ON STYLE, EXPRESSIBLITY, AND EFFICIENCY IN FUNCTIONAL PROGRAMMING LANGUAGES

Gabriel Robins

April 1988
CSD-880029

On Style, Expressibility, and Efficiency in Functional Programming Languages

Gabriel Robins
Computer Science Department
University of California, Los Angeles

Abstract

A functional style of programming was proposed by Backus as an alternative to conventional programming paradigms. Functional programming is intended to alleviate what Backus dubbed as the "Von Neumann Bottleneck," a restrictive set of biases deeply ingrained in the thought processes of programmers. The functional programming paradigm is investigated here through the actual implementation of a series of common algorithms. One of the novel features of this paradigm is the lack of side-effects; it is shown how this property can be used to achieve considerable efficiency improvement in arbitrary implementations of functional languages, using the idea of **value-caching**. This scheme is then shown to reduce the execution of certain computations from exponential-time to polynomial-time, a formidable speedup. It is also shown that the expressive power of functional languages can be greatly enhanced via the utilization of both **macros** and **idioms**, *without* changing the original semantics of the language.

1. Introduction

A functional style of programming was proposed in 1978 by John Backus [Backus] as an alternative to conventional programming paradigms. Functional programming is intended to alleviate what Backus dubbed as the "Von Neumann Bottleneck," a restrictive set of biases which by now is deeply ingrained in the thought processes of programmers. Backus argues quite eloquently that because the processor and the memory of a conventional computer are connected via a narrow bus, processors end up spending most of their time ferrying useless data to and from the main memory.

The more serious problem, however, is that the classical "word-at-a-time" model of computer design has over the years managed to permeate up through the abstraction levels and become fused into most programming languages, and indeed into the very thought processes of the majority of computer scientists. According to Backus, this phenomenon is directly responsible for many horrendously complex programming languages with convoluted and even inconsistent semantics, where it is often impossible to prove formally even the most trivial correctness properties.

Backus gave a syntax and semantics for a candidate functional programming language. Some of the novel features of his language are the *lack* of variable names, assignment statements, global state, and side-effects. Instead, several powerful combining operators can be used to compose functions, yielding clean and simple semantics, conciseness of expression, and easily-derivable formal correctness properties. See [Vegdahl] for a more detailed discussion of these issues. Considerable work regarding FP has also been performed at UCLA: [Alkalaj], [Alkalaj, Ercegovac, and Lang], [Arabe], [Meshkinpour], [Meshkinpour and Ercegovac], [Patel and Ercegovac], [Patel, Schlag, and Ercegovac], [Pungsornruk], [Sausville], [Schlag, 86], [Schlag 84], [Worley].

In this paper I investigate some of the claims made by proponents of functional programming, through the actual implementation of a series of common algorithms. Based on these examples, I then draw some conclusions with regards to the practical advantages and flaws of the functional programming paradigm. The language I use is Berkeley's functional programming language, FP [Baden]. All FP forms which appear in this paper were verified to be correct by actually typing them into the FP v. 4.2 (4/28/83) system. The rest of the paper assumes that the reader is familiar with Berkeley FP; for a brief refresher on the syntax and semantics of FP, the reader is referred to the appendix.

I show how the lack of side-effects can be used to achieve considerable efficiency improvement in arbitrary implementations of functional languages, using the idea of **value-caching**. This scheme is proved to reduce the execution time of certain FP computations from exponential to polynomial, a remarkable speedup. Finally, it is shown that the expressive power of functional languages can be greatly enhanced via the utilization of both **macros** and **idioms**, *without* changing the original semantics of the language.

2. FP Solutions to Some Common Problems

In this section we examine how various common problems and algorithms can be handled in FP. We do this in order to get some feeling as to what does "typical" FP code look like. The notation "X ==> Y" will be used to denote that when the form X is evaluated it yields the value Y, as in the following: "+ @ [%1 , %2] ==> 3".

2.1. Permutations

The following function produces a permutation of the second argument, according to the order specified by the first argument. That is, given the following data vector of length N: $<d_1, d_2, \ldots, d_N>$, and a permutation vector $<p_1, p_2, \ldots, p_N>$, where $1 \le p_i \le N$ and $p_i \ne p_j$ if $i \ne j$, we would like to return the vector $<d_{p_1}, d_{p_2}, \ldots, d_{p_N}>$. The following FP function will accomplish this task:

```
{PERM1 &pick @ distr}
```

For example:

```
PERM1 : <<3 1 2 5 4> <is going what here on>>  ==>  <what is going on here>
```

But now one might argue that any straight-forward implementation of FP would cause the "distr" operator to "copy" its second argument a number of times proportional to the length of the first argument, using $O(N^2)$ space in total. In attempting a response to such criticism we may modify our definition as follows:

```
{PERM2 (null @ 1 -> [] ; apndl @ [pick @ [1 @ 1, 2] , PERM2 @ [tl @ 1, 2]])}

PERM2 : <<2 4 3 1> <permuted 4 get things>>  ==>  <4 things get permuted>
```

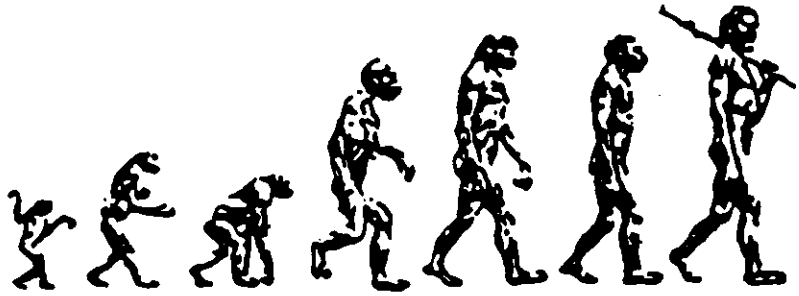
Now we are using tail-recursion to accomplish the same task. In constructing the seemingly "more space-efficient" function PERM2, we were presumptuous in assuming that the underlying FP implementation allocates space and executes FP forms in a certain way, and regardless of how close to the truth our assumptions may have been, these are the very kinds of

considerations Backus is trying to free us from in the first place.

