RECURSION EQUATIONS AS A PROGRAMMING LANGUAGE*

D. A. Turner

Computing Laboratory
University of Kent, UK

The last few years have seen a growing interest in functional (or applicative) languages as a potential alternative to conventional programming languages, particularly since Backus's Turing lecture (Backus 1978). This interest arises from two distinct causes, one coming from software considerations and the other from developments in the hardware. On the software side, there is mounting evidence that the collection of ideas that came to maturity in the late sixties and are loosely called "structured programming" have simply failed to deliver the reduction in software costs that was originally hoped for — perhaps because the break then proposed with traditional programming practices was insufficiently radical. At the same time, people on the hardware side are searching for new architectures, and therefore new methods of programming, that are capable of taking advantage of the possibility of a very large degree of concurrency in the machine, a possibility that is opening up because of the development of VLSI.

The importance of applicative languages lies in the fact that they hold out the promise of being able to solve both of these problems at the same time. In both cases, the key step is the abolition of the assignment statement and with it the notion of sequencing. Before looking in more detail at an applicative language and its properties it is worthwhile to review the software and hardware arguments for considering such a radical change in our programming practice.

The Software Crisis and its Causes

It is commonly observed that we have a software crisis and in a gathering of computer scientists it should not be necessary to multiply examples. Everyone has their own favourite horror story about a project that failed in some catastrophic way because of a bug in a program. Less spectacular but equally worrying is the high cost of producing software even for comparatively simple applications. It is by now clear that the largest single obstacle to the wider use of computers is our inability to produce cheap, reliable and manageable software. The more dramatic the advances on the hardware side the more embarrassing this fact becomes. Does it seem too unreasonable to suggest that there is something fundamentally wrong about the way in which we produce software?

* Originally published in *Functional Programming and its Applications*, eds. J. Darlington, P. Henderson and D. A. Turner © Cambridge University Press, 1982. Reproduced with permission.

I shall argue that the basic problem lies in the nature of existing programming languages. Existing programming languages emerged in a relatively short period between 1955 and 1960 — Fortran and Cobol set the pattern for later languages. They have evolved since primarily by way of becoming more complicated — the underlying principles have not changed. When we compare, say, Pascal with Fortran, the similarities are much more significant than the differences. More precisely, the differences are superficial but the similarities are fundamental. At a certain level of abstraction all the programming languages in production use today are the same. All are sequential, imperative languages with assignment as their basic action.

When we compare our programming languages with all previous mathematical notation, however, the differences are very striking. Mathematical notation has evolved over many centuries and obeys certain basic rules which are common to every area of mathematics and which give mathematical notation its deductive power. Let us briefly enumerate some of these basic properties of mathematical notation.

First of all mathematics is static. There is no equivalent in mathematics of the programming language notion of a procedure which gives a different answer each time you call it. The mathematical idea of a function is a fixed table of input–output pairs. Given the same f and the same x, the value of $f\ x$ must always be the same. This is so even when mathematics is used to describe processes of change as in physics. We proceed by making time a parameter. That is we formalise the notion of a three dimensional world which is changing by talking about a static four dimensional world. Physics as a science became possible only because Newton showed us how to reduce dynamics to statics in this way.

Secondly, and relatedly, there is in mathematics a certain kind of consistency in the use of names — consider for example the equation

$$x^2 - 2x + 1 = 0$$

which has only one solution, namely $x = 1$. Now, supposing someone were to say "no, there is another solution—we can take the first occurrence of x to be 3 and the second to be 5, giving $3\times3-2\times5+1 = 0$, which is also correct", what would we say? We would say that this proposed "solution" is invalid because it ignores a basic premise of the whole exercise, namely that x is supposed to stand for the *same* value throughout its scope, otherwise there would have been no point in always calling the value x. Paradoxical though it may sound, in mathematics variables do not vary; they stand for a constant value throughout their scope.

These basic properties of mathematical notation have been termed by logicians "referential transparency" (Russell & Whitehead 1925, Quine 1962). In mathematics, an expression is used only to refer to (or *denote*) a value and the same expression always denotes the same value (within the same scope).

