Palo Alto Research Center

Real Programming in Functional Languages

by James H. Morris

XEROX

Real Programming in Functional Languages

James H. Morris

CSL-81-11 July 1,1981

Abstract: The established properties of functional languages—easy to define semantics and mathematical elegance---are appealing to meta-programmers who study programming and programs at one remove. Most people believe that functional programming is inappropriate for real programmers who write programs that are judged on their behavior rather than their appearance. We shall explore this question by considering experience with two languages, Poplar and Euclid, that have a claim to being functional languages and to being used on real problems—string processing and system programming, respectively.

CR Categories: 4.2

Key words and phrases: Functional, Applicative, Languages

XEROX
PALO ALTO RESEARCH CENTER
3333 Coyote Hill Road / Palo Alto / California 94304

Introduction

The established properties of functional languages—easily defined semantics and mathematical elegance—are appealing to meta-programmers who study programming and programs at one remove. [Backus, Burge, Landin, McCarthy, Scott&Strachey] Real programmers, who write programs that are judged on their behavior rather than their appearance, believe that functional languages are not useful because they are inefficient and unnatural.

Functional languages as implemented make less efficient use of conventional machine resources than other languages. A programmer using a functional language has less control over the resources of the machine and therefore cannot squeeze out the best performance. This is also true of some non-functional languages. There are theoretical arguments based upon clever evaluation strategies, compilers, and multi-processor machine architectures that suggest that functional programs needn't be inefficient, indeed, can be more efficient; but nothing of a concrete nature has been shown. [Henderson&Morris, Friedman&Wise, Darlington&Burstall, Dennis] The major problem to be overcome is the cost of garbage collection. A conventional programmer has the advantage that he can overwrite storage rather than re-cycle it through a garbage collector. Unless the cost of collection can be made negligible, or eliminated by compile-time analysis, functional programming will always be relatively inefficient. Given all the theoretical work, much less than one would expect has been done in the area of clever implementations for any language. The challenge here is to produce an implementation of a functional language that makes better use of digital hardware than an average programmer using conventional methods can. It is the same sort of challenge FORTRAN met successfully many years ago.

Functional languages *are* unnatural to use; but so are knives and forks, diplomatic protocols, double-entry bookkeeping, and a host of other things modern civilization has found useful. Any discipline is unnatural, in that it takes a while to master and can break down in extreme situations. That is no reason to reject a particular discipline. The important question is whether functional programming is unnatural the way Haiku is unnatural or the way Karate is unnatural. Haiku is a rigid form of poetry in which each poem must have precisely three lines and seventeen syllables. As with poetry, writing a purely functional program often gives one a feeling of great esthetic pleasure. It is often very enlightening to read or write such a program. These are undoubted benefits, but real programmers are more results-oriented and are not interested in laboring over a program that already works. They will not accept a language discipline unless it can be used to write programs to solve problems the first time—just as Karate is occasionally used to deal with real problems as they present themselves. A person who has learned the discipline of Karate finds it directly applicable even in bar-room brawls where no one else know Karate. Can the same be said of the functional programmer in today's computing environments? No.

How could functional languages be made more convenient to use? If one takes a sufficiently elevated view, the semantics of functional programming languages are pretty much the same: virtually everything is a function that maps values in a heterogeneous data space into each other. The real differences among languages, and the part which should occupy more of each designer's attention is in their syntax. Far from being "mere" syntactic sugar, the way one writes and reads a language will tend to influence usage more than the underlying semantics. For example, consider the Strachey/Landin **let** statement. It is an inspired stroke of syntax design which showed us how to get the convenience of the assignment statement in a functional context. There are other features of conventional programming languages that have stood the test of time, but haven't been fully incorporated into functional languages: statement sequences, for-loops, and record structures.

We shall consider two languages, Poplar [Morris, *et al.*] and Euclid [Lampson, *et al.*], that have a claim to being functional languages and to being used on real problems—string processing and system programming, respectively. Poplar contains a purely functional sublanguage that is adequate for performing a large variety of string and list-processing applications. Its implementation is woefully inefficient, so its range of applications is limited. Euclid, on the other hand, is nearly a subset of Pascal and has been used to write compilers and operationing systems. The claim that it is a functional language, however, comes as a surprise to its designers and raises the question of what it means to be a functional language. These two languages represent quite diverse approaches to the question of syntax design for functional languages. Poplar introduced several new syntactic forms to make an obviously functional language somewhat easier to use. Euclid retains the basic Pascal syntax and leaves out features that prevent the language from being functional.

Poplar

Poplar is an experimental language for text and list manipulation. It has been used for testing some ideas about extending the powers of interactive text editors. We designed Poplar to encourage functional programming and tried to use it in that spirit. A recipe for it might read: start with pure LISP, replace atoms with decomposable strings, add SNOBOL4 pattern matching, build in implicit iteration over lists, sprinkle with untried ideas, add powerful primitives like sorting, fold into an APLish, post-fix syntax, and bake until half done.

Poplar was designed and implemented in 1978/9. It has received moderate use: there have been a few hundred pages of program written by about twenty professional programmers and computer scientists. They had a good display-oriented text editor, but no text-oriented language like SNOBOL4 [Griswold] or any UNIX facility like AWK [Aho] or LEX [Lesk]. It has received most of its use from people with a clerical task that is regular enough to be tedious, but not recurrent enough to justify a big programming effort in a more conventional language. A typical comment has been: "In a couple of hours I was able to learn Poplar and use it to solve a problem that would

have taken much longer otherwise." A few people wrote more serious programs: a report generation system for software projects, a family budget maintainer, a correspondence management system for academic journal editors, a purchase order management system. Large portions of some of these projects have been written functionally.

Simple Values

Poplar's value space is quite simple when described as a Scott domain:

V = Characters* + {fail} + V* + V → V

Strings (=Characters*) are written in quotes; e.g. "A string" and "". Concatenation of strings is denoted by juxtaposition:

"aaa" "bbb" = "aaabbb"

As in SNOBOL4, a number is simply a string of digits. The quotes can be omitted: "123" = 123. Addition and subtraction can be written as infix operations. In retrospect we made two mistakes here: The symmetrical double quotes and lack of an explicit concatenation operation made things hard to read.

The special primitive value *fail* plays the role normally played by Boolean values in Algol. Instead of providing **if-then-else** expressions we used three operations '>', '|', and '~', with the following definitions:

```
x > y  =  if x=fail then fail else y
x | y  =  if x=fail then y else x
~x  =  if x=fail then "" else fail
```

This allows one to write things like (BigExpression | u) rather than the more cumbersome

t ← BigExpression; **if** t=fail **then** u **else** t

The conventional **if** p **then** x **else** y could almost be achieved by (p > x | y), except when p was not *fail* but x was. In retrospect, this syntax was not a big improvement and caused some confusion.

Non-primitive values are either lists or functions. Lists are written like ["A", "list"] and []. Lists may be subscripted using the operator '/': ["A", "B"]/2 = "B". A negative subscript, -i, yields the list with its first i elements removed. ["A", "b", "c", "d"]/-2 = ["c", "d"]. Lists can be concatenated with the infix operator '„'.

