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.

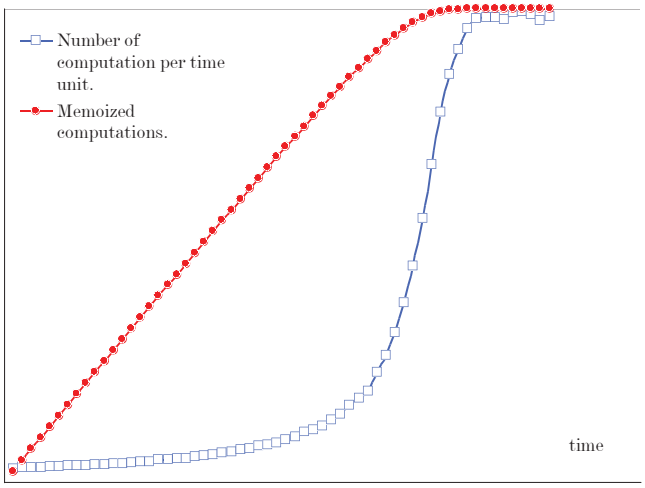
It is frequently hard to predict whether computation of some function can benefit from memoization. It depends on many factors, including, but not limited to the efficiency of the implementation of a memoization, time required for the computation of the function without memoization, probability that 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

••

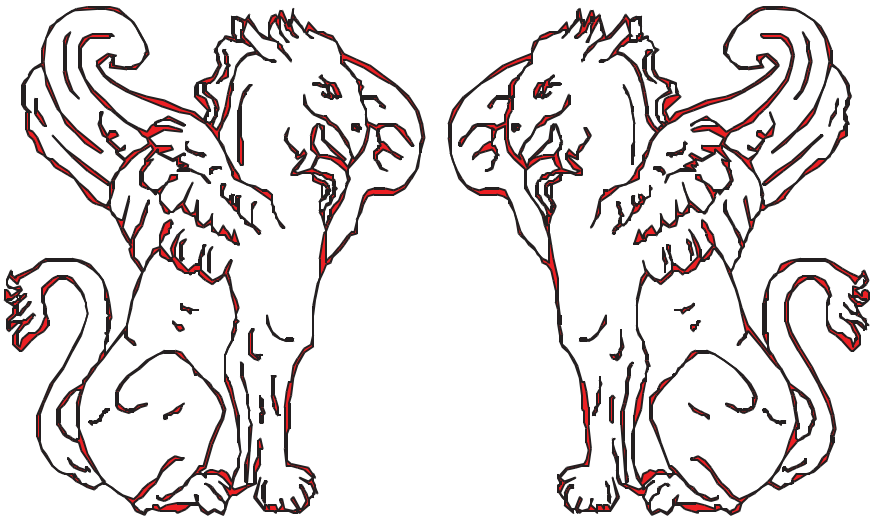
Memo3-ized recursive implementation (encode: procedure encode): 942
Memo3-ized populated recursive implementation (encode: procedure encode): 120
Memo3-ized generator (encode: procedure encode): 361
Memo3-ized populated generator (encode: procedure encode): 350

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/