Finally, we can relate this to the notion of equality, which plays a fundamental role in mathematical reasoning. Two expressions are said to be equal if and only if they denote the same value. An immediate consequence of referential transparency is that equality is substitutive — equal expressions are everywhere interchangeable. It is this which gives mathematical notation its deductive power.

Now, how do our conventional programming languages relate to this tradition? It is clear that they don't adhere to it, because in introducing the assignment statement, which can change the value of a variable in the middle of its scope, they have broken the basic ground rules of mathematical notation. Instead of being referentially transparent, programming languages are *referentially opaque*. In fact, the things that we call "variables" in languages like Algol are not really variables in the mathematical sense at all — in the last analysis they are names for registers in the store of a Von–Neumann computer.

It is because of this that it is difficult to reason about programs. Since expressions can change their value through time, equality is not substitutive. Indeed, in a programming language it does not even have to be true that an expression is equal to itself — because the presence of side effects may mean that evaluating the same expression twice in succession can produce two difference answers! In general it is not possible to reason about such programs on the basis of a static analysis of the program text — instead, we have to think of the program dynamically and follow the detailed flow of control, and this seems unreasonably difficult.

A particularly sharp symptom of the software crisis is the fact that after more than a decade of intensive effort — starting with say Floyd (1967) and Hoare (1969) — we still do not have anything resembling a practically viable set of techniques for giving formal proofs of program correctness on production programs. It seems likely that this is because existing programming languages lack the basic substitution properties on which a smooth running proof theory could be built.

Apart from the problems connected with their referential opacity, the other basic difficulty with existing programming languages is that they are very long–winded, in terms of the amount one has to write to achieve a given effect. Even a comparatively straightforward program like a compiler can easily run to ten thousand lines and there exist commercial packages up to a million lines long.

A number of studies carried out in industry have shown that a given programmer tends to produce a relatively fixed number of lines of code per year — typically around 1500 lines of debugged and documented code — and while the number of lines varies quite a lot from programmer to programmer, it is for a given individual *largely independent* of the language in which he is working — for example, it doesn't seem to matter whether it is assembly code or PL/1.

The significance of this result is that it means that the most important single variable in determining software production costs, apart from the quality of the programmers, is the level of language at which they are working. The reason why FORTRAN was such an enormous step forward, for example, is that programs written in FORTRAN are from five to ten times shorter than the equivalent assembly code. Other things being equal then, the FORTRAN programmer is from five to ten times more productive than the assembly code programmer.

Our problem today is that in the twenty five years that have elapsed since the invention of FORTRAN, we have failed to produce any further substantial improvement in this basic ratio of expressive power. If you compare a program written in a "modern" imperative language, such as PASCAL or ADA with its FORTRAN equivalent, you will not find it very much shorter — in fact, it might even be longer because of the extra declaratory information necessitated by the current fashion for very restrictive forms of strong typing.

It is becoming clear that in order to solve the software crisis, we have to find a way to move up to a whole new level of language that will be more expressive than our conventional high level languages by about the same ratio as our conventional languages were better than assembly code. That sort of increase in power can only arise by letting go of a level of detail that our present programming languages force us to express — which seems to imply a move to some kind of non-procedural language.

In fact, our experience to date with applicative programming is very promising in this respect. Programs written in languages like SASL (Turner 1976, 1981) are consistently an order of magnitude shorter than the equivalent programs in a conventional high level language. We will see some examples of programs in this style below, but at this stage we can give a general reason why the change from an imperative language to a descriptive one should lead to programs becoming so much shorter.

Expressed at an appropriate level of abstraction (for example as a dataflow graph—see Treleaven (1979)), an algorithm is a partially ordered set of computations, the partial ordering being imposed by data dependencies. In order to execute an algorithm on a Von–Neumann computer, however, we have to convert this to a total ordering, in one of the many possible ways, and organise storage for the intermediate results. In an imperative programming language both of these tasks must be carried out by the programmer with the result that he has to specify a great deal of extra information. A second reason why conventional high level languages are so long winded is that they lack certain necessary abstraction tools, in particular higher order functions, of which we shall say more below.