["A", "b"] „ ["c", "d"] = ["A", "b", "c", "d"]

The familiar Cons(x, y) operation of LISP can be accomplished with the idiom '[x] „ y' which places x in a list of length one and concatenates it with y. We found this syntax for list construction and selection to be quite congenial and didn't miss the primitive functions cons, head and tail.

The distinction between strings and lists broke Poplar into two pieces like SNOBOL4: the pattern sub-language and the general list-processing language. The shortcomings of this became clear when someone wanted to precede a parsing operation by a lexical analysis that produced a list of strings. The pattern language could not be used on the list! We might have preferred a design like LISP70 [Tesler] in which the basc data typc is character, and a string is just a list all of whose elements are characters.

Functions and Iteration

People reading or writing a program tend to think of it *doing something* in a sequential fashion and find it most natural if the order in which they read a program fragment follows that sequence. In a conventional language the programmer usually thinks of the state of the machine as the object being worked upon by sequential statements. In a functional language, the machine state is not an appropriate concept, but that is no reason to discard the sequence as a syntactic construct. We chose to make function application a post-fix operation and found it very natural to write long sequences of monadic function applications just as if they were statements. This turned out to be especially true for conversational programming.

Functions are denoted by lambda expressions except that instead of 'λx.' one writes 'x:'. List-structured formal parameters require structured actual ones, so

```
([x, [y, z]]: x+y+z) = (w: w/1 + w/2/1 + w/2/2)
```

The application of functions to parameters is written in post-fix notation again using the operator '/'.

```
3+9 / (t: t t t) = 121212
```

The apparent double use of '/' as both function application and subscripting is explained by regarding integers as denoting functions applicable only to lists. The precedence of ':' is such that one can conveniently use function application as a sort of post-fix assignment statement.

```
L / x: x+x / t: t t t = L/(x: x+x/(t: t t t))
```

The let expression is a much better substitute for assignment and was sorely missed.

We wish we had had a kind of conditional function application based on *fail*:

```
x\f = if x=fail then fail else x/f
```

because we often found ourselves writing

```
BigExpression/x: x > x/f
```

One of Poplar's more successful syntactic features was the use of several infix operators as a substitute for for-loops. Without them the functional style requires the use of many recursive function definitions, one for every loop. String concatenation and the arithmetic operations extend to lists of strings so that

```
["a", "b", "c"]  "x"  =  ["ax", "bx", "cx"]

[1, 2, 3] + [5, 6, 7]  =  [6, 8, 10]
```

There are infix operators for LISP's Maplist, APL's reduction operator, APL's ι operator, and a general iterative operator. Like function application these three operators are written with the function second rather than first.

```
[a, b, c]//f = [a/f, b/f, c/f]                    (Maplist)

[a, b, c]///f = [[a,b]/f, c]/f                    (Reduce)

4 -- 7 = [4, 5, 6, 7]                             (Generate)

x%f = if x/f then (x/f)%f else x                  (Iterate)
```

It should come as no surprise that the maplist operator was very handy and that the reduce operator was good for adding lists of numbers and like tasks. Somewhat more surprisingly: the reduce operator was used extensively in situations where one wanted to traverse a list and carry information along. Consider the general for-loop scheme

```
S ← InitState;
for x ← List, x.tail until x=NIL
   do S ← F(S, x.head)
```

This could be written as

```
[InitState] „ List///F
```

A list of functions applied to a value generates a list of results.

```
x/[f,g,h] = [x/f, x/g, x/h]
```

This feature paid off most when the 'functions' were numbers as in

```
[a, b, c, d, e]/(2--4) = [b, c, d]
```

There are many built in functions. For example, a list of equal length lists may be transposed.

```
[[a, b, c], [d, e, f]]/Transpose = [[a, d], [b, e], [c, f]]
```

Transpose is important because it allows one to generalize a non-unary function, f, to work on lists via the idiom

```
[List1, List2]/Transpose//f
```

The combination of built-in iterators and post-fix notation was very successful; succinct functional programs to do complicated things could be written easily without using recursion. Furthermore, writing such programs became a simple, even natural process, rather than a challenge to the intellect.

Pattern Matching

Patterns are a subset of the function space, String → V, and have a special syntax. In essence, the pattern sub-language is the language of regular expressions. A primitive pattern is either a string or the *ellipsis* '...' which matches anything. Larger patterns may be constructed from smaller ones by using four combination rules: if P and Q are patterns, then so are the following

P Q	concatenation
P | Q	alternation (i.e. or)
P! = P | P P | P P P etc.	one or more repetitions
P? = (P | "")	optional

The Kleene star pattern P* can be written as P!?. Every pattern is enclosed in braces '{}'.

A pattern is a function which can be applied to a string; the result can be *fail* or something derived from the string by a set of pattern composition rules. As the default, the matcher simply re-concatenates the pieces matched so that

```
"aazbbcz"/{... "z" ... "z"} = "aazbbcz"
```

By decorating the pattern appropriately one can arrange for different things to happen: Suffixing a component with * causes whatever it matches to be discarded.

```
"aazbbcz"/{... ("z"*) ... ("z"*)} = "aabbc"
```

One can replace pieces by suffixing the phrase '> newpiece'

```
"aazbbcz"/{... ("z" > "X") ... "z"} = "aaXbbcz"
```

One can make lists out of the pieces by inserting brackets and commas in the pattern

```
"aazbbcz"/{ [... "z" , ... "z"] } = ["aaz", "bbcz"]
```

Conceptually, it is best to think of a two-phase process: first the string is parsed, then one computes the result from the parse tree using the various signals attached to the pattern. Although it can be syntactically confusing to intertwine these two processes, it overcomes the fact that any division of the two phases can lead to them becoming inconsistent.

The operator ',!' parses things just like '!' but produces a list of the items matched rather than re-concatenating them. For example to apply the function F to each substring of s found before a 'z' one says

```
s/{(..."z"*),!}//F
```

A very general method for processing the outcome of a pattern match is to attach a function to a pattern element and apply it to the result of matching that element. One says

```
{(P / F)!}
```

and the result of a successful match is computed by applying F to each of the sub-strings which matches P and concatenating the results. This method is applicable in more general cases typified by the recursive patterns. Without functional attachment, such patterns are not useful if one wants to process the recursive structure. For example, to parse an expression and compute its value one can write

```
E ← {digit! | [ "("* E , "+"* E ")"* ] / Plus}
```

which is succinct if nothing else. Functional attachment was used extensively to build powerful patterns that simultaneously matched and transformed their input.

Non-functional Features

Poplar has a conventional assignment statement 'x ← e' that changes the values of simple variables. The extent of its use depended upon the programmer and the degree to which he was trying to write functionally. Input-Output was imperative, but it was not possible to use it to simulate assignment because a file could not be read after it was written during a single session. In other words, files were accessed via pure input or pure output streams.

Equality Assertions

Reading a program is a little like listening to one end of a telephone conversation. When listening to someone talking on the phone one can figure out what is going on only if he has a

good model for what the unheard person might be saying. When reading a program one needs a model for the data that is being manipulated. Data declarations in conventional programming languages serve this function to a degree, but often they don't say nearly as much about the content of the data as one might like. To make Poplar, which has no declaration facility, more readable there is a checked comment facility. Any function definition can be decorated with a set of assertions that constitute a test evaluation of the function.

