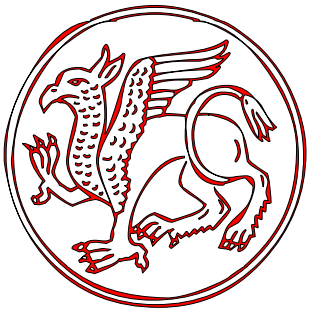
The Generator.

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 1. No 1.

March MMIV.

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 two standard formats, A4 and Letter. In some circumstances, use of the printing options *Auto-rotate and Center* and *Fit to paper* might be appropriate.

Call for Papers.

The Generator publishes 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 All articles contain some amount of the previously unpublished material; an exception is material previously published on less formal ways (preprints, mailing list and newsgroups posts etc.)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 writing style is preferred.
All submitted articles are reviewed.
The author permits unlimited publishing of the submitted article in The Generator.
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.
Used 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>.
Used fonts: Weatherly Systems Inc. Ramona, Bitstream De Vinne and `Pica10`, Corel Frankenstein.

Publisher

Published by K. Majorinc, Drvinje 20, Zagreb, Croatia

Launching The Generator.

This is the first 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** is inspired by **The Icon Analyst**, a publication edited by **Madge** and **Ralph Griswold** and **Gregg Townsend** from **1990** to **2001.** We can thank them for the excellent publication they produced[1], and hope that we'll see some their contributions in years to come. Although **The Generator** can hardly avoid comparisons with a publication that inspired it, reader should see **The Generator** not as a continuation of **The Icon Analyst**; but rather a continuation of its *tradition*.

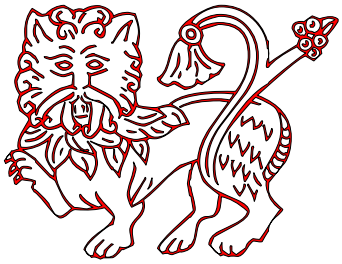
The arrival and spread of the INTERNET in the last decade of the 20th century allows easy and simple distribution of our publication without costs for authors and readers. Also, articles can be supported with unlimited amount of the source code, graphical or textual output.

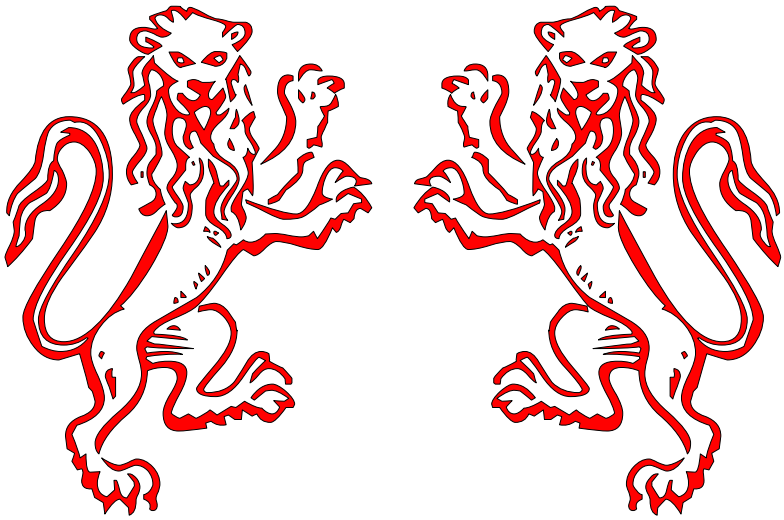
It is our understanding that information, experience or knowledge is too valuable for its presentation to be conditional upon its completeness, as is most frequently the case. If a reader uses Unicon or Icon for *any* purpose or in *any* way previously not described in the literature on these programming languages, no matter the level of complexity, **The Generator** is interested in publishing his experiences.

Finally, we want to thank all who worked or contributed to the Icon and Unicon projects, led by **R. Griswold** and later by **G. Townsend,** and **C. Jeffery,** respectively, and to all that contributed to the tradition of these two programming languages outside of the mentioned projects.

---

[1] All issues of **The Icon Analyst** and supplementary materials are available, free of charge, at Icon Project WWW site, <http://www.cs.arizona.edu/icon/analyst/ia.htm>.





Good Looking Circles.

Kazimir Majorinc.

We attempt to draw a simple illustration for a thermodynamics textbook. The molecules in fluid are represented with circles of equal size. The circles can touch, but not intersect each other. They should be densely positioned, i.e. it should be impossible to draw an additional circle without intersecting others. The whole picture should look *natural*. That condition is vague but unavoidable: the picture should not look like it is drawn by human, or by any other, unrelated algorithm. Also, the program does not need to be fast; only one good picture need to be drawn for the textbook.

The problem is solved in four attempts; all four are described in this article. A few elements are common to all attempts. The record *circle* and procedure *draw_circle* are defined; they only make the program slightly shorter and simpler. The distance between two circles, d, is defined as in geometry; actually, even bit more generally: it is assumed that the distance between intersecting circles is negative.

```
link random
link graphics
global SIZE, R
record circle(x, y, r)


procedure d(C1, C2)
  return sqrt((C1.x-C2.x) ^ 2+(C1.y-C2.y) ^ 2)-C1.r-C2.r
  end

procedure draw_circle(C)
  DrawCircle(C.x, C.y, C.r)
  return C
  end

procedure eliminate_intersecting(Candidates, c)
  every insert(ToDelete:=set(), d(c, c0:=!Candidates)<0 & c0 )
  return Candidates --:=ToDelete
  end


procedure main(args)
  SIZE:=150
  R:=10
  every (SIZE|R) := get(args)
  randomize()
  attempt1()
  attempt2()
  attempt3()
  attempt4()
  write("Done")
  WDone()
  end
```

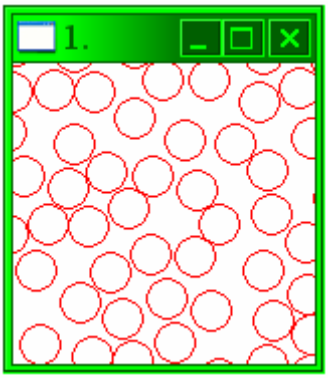
As the running time is not important, all algorithms are left in the simplest, unoptimized form. After a circle is drawn, in the next step, we form the set *Candidates* that contains all circles that satisfy criteria for a next step. Most importantly, they do not intersect any already drawn circles. One of the candidates is randomly chosen and drawn. This process repeats until *Candidates* is empty. In three of four attempts, the *Candidates* set initially contains all circles that could be drawn in quadrant of the given size.

The First Attempt.

In the first attempt no additional criteria are imposed. The procedure that draws a random set of the non-intersecting circles is short and simple.

```
procedure attempt1()
   &window:=WOpen("height="||SIZE, "width="||SIZE, "label=1.", "fg=red")
   every insert(Candidates:=set(), circle(-2*R to SIZE+2*R, -2*R to SIZE+2*R, R))
   while Candidates:=eliminate_intersecting(Candidates, draw_circle(?Candidates))
   end
```

Some circles are drawn outside of the visible part of the picture, to avoid possible edge effects on the picture.