In summary, a good case can be made out for saying that the fundamental cause of the software crisis is the imperative and machine oriented nature of our programming languages, and that to overcome it we have to abandon the use of side effects and programmer control of sequencing in favour of purely functional notation. The theoretical possibility of programming in a purely functional style has been known for two decades — the obstacle to its use in practice has always been the difficulty of achieving acceptable efficiency in the use of existing hardware while using such techniques. The current rebirth of interest in functional programming is largely triggered by the fact that developments are now taking place on the hardware side which seem likely to overturn this situation.

The Development of VLSI and the Challenge of Parallelism

The basic design of the computer was laid down by John Von Neumann in the 1940s and has remained largely unaltered since (see Fig. 1). There is a single active processor and a large passive store — the connection between the processor and the store is relatively narrow, only one word in the store can be accessed at a time. Initially, and for a long time afterwards, the processor and the store were made of two fundamentally different technologies, and the processor was very much the more expensive of the two.

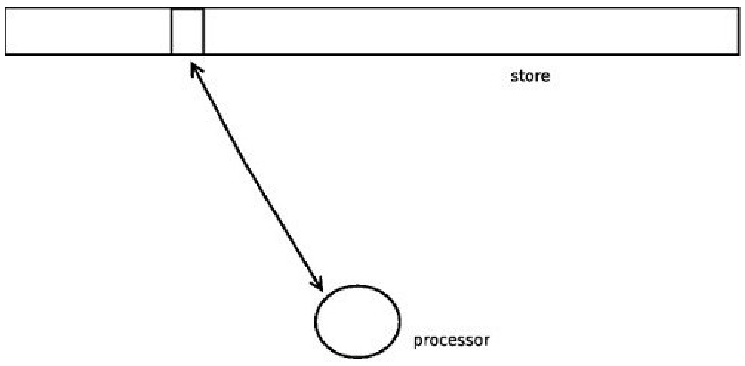


Fig. 1. the Von Neumann computer

The rationality of this arrangement is now being fatally undermined by the development of VLSI. First of all, note that processor and memory are now built of the same technology, namely VLSI chips. Moreover, processing power is becoming very cheap indeed. There is no longer any compelling reason for building mono–processor architectures — it would make equally good economic sense to build a machine which had a network of many processors.

Secondly, we have an obvious motive for doing so — to obtain increased performance. The speed of operation of a conventional Von Neumann computer is limited basically by the bandwidth of the connection between the processor and memory — Backus has called this “the Von Neumann bottleneck”. In order to make such a computer go faster we have to improve the technology out of which the components are built, and there are obvious limits to this process. In a multiprocessor architecture, by contrast, we can obtain arbitrary increases in speed simply by adding more processors to the network — provided of course, and this is crucial, that we can find ways of programming it that exploit the potential concurrency.

The first step towards using the possibilities for parallelism opened up by VLSI has been taken by the development of the various “array processors” now appearing on the market — for example, the ICL “DAP”. These capture a particular type of parallelism, which can be called “lockstep parallelism” in which the same instruction is performed simultaneously on a large number of data items.

This is appropriate only to certain specialised applications. A more general type of parallelism is where we have many processors each executing different instructions. A number of architectures of this general type are now under development, of which the best known are the various kinds of dataflow computer—see for example (Dennis 1979).

The potential performance of this type of architecture is enormous (thousands of megaflops, using current technology) but how can they be programmed? An idea that can be dismissed more or less straight away is that we should take some conventional sequential language and add facilities for explicitly creating and co–ordinating processes — the tasking facilities of ADA are an example of this approach. This may work where the number of processes is small, but when we are talking about thousands and thousands of independent processes, this cannot possibly be under the conscious control of the programmer.