For example, given the function (x: [x,x]/Conc/Reverse) one can add equality assertions to produce

```
x:                  = "foo";
[x,x]/Conc          = "foofoo"
/Reverse            = "oofoof"
```

which says: If the input is "foo" the value of [x,x]/Conc will be "foofoo" and the final value will be "oofoof".

This idea has worked out well: it is much easier to grasp what a program is doing if a well-chosen example is interleaved with it. Poplar is very difficult to read without them. The fact that the example is machine-checked makes it more credible than a normal comment. In practice, one needs mechanical aids to generate examples because of all the details (e.g., how many spaces are in " "?) which escape the reader, but not the checker.

An Example

Consider the key-word-in-context problem discussed by [Parnas]: given the list of book titles

Green Sleeves

Time Was Lost

generate the following alphabetized list, useful for looking up specific key words:

<Green> Sleeves
Time Was <Lost>
Green <Sleeves>
<Time> Was Lost
Time <Was> Lost

The procedure is as follows:

Break the text up into lines.
Break each line up into words.
For each line:

Generate a list of pairs, one for each word, consisting of the word, and a reconstruction of the line with brackets around the word.

Merge all these lists into one big one.
Sort the list by the words.
Discard the words.
Concatenate all the lines to form the final text.

Figure 1 shows the Poplar program to do this, and Figure 2 shows the same program decorated with equality assertions. The major steps correspond to the informal steps above. The character '↵' stands for carriage-return. The function Lines is a pattern that breaks up the string a each carriage return. Words is a pattern that produces a list of words from its input discarding all punctuation. Append concatenates pairs of lists; Conc concatenates pairs of strings. The phrase '//2' applies 2 to each pair on the list.

```
KWIC ← (s:s/Lines
        //Words
        //Generate
        ///Append
        /Sort
        //2
        ///Conc);

Lines ← {(... "↵"*),!};

Words ← {(...* Letter!),! ...*};


Generate ← (w:1--(w/length)
            //k: 1--(w/length)
            //(j:  j/{k} > "<" (w/j) ">" |  w/j)
            /t: " " t///Concat "↵"
            /u: [w/k, u]);
```

Figure 1. A Poplar Program for Key-Word-in-Context

```
KWIC ← (s:          ="Green Sleeves↓Time Was Lost↓";
        s/Lines     = ["Green Sleeves", "Time Was Lost"]
        //Words     = [["Green", "Sleeves"], ["Time", "Was", "Lost"]]
        //Generate  =[[["Green", " <Green> Sleeves↓"],
                      ["Sleeves", " Green <Sleeves>↓"]],
                     [["Time", " <Time> Was Lost↓"],
                      ["Was", " Time <Was> Lost↓"],
                      ["Lost", " Time Was <Lost>↓"]]]
        ///Append   =[["Green", " <Green> Sleeves↓"],
                      ["Sleeves", " Green <Sleeves>↓"],
                      ["Time", " <Time> Was Lost↓"],
                      ["Was", " Time <Was> Lost↓"],
                      ["Lost", " Time Was <Lost>↓"]]
        /Sort       =[["Green", " <Green> Sleeves↓"],
                      ["Lost", " Time Was <Lost>↓"],
                      ["Sleeves", " Green <Sleeves>↓"],
                      ["Time", " <Time> Was Lost↓"],
                      ["Was", " Time <Was> Lost↓"]]
        //2         =[" <Green> Sleeves↓",
                      " Time Was <Lost>↓",
                      " Green <Sleeves>↓",
                      " <Time> Was Lost↓",
                      " Time <Was> Lost↓"]
        ///Conc     =" <Green> Sleeves
                      Time Was <Lost>
                      Green <Sleeves>
                      <Time> Was Lost
                      Time <Was> Lost
                     ");

Generate ← (w:                              = ["Time", "Was", "Lost"];
        1--(w/length)                       = [1, 2, 3]
        //(k:                               = 2;
          1--(w/length)
          //(j: j/{k} > "<" (w/j) ">" | w/j)  = [ "Time", "<Was>", "Lost"]
          /t: " " t///Concat "↓"            = " Time <Was> Lost↓"
          /u: [w/k, u]                      = [ "Was", " Time <Was> Lost↓"]
          )                                 = [["Time", " <Time> Was Lost↓"],
                                               ["Was", " Time <Was> Lost↓"],
                                               ["Lost", " Time Was<Lost>↓"]];
```

Figure 2. The KWIC Program Annotated with Equality Assertions

The function Generate could be described informally as follows:

> For each element, k, of the list w
> Generate a new list, t, identical to w except that the kth element of w has had brackets placed around it.
> Put a space in front of each element of t, and concatenate all the elements following with a ")", producing u.
> Return the pair consisting of the kth element of w and u.

Notice that the informal description of procedures KWIC and Generate consist of quite imperative statements while the program itself is entirely functional! This is the advantage of post-fix syntax. Many programs have been written in this style, often interactively. This program is rather inscrutable, but we believe that translating it to a more conventional notation makes it worse. In Figure 3 the program appears written in an Algol/LISP style of syntax, i.e., changed to a prefix notation with all the maplist and reduce operations explicit. To make the nesting tolerable, we introduced many assignment statements; imagine how the program would look if we eliminated them by back-substituting! Of course the assignment statements give one the opportunity to introduce a mnemonic identifier to describe the intermediate result. Thus the opaqueness of the program is as much due to the style of expression as the syntax of the language. We found than Poplar's syntax allows one to write extremely succinct programs. They are also quite unreadable. There seems to be a limit to how compact a notation should be and Poplar exceeded it.

Records

Poplar, and any other functional language to be used for real programming, should have record data types. Consider the following common sort of program:

```
w ← 0; L ← NIL;
for x ← List, x.tail until x=NIL
   do begin
      w ← w + x.head;
      if L= NIL or not (x.head = L.head) then L ← Cons(x, L)
      end
```

A Poplar equivalent, employing the reduce operator is

```
[[0, []]] „ List
///[[w, L], xhead]:
   [w+xhead,
    (L/isnull | ~(L/1/{xhead})) > [xhead] „ L | L ]
```

```
procedure KWIC(s);
        begin
        ListofLines ← Lines(s);
        ChoppedLines ← maplist(ListofLines, Words);
        ListofListsofPairs ← Generate(ChoppedLines);
        ListofPairs ← reduce(ListofListsofPairs, Append);
        OrderedList ← Sort(ListofPairs);
        ListofStrings ← maplist(OrderedList, λx.x/2);
        return reduce(ListofStrings, Conc)
        end

procedure Generate(w)
        begin
        return(
        maplist(GList(1,Length(w)),
                λk. t ← maplist(GList(1, Length(w)),
                                λj. if j=k then "<" (w//j) ">" else w/j);
                    u ← Concat(" ", t) "↵";
                    return Cons(w/k, u)))
        end
```

Figure 3. The KWIC Program Written in Algol/LISP

This kind of thing was hard to read and write. One problem was figuring out the correspondence between the various structures and the identifiers w, L and xhead. It was easy to get these structures wrong. Pascal's record construct would help this problem. For example, suppose we invent the record notation

```
<x ← E, y ← F>
```

to denote the record with fields x and y initialized to E and F. The above could have been written

```
[<w ← 0, L ← []>] ,, List
/// [r, Hx]:
        <w ← r.w+Hx,
        L ← (L/isnull | L/1/{Hx}) > [Hx],,L | L >
```

This notation would have helped a great deal in making patterns more readable. For example, one might write

```
Exp ← {< "("  leftOperand ← Exp "+" rightOperand ← Exp ")" > }
```

This would achieve some of the mnemonic value of the SNOBOL4 conditional assignment operation without introducing side effects.