This picture is not satisfactory: the circles are not dense enough and the parts left uncovered by circles are too big. As all circles are chosen randomly, the program can, theoretically, draw denser picture; however, it is not likely to happen in a reasonable number of executions.

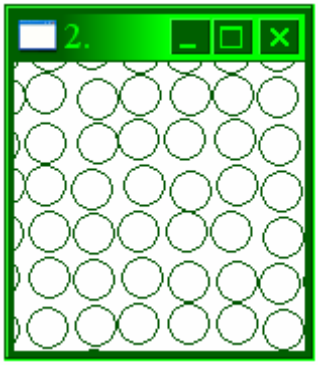
The Second Attempt.

It was suggested that the program could choose circles that form a perfect quadratic or hexagonal pattern, and then slightly translate individual circles in the random direction. Ideally, both higher density and randomness can be achieved.

```
procedure attempt2()
   distance_between_centers:=(2*R+3)
   tolerance:=3
   &window:=WOpen("height="||SIZE,"width="||SIZE,"label=2.","fg= dark green")
   every insert ( Candidates:=set(),
                  i:=-2*R to SIZE+2*R &
                  j:=-2*R to SIZE+2*R &
                  (i+100) % distance_between_centers <= tolerance &
                  (j+100) % distance_between_centers <= tolerance &
                  circle(i, j, R)
                )
   while Candidates:=eliminate_intersecting(Candidates, draw_circle(?Candidates))
   end
```

The algorithm is similar to the previous one. Additionally, an initial choice of the candidates is narrowed to the circles with coordinates of the centres in the segments $[k(2R+3), k(2R+3)+3]$, $k=0, 1, 2 \ldots$ We added the constant 100 to avoid aperiodicity of the operation % around zero.

However, the result of the second attempt is even less satisfactory.



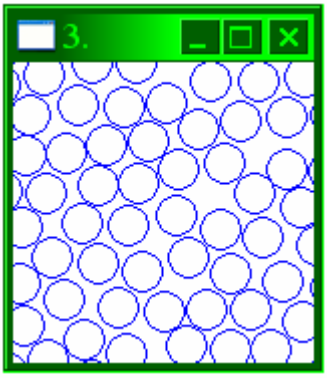
Although the circles are dense, their distribution is too regular. The square pattern is clearly visible, especially if seen from a distance. Variations of the values *tolerance* and *distance_between_centers* do not help: the pattern can become less obvious only if the density is significantly decreased.

The Third Attempt.

In the third attempt we applied a similar idea, but more locally. If circles are chosen so they touch at least two already drawn circles, then the result is a perfect hexagonal pattern. We expected that, using a slightly looser requirement: circles are drawn so they are close to two other circles, density can be preserved, while irregularities will accumulate to the degree that hexagonal pattern will not be clearly visible in picture.

```
procedure attempt3()
   min_allowed_d:=0
   max_allowed_d:=5
   &window:=WOpen("height="||SIZE, "width="||SIZE, "label=3.", "fg=blue")
   every insert(Candidates:=set(), circle(-2*R to SIZE+2*R, -2*R to SIZE+2*R, R))
   Drawn:=[ ]
   repeat
   { Candidates2:=[ ]
      every c:=!Candidates do
      { close_circles:=0
         every (min_allowed_d <= d(!Drawn, c) <= max_allowed_d)
         do close_circles+:=1
         if (close_circles > 1) | (close_circles = *Drawn <= 1)
         then put(Candidates2, c)
      }
      if not(c:=?Candidates2) then fail
      Candidates:=eliminate_intersecting(Candidates, draw_circle(c))
      put(Drawn, c)
   }
   end
```

Our expectations are fulfilled and resulting picture looks significantly better; it is both dense and irregular.

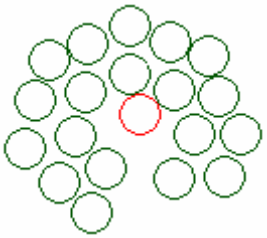


Some circles form hexagonal pattern, but it looks more like the natural tendency of the densely packed circles than the result of some inadequate algorithm.

However, as can be seen from the picture, some relatively large empty areas occurred again on pictures drawn with *attempt3*. Although it is not obvious from the above static picture, observation of the program during work clearly reveals the origin of such empty spaces.

The set of the all drawn circles can form figures with large concavities that are bigger than one circle. Some of these concavities can not be filled with densely packed circles.

See, for example, the picture below: the green circles form a concave figure, and after red circle (one in the center if you see this article in monochrome) is drawn, the large area around it can not be filled any more.

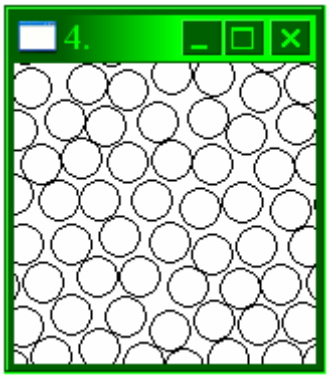


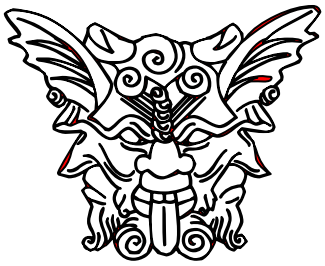
The Fourth Attempt.

The problem with the previous attempt can be solved easily. We define the center of the first drawn circle as the *Center* of the whole picture. Beside the previously mentioned criteria, drawn circles are chosen to be relatively close (variable *tolerance*) to the *Center*. If *tolerance* is 0, the circle closest to the center that does not intersect with other circles is drawn. This ensures that figures formed by the circles have no large concavities.

```
procedure attempt4()
   min_allowed_d:=0
   max_allowed_d:=5
   tolerance:=5
   &window:=WOpen("height="||SIZE, "width="||SIZE, "label=4.", "fg=black")
   every insert(Candidates:=set(), circle(-2*R to SIZE+2*R, -2*R to SIZE+2*R, R))
   Drawn:=[ ]
   repeat
   { if /Center
        then c:=(Center:=?Candidates)
        else { Candidates2:=[ ]
               min_d:=1E5
               every c:=!Candidates do
               { close_circles := 0
                 every (min_allowed_d <= d(!Drawn, c) <= max_allowed_d)
                    do close_circles+:=1
                 if (close_circles > 1) | (close_circles = *Drawn <= 1)
                    then { put(Candidates2, c)
                           min_d >:= d(Center, c)
                         }
               }
               every put(Candidates3:=[], (d(Center, c:=!Candidates2)<=min_d+tolerance) & c)
               if not(c:=?Candidates3) then fail
             }
     put(Drawn, c)
     Candidates:=eliminate_intersecting(Candidates, draw_circle(c))
   }
   end
```

Finally, the result satisfied all our criteria; at least sufficiently well for the textbook illustration.





Fun With Co-expressions, Part One.

Steve Wampler.

Unicon (and Icon) users are likely to be familiar with the role that result sequences play in the language[1]. Result sequence is the term coined to describe the sequence of results that an expression is capable of producing during goal-directed evaluation. For example, the result sequence of 1 to 4 is {1, 2, 3, 4}.