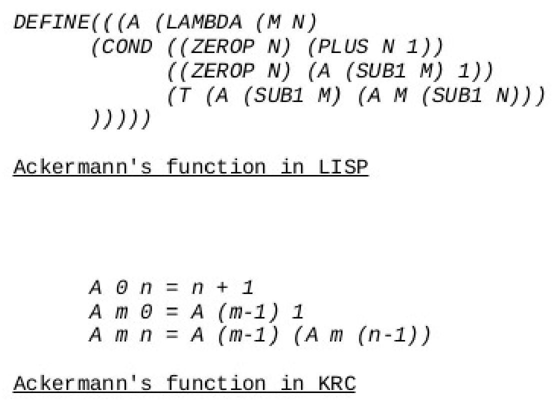
Parallelism on this scale can only arise from some basic asynchronousness of the language being used. Workers in dataflow are converging on the use of functional languages as a solution to this problem—see Ackermann & Dennis (1979), Arvind, Gostelow & Plouffe (1978). Paradoxically then, notwithstanding their former reputation for inefficiency, it is precisely the need for higher performance that may ultimately force the adoption of functional languages.

Incidentally, the historical efficiency disadvantage of functional languages arises partly from the fact that they have been running on machines with inappropriate instruction sets. There are simple theoretical arguments which show that given an appropriately designed instruction set there is no reason in principle why functional programs should be less efficient than the corresponding imperative programs, even on a Von Neumann machine. It is therefore welcome that in addition to the work currently being done with parallel architectures, some recent efforts have been directed towards the development of sequential machines specifically adopted to functional languages (Clarke et al. 1980, Holloway et al. 1980).

A Simple Language Based on Higher Order Recursion Equations

For the sake of having a definite syntax to work with, I will give the ensuing examples of functional programming in the notation of KRC ("Kent Recursive Calculator") a system I have implemented at the University of Kent and which I have been using for teaching purposes. It is fairly closely based on the earlier language SASL (Turner 1976) which I developed while working at the University of St Andrews in the period 1972–1976, but I have added a new language feature based on Zermelo–Frankel set abstraction.

Perhaps I should explain why I don't teach my students LISP (McCarthy et al. 1962) which is still the language most people first think of when functional programming is mentioned. There are two reasons — the first is that the syntax of LISP is so clumsy that it constitutes a real obstacle to comprehension. Fig. 2 illustrates this with a definition of Ackermann's function in LISP — note that



```
DEFINE(((A (LAMBDA (M N)
      (COND ((ZEROP N) (PLUS N 1))
            ((ZEROP N) (A (SUB1 M) 1))
            (T (A (SUB1 M) (A M (SUB1 N)))
      )))))
```

Ackermann's function in LISP

```
      A 0 n = n + 1
      A m 0 = A (m-1) 1
      A m n = A (m-1) (A m (n-1))
```

Ackermann's function in KRC

Fig. 2. Ackermann's function in LISP and KRC

there are eighteen pairs of parentheses! — and for contrast a definition of the same function in KRC, which looks more like a piece of ordinary mathematics. The second and more serious reason is that the semantics of LISP are rather complicated and include a number of features which could in no sense be regarded as functional. The nett effect, at least in my experience, is that as a vehicle for teaching people about functional programming, LISP is apt to cause more confusion than enlightenment and I prefer to avoid using it.

KRC is purely a functional language — there are no side effects and no concept of flow of control. A program in KRC (actually, we call it a "script") is a set of equations giving mathematical definitions of various entities in which the user is interested. For example, a simple script might be

```
r = u / v
u = x + y
v = x - y
x = 23
y = 10
```

The order in which the above equations are listed is of no significance — we have shown them in alphabetical order but that is purely for clerical convenience. The KRC system is interactive and includes built in commands for editing scripts, saving them in and retrieving them from, files and so on. In particular, the user can ask to have expressions evaluated in the environment established by the script. So for example, typing

```
r?
```

here causes the value of r to be printed at the terminal.

The only ordering of calculations established by a KRC script is that implied by the data dependencies — so for example, in the above case u and v must be calculated before r, but that is the only constraint — note in particular that u and v could be calculated in parallel.

