait for a result and propagate it back up a calling

sequence. Instead, the final component in the pipeline can directly return the result to the top of the tree using a throw.



There are many situations that lend themselves naturally to a pipeline system: a sequence of graphical data transformations is a common one, particularly when non-affine transformations are involved. The pipelines available typical Unix shells are the prototypical example – a pipeline system that is routinely used for many complex tasks. Tasks in a pipeline system implemented in Unicon using the approach outlined here may be procedures, co-expressions, or even separate pipeline systems.

Underdocumented Unicon

This is the first of a regular column providing juicy details on features of general interest that have been added to the Unicon language, but have so far slipped through the cracks of documentation. Often a feature familiar to Icon programmers has been extended in Unicon, but the extension may be under documented. In its broadest interpretation, this column is about differences between Icon and Unicon which may be of general interest, as opposed to extensions that are specific to particular application domains such as graphics. A rich source for such material is to examine the CVS logs of the Unicon project, and simply look for interesting log messages.

The first subject of this column is the tried and true function delete(X,x). In Icon, this important function deletes set element or table key x from set or table X. In Unicon, it takes an arbitrary number of arguments, so you can delete as many elements or keys as you wish. The call delete(L,x_1,...,x_n) deletes elements at subscripts x_1...,x_n from list L, by analogy to tables. The current implementation is not as clever as you would like for frequent deletes on very large lists; each call to delete(L) is proportional to the size of L...but much faster than if you write your own procedure to do the job. It is much faster to call delete(L) once with many arguments, than it is to call delete(L,x) many times. If you have a list of subscripts to delete stored in variable Lsubs, you might try

```
delete ! [L] ||| Lsubs
```

instead of

```
every i := 1 to *Lsubs do delete(L, Lsubs[i] – (i-1))
```

By the way, why is this code subtracting (i-1) from the subscript?

For both delete(T) and delete(L), one other disturbing question is: how does one delete elements by their value, rather than their key or index? The answer, unfortunately, is that it is easy but painfully slow, with techniques such as

```
while T[k := key(T)]===v do delete(T, k)
```

or for some set S of values to delete:

```
len := *L; every i := 1 to *L do if not member(S, (x := pop(L))) then put(L, x)
```

Recently a code review of an application turned up a circumstance where this operation needed to be performed on a list of 6,000 to 10,000+ elements as fast as possible (under 1/30th of a second would be nice). Watch Unicon CVS logs and see if delete(L, S) is a built-in yet. Unfortunately it is not straightforward to extend delete(T) in this way, as delete(T,S) is in principle already defined. If you have to delete elements by value from a table fast, try a second table which contains for every value, the list or set of keys that map to it. You probably want to encapsulate this with a class so you control access to T and keep the 2nd table in sync. If you have another good solution, please share it!