Better iterative operators

Although the built-in iterators were successful in general, we now have a better idea of what they should be. The maplist operator had the feature that if a value in the output list was *fail* it was omitted. This was handy, but occasionally it tended to bury errors one would like to discover. There should be separate operators to accomplish this, perhaps

```
[l, p]/Filter  =  sheep
[l, p]/Split  =  [sheep, goats]
```

where sheep is the list of items on list l for which p is true (i.e., not *fail*) and goats is a list of all the others. The absence of a Split operator like this caused people to use assignment statements. They would write

```
goats ← [];
sheep ← l//(x: if x/p then x else (goats ← [x],, goats; fail))
```

A problem with the reduce operator was that it was clumsy to produce lists as answers. In those situations a reduce functional that worked in the opposite direction might have been much more congenial. For example

```
[x,y,z]\\\F  =  [x, [y, [z, []]/F]/F]/F
```

It processes the list from right to left and applies F to the last element and the empty list. For example, this would allow us easily to solve the otherwise bothersome problem of eliminating adjacent repetitions from a list.

```
[1,1,3,4,1,2,2]
\\\([x,y]:  ~y/isnull > y/1/{x} > y | [x],,y)
=  [1,3,4,1,2]
```

Notice that when x is a non-null list

```
x \\\ Cons  =  x
x \\\ ([h,t]:  [h/f],,  t)  =  x//f
```

This form of the reduce operator is very close to Strachey and Barron's *lit* (list iterate) functional:

```
lit a g x = if x=NIL then a else g (x.head) (lit a g(x.tail))
```

If one has the possibility of multi-processing then there should be a reduce operator suitable for use with associative functions that is free to do things in any order it likes.

The general iteration operator '%' was not very useful and suffered from the lack of a record notatation as discussed above. The need was felt for ways other than the sequence operator to generate lists from whole cloth. For example, the following function might be useful:

```
[a, f]/GenList = ~a > [] | [a] ,, ([a/f, f]/GenList)
```

Clever Implementation is essential

We have experimented with an implementation that uses the lazy evaluation strategy described in [Henderson&Morris], but most programs were written for a standard evaluator. In any case, we found that assuming a lazy evaluator can have a very liberating effect on how one programs. For example, the KWIC program is very inefficient by contemporary standards. Every line seems to create a large new structure which the following line consumes. Improvements in this algorithm's performance can be made by a little cleverness in the evaluation strategy so these multi-pass operations are merged. The essence of the technique is that nothing is evaluated until it absolutely must be. Under this regime lists often behave like streams because their tails remain unevaluated until they are needed. In the case of KWIC the first operator that forces any sort of evaluation is Sort which demands that it receive a list of lists, each of whose first components is a fully evaluated string. This causes the Append reduction to be completed, but the second component of each pair remains unevaluated until the final reduction using Conc. Thus, in principle, this program requires only enough space to create a list of all the individual words and does not require space proportional to its output, which approximates the square of the input. Of course, a run-time lazy evaluation strategy does not reduce the amount of garbage this program will generate, it only reduces the maximum amount of storage it requires at any one time. A deeper, compile-time analysis *a la* Darlington and Burstall would be necessary to eliminate the creation of temporary storage.

Notice that the revised definition of the reduce operator works much better under lazy evaluation. For example, the beginning of the value of L\\\\Append can emerge before L has been completely traversed.

Since lists are never fully evaluated one can even deal with infinite lists. The Fibonacci

numbers may be described by the recursively defined list Fib.

```
Fib ← [1,1] „ (Fib + (Fib/-1))
```

Suppose one want to find the first Fibonacci number that is divisible by 3. He can say [Fib,div3]/Filter/1. This will not involve computing any more elements than a more conventional recursive program would. In general, any **while** loop could be written in this way:

```
s ← a; while P(s) do s ← F(s)
```

can be simulated by

```
[[a, F]/GenList, P]/Filter/1
```

There were many situations in which lack of lazy evaluation encouraged programmers to use assignments. For example, one wrote a function GetParagraph that transforms a string into a pair consisting of the first paragraph in the string and the remainder of the string. He then wrote an expression

```
(1--N)//(x: File/GetParagraph/[p, f]: File ← f; p)
```

that produces the first N paragraphs of the string and leaves File holding what is left over. There are a variety of ways to do this without assignment. The most straightforward is to parse the entire file into a list of paragraphs once and for all and then grab N items. One would write

```
File/{GetParagraph,!}
/ParsedFile: [ParsedFile/(1--N), ParsedFile/-N]
```

to produce the two pieces. However, the file in question was very large and this program could not have worked if the parsed file were fully materialized by the non-lazy evaluator.

One of the things that has always been hard for functional programs is achieving the efficiencies associated with loop exits. If we are going to change loop constructs into maplist-like, operators how do we deal with premature exits from for-loops? E.g.,

```
for i in 1 .. N do
  if Q(i) then begin k ← i exit end
```

This translates easily into

```
k ← 1--N // (i: i/Q > i) / 1
```

which seems to say: map over the list 1--N producing all the values for which Q is true, then select the first. Under lazy evaluation, however, Q will not be applied any more times than in the for-loop program.

Euclid

Euclid was designed as a restriction of Pascal, leaving out all those things that made proofs difficult—most notably gotos and hidden side effects. These restrictions make Euclid easily translatable into a functional language—so easily that we argue that it is a functional language disguised in Pascal syntax. It omits the Pascal features of gotos, concealable side effects, and procedures as parameters. What remains is a highly constrained language, but one which, to the surprise of many, has been used to write large programs. A group of programmers have enthusiastically adopted Euclid and written a real compiler and a toy operating system in it. [Wortman, Holt]

Euclid's Restrictions

The arbitrary goto has been eliminated. This guarantees that every procedure returns to its caller.

Pointers are treated as indices of special kinds of arrays called *collections.* Collections are actually a resurrection of the *class* concept of an early version of Pascal. A collection C may be thought of as an array except that it has a special type of indices called pointers which admit to no arithmetic operations. These pointers can be obtained only by calling the procedure C.New; the array has no explicit bounds. The dereferencing operation 'p↑' should be thought of as an array subscription operation, C(p). The advantage of this for program verification is that each pointer assignment has its scope of influence explicitly limited. If one makes the declarations

```
type CT = collection of integer
type CU = collection of integer
type T = ↑CT
type U = ↑CU
t: T
u: U
```

then the theorem

$$\{u\uparrow = 5\}\ t\uparrow := 6\ \{u\uparrow = 5\}$$

is treated exactly like

$$\{CU(u)=5\}\ \ CT(t)\ :=\ 6\ \ \{CU(u)=5\}$$

which is obviously true, given that CT and CU are disjoint.