The term is, however, artificial; there is no language entity that is a result sequence. (Ralph Griswold did experiment with a language Seque[2] in the late 1980's that did elevate result sequences into first class data objects, however.) Expressions simply produce results when evaluated, with the ability to provide alternative results when backtracked into during goal-directed evaluation.

Nevertheless, the idea of result sequences as language objects is appealing. It's even possible to think of expressions as being the language's way of expressing result sequences - the representation of a result sequence then becomes the code that is capable of producing it. Hiroshi Shinohara has been experimenting with this in his work on using generators as filters as seen recently on the Unicon mailing list. (I'm not sure Hiroshi thinks of his work in quite this fashion, but that's how I see it!)

The biggest problem one encounters when attempting to work with result sequences as language objects is that the representation of any result sequence (i.e. the code) is lexically fixed to the point in the program where that code appears. This severely restricts the flexibility of this approach to programming.

There is, however, a way forward. You can't separate a result sequence from the code that generates it, but you can capture that code in a co-expression. The effect is one of 'freeing' a result sequence from a single lexical location in a program, allowing the manipulation of the result sequence, if not as a first class data type in Unicon, at least as a second class data type. I'll call this new data entity a Result Sequence, using capitalization to distinguish it from the more general result sequences found throughout Unicon code evaluation.

You can assign the Result Sequence to a variable, pass it to functions and procedures, and include it in computations. In short, you can treat it nearly the same as you can other structured data entities. (Note the 'nearly' - a Result Sequence is not a list. If you want a list, use a list.)

No data entity is useful unless the language provides operations for manipulating that data entity and Result Sequences are no exception. Certainly the primary Result Sequence operation is goal-directed evaluation - a key feature in Unicon and Icon that provides much of the expressiveness inherent in those languages. By capturing a result sequence as a co-expression, several other operations become available for manipulating that Result Sequence.

First, there is the obvious operation: activation (@). Activation simply produces the next result from the sequence and fails if there are no results left. Activation is the foundation for all actions involving co-expressions and, hence, is the gateway required to gain access to the result sequence the co-expression represents. Most other Result Sequence operations are simply 'normal' Unicon operations layered on top of activation.

---

[1] For introduction see Result Sequences, The Icon Analyst, No 7, August 1991, pp. 5-8. [ed.]

[2] For more details see Lost Languages - Seque, The Icon Analyst, No 19, August 1993, pp. 1-4 [ed.]

Activation is, by design, not a generating operation. When back-tracked into during goal-directed evaluation, no further results are produced, even if the Result Sequence contains additional results. One of the first operations that can be layered on top of activation is repeated alternation (unary |), which turns any expression into a generating expression. So, the combined operation |@ allows the use of a Result Sequence in backtracking contexts.

What can you do with a Result Sequence that might be hard to do when you're forced to access a result sequence at a single lexical place? Lots of things.

How about producing every other result? If x is a Result Sequence, then: |{@x; @x} does the trick. All results after the first n?

{**every** |@x\n; |@x} works just fine.

Want to interleave the results of two expressions? If x and y are Result Sequences for the two expressions, then use: |@(x|y).

Some things are bit trickier, such as producing every nth result from Result Sequence x: |{|@x\(n-1) & &**fail**; @x}. The &**fail** is needed to force the first subexpression (which is just an expression to produce the first n-1 results from a Result Sequence) to generate all of its alternatives before moving to the second subexpression. If you find the appearance of & &**fail** unsettling, as I do, you can use **Steve Hunter's** technique for achieving the same effect: |{\/|@x\(n-1); @x}. (This is visually appealing as \/ is reminiscent of the 'for all' operator common to logic systems - though it can take a bit of thought to realize just why it works as a replacement for appending & &**fail.** Unfortunately, it's not safe to carry the relationship with logic systems too far, as /\ is not the same as 'for any' but rather equivalent to \/!) Perhaps just rewriting the above as |{**every** |@x\(n-1); @x} is the clearest approach.

All of the above examples are concise and show the expressive power available in Unicon. However, they also are perhaps too 'information dense' - the conciseness that authors of such code find appealing can make it difficult for others to understand the meaning. For some reason it's psychologically unappealing to have to spend so much time grasping the behaviour of so little code. So perhaps it's better to spread out the code more, but still make it easy to use by placing such operations into procedures. For example, the first example |{@x; @x} can be rewritten as a procedure:

```
procedure evenResults(x)
  while @x do
    suspend @x
end
```

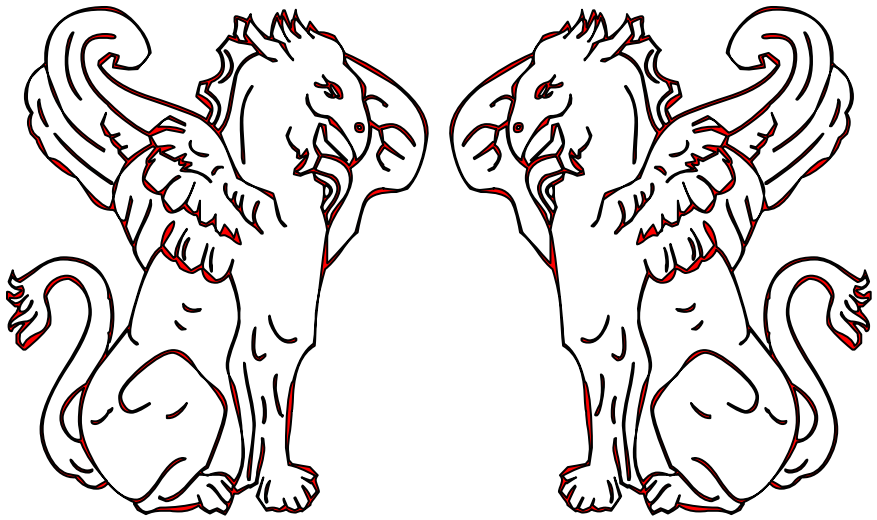
This becomes even more useful as the complexity of the operation increases.

This approach also allows the use of a meaningful name in place of a complex expression and allows one to build up a package of useful Result Sequence operations. It might be a good exercise to rewrite each of the above expressions as procedures (and no, simply embedding the expression into a procedure, as in:

```
procedure evenResults(x)
  suspend |{@x; @x}
end
```

doesn't count!).

Thinking in terms of operating on sequences of results takes some getting used to, in part because there are very few languages that provide the tools to do so. (I once asked a group of people to give me a representation for the phrase “the value of x lies between 1 and 10” - the programmers in the group [none of whom had seen Icon or Unicon] tended to give '$(1 < x)$ and $(x < 10)$'. Only a mathematician suggested '$1 < x < 10$'. What the mathematician saw as obvious wasn't obvious to people thinking in terms of C or FORTRAN.) Result sequences are a major feature of Unicon and using co-expressions to turn result sequences into Result Sequences can be a powerful tool for expressing problem solutions.



The Generator
Vol 1. No 1.
March MMIV.

Unification in Propositional Calculus.