The types of object in KRC's universe of discourse are — numbers, strings, written e.g. `"pig"`, lists and functions. Numbers and strings have the sorts of properties one would expect with the usual sorts of operators defined on them. Lists are written using square brackets and commas, thus

```
days = ["mon", "tue", "wed", "thu", "fri", "sat", "sun"]
```

elements of a list are accessed by indexing[1]. So for example, the expression

```
days 0
```

would here have the value `"mon"`. The operator `#` takes the length of a list, so

```
# days
```

is here 7. Another important operator on lists is "`:`" which adds a new element at the front, corresponding to the LISP function "CONS". So for example, the expression

```
0 : [1, 2, 3]
```

takes the value [0, 1, 2, 3]. The elements of a list can be of any type — enabling us to use lists of lists to represent matrices for example — and can also be of mixed type, enabling us to represent trees, etc.

Lists can be concatenated using an infix "`++`" operator and there is also a list difference operator, written "`--`". So for instance

```
[1, 2, 3, 4, 5] -- [1, 3, 5]
```

has the value `[2, 4]`.

Finally, a useful piece of shorthand is the ".." notation, allowing, for example,

```
[1..100]
```

as a notation for the list of integers from 1 through 100. An interesting property of the implementation in this latter case, by the way, is that this list does not immediately occupy 100 words of store, but only about 3 — enough to store a formula for calculating the elements when they are accessed.

This is part of a general strategy called "lazy evaluation" (Henderson & Morris 1976, Turner 1976) whereby the KRC system consistently avoids performing any calculation until it becomes necessary. Perhaps the most important consequence of this is that it permits the system to accept definitions involving infinite data structures as well as finite ones. For example, the equation

[1] KRC originally indexed lists starting at 1 rather than 0 but I have changed the examples in the paper to the modern convention of indexing from 0, which is the behaviour of Unix KRC. The only example affected is the eight queens problem.

```
    x = 2 : x
```

defines x to be the infinite list all of whose elements are 2, and we also permit the form, e.g.

```
    [1..]
```

meaning the list of all the natural numbers starting at 1.

Notice by the way that in the applicative style appropriate to a language like KRC, the use of explicit lists of values replaces the use of loops in an imperative language. For example, suppose we wanted to calculate the sum of the numbers from 1 to 1000. We would write

```
    sum [1..1000]?
```

in which we first set up the list of values in which we are interested and then apply the library function for summing a list. Lazy evaluation enables us to set up intermediate data structures in this way without incurring a space penalty.

The fourth and final type of object in KRC's universe of discourse is the function. Functions are defined by including in the script one or more equations with the name of the function followed by some formal parameters on the left and an expression describing the corresponding value in the right. For example the factorial function could be defined by the following equation (`product` is a library function)

```
    factorial n = product [1..n]
```

Sometimes there are several possible right hand sides — we can show them differentiated by "guards" (a guard is Boolean expression written on the far right of the equation, after a comma). Consider for example the following definition of a function for calculating greatest common divisor by Euclid's algorithm

```
    gcd a b = a,            a==b
            = gcd (a-b) b, a>b
            = gcd a (b-a), b>a
```

An alternative to the use of guards on the right is the use of pattern matching on the left, in which one or more of the formal parameters are replaced by constants — an example of this is shown by the definition of Ackermann's function given in figure 3 above.

A more sophisticated type of pattern matching involves the use of list structures in formal parameter positions, for example in the following definition of a function which takes a pair of 2–lists representing complex numbers and returns their complex product

```
    mult [a,b] [c,d] = [a*c - b*d, a*d + b*c]
```

The operator "`:`" is also allowed in pattern matching, as in this definition of the library function `sum`

```
sum [] = 0
sum (a:x) = a + sum x
```

Here [] represents the empty list and in the second equation, the formal parameter matches any non–empty list, whose first member corresponds to a and whose rest or "tail" to x.

Notice by the way that because of the absence of side effects, KRC functions are purely static in nature — applied to the same argument, a given function always gives the same answer. Note also that functions are "first class citizens" — they can be made elements of lists, passed as parameters and returned as results.