In Euclid procedures and functions cannot be passed as parameters. The most important proof rule for Euclid, or any other language, is the procedure call rule. The idea behind a procedure call rule is that one proves a general fact about a procedure in terms of its formal parameters and then uses the procedure call rule to particularize the general theorem. We shall not describe such a rule in detail but merely point out that to have such a rule one must be able to describe the effect of a call on any procedure at the position of the call. Sometimes this is hard to do in Pascal. Consider the program

```
begin procedure P;
        begin x:integer;
        procedure Q(var z: integer);
                begin x := z+x; z := x end;
        x:= 0; R(Q);
        end;
    procedure R (s: procedure(var integer));
        begin t: integer;
        t:= 1; {?} s(t); {?} s(t); {?} Print (t);
        end;
P()
end;
```

In this program, P calls R, passing its internal procedure Q which references a variable x that is inaccessable to R. The question then arises, how can we describe the effect of calling s within R? Clearly we cannot make any assertions about x inside R since x is at best inaccessible and at worst non-existent. Euclid's solution to this problem is to ban procedures as parameters. Then it is easy to see that any variables accessible to a procedure are also accessible at the point of its call.

Each procedure is required to list, in an *imports list*, all of the free identifiers it mentions and attach the **var** attribute to any it assigns to. Note that if a procedure dereferences a pointer, it must import its collection. The programmer is required only to list all the free identifiers of each procedure; the compiler uses these lists to produce expanded imports lists by adding *thus clauses* (which it prints in the output listing) that include all the identifiers implicitly imported. In other

```
type PuzzleSolver = module imports (writeln) thus (var output)

pervasive const   size =  343
pervasive const classMax =  4
pervasive const typeMax  =  13
pervasive type    pieceClass =    1..classMax
pervasive type    pieceType =     1..typeMax
pervasive type    position =  1..size

pervasive const   class:    array pieceType of pieceClass = [1,1,1,1,1,1,2,2,2,3,3,3,4]
pervasive const   pieceMax:   array pieceType of position
                                        = [11,149,71,23,53,155,3,15,99,9,51,57,58]
function Index(i,j,k: integer) returns integer =
  begin return(i+7*(j+7*k)+1)end

function InitP (x : integer) returns
     p: array pieceType, position of boolean =
begin
 procedure SetPiece(t: pieceType, I, J, K: position) =
     imports (var p)
     begin  for i in 0 .. I loop for j in 0 .. J loop for k in 0 .. K loop
                  p(t, Index(i,j,k)) := true end loop end loop end loop
     end
 for i in 1 .. typeMax loop for m in 1 .. size loop p(i, m) := false end loop end loop
 SetPiece(1, 3, 1, 0); SetPiece(2, 1, 0, 3); SetPiece(3, 0, 3, 1)
 SetPiece(4, 1, 3, 0); SetPiece(5, 3, 0, 1); SetPiece(6, 0, 1, 3)
 SetPiece(7, 2, 0, 0); SetPiece(8, 0, 2, 0); SetPiece(9, 0, 0, 2)
 SetPiece(10, 1, 1, 0); SetPiece(11, 1, 0, 1); SetPiece(12, 0, 1, 1)
 SetPiece(13, 1, 1, 1)
end

pervasive const   p:   array pieceType, position of boolean = InitP(0)
pervasive type    PC =    array pieceClass of 0..13
var pieceCount:  PC
pervasive type    PZ =array position of boolean
var puzzle:   PZ

function fit (i : pieceType, j : position) returns boolean =
imports (puzzle)
begin
 for k in 1 .. pieceMax(i) loop
     if p(i,k) then if puzzle(j+k-1) then return(false) end loop
 return(true)
end

procedure remove (i : pieceType, j : position) =
imports (var puzzle, var pieceCount)
begin
 for k in 1 .. pieceMax(i) loop
     if p(i,k) then puzzle(j+k-1) := false endloop
 pieceCount(class(i)) := pieceCount(class(i)) + 1
end
```

Figure 4. A Euclid Program

```
procedure place (i : pieceType, var j : position) =
imports(var puzzle, var pieceCount, writeln) thus (var output)
begin
 for k in 1 .. pieceMax(i) loop
      if p(i,k) then puzzle(j+k-1) := true end loop
 pieceCount(class(i)) := pieceCount(class(i)) - 1
 for k in j .. size loop
      if not puzzle(k) then begin j:=k; return end end loop
 writeln('puzzle filled')
 j := 1
end

procedure trial (j: position, var ans: boolean) =
imports(var kount, trial, pieceCount, fit, place, remove, writeln)
 thus (var puzzle, var pieceCount, var output)
begin
var k: position
 for i in 1 .. typeMax loop
      if pieceCount(class(i)) <> 0 then
           if fit (i, j) then
                begin
                     k := j
                     place (i, k)
                     trial(k, ans)
                     if ans or (k = 1) then
                          begin
                          writeln ('piece', i, ' at', k)
                          ans := true
                          kount := kount+1
                          return
                          end
                     else remove (i, j)
                end end loop
 ans := false
kount := kount + 1
end

var  m:  position
var  a:   boolean
 for m in 1 .. size loop puzzle(m) := true end loop
 for i in 1 .. 5 loop for j in 1 .. 5 loop for k in 1 .. 5 loop
      puzzle(Index(i,j,k)) := false end loop end loop end loop
 pieceCount(1) := 13
 pieceCount(2) := 3
 pieceCount(3) := 1
 pieceCount(4) := 1
 m := Index(1,1,1)
 kount := 0
 if fit(1, m) then place(1, m) else writeln('error 1')
 trial(m, a)
 if a then writeln('success in', kount, ' trials')
 else writeln('failure')

end.
```

Figure 4. (continued) A Euclid Program

words, the compiler computes a transitive closure of identifiers referenced and changed starting from those referenced and changed directly. The existence of these imports lists, however arrived at, is important for checking the following vital rule.

Any two identifiers known in a scope are guaranteed to be represented by disjoint storage. This rule is required to support Hoare's basic assignment axiom, {P(e)} x := e {P(x)}, which says that to prove that P(x) holds after "x:=e" has been executed on must show that P(e) holds before where P(e) is derived from P(x) by uniform substitution of e for x. This rule fails to hold in the following Pascal program:

```
begin z: integer;
procedure P (var x: integer);
    begin {z+1>z} x := z+1 {x>z} end;
z:=5; P(z);
end;
```

Euclid's solution to this is to disallow the overlapping of variables. This restriction is enforced by the following sort of rule: Consider the actual parameters of a procedure call together with its imports list, which would include *z*, as an extended actual parameter list. Consider each actual corresponding to a var parameter. If it is a simple identifier, field extractor, array, or collection, it cannot appear as any other actual. If it is an array or pointer dereference, the array or collection in question cannot be passed as one of the other parameters. Where the overlap is only potential, as in a call like P(a(i), a(j)), the language definition requires that a *legality assertion* to the effect that $i \neq j$ precede the call and that this assertion be verified at compile time or checked at run time.

There are also a number of ways in which Euclid extends Pascal, but we shall not deal with them here. None of these extensions seems to compromise the arguments we shall make, but they add complications.

Translating an example

To illustrate how close Euclid is to being a functional language we shall translate a typical program to a functional form. We have chosen the program *PuzzleSolver* written by Forest Baskett [Baskett] to be used as a benchmark for various systems. The original program is shown in Figure 4. We have included all the thus clauses in the imports lists. Note that pervasive constants such as the arrays *class*, *pieceMax*, and *p* need not be mentioned in imports lists since they cannot be assigned to after initialization.