Kazimir Majorinc

Propositional calculus is perhaps the simplest and the best known example of a formal theory[1]. The formulas of propositional calculus are either (1) *propositional variables* (for example A, B, C, ...) or (2) constructed from other formulas using unary (~) or binary connectives (|, &, >, ...). If infix notation for binary connectives is used, these formulas are similar to the usual algebraic formulas: for example (~A), (A>B), (A>(~(A>B))) are formulas, while A~B>B is not.

Somehow, surprisingly, a standard set of axioms for propositional calculus has not been established; one can hardly find two books that use identical sets of axioms.

There is more agreement about rules of inference. Two of the most frequently used rules are (1) *substitution*: if F is a theorem, X_1, ..., X_n are some of the variables occurring in F and G_1, ..., G_n are formulas without occurrences of X_1, ..., X_n then the formula F' obtained from F by simultaneous substitution of all occurrences of X_1, ..., X_n with G_1, ..., G_n respectively is also the theorem and (2) *modus ponens:* if formulas $(F>G)$ and F are theorems, then G is also theorem. Additional rules of inference are not necessary. Axioms are, by definition, also theorems of propositional calculus.

A logician usually defines propositional calculus syntactically, because syntax is finite and even - visible, and as such it raises less doubts than any semantics[2]. However, the usual intention is to finally add semantics to the defined syntax. Typically, variables are interpreted as statements of the natural language (including mathematical extensions) and connectives ~, |, & and > as logical operators "not", "or", "and" and "implies" respectively. With proper choices of axioms and inference rules, propositional calculus is *complete*, i.e. all *tautologies* (i.e. statements that are true no matter which natural language statements are represented by variables) and only tautologies are theorems of propositional calculus.

Although the tautology concept might seem trivial and useless, it is not. For example, if we know that $(F_1>F_2)$ is a tautology, we also know that F_2 is true whenever F_1 is true; certainly, such a conclusion is not trivial for every possible interpretation of F_1 and F_2 in natural language.

Definition of the propositional calculus is constructive; in principle, one can make a program that derives all theorems of propositional calculus. However, after half a century of research, computers have only established a marginal role in the development of the mathematical knowledge. Furthermore, difficulties in designing programs that match human capabilities in games such as chess, or especially *go* are not encouraging as it is probable that mathematics is more complicated than these games.

---

[1] Any introductory text in mathematical logic will contain an extensive survey of important results, for example any edition of **E. Mendelson,** Introduction to Mathematical Logic.

[2] For example, some logicians do not accept that double negation implies affirmation. They, however, find formula as ((~(~A))>A) acceptable if it is defined as string of characters, without meaning.

Theorems, as found in books are both *true* and *interesting* mathematical statements. In attempts to automate their derivation[1], two general approaches are used. In the first approach, one starts from an interesting statement and tries to determine whether it is true. In the second approach, one starts from statements known to be true and tries to find interesting consequences. These two approaches are called *automated theorem proving* and *automated theorem finding*[2]. (far less researched with exception of the works of **S. N. Vassilyev**[3].)

In practice, the problem with automated theorem finding is always the same: naive algorithms for deriving tautologies generate many obviously trivial, weak or other less than interesting theorems. If interesting theorems are derived at all, it is not known how they could be identified and isolated from myriads of others also generated in the process.

For example, substitution can be applied to any formula and infinitely many formulas can be derived from it. Unfortunately, all these derived formulas are longer and weaker than the premise. These are weaker in the usual mathematical sense that can be easily recognized but is hard to formalize.

Modus ponens is different: the consequence is shorter and stronger than the longer one of the two premises. Unfortunately, modus ponens can rarely be applied; almost certainly it cannot be applied on two randomly chosen theorems of propositional calculus.

This difference suggests that integration of these two rules in some combined rule can both reduce combinatorial explosion caused by substitution and increase the frequency of *successful* application of modus ponens in the process of the development of propositional formulas. One possible *combined rule* is (3) for two theorems F and $(G{>}H)$, if there are substitutions s and t such that $s(F){=}t(G)$ then $t(H)$ is also a theorem.

The combined rule is not trivial any more. The essential part of the problem is determining whether for given formulas F and G there exist substitutions s and t such that $s(F){=}t(G)$. That problem also occurs in other contexts and the commonly used name for it is *unification*[4] of formulas.

An algorithm for unification of two propositional formulas is easily implemented in a Unicon program of about hundred and fifty lines.

```
link strings
link sets

$define NL "\n"
$define LINE repl("=", 20)
$define TRUE 1
$define FALSE 0
```

---

[1] Good survey article is **M. Beeson,** The Mechanization of Mathematics in **C. Teuscher,** (ed.) ***Alan Turing****: Life and Legacy of a Great Thinker,* Springer-Verlag, Berlin, **2003**.

[2] **L. Wos,** The Problem of Automated Theorem Finding. *Journal of Automated Reasoning,* Vol. **10**(1), **1993**, pp. 137-8.

[3] Probably the best review of his work is **S. N. Vassilyev,** Machine Synthesis of Mathematical Theorems, *Journal of Logic Programming*, Vol 9, **1990**, pp. 235-66.

[4] Extensive review can be found in **F. Baader**, **W. Snyder**, Unification Theory, Chapter 8, pp. 439-526 in **A. Robinson**, **A. Voronkov** (ed.), *Handbook of Automated Deduction,* Elsevier/MIT Press, **2001**.

```
$define xxxxxx1 (
$define init_to :=:temp_init_to):=(if \temp_init_to then temp_init_to else
$define xxxxxx2 )

procedure main()
   every k:=1 to 4 do
   {F:= [  ["(A>((B>(C>B))>D))","((a>(b>c))>((a>b)>(a>c)))"],
        ["(A>(~A))","((~B)>B)"],
        ["(A>(~B))","(B>(~A))"],
        ["(A>(~B))","(B>(~A))"]
       ][k]
    Fca:=[[TRUE, TRUE], [TRUE, TRUE], [TRUE, FALSE], [TRUE, TRUE]][k]
    write(LINE, NL, "Unification of: ")
    every i:=idx(F) do write(F[i], ", changes allowed: ", Fca[i])
    write("Unification succeeded: ", unified( F, Fca ).formula)
   }
 end
```

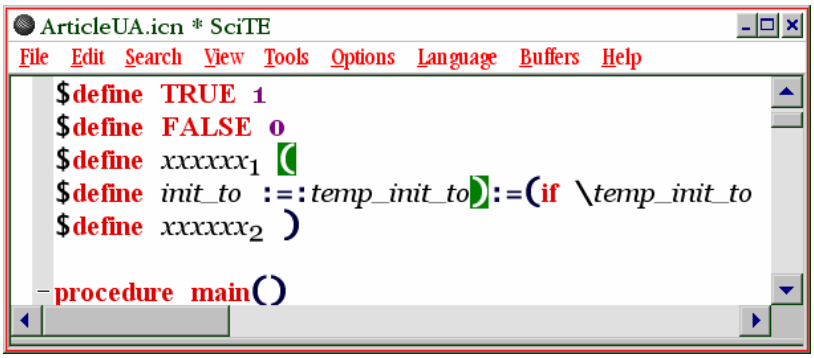
The program links to standard libraries "strings" and "sets." A few simple macros are defined. Only the infix macro operator *init_to* deserves some comment. The expression ((*x init_to* expr)) is equivalent to longer expressions like {/*x*:=expr; *x*} or {**if** /*x* **then** *x*:=expr **else** *x*} that allow initialization of variables in the same place they are used in loops. The macro *init_to* is usually slower than initialization outside of the loop. In some expressions, like **every** ((*x init_to* 0))+:=1 **to** *n*, one can replace ((*x init_to* 0)) with simple (*x*:=0). Also, one execution of the macro in some procedure must be completed before another execution is started. Hence, nested expressions like ((*x init_to* ((*y init_to* 0)))) do not work correctly.[1] However, we believe that replacement of the frequently occurring idioms or patterns in the programs with simple, non-redundant syntactical constructs reveals the logic of the programming itself, so the price appears to be acceptable.[2]

The seemingly strange macros *xxxxxx1* and *xxxxxx2* have only one role: to balance the parenthesis left open by *init_to* and prevent errors in text editors with integrated parentheses matching; the definition of the macro *init_to* uses parentheses in an unusual way; for an excellent example see **N. Hodgons's** SciTE[3].