Partial Application and Higher Order Functions

A particularly powerful kind of abstraction is a higher order function — that is, a function that returns another function as a result. In KRC, these are made available through the following very simple mechanism. If a function is to defined to have, say, n arguments, it can always be applied to less than n arguments, say m, and the result is a function of $(n - m)$ arguments in which the first m arguments have been "frozen in".

So for example, if we define

```
twice f x = f (f x)
sq x = x * x
f = twice sq
```

what is the effect of the function f? Answer — it takes the fourth power; twice is here being used as a higher order function.

Higher order functions can be used to bring about an extremely compressed programming style. Consider, for example, the definition of sum given earlier. This represents an extremely common pattern of recursion for "folding" a list using a given binary operator, in this case "+", and a given start value, in this case "0". We can capture the general pattern in the following definition of a 3 argument function, fold[2]

```
fold op s [] = s
fold op s (a:x) = op a (fold op s x)
```

We can now define sum in a single line by partially applying fold

```
sum = fold '+' 0
```

The advantage of doing things this way is that a great number of analogous functions become available without any further effort. For example

```
product = fold '*' 1
```

Note the use of single quotes, e.g. '+', to denote an infix operator as a function.

[2] This function would now be called *foldr*; *foldl* and *foldr* with their now familiar definitions first appeared in the 1983 release of SASL.

The Use of ZF Expressions[3]

It is extremely useful to be able to quantify over a collection of objects without having to explicitly recurse down them. To this end, KRC includes a facility based on Zermelo–Frankel set abstraction. The basic idea of this kind of set abstraction is that one is allowed to write

$$\{f x \mid x \in S\}$$

meaning "the set of all $f\ x$ such that x is a member of S". Notice that x is a local variable of the above construction. We make this into a KRC language feature by working with lists, rather than sets, and using "`;`" for *such that* and "`<-`" for the membership sign. The variable binding construct "`var<-list`" is called a "generator". So for example

```
{x * x; x<-[1..100]}
```

is a list of the squares of the first hundred numbers[4]. We further extend the power of the notation by allowing more than one generator, separated by semicolons, and also one or more "filters"—these are logical expressions restricting the values which can appear in the list.

So we can define the Cartesian product of two lists (a list of all pairs formed with a member from each) as follows

```
cp x y = {[a,b]; a<-x; b<-y}
```

and a list of all Pythagorean triangles with sides less than 30 can be written

```
{[a,b,c]; a,b,c<-[1..30]; a*a+b*b==c*c}
```

notice that in an obvious piece of shorthand, we allow more than one variable to be bound by the same generator.

ZF expressions are particularly powerful when combined with recursion, as in the following definition of a function for generating all the permutations of a given list

```
perms [] = [[]]
perms x = {a:p; a<-x; p<-perms (x -- [a])}
```

[3] These are now called "list comprehensions", a term coined by Phil Wadler in 1985. The ZF expressions of KRC differ from modern list comprehensions in one significant respect — outputs of multiple generators are interleaved to ensure that all combinations are reached, even with two or more infinite generators.

[4] The original motive for choosing ";" rather "|" as the *such that* sign was to avoid confusion with KRC's use of "|" to mean logical "or". However, if *such that* is immediately followed by a generator it can safely be written "|", which in the context cannot meaningfully be interpreted as "or". All the ZF expressions in this paper can be written with either "|" or ";" as the initial separator. The only situation where *such that* cannot be written "|" is when followed by a *filter* rather than a generator. For example in $\{a;b \mid c\}$ which evaluates to $[a]$ or $[]$ depending on the value of the logical expression following the semicolon.

Generators come into scope from left to right, so later generators can involve earlier ones, but not vice versa.

This more or less completes our survey of KRC (it is not a very large language) and we now turn to some programming examples.