First we shorten the imports clauses by promoting all the imported variables to be explicit

parameters. This requires that each call pass those parameters explicitly. Figure 5 shows the changes to two of the procedures, *place* and *trial.* It should be noted that this transformation would not always work in Pascal because, as we showed earlier, it is possible to pass procedures around in such a way that variables implicitly imported by a procedure cannot be accessed by the procedure's caller.

```
procedure place (i : pieceType, var j : position, var puzzle: PZ,
                 var pieceCount: PC, var output: Stream) =
imports(writeln)
begin
for k in 1 .. pieceMax(i) loop
    if p(i,k) then puzzle(j+k-1) := true end loop
pieceCount(class(i)) := pieceCount(class(i)) - 1
for k in j .. size loop
    if not puzzle(k) then begin j:=k; return end end loop
writeln(output, 'puzzle fil'ed')
j := 1
end

procedure trial (j: position, var ans: boolean, var kount: integer,
                 var puzzle: PZ, var pieceCount: PC, var output: Stream) =
imports(trial, fit, place, remove, writeln)
begin
var k: position
for i in 1 .. typeMax loop
    if pieceCount(class(i)) <> 0 then
        if fit (i, j, puzzle) then
            begin
                k := j
                place (i, k, puzzle, pieceCount, output)
                trial(k, ans, kount, puzzle, pieceCount, output)
                if ans or (k = 1) then
                    begin
                    writeln (output, 'piece', i, ' at', k)
                    ans := true
                    kount := kount+1
                    return
                    end
                else remove (i, j, puzzle, pieceCount)
            end end loop
ans := false
kount := kount + 1
end
```

Figure 5. Imports transformed into explict parameters

Next we eliminate all the var parameters by turning the procedure into a function and returning them as values. Each of the formal parameters *x* that was a var parameter is replaced by the parameter *oldx*; and upon entry to the procedure *oldx* is copied into the result identifier *x*. Figure 6 shows the change to *place and trial.* To make this convenient, we have had to pretend that Euclid allows multiple results to be returned.

```
function place (i : pieceType, oldj : position, oldpuzzle: PZ,
                    oldpieceCount: PC, oldoutput: Stream)
returns (j : position, puzzle: PZ, pieceCount: PC, output: Stream) =
imports(writeln)
begin j := oldj; puzzle := oldpuzzle; pieceCount: =oldpieceCount;
      output: = oldoutput;
for k in 1 .. pieceMax(i) loop
        if p(i,k) then puzzle(j+k-1) := true end loop
    pieceCount(class(i)) := pieceCount(class(i)) - 1
    for k in j .. size loop
        if not puzzle(k) then begin j: =k; return end end loop
    output := writeln(output, 'puzzle filled')
    j := 1
end

function trial (j: position, oldans: boolean, oldkount: integer, oldpuzzle: PZ,
                    oldpieceCount: PC, oldoutput: Stream)
returns (ans: boolean, kount: integer, puzzle: PZ,
                                pieceCount: PC, output: Stream) =
imports(trial, fit, place, remove, writeln)
begin var k: position
    ans: =oldans; kount: =oldkount; puzzle: =oldpuzzle;
    pieceCount: =oldpieceCount; output: =oldoutput;
    for i in 1 .. typeMax loop
        if pieceCount(class(i)) <> 0 then
            if fit(i, j, puzzle) then
                begin
                    k := j
                    (k, puzzle, pieceCount, output) :=
                                place (i, k, puzzle, pieceCount, output)
                    (ans, kount, puzzle, pieceCount, output) :=
                                trial(k, ans, kount, puzzle, pieceCount, output)
                    if ans or (k = 1) then
                        begin
                        output: = writeln (output, 'piece', i, ' at', k)
                        ans := true
                        kount := kount+1
                        return
                        end
                    else (puzzle, pieceCount)
                                := remove (i, j, puzzle, pieceCount)
                end end loop
    ans := false
kount := kount + 1
end
```

Figure 6. Procedures transformed into functions

Is this transformation always valid? It is clear that assignment statement above will have the same effect as the original procedure call (albeit at possibly greater expense) if the returned values are equal to what the values of the original var parameters were when the original procedure

returned. However, the computation inside *place* is different. Instead of updating the original variables we copy them and use only the copies inside the function. The non-overlap rule now come into play: since all the parameters of *place* were disjoint to begin with, copying them has no qualitative effect on the behavior of *place.* The only possibility left is that *place* calls another procedure or function that somehow accesses the original variables. For example, suppose *place* contained a call to a procedure PeekAtPuzzle(i) which imported puzzle and returned the ith element. But PeekAtPuzzle has also been subjected to this rewriting so that its access to variables has been made entirely explicit in its parameter list; i.e., its call would read PeekAtPuzzle(*i, puzzle*) so it must be working on the copy too. By induction on the depth of function calls, this call on PeekAtPuzzle has the same result as it did before. Finally, note the impossibility of a non-local goto that could terminate the execution of *place* and by-pass the assignment of the new results.

Once we have carried out the procedure to function transformation on the program it remains only to make each function by itself more obviously functional. This is basically a mopping up operation familiar to any student of functional languages. Figure 7 illustrates how one eliminates for loops and array assignments, assuming that arrays are represented as functions. The *lit* functional and Poplar list generator described earlier are employed.

```
function place (i : pieceType, oldj : position, oldpuzzle: PZ,
                oldpieceCount: PC, oldoutput: Stream)
returns (j : position, puzzle: PZ, pieceCount: PC, output: Stream) =
imports(writeln)
begin
  puzzle := (λn. p(i, n-oldj+1) or oldpuzzle(n))
  pieceCount := (λn. if n=class(i) then oldpieceCount(class(i)) - 1
                                   else oldPieceCount(n))
  j := lit 1 (λ(k,r). if not puzzle(k) then k else r) (oldj -- size)
  output := if j=1 then writeln(oldoutput, 'puzzle filled') else output;
end
```

Figure 7. For loops and array assignments eliminated

The important thing about this demonstration is that it shows that one of the major aspects of functional programming, the absence of hideable side effects, is also a property of Euclid. The Euclid programmer is not required to make all side effects explicit at the point of the procedure call, but they are explicit in the declaration in the imports clauses.

Euclid vs. Pascal

Why don't we argue that Pascal, too, is a functional language? After all, *any* language can be given functional semantics if one is willing to work hard enough. The difference between Euclid and Pascal can be captured in the types of data spaces one must deal with in order to provide functional semantics. To simplify the comparison let us ignore records and functions. To provide Scott/Strachey semantics for Euclid one would need the following sort of data spaces:

Basic = Boolean + Integer + Pointer

Array = Integer → Value

Collection = Pointer → Value

Value = Basic + Array + Collection

Environment = Identifier → Value

Procedure = Environment → Value* → Value*

A procedure like *remove* with the heading

procedure remove (i: **integer**, j: **integer**) =

imports (**var** puzzle, **var** pieceCount)

could then map into a function with the type

Environment → (Integer × Integer × Array × Array)

→ (Array × Array)

where the four input types correspond to *i*, *j*, *puzzle*, and *pieceCount*, and the two output types correspond to *puzzle* and *pieceCount*. This mapping follows the same path that the translation in the previous section did. The environment is used to look up the values of constants like *p*.