[1] It seems to be one of the most frequent problems with macros..

[2] **C. Evans** implemented a more powerful macro system and special syntax (x :$ expr) for {/x:=expr; x} in his private build of Unicon.

[3] <http://www.scintilla.org/SciTE.html>.



For testing and demonstration purposes, four pairs of formulas are stored in the list F and passed as an argument to the procedure *unified*. That procedure returns a record consisting of (1) the formula resulting from unification and (2) a list of all performed substitutions.

The procedure *unified* accepts another argument, *Fca*, a list of Boolean values. For clarity of intention the macros FALSE and TRUE are used respectively. In this implementation of *unified*, only formulas $F[i]$ such that $Fca[i]$=TRUE can be changed. If unification succeeds and $Fca[i]$ was FALSE, then $F[i]$ can be obtained from $F[3\text{-}i]$ by substitution. Less formally, $F[i]$ is a special and weaker case of $F[3\text{-}i]$.

Some procedures used in the program can be useful in a more general context. They are copied from other programs or generalized and extracted elements of the early working versions of this program.

```
procedure is_true(B)
  if B==TRUE then return TRUE
  end

procedure card(predicate, X)
  every ((result init_to 0))+:=( predicate(!X) & 1 )
  return result
  end

procedure card_nulls(X)
  every ((result init_to 0)) +:= ((/!X) & 1 )
  return result
  end

procedure card_columns(LL)
  every ((result init_to *?LL)) <:= *!LL
  return result
  end

procedure idx(L)
  suspend 1 to *L
  end
```

```
procedure jdx(LL)
  suspend 1 to card_columns(LL)
  end

procedure column(LL, j)
  if /j then suspend column(LL, jdx(LL))
       else  {  every L:=!LL do
                  put( ((C init_to [])) , if j <= *L then L[j] else &null)
                  return C
                }
  end


procedure projection(XX, index)
  if  type(XX)=="list"
    then {  result:=[];
              every X:=!XX do put(result, X[index])
              return result
            }
  end

procedure is_simple_type(x)
  if type(x)==("real"|"integer"|"string") then return x
  end

procedure generalized_application(p, L)
  every put(result:=[], p(!L))
  return result
  end

procedure equal_by_value(X)
  if not different_by_value(X) then return X
  end


procedure different_by_value(L)
  S:=set(L)
  if member(S, "&equal") & member(S, "&different")
     then error("Ambivalent different/equal_by_value.")
  if member(S, "&equal") then fail
  if member(S, "&different") then return L
    case card_nulls(L) of { 1: return L; 2: fail }
    return case card(is_simple_type, L) of
    {  1: L
       2: if L[1] ~=== L[2] then L else &fail
       0: if different_by_value(column(L)) then L else &fail
    }
  end
```

The predicates *is_true* and *is_false* allow convenient combination of Boolean and success/ failure program control flow.

The generator *jdx*(*LL*) accepts a list of lists as an argument; if understood as two-dimensional array, *jdx* generates indexes of its columns. The procedure *column*(*LL*, *j*) returns *j*-th column of such an array, i.e. list [*LL*[1][*j*], ..., *LL*[**LL*][*j*]]; if the second argument is omitted, it generates all columns of *LL*. Expressions symmetrical to *jdx*(*LL*), *column*(*LL*, *j*) and *column*(*LL*) are 1 **to** **L*, *LL*[*j*] and !*LL* respectively. Syntactical symmetry can be achieved by implementation of procedures *idx*(*LL*)[1] and *row*(*LL*, *j*).

The procedure *projection*(*X*, *index*) is a generalization of the procedure *column*(*LL*, *j*); it accepts a list of tables as an argument and *index* can be any key in the table. Further generalization can be useful.

The procedure *generalized_application*(*p*, *L*) returns the list [*p*(*L*[1]), ..., *p*(*L*[**L*])]. It is similar to **R. Griswold's** *apply* in the Icon Program Library, file "`apply.icn`". Further generalization can be useful.

A few procedures with names containing the prefix *card*[2] count elements of the structures satisfying given criteria.

Unicon's built in operator === and its negation ~=== compare equality of the two structures "*by reference.*" Although there are few similarities with set-theoretical equality, === does not satisfy the axiom of extensionality[3]. For example, {1, 2}={1, 2} is true in set theory, while its Unicon equivalent **set**([1, 2]) === **set**([1, 2]) does not necessarily succeed[4].

Design and implementation of a relation more similar to set theory equality has been addressed in the past[5].

The procedures *different_by_value* and *equal_by_value* presented here are more limited than **J. P. de Ruiter's** procedure. However, they have one useful additional property. Pseudo- keywords "`&equal`" and "`&different`", are defined as *equal_by_value* and *different_by_value* to any value. Comparison between "`&equal`" and "`&different`" is not defined and will result, in a runtime error if attempted.

```
procedure is_variable(F)
  if find(F, &letters) then return F
  end

record character_index_level_type(character, index, level)

procedure character_index_level(F)
  suspend character_index_level_type(
```

[1] The function **key** is equivalent to *idx*.

[2] The name of the procedure is inspired by the set-theoretical concept of the cardinal number.

[3] Sets are uniquely defined by their members, i.e. $(\forall x)(\forall y)(((\forall z)(\, z\in x \leftrightarrow z\in y)) \leftrightarrow (x=y))$

[4] Actually, **set**([1, 2]) === **set**([1, 2]) never succeeds in Unicon.

[5] **R. Griswold's** procedure *equiv* (`equiv.icn`, I.P.L.) and **J. P. de Ruiter's** procedure *same_value* (`mset.icn`, I.P.L.) should be mentioned.

```
    cF:=!F,
    ((i init_to 0)) +:= 1,
    ((lev init_to 0)) +:= case cF of { "(": 1; ")": -1; default: 0}
    )
  end

procedure main_connective(F)
  return equal_by_value([ character_index_level(F), ["~"|">", "&equal", 1]])[1]
  end


procedure analysed_formula(F)
  T:=table()
  if is_variable(F)
    then T["variable"]:=F
    else{m:=main_connective(F)
        T["connective"]:=m.character
        T["left"]:=F[2:m.index]
        T["right"]:=F[m.index + 1: -1]
       }
  return T
end
```

The predicate *is_variable* allows all lowercase and uppercase letters as propositional variables.