In programming in FP we must give up many familiar and useful concepts, such as variable names, a state, side-effects, etc. But we gain many pleasant properties in return for our "sacrifice": conciseness, more easily provable correctness, better parallelism, and freedom from details. If during FP programming we worry about efficiency and implementation issues, the efficiency of our code will become implementation-dependent, and our minds will be shackled by the very details from which we escaped when we sought refuge from the "Von Neumann bottleneck" in the first place.

If we are to enjoy the full benefits of a functional programming system, we must delegate most of the efficiency and implementation issues to the underlying (optimizing) compilers. One of the main benefits of FP is that it has very clean semantics, yielding relatively painless correctness proofs, and it is amenable to simple and powerful algebraic manipulations and transformations. Such algebraic transformations could be used to automatically convert/compile an FP program into a version which would take advantage of the inherent parallelism or other special features of the architecture we happen to be executing on.

In summary, I am **not** arguing that efficiency considerations are not important; I am simply arguing that it is more the responsibility of the machines to catch up with functional paradigms of thinking and *not* the burden of functional programming to accommodate archaic architectures. The situation is dramatized in the following diagram:



Functional Programming

Hardware Support

2.2. List Reversal

Suppose we wanted to reverse a list. What first comes to mind is a recursive definition for the reversal operation, namely that the reversal of a list is equal to the last element followed by the reversal of the rest of the list. In FP this could be expressed as follows:

```
{REV1 (null -> id; apndl @ [last,REV1 @ tlr])}
```

For example:

```
REV1 @ iota : 13      ==>    <13 12 11 10 9 8 7 6 5 4 3 2 1>
```

In trying to determine whether any parallelism is inherent in list reversal, we try to apply the highly-parallel "tree-insert" FP operator, as in the following:

```
{REV2 | [2,1]}

REV2 @ iota : 13      ==>    <<<13 <12 11>> <10 <9 8>>> <<7 <6 5>> <<4 3> <2 1>>>>
```

We observe that this simple function is *almost* what we want, except for the extra levels of list depth, so we revise our definition as follows:

```
{REV3 | (concat @ [(atom @ 2 -> [2] ; 2) , (atom @ 1 -> [1] ; 1)] )}

REV3 @ iota : 13      ==>    <13 12 11 10 9 8 7 6 5 4 3 2 1>
```

Now we have obtained a highly parallel, yet simple and concisely-expressed function to reverse a list. I consider this to be functional programming at its best.

2.3. Membership in a List

Suppose we needed a function that given an element and a list, would determine whether that element appears in that list. Our first attempt is the following:

```
{MEMB ! or @ &= @ distl}

MEMB  : <13 <3 6 4 13 98 -45 13 73>>          ==>  T
```

The idea here is to distribute the search key over the entire list and "or" together the results of all the comparisons of these pairs. Now suppose we know ahead of time that our list was sorted. Using the classical binary-search technique, we could speed up the computation to "require" at most O(log N) time by recursively searching half-lists, as follows:

```
{BS (null @ 3 -> = @ [1,1 @ 2];
        (<= @ [1,1@rotr@2] -> BS @ [1,2]; BS @ [1,3])) @ apndl @ [1,split @ 2]}
```

Note that although this "optimization" appears to reduce the evaluation time, this may not really happen in reality. For example, a close examination of the code for BS would reveal that we have assumed that the operation "split" operates in constant time. If indeed split is implemented in such a way as to require time linear in the size of its argument, BS may in fact run slower than its seemingly "less efficient" counterpart MEMB.

This example again illustrates our previous argument that FP programmers should not try to "second-guess" the implementation, because they may loose. It is conceivable that in the future intelligent compilers could algebraically manipulate such expressions and, for example, replace a linear search with a binary search whenever it finds a monotonicity property to be preserved. So again, I claim that such "optimizations" should be left to the compilers.

Of course one must draw the line somewhere. Given that the problem of whether two programs are equivalent or not is undecidable, in the general case it is not possible to perform these optimizations automatically. Yet even undecidability results do not mean that special classes of instances could not be handled automatically, as well as efficiently. For example, it is probably too much to hope that a compiler would recognize an $O(N^5)$-time max-flow algorithm and replace it with an equivalent but more efficient $O(N^3)$ time algorithm; on the other hand it is not far-fetched to expect that an intelligent compiler could determine that a certain piece of code behaves like a look-up table and consequently implement it as a hash-table rather than a list, yielding better efficiency.

2.4. Fibonacci Numbers

The Fibonacci numbers are defined as the sequence of integers beginning with two 1's with each subsequent integer being the sum of the previous two. This recursive definition immediately yields an FP program:

```
{F (<=@[id,%2] -> %1; +@[F @ D,F @ D @ D])}  # returns the ith Fibonacci number
{D  -@[id,%1]}                               # decrements by 1
{FIB &F @ iota}                              # returns the first n Fibonacci #'s

FIB : 18          ==> <1 1 2 3 5 8 13 2134 55 89 144 233 377 610 987 1597
2584>
```

Although this program is both elegant and concise, it is quite "inefficient", because any conventional implementation of it would force the re-computation of the function on many of the same arguments. We can alleviate this flaw as follows:

```
{FIB2 tlr @ tlr @ FF @ [iota , [%1,%1]]}
{FF (null @ 1 -> 2 ; FF @ [tl@1 , apndr @ [2, + @ [last@2 , last@tlr@2]]])}
```

Now successive Fibonacci numbers are appended to a growing list, where the last two elements are used directly to compute the next Fibonacci number. This optimization is essentially an algorithmic one and reduces the "execution" time from exponential-time to polynomial-time. a considerable savings. I believe such programmer-affected optimization to be quite valid, and also one which FP programmers *should* endeavor to concern themselves with. Later in this paper, we describe a method of achieving considerable running-time improvement while preserving the functional appearance of functions such as the Fibonacci numbers function; this is accomplished via the idea of value-caching. In other words, sometimes we can "have the cake and eat it too," functionally speaking.