To provide Scott/Strachey semantics for Pascal one would need the following sort of data spaces:

Basic = Boolean + Integer + Pointer

Array = Integer → Pointer

Value = Basic + Array + Continuation + Procedure

Environment = Identifier → Pointer

State = Pointer → Value

Continuation = State → State

Procedure = Environment → Continuation → State → Value* → State

Now, the procedure *remove* would map into a function with the type

Environment → Continuation → State → (Integer × Integer) → State

Here the environment must be used to look up the values of *puzzle* and *pieceCount* because, for the reason discussed earlier, they cannot be treated like explicit parameters. The continuation, which is the Scott/Strachey analogue of the machine language return address, is necessary because *remove* may, in general, exit with a non-local goto by applying a continuation other than the one it was passed. The state, which represents the entire data memory of the machine, must be passed around because remove may, in general, have a side effect on any location it pleases.

There is an enormous difference between Euclid and Pascal when viewed in this way. All the data spaces needed to support Euclid, except the Environment, are equivalent to data types the Euclid programmer declares explicitly; that is why we were able to translate *PuzzleSolver* into functional form without leaving the Euclid language in any serious way. The data spaces needed to support Pascal involve the extra-linguistic concepts of State and Continuation, not to mention recursive, higher order function types. It is impossible for the Pascal programmer to describe these objects in Pascal, partly because they are polymorphic. The average Pascal programmer thinks of the state only in a vague way and never thinks about continuations. Thus the functional semantics of Pascal do not reflect the way programmers think in the language nearly as well as Euclid's semantics do. Figuring out how to provide functional semantics for Pascal or Algol-60 was hard and occupied some brilliant minds for several years. The functional semantics of Euclid are trivial by comparison; even a programmer can understand them! We claim that the semantics of Euclid are much closer to Poplar's than Pascal's; neither Euclid nor Poplar involve global states or continuations. Indeed, it can be argued that Euclid's data space is simpler than Poplar's since it does not allow functions as values.

We have now completed our argument that Euclid is a disguised functional language. Why

would one not consider Euclid a functional language? What virtues of functional languages does Euclid fail to possess? It would appear that the pragmatic possibilities for verification or optimization are as good for Euclid as for any functional language. The only non-syntactic difference we see between Euclid and languages commonly accepted as functional is that it is a rather low-level language that discourages its programmers from writing programs that manipulate list structures or other large data aggregates. As a final example, Figure 8 shows a version of Trial written in Poplar, suitable only for lazy evaluation. We assume that *puzzle* and *pieceCount* are functions and that *place* returns an index and new versions of the functions. An informal description of Trial is: If j is 1 the puzzle is filled (because *place* returned a 1); return the string saying so. Otherwise, for each type of piece, i, if there is one left, and it fits, place it, and invoke Trial on the resulting *position*, *puzzle*, and *pieceCount*. If Trial does not return *fail*, concatenate the appropriate string to its value and return it; otherwise return *fail*. The maplist over the types will produce a list of solutions. If the list is empty return *fail*, otherwise return the first element. If this program is submitted to a lazy evaluator, no work will be expended discovering alternate solutions. Notice that, were a multi-processor available, we could suggest that it pursue the maplist operation in parallel by an operation that said select any member of the list of solutions rather than the first.

```
Trial ← ([j, puzzle, PieceCount]:
            j/{1} > "puzzle filled↵"
            | 1 -- typeMax
             //(i: ~(i/class/pieceCount/{0}) >
                    [i, j, puzzle]/fit >
                        [i, j, puzzle, pieceCount]
                            /place
                            /Trial
                            /output: output >
                                output "piece" i " at" k "↵")
             /Solutions: ~(Solutions/isnull) > Solutions/1);
```

Figure 8. Trial written in Poplar

Experience with Euclid

How people use Euclid should be of great interest to functional language advocates and other foes of hidden side effects. The preliminary evidence from people programming in Euclid [Wortman, Horning] indicates that it is not difficult to program real systems in it, that the ability to hide side effects is not essential, and that significant pragmatic benefits are associated with the language's restrictions. On the other hand, there is some evidence that the programmers are not programming in a functional spirit. The original Euclid design required the programmer to list all

of the imported identifiers, including the implicitly referenced ones. The implementors of the Euclid compiler, who were also the first people to program in Euclid, invented the **thus** clause and provided its automatic generation because they got tired of creating the imports lists by hand. Presumably they would *really* object if they were asked to make all the imports into explicit parameters. One might say that Euclid requires one to make the side-effects of a procedure easily discernable, while a strict functional form requires one to make them painfully obvious.

Another complaint was that Euclid requires one to declare functions with "benevolent" side effects to be procedures. Consider the case of a memo function that uses a cache to remember some of the previously computed values.

```
begin
Arg, Ans: array 1..10 of integer;
p: 1..10;
function Fibbonaci(n: integer);
        begin
        for i ← 1, i+1 until i=10 do
            if n=Arg[i] then return Ans[i];
        p ← (p+1) mod 10;
        Arg[p] ← n;
        if n=1 or n=2 then Ans[p] ← 1
        else Ans[p] ← Fib(n-1)+Fib(n-2)
        return Ans[p]
        end;
for p ← 1, p+1 until 10 do begin Arg[p] ← 1; Ans[p] ← 1 end;
p ← 1;
...
end
```

They argue that it only muddies the waters to require Fibbonaci to be a procedure which imports the cache. On the other hand one can argue, somewhat rudely, that the "functionalness" of this function depends upon the programmer not having made any mistakes, so the prudent course is to require that the cache be made explicit. Besides benevolent side effects there are also *irrelevant* side effects such as a random number generator's storage of its state. In this case the procedure is representing a stream or list, and the fact that it uses intermediate state to produce values is not relevant.

Conclusions

Our experience with Poplar leads us to believe that functional programming needn't be difficult or unnatural if one is willing to ignore efficiency considerations and program in the APL style. As the KWIC program illustrates Poplar makes it quite simple to write a long series of statements, each one of which transforms all the data in a small way. In a conventional language, the syntactic overhead of a **for** loop or a recursion is large and makes one try to merge as much processing into each loop as possible. Of course, this impulse is one of survival, given today's implementations. The large number of maplist-like operations in Poplar programs have made them excellent benchmarks for garbage collectors. We look forward to the day when they can be used as benchmarks for program transforming compilers that merge loops and eliminate unneeded list creation.

Preliminary experience with Euclid indicates that side effects can be tamed, and that the difficulty with making the programs look functional is, again, a matter of machine efficiency. The arrays and collections that a Euclid program manipulates are likely to be rather large and to be represented by blocks of storage. Passing them by value might not be possible, altough it was actually done for the relatively small arrays in the PuzzleSolver program without serious degradation of performnace.

However, there are a lot of questions that need to be answered before one can see a clear path to the use of functional languages for real programming:

How should interaction with a user be carried out? In our environment it is the norm to write programs that interact with a person through a keyboard, screen, and pointing device. To describe such things functionally one can describe each program as a function that maps each "input" into its output response, or better, an input *stream* into an output stream as [Friedman&Wise77] have done. This model doesn't fit very well with making random changes to a two-dimensional display, however.