Some Programming Examples

Primes by the Method of Eratosthenses

Our first example is generating prime numbers by the sieve of Eratosthenes (“%” is the remainder operator)

```
primes = sieve [2..]
sieve (p:x) = p : sieve {n; n<-x; n%p>0}
```

Given the above as the script the command

```
primes?
```

will cause the KRC system to print prime numbers indefinitely (or rather until it runs out of space) producing the output

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31 ... etc
```

It is instructive to compare this definition of the sieve of Eratosthenes using infinite lists, with a more operational one involving explicit notions of co–operating process—see (Kuo, Linck & Saadat 1978) for a solution in Hoare’s CSP notation.

The Digits of *e*

Another example which demonstrates nicely the convenience of being able to work with infinite lists is the following program for printing the digits of e, for the idea of which I am grateful to E. W. Dijkstra. We know that e can be defined, writing $i!$ for $factorial(i)$

$$e = \sum_{i=0}^{\infty} 1/i!$$

Now we can choose to represent fractional numbers using a peculiar base system, in which the weight of the i’th digit is 1/i! — so note that the “carry factor” from the i’th digit back to the (i-1)’th is i. Written to this funny base e is just

$$2.1111111\ldots$$

the problem now being to convert from the funny base to decimal. The general algorithm for converting from any base to decimal can be stated in words as follows.

First print the integer part of the number, then take the remaining digits, multiply them all by 10 and renormalize using the appropriate carry factors

— the new integer part will be the next decimal digit and this process can be repeated indefinitely. We can capture this in a purely functional way by representing the number before and after conversion as two infinite lists, and defining a function `convert` recursively, thus

```
    e = convert (2 : ones)
    ones = 1 : ones
    convert (d:x) = d : convert (normalise 2 (0 : mult x))
    mult x = {10 * a; a<-x}
    normalise c (d:x) = carry c (d : normalise (c+1) x)
    carry c (d:e:x) = d+e/c : e%c : x
```

The above will not quite do, however — if we try to print `e`, we get the first digit `2`, followed by a long silence. The problem is that the recursion for `normalise` is not well founded — it tries to look infinitely far to the right before producing the first digit. We need some result which limits the distance from which a carry can propagate, or else we are stuck.

The necessary cut–off rule is provided by the observation that in the above conversion, the maximum possible carry from a digit to its immediately leftward neighbour is 9 (we leave the proof of this as an exercise for the reader). This leads us to rewrite the definition of `normalise` in the following more cautious form

```
    normalise c (d:e:x) = d : normalise (c+1) (e : x), e+9 < c
                        = carry c (d : normalise (c+1)(e : x))
```

This simple modification is all that is required. If we now issue the command

```
    e!
```

the system responds by printing the digits of e indefinitely (subject to space limitations)

```
    2.7182818284590 ...
```

Note on KRC Printing Conventions The use of "!" rather than "?" causes lists to be printed unformatted, that is without surrounding square brackets or commas between the items—likewise recursively for sublists. This is useful in case the user wants to organise his own formatting by including layout characters at various points in the data structure being printed.

The reader should study the function `show` defined in the KRC Prelude[5]—the instruction "`x?`" is actually equivalent to "`show x : ["\n"]!`".

[5] The Prelude and other information about KRC including downloads can be found at *http://krc-lang.org*.

The Eight Queens Problem

For our final example of functional programming, we shall take the well known eight queens problem, which is representative of a large class of problems which, at least when programmed imperatively, seem to require the use of backtracking.

We have to find a way of placing eight queens on a chess board so that no queen is in check from any other. Queens can give check vertically, horizontally or diagonally (in two ways). A moment's reflection tells us that in any solution, there must be exactly one queen in each column. So an obvious way to proceed is to start with an empty board and proceed from left to right , say, placing one queen in each column, always putting the new queen in a position where it cannot be checked by those already there — and if there is no such position, we have boxed ourselves into a blind alley. A reasonable representation of a board is as a list of integers, giving the row numbers of the queens so far placed on it. So for example

```
[2,5,3]
```

represents a board with queens in the first three columns at the positions shown. So the empty board is represented just by the empty list, `[]`.

We proceed by defining a function `queens n` that returns a list of all the solutions to the "*n* queens" problem — that is the problem of placing *n* queens on an *n* by 8 board