The procedure *character_index_level*(*F*) generates records containing successive individual characters of the formula *F*, the position index of the character in the formula and the number of opened and unclosed parentheses before that position. Note that **suspend**, aside from its primary role, resumes all generators like **every**.

Perhaps the most elegant procedure in the whole program, *main_connective*(*F*) returns a connective ("~" or ">") enclosed in exactly one pair of parenthesis in the formula *F* and its position in that formula.

The procedure *analysed_formula* accepts a formula as an argument and returns a table containing the main connective and both the left and right subformulas of a given formula. If the main connective is unary, i.e. "~", the left subformula is by the definition empty string.

Finally, we approach the most specific parts of the program.

```
record substitution(variable, formula)

procedure forced_substitution(F, Fca)
  if is_variable(!F)
    then if i:=idx(F) & is_variable(F[i]) & is_true(Fca[i])
      then return substitution(F[i], F[3-i] )
      else {write("No substitution: formulas differ in variable "||
                  "but substitution is not allowed.")
            fail
           }

  AF:=generalized_application(analysed_formula, F)
```

```
  "~==" !( D:=projection(AF, j:=!["connective", "left", "right"]))
  if j=="connective"
    then write("No substitution: different main connectives.")
    else return forced_substitution(D, Fca)
  end

procedure substitute(F, Fca, s)
  every is_true(Fca[i:=idx(F)])
     do F[i]:=replace(F[i], s.variable, s.formula)
  if find(s.variable, !F)
    then write("Substitution failed: ", s.variable, " cannot be eliminated.")
    else return s
  end

record unified_type(formula, substitution)

procedure unified(F, Fca)
  while different_by_value( F ) do
    if not(s:=forced_substitution(F, Fca) &
         write( LINE, NL, "Substitution ", s.formula,
               " for ", s.variable, " suggested."
              ) &
         substitute(F, Fca, s) &
         write("Substitution succeeded.", NL, F[1], NL, F[2]) &
         put( ((applied_substitutions init_to [])), s)
       )
    then fail
  return unified_type(?F, applied_substitutions)
  end
```

The procedure *unified* contains a loop that is repeated as long as formulas $F[1]$ and $F[2]$ are different. In the loop two elementary operations are performed, (1) searching for substitutions that need to be performed and (2) performing the substitutions. If any of these two fail, unification also fails. Those two operations are delegated to the procedures *forced_substitution* and *substitute*.

The prefix “forced” in *forced_substitution* suggests that a found substitution has to be applied; otherwise, it would be impossible to unify two formulas. The *forced_substitution* first searches for the difference between two formulas, translating them into the form of a tree 'on the fly' and then tries to match these trees. There are a few different cases, dependent on the difference between formulas $F[1]$ and $F[2]$.

In the simplest case exactly one of the formulas is a propositional variable; let us denote it with $F[i]$. If changing $F[i]$ is allowed then substitution of $F[3\text{-}i]$ for $F[i]$ is necessary for unification. If changes to the formula $F[i]$ are not allowed, then $F[1]$ and $F[2]$ cannot be unified.

If both formulas are variables, then either of the substitutions $F[1]$ for $F[2]$ or $F[2]$ for $F[1]$ can be chosen.

If neither one of the formulas in F is variable and they differ in the main connective then no substitution can unify them.

Finally, if both formulas in F are complex, (i.e. not variables) and have the same main connectives and differ in at least one of the corresponding subformulas then further searching is performed recursively.

Once found, substitution can be performed easily. The procedure **replace** from "`strings.icn`" in the Icon Program Library can be used for formulas in the form of the string.

Under some circumstances substitution fails, i.e. when a substituted variable still occurs in some part of the formula F. This can happen if (1) the formula to be substituted for a variable contains the same variable[1]; for example, if (∼B) is substituted for B; or (2) when a substituted variable occurs in a formula where changes are not allowed. If substitution fails, again, unification of the formulas is impossible.

After the formulas are unified it does not matter which one is returned as result of the unification; so a random choice is returned. Output produced by the program is relatively readable.

```
==========
Unification of:
(A>((B>(C>B))>D)), changes allowed: 1
((a>(b>c))>((a>b)>(a>c))), changes allowed: 1
==========
Substitution (a>(b>c)) for A suggested.
Substitution succeeded.
((a>(b>c))>((B>(C>B))>D))
((a>(b>c))>((a>b)>(a>c)))
==========
Substitution a for B suggested.
Substitution succeeded.
((a>(b>c))>((a>(C>a))>D))
((a>(b>c))>((a>b)>(a>c)))
==========
Substitution (C>a) for b suggested.
Substitution succeeded.
((a>((C>a)>c))>((a>(C>a))>D))
((a>((C>a)>c))>((a>(C>a))>(a>c)))
==========
Substitution (a>c) for D suggested.
Substitution succeeded.
((a>((C>a)>c))>((a>(C>a))>(a>c)))
((a>((C>a)>c))>((a>(C>a))>(a>c)))
Unification succeeded: ((a>((C>a)>c))>((a>(C>a))>(a>c)))
==========
Unification of:
(A>(∼A)), changes allowed: 1
((∼B)>B), changes allowed: 1
==========
```

[1] The occur-check test is frequently discussed in the context of Prolog. Most implementations do not perform occur-check.

Substitution (~B) for A suggested.
Substitution succeeded.
((~B)>(~(~B)))
((~B)>B)
==========
Substitution (~(~B)) for B suggested.
Substitution failed: B cannot be eliminated.
==========
Unification of:
(A>(~B)), changes allowed: 1
(B>(~A)), changes allowed: 0
==========
Substitution B for A suggested.
Substitution failed: A cannot be eliminated.
==========
Unification of:
(A>(~B)), changes allowed: 1
(B>(~A)), changes allowed: 1
==========
Substitution B for A suggested.
Substitution succeeded.
(B>(~B))
(B>(~B))
Unification succeeded: (B>(~B))

For some pairs of formulas, for example (B>(C>(D>((a>a)>((b>b)>((c>c)>d))))))) and ((A>A)>((B>B)>((C>C)>(b>(c>(d>D)))))), unification requires exponential running time.