2.5. Sorting

Sorting a list entails rearranging its elements in such a way as to preserve some monotonicity property with respect to successive elements. For simplicity, we assume that our elements are integers and that the monotone property we wish to sort with respect to is the

normal arithmetic "less-than" or "<". Our first sorting algorithm is fashioned after the classical **selection-sort**:

```
{SS (null -> [] ; apndl @ [1, SS @ SUB] @ [MIN,id])}
{SUB concat @ &(= -> [] ; [2]) @ distl}
{MIN | (< -> 1 ; 2)}

MIN : <3 5 0 9 213 -7 1>              ==> -7
SUB : <13 < 2 13 a 5 w 13>>           ==> <2 a 5 w>
SS : <4 7 5 1 0 9 11 7>               ==> <0 1 4 5 7 9 11>
```

The function MIN determines the least element in a given list, while the function SUB returns a new list without any occurrences of a given element. Sorting is accomplished by finding the least element, putting it first, followed by the sorted list of the remaining elements.

Next we try another method for sorting, which is similar to our selection-sort algorithm. To sort a list, we keep rotating it until the first element is least. We then continue this process on the tail of the list, until the entire list is sorted. We call this algorithm **rotation-sort:**

```
{RS (null -> []; (= @[1,MIN] -> apndl @ [1, RS @ tl]; RS @ rotr))}
{MIN | (< -> 1 ; 2)}

RS : < 5 3 13 -2 -13 13 0 99 213>     ==> <-13 -2 0 3 5 13 13 99 213>
```

As an interesting aside, our list is now allowed to have duplicates and such duplicate elements are not eliminated from the list. Although our rotation-sort algorithm is specified very concisely, it does not exhibit much parallelism and would require "quadratic" time to run in any conventional implementation. Another method of sorting entails swapping adjacent pairs of elements until the list becomes monotonic. We call this scheme **odd-even-sort**:

```
{SORT (while UNORD (apndl @ [1,SWAP @ tl] @ SWAP))}
{UNORD |or @ &> @ trans @ [tlr,tl]}
{SWAP concat @ &FLIP @ pair}
{FLIP (null @ tl -> id; (> -> [2,1]; id))}

UNORD : <-5 2 5 9>                    ==> F
FLIP : <13 2>                         ==> <2 13>
SWAP : <76 13 0 12 33 22 0 13>        ==> <13 76 0 12 22 33 0 13>
SORT : <76 13 -12  22 0 13 213 0 >    ==> <-12 0 0  13 13 22 76 213>
```

The idea here is to look at the pair of elements 1 and 2, 3 and 4, etc, and swap each pair in-place if they are out of order. During the second pass, we similarly treat the pairs 2 and 3, 4 and 5, etc. We keep repeating this two-phased pass over the data until the list becomes sorted. Note the high degree of inherent parallelism here, since all the pairs may be examined/swapped in parallel,

A common method for sorting with a good average-time behavior is quicksort. The quicksort algorithm selects a pivot element and splits the input into two sublists, one consisting of elements less than (or equal-to) the pivot, and the other list consisting of elements greater-than the pivot element. Each of these two sublists is similarly sorted in a recursive fashion, and the final result then consists of the first list, followed by the pivot element, followed by the second list. The FP implementation of **quicksort** follows:

```
{QS (<= @ [length,%1] -> id; concat @ [QS @ L, [1], QS @ R])}
{L concat @ & (>  -> tl; []) @ distl @ [1,tl]}
{R concat @ & (<=-> tl; []) @ distl @ [1,tl]}

L : <7 3 9 -4 13 27>                    ==> <3 -4>
R : <7 3 9 -4 13 27>                    ==> <9 13 27>
QS : <76 13 -12  22 0 13 55 213 0 >  ==> <-12 0 0  13 13 22 55 76 213>
```

Again, we note the conciseness and elegance of the FP specification. Moreover, it is well-known that quicksort has an O(NlogN) average-time behavior (although the normal worst-case behavior is still quadratic). Later in this paper we describe how the FP definition of quicksort can be made even more concise using macros.

2.6. Prime Numbers

A prime number is a positive integer with **exactly** two distinct divisors: itself and 1. We would like to specify a functional program which given an integer N, returns a complete list of primes between 2 and N, inclusively. Here is our first pass at this problem:

```
{PRIMES concat @ &(PRIME -> [id] ; []) @ tl @ iota}
{PRIME |and @ &(not @ = @ [%0,id]) @ &mod @ distl @ [id,tl@iota@-@[id,%1]]}

PRIMES : 20                             ==> <2 3 5 7 11 13 17 19>
```

The idea here is to process all the integers between 2 and N, trying to determine which ones have no factors other than themselves and 1. All integers satisfying this property are primes and are therefore returned as a list. The function PRIME is a predicate which tests a single integer for primality, while the function PRIMES maps PRIME onto an entire list of candidates for primes.