How does one debug a program with a surprising evaluation order? Our attempts to debug programs submitted to the lazy implementation have been quite entertaining. The only thing in our experience to resemble it was debugging a multi-programming system, but in this case virtually every parameter to a procedure represents a new process. It was difficult to predict when something was going to happen; the best strategy seems to be to print out well-defined intermediate results, clearly labelled.

How does one predict performance? Never mind that lazy evaluation, or any other clever strategy, will make the program perform better than it would have otherwise—ultimately one depends upon his understanding of the machine to design things so that they run reasonably. If

the machine is clever it is probably harder to understand, especially if it employs various *ad hoc* heuristics, based upon expectations of what sort of programs people write.

How does one arrange meaningful checkpoints? Even if one's computation has no bugs and is non-interactive, the order in which things are done can be relevant. When one's computation takes a long time he would like to save intermediate states that have meaning to the programmer. For example, in a correspondence management system we found it desirable to produce a letter and record the fact that it had been sent as an atomic action. Typically one might request the system to send many letters and expect that one or two requests would cause trouble for reasons ranging from hardware errors, to software errors, to improper requests. Also, one occasionally wanted to interrupt the process to do something else with the machine. Since there is no interdependence between these requests and the operation takes a non-trivial amount of time, one would like all but the troublesome requests to be completed. We attempted to solve this problem through the use of explicit writes on files—a highly non-functional operation. If one attempted to describe the operation as a whole, surrendering control of what happens to the system, any mishap forces one to start over entirely.

To summarize, the potential practical benefit of a functional language is that its implementation has much more running room in which to be clever since the order in which operations are performed is constrained only by the data flow. Examples of such cleverness are lazy evaluation, compile-time loop integration, and parallel processing. On the other hand, computing is an activity that goes on in time and space. In situations where one cares about the time and space aspects of an operation as much as the qualitative result, functional programming is less applicable. Furthermore, the personal, interactive mode of computing tends to increase the frequency of these situations. The view that computing is functional was much more plausible in the days when one interacted with a single computer at a leisurely pace. Then the computer itself—when it was working—seemed to represent a function from program and data to answers. Now many computers are parts of networks, and a large part of their activity is devoted to receiving and transmitting information on the networks. It is possible that God can look at the whole thing as a function, but any individual program or machine is dealing with a highly non-deterministic, non-functional world.

One way to compromise between functional and imperative programming is to recognize that every language has a functional subset, usually associated with the set of expressions in the language. However, it is rarely useful to take advantage of the functional subset via clever evaluation or compilation strategies because expressions in that subset usually represent very small intervals of computation or leave the functional subset via calls on functions with side effects.

Farther out reflections

Functional languages as a minority doctrine in the field of programming languages bear a certain resemblance to socialism in its relation to conventional, capitalist economic doctrine. Their proponents are often brilliant intellectuals perceived to be radical and rather unrealistic by the mainstream, but little-by-little changes are made in conventional languages and economies to incorporate features of the radical proposals. Of course this never satisfies the radicals, but it represents progress of a sort.

A little appreciated role of functional programming, goto-less programming, and other stylish forms of programming is as an indicator of a programmer's morale. When one comes across a program with a rat's nest of gotos, or large amounts of pointer arithmetic one says to himself, "This programmer was barely able to solve the problem he was working on. If he had the intellectual problem well under control, then he could have devoted some of his brainpower to making it look pretty according to generally accepted standards, e.g., eliminating gotos."

Even if they never become useful for real programmers functional languages are useful objects of study. Functional languages are entirely mathematical, so the places where they don't work show where computing is not mathematics and help to illuminate both fields.

References

[Backus] John Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Comm. ACM 21*, 8 (Aug. 1978), 613-641.

[Burge] William. H. Burge, *Recursive Programming Techniques*, Addison-Wesley, Reading Mass., 1975.

[Landin] Landin, P.J. The next 700 programming languages. *Comm. ACM 9*, 3 (March 1966), 157-164.

[McCarthy] John McCarthy, Recursive functions of symbolic expressions and their computation by machine, *Comm. ACM 3*, 4 (April 1960) 185-195.

[Scott & Strachey] D.Scott, C. Strachey, Toward a Mathematical Semantics for Computer Languages, 1971 Symposium on Computers and Automata, Microwave Research Institute Proceedings, Vol. 21, Polytechnic Institute of Brooklyn, 1972.

[Henderson & Morris] P. Henderson, J. H. Morris, A lazy evaluator. Proc. 3rd annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, Atlanta, 1976, 95-103.

[Friedman & Wise] D. P. Friedman and D. S. Wise, CONS should not evaluate its arguments. In Automata, Languages and Programming, Michaelson and Milner, eds., Edinburgh University Press, 1976, 257-284.

[Darlington & Burstall] J. Darlington, R. Burstall, A transformation system for developing recursive programs, *JACM 24*, 1, (January 1977), pp 44-67.

[Dennis] J. B. Dennis, D. P. Misunas, A preliminary architecture for a basic data-flow processor, Proc. Second Annual Symposium on Computer Architecture, Computer Architecture News, Vol. 3, No. 4, Jan. 1975, pp 126-132.

[Morris *et al.*] J. H. Morris, Eric Schmidt, Philip Wadler, Experience with an applicative string processing language, in Proc. 7th annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, Las Vegas, Nevada, 1980

[Lampson *et al.*] B. Lampson, J. J. Horning, R. W. London, J. G. Mitchell, G. J. Popek, Report on the Programming Language Euclid, Technical Report, Computer Systems Research Group, University of Toronto, 1981.

[Griswold] R.E. Griswold , J.F. Poage , and J.P. Polonsky, *The Snobol-4 Programming Language*, Prentice-Hall, 1971.

[Aho] A. V. Aho, B. W. Kernighan, and P. J. Weinberger, Awk - A Pattern Scanning and Processing Language, Bell Laboratories Internal Memorandum, Murray Hill, N. J., 1978.

[Lesk] M. E. Lesk, and E. Schmidt, Lex - A Lexical Analyzer Generator, Bell Laboratories Internal Memorandum, Murray Hill, N. J., 1978.

[Tesler] L. Tesler, H. Enea, D. Smith, The LISP70 pattern matching system, Proceedings of the International Joint Conference on Artificial Intelligence, Stanford, 1973.

[Parnas] D. Parnas, On the criteria to be used in decomposing systems into modules, *Comm. ACM 15*,12, (Dec 1972).

[Wortman] D. B. Wortman, J. R. Cordy, Early Experience with Euclid, The Fifth International Conference on Software Engineering, San Diego, March, 1981.

[Holt] R. C. Holt *et al.* TUNIS: A UNIX-like operating system written in Euclid, Technical Note 16, Computer Systems Research Group, University of Toronto, 1980

[Baskett] F. W. Baskett, personal communication, Xerox PARC, 1981.

[Horning] J. J. Horning, personal communication, Xerox PARC, 1981.

[Friedman&Wise77] D. P. Friedman and D. S. Wise, Aspects of applicative programming for file systems, SIGPLAN notices, vol. 12 no. 3 (March 77), p. 41-55.

XEROX

XEROX

Real Programming in Functional Languages

by James H. Morris

Xerox Corporation
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

Printed in U.S.A.

CSL-81-11