==========
(B>(C>(D>((a>a)>((b>b)>((c>c)>d))))))), changes allowed:
1
((A>A)>((B>B)>((C>C)>(b>(c>(d>D)))))), changes
allowed: 1
==========
Substitution (A>A) for B suggested.
Substitution succeeded.
((A>A)>(C>(D>((a>a)>((b>b)>((c>c)>d))))))
((A>A)>(((A>A)>(A>A))>((C>C)>(b>(c>(d>D))))))
==========
Substitution ((A>A)>(A>A)) for C suggested.
Substitution succeeded.
((A>A)>(((A>A)>(A>A))>(D>((a>a)>((b>b)>((c>c)>d)))
)))
((A>A)>(((A>A)>(A>A))>((((A>A)>(A>A))>((A>A)>(A
>A)))>(b>(c>(d>D))))))
==========
Substitution (((A>A)>(A>A))>((A>A)>(A>A))) for D
suggested.
Substitution succeeded.
((A>A)>(((A>A)>(A>A))>((((A>A)>(A>A))>((A>A)>(A

>A)))>((a>a)>((b>b)>((c>c)>d))))))
((A>A)>(((A>A)>(A>A))>((((A>A)>(A>A))>((A>A)>(A
>A)))>(b>(c>(d>(((A>A)>(A>A))>((A>A)>(A>A)))))))))
==========
Substitution (a>a) for b suggested.
Substitution succeeded.
((A>A)>(((A>A)>(A>A))>((((A>A)>(A>A))>((A>A)>(A
>A)))>((a>a)>(((a>a)>(a>a))>((c>c)>d))))))
((A>A)>(((A>A)>(A>A))>((((A>A)>(A>A))>((A>A)>(A
>A)))>((a>a)>(c>(d>(((A>A)>(A>A))>((A>A)>(A>A)))))
))))
==========
Substitution ((a>a)>(a>a)) for c suggested.
Substitution succeeded.
((A>A)>(((A>A)>(A>A))>((((A>A)>(A>A))>((A>A)>(A
>A)))>((a>a)>(((a>a)>(a>a))>((((a>a)>(a>a))>((a>a)>(
a>a)))>d))))))
((A>A)>(((A>A)>(A>A))>((((A>A)>(A>A))>((A>A)>(A
>A)))>((a>a)>(((a>a)>(a>a))>(d>(((A>A)>(A>A))>((A
>A)>(A>A)))))))))
==========
Substitution (((a>a)>(a>a))>((a>a)>(a>a))) for d suggested.
Substitution succeeded.
((A>A)>(((A>A)>(A>A))>((((A>A)>(A>A))>((A>A)>(A
>A)))>((a>a)>(((a>a)>(a>a))>((((a>a)>(a>a))>((a>a)>(
a>a)))>(((a>a)>(a>a))>((a>a)>(a>a)))))))))
((A>A)>(((A>A)>(A>A))>((((A>A)>(A>A))>((A>A)>(A
>A)))>((a>a)>(((a>a)>(a>a))>((((a>a)>(a>a))>((a>a)>(
a>a)))>(((A>A)>(A>A))>((A>A)>(A>A)))))))))
==========
Substitution A for a suggested.
Substitution succeeded.
((A>A)>(((A>A)>(A>A))>((((A>A)>(A>A))>((A>A)>(A
>A)))>((A>A)>(((A>A)>(A>A))>((((A>A)>(A>A))>((A>
A)>(A>A)))>(((A>A)>(A>A))>((A>A)>(A>A)))))))))
((A>A)>(((A>A)>(A>A))>((((A>A)>(A>A))>((A>A)>(A
>A)))>((A>A)>(((A>A)>(A>A))>((((A>A)>(A>A))>((A>
A)>(A>A)))>(((A>A)>(A>A))>((A>A)>(A>A)))))))))
Unification succeeded:
((A>A)>(((A>A)>(A>A))>((((A>A)>(A>A))>((A>A)>(A
>A)))>((A>A)>(((A>A)>(A>A))>((((A>A)>(A>A))>((A>
A)>(A>A)))>(((A>A)>(A>A))>((A>A)>(A>A)))))))))

The resulting formula is exponentially longer than the input of the program. Hence, improvement of the algorithm is not possible without redefinition of the propositional calculus. This important negative result is, however, not completely surprising. Similar inefficiencies are observed in the related fields of propositional calculus, and relative improvements are achieved through introduction

of the equality in language or equivalent use of alternative data structures[1]. That idea is, also, fruitfully applied on the unification problem.[2]

---

[1] The most important examples are described in **G. S. Tseitin,** On the Complexity of Derivation in Propositional Calculus, in *Studies in Constructive Mathematics and Mathematical Logic,* Part 2. Consultant Bureau, New York **1968**, pp. 115-25. and **S. A. Cook** and **R. A. Rechkow,** The Relative Efficiency of Propositional Proof Systems. *Journal of Symbolic Logic* 44 (**1979**), pp. 36-50. We addressed similar problem in **K Majorinc,** Extension Rule for Non-Clausal Propositional Calculus, *Fundamenta Informaticae,* Vol 31, No 2, August **1997**, pp. 107-16.

[2] Few quadratic and linear time algorithms for unification in more general sense are reported. Perhaps the best known one is described by **A. Martelli** and **U Montanari** in An Efficient Unification Algorithm, *ACM Transactions on Programming Languages and Systems* **4**(2), **1982,** pp. 258-82.

From Mailing List Archives.

Icon's mailing list archive contains lots of interesting ideas, notes and programs. However, its huge size of about 10 000 pages, Spartan format, and unavoidable redundancies might discourage many - if not the majority - of Unicon users from the study of that valuable resource. In the following issues of **The Generator** we'll republish selected posts or their excerpts that, we believe, deserve to be more readily available to Unicon users.

In this issue, **G. Yee's** implementation of mathematical functions from **1986** is presented. Note that functions fail (instead of producing runtime error) if argument is not in the domain of definition, e.g. *sqr*(-1), *log*(-1) etc. All procedures except *floor*(x) and *ceil*(x) have been incorporated into Unicon and Icon as functions. Functions *atan2*(y, x) and *atan*(x) have been coalesced into *atan*($r1$, $r2$) while *log*(x) and *log10*(x) have been coalesced into *log*(x, b).

Yee's Mathematical Procedures.

From ralph Sat Mar 15 08:42:07 **1986**
From: "**Ralph Griswold**" <ralph>
Subject: math procedures