We note however, that to test for primality, we need only try to divide by factors no greater than the square root of our candidate integer, since if our integer is divisible by a factor which is greater than the square root, the result of the division is less than the square root and so we would have tried that integer earlier. Furthermore, other than the integer 2, only odd integers need be considered for primality, and only odd integers (up to the square-root of the candidate) need be considered as factors, since an even integer can not divide an odd one. These "optimizations" are reflected in a new version of our prime number generator:

```
{PRIMES apndl @ [%2, concat @ &(PRIME -> [id] ; []) @ ODDS]}
{PRIME |and @ &(not @ = @ [%0,id]) @ &mod @ distl @ [id,ODDS @ SQRT]}
{SQRT 1 @ (while (< @ [*@[id,id]@1, 2]) [+@[1,%1],2]) @ [%1,id]}
{ODDS apndl@[%2,&(+@[*@[id,%2],%1])
                    @iota@-@[+@[/@[id,%2],mod@[id,%2]],%1]]}

PRIMES : 50                     ==> <2 3 5 7 11 13 17 19 23 29 31 37 41 43 47>
```

Note that while this scheme is "faster" than the previous one, the code is beginning to look rather ugly. In fact, further speedup is possible since when checking for the primality of a candidate integer, we need only to consider other primes (up to the square-root of the candidate) as possible factors. The lesson here again is that the more "efficient" a functional program is, the messier it tends to become.

In the case of prime numbers, for example, I think that only the first "optimization" (the one about going only up to the square root) is of the type with which the functional programmers need concern themselves, because it is "algorithmic" in nature, and relies on a non-trivial number-theoretic fact; it so happens that it is also the only one which increases the "efficiency" of our functional program by a non-constant factor.

We give another algorithm for prime generation, the **sieve of Erastostenes**:

```
{SIEVE concat @ &(MEMB -> [];[1]) @ distr @
          [id,concat @ &&* @ &distl @ distr @ [id,id]] @ tl @ iota}
{MEMB !or @ &= @ distl}

SIEVE : 18                          ==> <2 3 5 7 11 13 17>
```

The idea here is to list all the integers and erase all factors of 2, all factors of 3, and so on. Everything which remains after this process of elimination is by necessity a prime. Actually, in this implementation we do the opposite: we form all possible products and then consider as prime all the "missing" integers. Again we see an example of the trade-off between concise specification and computational "efficiency".

3. FP vs. APL

There are some notable similarities between APL and FP is quite similar in spirit and syntax to APL. Some functions and operators in FP are taken directly from APL; for example, the FP iota function is equivalent to the APL iota function ("ι"). Similarly, the FP operator "right-insert" ("!") is equivalent to the APL operator "reduce" ("/"). The difference with respect to these operators is that in FP one can compose any number of operators together in a concise and semanticaly clean manner, while in APL only a pre-defined limited set of types of compositions are allowed.

Both in FP and APL complex operations may be defined very concisely, often giving rise to "one-line" programs. Yet unlike FP, the semantics of APL are very complex, being muddled by state, local and global variables, parameter passing, operator overloading, goto statements, and the existence of several hundreds of different functions and operators. I have seen one-line APL programs which take the better part of an hour to understand; a possible moral here is that "absolute power corrupts absolutely." While it is true that some pieces of FP code may also seem rather dense, the situation in FP is still much better than in the notorious APL "one-liners."

4. FP vs. LISP

FP is also similar in many respects to LISP. Both languages rely heavily on lists as their primary data structures, and both languages provide a rich set of list-manipulation primitives. On the other hand, LISP functions are more defined than their FP counterparts; for example in FP **tl : <>** is undefined, while in LISP **(CDR NIL)** is still **NIL**. Given that FP functions are bottom-preserving, it is often the case that an FP function would return "?" on a boundary or pathological datum, while the LISP equivalent would still return the "right" answer.

Here is an example of when it is preferable to have functions which are more defined on "boundary conditions". Given an atom x and a list of atoms y, we would like to return the set of all atoms in y which directly follow occurrences of x in y. The obvious way of specifying this in

FP elegantly is:

```
{F concat @ &(= @ [2,1@1] -> [2@1] ; []) @ distr @ [trans @ [tlr,tl] @ 2 , 1]}

F : <3 <m 3 4 5 6 3  3 x 3>>          ==>      <4 3 x>
F : <y <>>                            ==>      ?
```

The problem here of course is that our function is not defined for the pathological case when the second argument is the empty list, while it would be preferable to define the answer in that case to be the empty list. We could modify our original function to check for that boundary condition as a special case, but if the FP functions **tl** and **tlr** were originally designed to return the empty list when called upon the empty list, our function F would not have the problem mentioned here. For example, if **tl**, **tlr**, and **trans** were redefined thus:

```
{TL (= @ [ length , %0 ] -> [] ; tl )}
{TLR (= @ [ length , %0 ] -> [] ; tlr )}
{TRANS (|and @ &null -> [] ; trans)}
```

F could now use these "more defined" specification to work correctly in all cases, including the boundary conditions:

```
{F concat @ &(= @ [2,1@1] -> [2@1] ; []) @ distr @ [TRANS @ [TLR,TL] @ 2 , 1]}

F : <y <x>>                           ==>      <>
F : <y <>>                            ==>      <>
```

The point we wish to stress is that had some of the primitive functions been just a little more "defined", the programmer would not have had to worry about these boundary conditions, and freeing the programmer from such details is clearly a goal of the functional-thinking paradigm.

5. Macros in FP

FP currently does not allow the use of macros. This is, however, an artificial restriction since macros can be specified and used in a way that will not alter the original semantics of FP whatsoever. FP macros specify simple "static" substitution rules that would "expand" certain pieces of FP code. In other words, a macro-expansion preprocessor would be invoked before execution ever commences.

To accommodate macros, the original FP syntax would be extended as follows:

```
macro-definition ==> '#define' name '(' argument-names ')' macro-body
name ==> letter (letter | digit | '_')*
argument-names ==> name ( ',' name)*
macro-body ==> (function-form | name)*
function-form ==> simple-function | composition | construction
                        | conditional | constant | insertion | alpha
                        | while | '(' function-form ')' | macro-expansion
macro-expansion ==> name '(' arguments ')'
arguments ==> string ( ',' string)*
string ==> (letter | number | special-character)*
```

It is understood that a macro expansion form and the corresponding macro-definition must agree on the number of arguments. It is further observed that arguments to macros may include arbitrary strings, not just FP objects, as long as a valid FP form is obtained when macro-expansion is terminated. Moreover, macro-expansion is a recursive process, where macros may refer to other macros, as long as termination of the macro-expansion process is guaranteed.