```
queens 0 = [[]]
queens n = {q : b; q<-[1..8]; b<-queens (n-1); safe q b}

safe q b = and {\checks q b i; i<-[0..#b-1]}
checks q b i = q == b i | abs (q - b i) == i + 1
```

This is everything that is needed, to print the solutions, we can say

```
layn (queens 8)!
```

which will cause them to be printed one per line, with numbered lines (`abs`, `and`, `layn` are defined in the KRC prelude).

Notice by the way that if we decide to print only the first solution, e.g. by saying

```
hd (queens 8)?
```

then because of lazy evaluation, the other solutions do not even get generated. So we would still program in the above way, even if we only wanted one solution. The key abstraction that enables us to get rid of the whole problem of backtracking (here and in all similar cases) is to think in terms of a function that returns all the solutions at a given level, instead of only one of them.

Acknowledgements This paper was written in October 1981 and published in Darlington, Henderson and Turner (1982). I am grateful to Cambridge University Press for permission to reproduce it[6]. A version also appears in *A List of Successes that Can Change the World* eds. Lindley, McBride, Sanella and Trinder, Springer LNCS 9600 (2016), a Festschrift in honour of Phil Wadler on his 60th birthday.

The paper as originally published included as an Appendix the KRC Prelude. Both the current Prelude and a historical one dating from 1981 can be found at *http://krc-lang.org*, from where a working implementation of Kent Recursive Calculator can be downloaded—I am grateful to Martin Guy for translating my code from an obsolete dialect of BCPL to C and getting it working under Unix.

References

Ackermann, W.B., Dennis, J.B.: VAL – preliminary reference manual. MIT Laboratory for Computer Science (June 1979)

Arvind, Gostelow, K.P., Plouffe, W.: An Asynchronous Programming Language and Computing Machine. University of California at Irvine (December 1978)

Backus, J.: Can Programming be liberated from the Von Neumann style: A functional style and its algebra of programs. CACM 21(8):613–641 (August 1978)

Clarke, J.W., Gladstone, P.J.S., Maclean, C.D., Norman A.C.: SKIM – S, K, I reduction machine. Proceedings LISP conference, Stanford (1980)

Darlington, J., Henderson, P., Turner, D.A. (eds): Functional Programming and its Applications Cambridge University Press (1982)

Dennis, J.B.: The varieties of Data Flow Computers. MIT Computation Structures Group, Memo 183 (August 1979)

Floyd, R.W.: Assigning meanings to programs. Proc Amer Math Soc Symposia in applied mathematics, 19:19–31 (1967)

Henderson, P., Morris, J.M.: A lazy evaluator. Proceedings 3rd POPL symposium, Atlanta, Georgia (1976)

Hoare, C.A.R.: An axiomatic basis for Computer Programming. CACM 12(10):567–583 (October 1969)

Holloway, J., Steele, G., Sussman, G.J., Bell, A.: The Scheme 79 Chip. Proceedings LISP conference, Stanford (1980)

Kuo, S.S., Linck, M.H., Saadat, S.: A guide to CSP. Oxford University Programming Research Group, Technical Monograph PRG–14 (August 1978)

McCarthy, J., et al.: LISP 1.5 Programmers Manual. MIT Press (1962)

Quine, W.V.O: Word and Object. MIT Press, Cambridge, Mass. (1960)

Russell, B., Whitehead, A.N.: Principia Mathematica. Cambridge University Press (1925)

Treleaven, P.C.: Exploiting Program Concurrency in Computing Systems. IEEE "Computer", 42–50 (January 1979)

Turner, D.A.: SASL Language Manual. St Andrews University Department of Computational Science Technical Report (1976)

Turner, D.A.: Aspects of the Implementation of Programming Languages. Oxford University D. Phil. Thesis (1981)

Turner, D.A.: Recursion Equations as a Programming Language. pp 1–28 of Darlington, Henderson and Turner (1982)

[6] This version last revised 2016.3.13