George Yee, a graduate student in the Department of Computer Science at the University of Arizona, has written a package of math procedures in Icon. A UNIX-style manual page and the source code for these procedures follow:

```
------------------------------------------------------------------------
MATH(3.icon)          Icon Program Library          MATH(3.icon)
NAME
     sin, cos, tan, asin, acos, atan, atan2 - trigonometric func-
     tions and their inverses
SYNOPSIS
     link "math"
     sin(x)
     cos(x)
     tan(x)
     asin(x)
     acos(x)
     atan(x)
     atan2(y, x)
     dtor(deg)
     rtod(rad)
DESCRIPTION
     Sin, cos and tan return trigonometric functions of radian
     arguments x.
     Asin returns the arc sine in the range -pi/2 to pi/2.
     Acos returns the arc cosine in the range 0 to pi.
     Atan returns the arc tangent in the range -pi/2 to pi/2.
     Atan2(y, x) :=  atan(y/x)                    if x > 0,
                     sign(y)*(pi - atan(|y/x|))   if x < 0,
                     0                            if x = y = 0, or
                     sign(y)*pi/2                 if x = 0 ~= y.
     Dtor converts degrees to radians, while rtod converts radi-
```

```
     ans to degrees.
DIAGNOSTICS
     If |x| > 1 then asin(x) and acos(x) will fail.
MATH(3.icon)          Icon Program Library          MATH(3.icon)
NAME
     sqrt - square root
SYNOPSIS
     link "math"
     sqrt(x)
DESCRIPTION
     Sqrt(x) returns the square root of x.
DIAGNOSTICS
     Sqrt(negative) fails to produce a result.
MATH(3.icon)          Icon Program Library          MATH(3.icon)
NAME
     exp, log, log10 - exponential and logarithm
SYNOPSIS
     link "math"
     exp(x)
     log(x)
     log10(x)
DESCRIPTION
     Exp returns the exponential function of x.
     Log returns the natural logarithm of x.
     Log10 returns the logarithm of x to base 10.
DIAGNOSTICS
     Log(negative) and log10(negative) fail to produce a result.
MATH(3.icon)          Icon Program Library          MATH(3.icon)
NAME
     floor, ceil - floor and ceiling
SYNOPSIS
     link "math"
     floor(x)
     ceil(x)
DESCRIPTION
     Floor returns the largest integer no greater than x.
     Ceil returns the smallest integer no less than x.
```

```
# math.icn - mathematical procedures for Icon programming language
#
#   Version 1.0  created on 10 February 1986.
#
#   Procedures developed in Icon by  George D. Yee
#                        1847 N. Frances Blvd.
#                        Tucson, AZ  85712
#
#   Free distribution and use of this material is granted provided the
#   above credit is left intact on all source copies.  No warranties
#   are made as to the correctness or suitability of these procedures
#   for any purpose.  Please send any suggestions to me at the above
```

```
#    address.
#

procedure sin(x)
   return _sinus(numeric(x), 0)
end

procedure cos(x)
   return _sinus(abs(numeric(x)), 1)
end

procedure tan(x)
   return sin(x) / (0.0 ~= cos(x))
end

# atan returns the value of the arctangent of its
# argument in the range [-pi/2, pi/2].
procedure atan(x)
   if numeric(x) then
      return if x > 0.0 then _satan(x) else -_satan(-x)
end

# atan2 returns the arctangent of y/x
# in the range [-pi, pi].
procedure atan2(y, x)
   local r
   static pi
   initial pi := 3.14159265358979323846264 3
   return if numeric(y) & numeric(x) then {
      if x > 0.0 then
         atan(y/x)
      else if x < 0.0 then {
         r := pi - atan(abs(y/x))
         if y >= 0.0 then r else -r
         }
      else if x = y = 0.0 then
         0.0         # special value if both x and y are zero
      else
         if y >= 0.0 then pi/2.0 else -pi/2.0
      }
end

procedure asin(x)
   if abs(numeric(x)) <= 1.0 then
      return atan2(x, (1.0-(x ^ 2)) ^ 0.5)
end

procedure acos(x)
   return 1.570796326794896619231e0 - asin(x)
end
```

```
procedure dtor(deg)
   return numeric(deg)/57.29577951308232
end

procedure rtod(rad)
   return numeric(rad)*57.29577951308232
end

procedure sqrt(x)
    return (0.0 <= numeric(x)) ^ 0.5
end

procedure floor(x)
   return if numeric(x) then
      if x>=0.0 | real(x)=integer(x) then integer(x) else -integer(-x+1)
end

procedure ceil(x)
   return -floor(-numeric(x))
end

procedure log(x)
   local z, zsq, ex
   static log2, sqrto2, p0, p1, p2, p3, q0, q1, q2
   initial {
      # The coefficients are #2705 from Hart & Cheney. (19.38D)
      log2  := 0.69314718055994530 9e0
      sqrto2 := 0.70710678118654752 4e0
      p0    := -0.24013917955921051 0e2
      p1    := 0.30957292821537650 1e2
      p2    := -0.96376909336868659 3e1
      p3    := 0.42108737121797971 4e0
      q0    := -0.12006958977960525 5e2
      q1    := 0.19480966070088973 1e2
      q2    := -0.89111090279831233 7e1
      }
   if numeric(x) > 0.0 then {
      ex := 0
      while x >= 1.0 do {
         x /:= 2.0
         ex +:= 1
         }
      while x < 0.5 do {
         x *:= 2.0
         ex -:= 1
         }
      if x < sqrto2 then {
         x *:= 2.0
         ex -:= 1
         }
      return ((((p3*(zsq:=(z:=(x-1.0)/(x+1.0))^2)+p2)*zsq+p1)*zsq+p0)/
```

```
            (((1.0*zsq+q2)*zsq+q1)*zsq+q0))*z+ex*log2
      }
end

procedure exp(x)
   return 2.71828182845904523536028 ^ numeric(x)
end

procedure log10(x)
   return log(x)/2.30258509299404568402
end

procedure _sinus(x, quad)
   local ysq, y, k
   static twoopi, p0, p1, p2, p3, p4, q0, q1, q2, q3
   initial {
      # Coefficients are #3370 from Hart & Cheney (18.80D).
      twoopi := 0.63661977236758134308
      p0     := 0.13578840978773756690926 80e8
      p1     := -0.49429081009028441611586 27e7
      p2     := 0.44010305353752665019449 18e6
      p3     := -0.13847272499824528730544 57e5
      p4     := 0.14596884066657687222269 59e3
      q0     := 0.86445586529225344299151 49e7
      q1     := 0.40817922523432997493957 79e6
      q2     := 0.94630961015382081805712 57e4
      q3     := 0.13265349087861363589114 94e3
      }
   if x < 0.0 then {
      x := -x
      quad +:= 2
      }
   y := (x *:= twoopi) - (k := integer(x))
   if (quad := (quad + k) % 4) = (1|3) then
      y := 1.0 - y
   if quad > 1 then
      y := -y
   return (((((p4*(ysq:=y ^ 2)+p3)*ysq+p2)*ysq+p1)*ysq+p0)*y) /
          ((((ysq+q3)*ysq+q2)*ysq+q1)*ysq+q0)
end
procedure _satan(x)
   static sq2p1, sq2m1, pio2, pio4
   initial {
      sq2p1 := 2.41421356237309504880 2e0
      sq2m1 := 0.41421356237309504880 2e0
      pio2  := 1.57079632679489661923 1e0
      pio4  := 0.78539816339744830961 5e0
      }
   return if x < sq2m1 then
             _xatan(x)
          else if x > sq2p1 then
```

```
        pio2 - _xatan(1.0/x)
      else
        pio4 + _xatan((x-1.0)/(x+1.0))
end
procedure _xatan(x)
  local xsq
  static p4, p3, p2, p1, p0, q4, q3, q2, q1, q0
  initial {
    # coefficients are #5077 from Hart & Cheney. (19.56D)
    p4  := 0.16153641298223022826 2e2
    p3  := 0.26842548195503973794141e3
    p2  := 0.11530293515404850115428136e4
    p1  := 0.17804063164331969710546458 7e4
    p0  := 0.89678597403663861959987488e3
    q4  := 0.58956970508444622227 91e2
    q3  := 0.53626537403121531510423 5e3
    q2  := 0.16667838148816337184521798e4
    q1  := 0.20793349744454098128727592 6e4
    q0  := 0.89678597403663861962481162e3
    }
  return x * ((((p4*(xsq:=x^2)+p3)*xsq+p2)*xsq+p1)*xsq+p0) /
      (((((xsq+q4)*xsq+q3)*xsq+q2)*xsq+q1)*xsq+q0)
end
```