As an example, the macro *#define add1(x) + @ [x,%1]* would expand the form *add1(k)* into the form + @ *[k,%1]*. As a second example of using macros, our previous definition of quicksort could be specified more concisely as:

```
#define S(X) concat @ &(X -> tl; []) @ distl @ [1,tl]
{QS (<= @ [length,%1] -> id; concat @ [QS @ S(>=), [1], QS @ S(<)])}
```

The above form, after the macro-expansion into "pure" FP, will become:

```
{QS (<= @ [length,%1] -> id; concat @ [QS @
                    concat @ &(>= -> tl; []) @ distl @ [1,tl], [1], QS @
                    concat @ &(< -> tl; []) @ distl @ [1,tl]])}
```

Note that in this example, we can not achieve the same effect by simply defining a new function, because the argument we really need to pass is a functor itself, not a value. In general, macros will increase the readability of FP code by making the FP representation of a function more concise, whenever certain pieces of code are identical or at least very similar. A macro can then be used to "factor out" that common piece of code, so that it does not have to be specified or expressed more than once; in particular, macros would allow the definition of an APPLY template, similar in spirit to the LISP apply function, something that is impossible to implement in "pure" FP.

5.1. The Implementation of Macros

We emphasize that the usage of macros as defined here does not extend nor change the original semantics of FP in any way. In fact, an implementation of this idea might consist of a macro-expansion pre-processor which would take as input a specification in macro-enhanced FP (that is, an FP program with some macros) and output an equivalent "pure" FP program after properly doing the requisite macro expansion and substitution. The result would then be fed into the original FP processor.

For example, to implement a macro pre-processor for Berkeley FP, one could extend the original FP syntax to accommodate macros as described above, and construct a UNIX lex(1) and yacc(1) specification to lexically analyze and parse the new syntax. The associated production rules would perform the required macro expansion, producing as output an equivalent FP program in the original "pure" syntax. The function "yyparse" can then be used as a "filter" to FP.

An easier alternative to using heavy-artillery such as yacc(1) and lex(1) consists of the following scheme: since many existing languages already have rather fancy macro-preprocessors, why not "fool" such a tool into doing our FP macro-processing work for us? For example, the standard UNIX C compiler has such a macro-preprocessor, and in fact it may be invoked without the actual C compilation. So an implementation of this scheme would consist of the following two steps:

1) Run the file containing "macro-enhanced" FP program through the C macro

preprocessor and collect the resulting "pure" FP output in a file. This is accomplished by specifying the -E flag: **cc -E macro-prog.fp > pure.fp**

2) Run the resulting file as usual through the normal FP system: **fp < pure.fp**

I have already tested this scheme with several examples, including the quicksort example stated above, and it worked flawlessly. In fact, if one combines the above two steps into one UNIX command aliased to an appropriate name, the entire macro-expansion process would be transparent to the user. Of course, if the format of our proposed FP macros deviates at all from that of C, this trivial scheme would fail and we would be forced to revert back to our direct-translation scheme using lex(1) and yacc(1).

Macros could also be used to "modify" the syntax of FP by providing alternate ways of expressing FP constructs; such extensions to FP in no way alter the original semantics of FP but simply provide some "syntactic sugar" to ease the programming task.

6. The Need for Idioms

I found that during FP programming, many constructs repeat themselves. For example, the function to decrement a counter, to sort, etc. Such common and repeatedly used functions can and should be provided for in the implementation of an FP-like language. Indeed such functions can be easily specified within FP itself, thereby not adding to the original set of "primitives", but instead be provided as a library of functions or macros.

I claim that it would be very convenient to define certain idioms within FP. Note that this convenience is not in conflict with the fact that such idioms may be easily specified using other FP forms. For example, the original definition of FP still contains the idiom "length", even though "length" may easily be defined as a recursive application of tl, or perhaps even more cleverly as:

{length | + @ &%1}

Perlis and Rugaber have already devised such a set of idioms for APL [Perlis and Rugaber]. I believe that a similar set should be identified and defined for FP. In particular, I recommend the following list of possible idioms as a starting-point for the further development of this idea:

Removal of duplicates from a list - given a key x and a list y, return a new list z which is a copy of y except that all elements equal to x have been removed.

Sorting a list given a specified partial-order predicate - given a list, return the list sorted according to a specified (or default) comparison predicate. Since the said predicate is an argument to our "sort" routine, an implementation using macros would probably be best here.

Generalized "iota" - given A, B, and C, return a sequence of all multiples of C, beginning with A and no larger than B.

Vector element uniqueness - given a sequence of objects, return "true" if there exists a repeated element in this sequence, or "false" otherwise.

Is X a permutation of Y? - given two sequences x and y, determine whether x is a permutation of y. This may be accomplished by sorting both into some "canonical order" and then comparing.

Minimum/maximum element - find the minimum (or maximum) element in a given sequence, using a specified min (or max) function, or else use a default comparison function.

Membership in a vector/set - test whether a given element x is a member of vector/set y.

Union of vectors/sets - given two vectors/sets x and y, return a vector/set of elements which are either in x or in y, without duplicates.

Intersection of vectors/sets - given two vectors/sets x and y, return a vector/set of elements which are both in x and in y, without duplicates.

Subtraction of vectors/sets - given two vectors/sets x and y, return a vector/set of elements which are in x but not in y, without duplicates.

Subset of a vector/set - given two vectors/sets x and y, determine whether each element of x is also an element of y.

Transitive closure - given a sequence of pairs or objects, return the transitive-closure of this set, interpreting the pairs as a specification of a binary relation.

Math functions - square root, exponentiation, factorials, binomial coefficients, successor, predecessor, etc.

Flatten a list - return a list of all atoms in a given list, disregarding the original level (or nesting-depth) of each.

7. Improved Efficiency Through Value-Cashing

We have observed that there exists a sharp trade-off in certain computations between conciseness-of-expression and "computational efficiency." For example, in the case of generating Fibonacci numbers, the elegant version of the algorithm appears as follows:

```
{F (<=@[id,%2] -> %1; +@[F @ D,F @ D @ D])}  # returns the ith Fibonacci number
{D - @[id,%1]}                               # decrements by 1
```

We also observed earlier that alternative ways of specifying the Fibonacci function yield more complex and difficult to read specifications. So the question now is how efficiently can the current specification be used to compute the i^{th} Fibonacci number on a conventional architecture, and whether the situation can be improved. We settle the former part of the question first, and argue that a conventional implementation would require exponential time to compute the current Fibonacci numbers specification.

It is well-known that the N^{th} Fibonacci number is given by the expression:

$$F_n = [\phi^n - (1-\phi)^n] / (2\phi - 1) \quad \text{where } \phi = (1 + \sqrt{5}) / 2$$

The constant $\phi \approx 1.618033989$ is also known as the golden ratio, and when N approaches infinity, the term $(1-\phi)^n$ approaches zero and therefore F_n approaches $\phi^n / (2\phi - 1)$ but since

ϕ is a constant, we have $F_n = \theta(\phi^n)$ or that F_n grows exponentially. Since a conventional implementation can not be expected to solve such recurrence relations in closed-form, it must use recursion to compute F_n. But the problem is that F will be recomputed many times at the same values, and since the value of F_n must be obtained from the lowest levels of the recursion via repeated additions of the constant 1, the total number of additions will be equal to the value of F_n, or proportional to ϕ^n. It follows that a naive computation of the Fibonacci sequence from the recursive specification will require exponential time.

Of course, in order to remain within polynomial-time, we could explicitly remember old values, but this is contrary to the spirit of the functional paradigm. We would like to combine the elegance of the concise recursive specification with the "efficiency" of the "optimized" version.

A hybrid can actually be achieved through what I call **value-caching**. The idea is to save argument/value pairs for certain difficult-to-compute functions, so that those values could be looked up in the future instead of having to be recomputed. We define a new FP operator @$ (read compose-cache) to have the exact same semantics as the familiar FP operator @, except that after the composition is computed, the result is stored away in such a way that next time the same composition needs to be performed, the result is already known and can be used immediately without any recomputation or further delay. Clearly, the new composition operation so defined has the exact same semantics as the normal composition operator, and moreover, this entire scheme remains transparent to the programmer.

Because in FP there are no side-effects, composing a function f with an argument g will yield the exact same result at **any** stage of the computation, and that is why old results will **always** be valid in the future. This scheme can be implemented using hash-tables to store the value/result pairs for each function we would like to thus speed up. When a function is "called" upon an "argument", the argument is looked up in the hash-table associated with that function, and if a result is found it is returned immediately. If not, the result is computed from scratch, and then it is saved in the hash-table using the argument as a key, whereupon the result is passed on to the rest of the computation.

Of course not all functions should be so treated, only those that are non-trivial to compute. For example, using this scheme, our Fibonacci function would now appear as follows:

```
{F (<=@[id,%2] -> %1; +@[F @$ D,F @$ D @ D])}   # returns the ith Fibonacci num
{D  -  @[id,%1]}                                 # decrements by 1
```

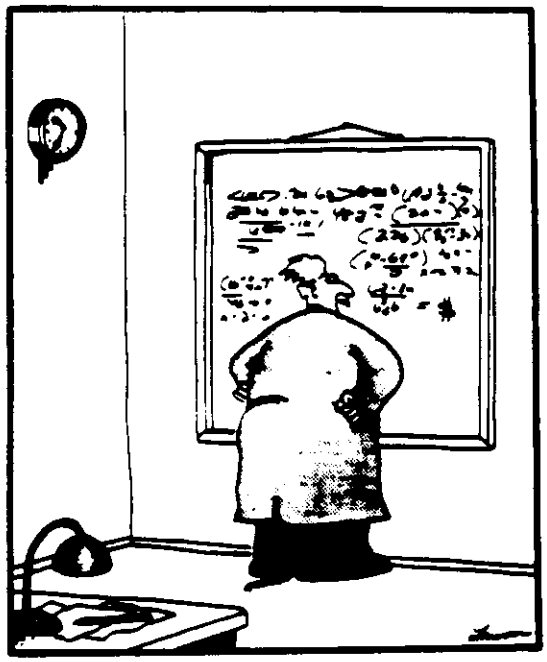
Notice that we have not lost any of the conciseness or elegance apparent in our original specification. And assuming the cache/lookup process takes near-constant time, the Fibonacci function now will run in near-linear time on any conventional implementation so enhanced, an exponential improvement!

This scheme will work for arbitrary functional languages, not just FP. Moreover, the @$ operator need not be used manually by the programmer, but simply be invoked automatically by the implementation whenever it is determined that doing so is likely to improve the efficiency. Some possible criteria for automatically performing value-caching are when considerable time has been spent inside some particular function that has relatively short input and output, when a particular function is being called a very large number of times on only a small set of different values, and when a function (or a set of functions) is highly recursive.

All of these situations can be detected automatically by the implementation, so it is

possible to automate the idea of speed-up by value-caching. Sometimes, however, it may be the case that the programmer can better determine where caching should and should not be performed, so it may be desirable to give the programmer a "manual-override" capability with respect to value-caching. I propose two new variations on the composition operator, and a slightly modified behavior of the old composition operator:

@$ - value-caching is **always** performed.
@^$ - value-caching is **never** performed.
@ - architecture automatically decides when cashing should be performed.



Einstein discovers that time is actually money.

Other instances where value-caching will yield considerable increase in execution efficiency are applications where computations have a geometric meaning in Euclidean D-space. For example, Conway's game of "life", where a cellular automata in the 2-dimensional plane is simulated and where the "next-generation" rule is a function of the local neighborhood surrounding the cell. A straight-forward recursive computation to determine the state of a given cell would run in time exponential in the magnitude of the generation to be computed, where the base corresponds to the size of the neighborhood around cells. On the other hand, the same algorithm running in a value-caching environment would run in polynomial time.

Generally speaking, whenever a dynamic-programming algorithm of some kind is used, new values are dependent on certain old values (with respect to some set of indices), an therefore an ordinary functional solution would likely to give rise to an inefficient execution. But if value-caching is used in the implementation, the execution time of the exact same algorithm is likely to improve drastically.

Recently it has come to my attention that the idea of caching values in order to avoid recomputations is also discussed in [Keller and Sleep]. They discuss several ways of storing and using old values in order to improve execution efficiency, and also address the problem of "purging" or keeping the cache size from growing without bounds during a computation. They also give as example another application area where value-caching would greatly speed up the execution time, namely in the finite-element method.

8. Conclusion

Through a series of examples I have investigated programming style and practical considerations in FP. I have demonstrated that it is often the case in FP that concise functions become verbose and complicated when certain "efficiency optimizations" are attempted. I have also argued that certain so-called "optimizations" may really not be optimizations at all, and should not be undertaken by an FP programmer.

I have briefly discussed the similarities between FP and APL and LISP, and argued that certain FP functions should be "more defined" and that there is a need for some additional FP "primitive" functions or macros. I described a scheme where macros can be implemented in a clean way, leaving the original semantics of FP intact. I have argued that FP would benefit from a library of idioms, or commonly-used constructs upon which the programmer can draw.

It is possible via novel architectures and intelligent compilers to support efficient FP-like languages, where many optimizations can take place automatically, leaving the programmer free to see much of the high-level picture without drowning in the details. I introduced one such possible optimization, value-caching, and showed that it can reduce the execution time of certain computations by as much as an exponential amount.

9. Acknowledgments

I would like to thank Professor Milos Ercegovac for introducing me to FP, and for the numerous illuminating discussions which inspired this paper. Warm thanks go to Professor Sheila Greibach for encouraging me to polish up this paper and submit it to the quarterly; she made numerous thoughtful comments, and also helped in unsplitting my infinitives... ☺

10. Bibliography

Alkalaj, L., A Uniprocessor Implementation of FP Functional Language, MS Thesis, UCLA Computer Science Department TR CSD-860064, April 1986.

Alkalaj,L., Ercegovac, M., and Lang, T., A Dynamic Memory Management Policy for FP, 1987 Hawaii International Conference on Systems Science.

Arabe, J., Compiler Considerations and Run-Time Storage Management for a Functional Programming System, Ph.D. Dissertation, UCLA Computer Science Department, CSD TR 86004, August 1986.

Backus, J., Can Programming Be Liberated from the Von Neumann Style? A Functional Style and its Algebra of Programs, Communications of the ACM, Vol 21, No. 8, August, 1978, pp. 613-641.

Baden, S., Berkeley FP User's Manual, Rev. 4.1, September 29, 1987, 33 pages.

Keller, R., Sleep, R., Applicative Caching: Programmer Control of Object Sharing and Lifetime in Distributed Implementations of Applicative Languages, Proceedings of the 1981 ACM Conference on Functional Programming Languages and Computer Architectures, October, 1981, pp. 131-140.

Meshkinpour, F., On Specification and Design of Digital Systems Using and Applicative Hardware Description Language, M.S. Thesis, UCLA Computer Science Department, Technical Report No. CSD-840046, November 1984.

Meshkinpour, F., and Ercegovac, M., A Functional Language for Description and Design of Digital Systems:Sequential Constructs, IEEE Proc. of the 22nd ACM/IEEE Design Automation Conference, 1985, pp.238-244.

Patel, D., and Ercegovac, M., A High-Level Language for Hardware-Oriented Algorithm Design, presented at the Symposium Electronics and Communications in the 80's, Indian Institute of Technology, Bombay, February 1983.

Patel, D., Schlag, M., and Ercegovac, M., yFP: An Environment for the Multi-Level Specification, Analysis, and Synthesis of Hardware Algorithms, Proc. 1985 ACM Conference on Functional Programming Languages and Architectures, Nancy, France, Springer-Verlag Lecture Notes 201, 1985.

Perlis, A., Rugaber, S., The APL Idiom List, Research Report #87, Yale University.

Pungsornruk, S., A Threaded FP Interpreter/Compiler, M.S. Thesis, UCLA Computer Science Department TR CSD-860061, March 1986.

Sausville, M., Gathering Performance Statistics on Hardware Specified in FP, UCLA Computer Science Department Internal Report, March 20, 1986.

Schlag, M., Layout from a Topological Description, Ph.D. Dissertation, UCLA Computer Science Department, Summer 1986.

Schlag, M., Extracting Geometry from FP Expressions for VLSI Layout, UCLA Computer Science Report CSD-840043, October 1984.

Vegdahl, S., A survey of Proposed Architectures for the Execution of Functional Languages, IEEE Transactions on Computers, Vol c-33, No. 12, December, 1984, pp. 1050-1071.

Worley, J., A Functional Style Description of Digital Systems, M.S. Thesis, UCLA Computer Science Department, Technical Report CSD-860054, February 1986.

10. Appendix: A Brief Overview of FP

In order to keep this paper self-contained, we briefly review the syntax and semantics of FP; for a more detailed description of FP, see [Baden].

The notation "X ==> Y" is used to denote that when the form X is evaluated it yields the value Y; this is simply a notational convenience. FP objects consist of numbers, strings, atoms,

and lists; examples are %213, "hi there", FOO, <I am <a list>>, and <>. An integer constant must be preceded by a percent sign, to distinguish it from the primitive selector functions. We use the form "F : x" to denote the result of the function F applied to the argument x; for example, + : <%2 %3> ==> 5. The unique error atom (or "bottom") is denoted as "?". In what follows, the formal FP specifications are also explained informally for clarity.

K : x ==> if x=<x_1, x_2, ... , x_n> and 1≤k≤n then x_k else ?

For every integer K there exists a primitive selector function **K** that returns the K^{th} element of a list. For example:

3 : <fi fy foo fum> ==> foo.

Other selector primitives:

pick : x ==> if x=<y,<x_1, x_2, ... , x_n>> and 1≤y≤n then x_y else ?
last : x ==> if x=<> then <> else if x=<x_1, x_2, ... , x_n> and 1≤n then x_n else ?
first : x ==> if x=<> then <> else if x=<x_1, x_2, ... , x_n> and 1≤n then x_1 else ?
tl : x ==> if x=<y> then <> else if x=<x_1, x_2, ... , x_n> and 2≤n then <x_2, x_3, ... , x_n> else ?
tlr : x ==> if (x=<>) or (x=<y>) then <> else if x=<x_1, x_2, ... , x_n> and 2≤n
then <x_1, x_2, ... , x_{n-1}> else ?

pick selects the y^{th} element out of a given list, **last** selects the last element, **first** selects the first elememt, **tl** chops the first element off and returns the rest (like the LISP cdr function), and **tlr** chops the last element off and returns the rest. Examples are:

pick : <2 <a b c>>	==>	b
last : <w x y z>	==>	z
first : <<1 2> <3 4 5> <6 7 8 9>>	==>	<1 2>
tl : <who are you>	==>	<are you>
tlr : <fi fy fo fum>	==>	<fi fy fo>

Structural and list-manipulation primitives:

distl : x ==> if x=<y,<>> then <>
else if x=<y,<x_1, x_2, ... , x_n>> and 1≤n then <<y,x_1>,<y,x_2>, ... , <y,x_n>> else ?
distr : x ==> if x=<<>,y> then <>
else if x=<<x_1, x_2, ... , x_n>,y> and 1≤n then <<x_1,y>, <x_2,y>, ... , <x_n,y>> else ?
apndl : x ==> if x=<y,<>> then <y>
else if x=<y,<x_1, x_2, ... , x_n>> and 1≤n then <y,x_1, x_2, ... , x_n> else ?
apndr : x ==> if x=<<>,y> then <y>
else if x=<<x_1, x_2, ... , x_n>,y> and 1≤n then <x_1, x_2, ... , x_n,y> else ?
trans : x ==> if x=<<x_1, x_2, ... , x_n>,<y_1, y_2, ... , y_n>,...,<z_1, z_2, ... , z_n>> and 1≤n
then <<x_1, y_1, ... , z_1>,...,<x_n, y_n,... , z_n>> else ?
reverse : x ==> if x=<> then <> else if x=<x_1, x_2, ... , x_n> and 1≤n
then <x_n, x_{n-1}, ... , x_1> else ?
rotl : x ==> if x=<> then <> else if x=<y> then <y> else if x=<x_1, x_2, ... , x_n> and 1≤n
then <x_2, x_3, ... , x_n, x_1> else ?

rotr : x ==> if x=<> then <> else if x=<y> then <y> else if x=<x_1, x_2, ... , x_n> and 1≤n
then <x_n, x_1, x_2, ... , x_{n-2}, x_{n-1}> else ?

concat : x ==> if x=<<x_1, x_2, ... , x_n>,<y_1, y_2, ... , y_m>,...,<z_1, z_2, ... , z_l>> and 1≤n,m,...,l
then <x_1, x_2, ... , x_n,y_1, y_2, ... , y_m,...,...z_1, z_2, ... , z_l> else ?

pair : x ==> if x=<x_1, x_2, ... , x_n> and 1≤n is even then <<x_1, x_2>, ... , <x_{n-1},x_n>>
else if x=<<x_1, x_2, ... , x_n> and n is odd then <<x_1, x_2>, ... , <x_n>> else ?

split : x ==> if x=<y> then <<y>,<>> else if x=<x_1, x_2, ... , x_n> and 1≤n
then <<x_1, x_2, ... ,$x_{floor(n/2)}$>,<$x_{floor(n/2)+1}$, ... ,x_n>> else ?

distl (**distr**) pairs from the left (the right) the first (second) argument with each element of the second (first) argument, **apndl** (**apndr**) appends the first (second) element to the end of the list given by the second (first) argument, **trans** treats the argument lists as the rows of a matrix and transposes this matrix, **reverse** reverses a list, **rotl** (**rotr**) does a left (right) circular shift on a list, **concat** concatenates two lists, **pair** pairs up the elements in a list, and **split** divides a list into nearly equal halves. Examples follow:

distl : <u <r m 2>>	==>	<<u r> <u m> <u 2>>
distr : <<i u voo> doo>	==>	<<i doo> <u doo> <voo doo>>
apndr : <<hi di> ho>	==>	<hi di ho>
apndl : <r <2 d 2>>	==>	<r 2 d 2>
trans : <<1 2 3> <a b c>>	==>	<<1 a> <2 b> <3 c>>
reverse : <to be or not to be>	==>	<be to not or be to>
rotl : <10 20 30 40>	==>	<20 30 40 10>
rotr : <<1 2> <3 4> <x y z>>	==>	<<x y z> <1 2> <3 4>>
concat : <<x> <y z> <u q t>>	==>	<x y z u q t>
pair : <romeo juliet samson delila>	==>	<<romeo juliet> <samson delila>>
split : <hair1 hair2 h3 h4 h5>	==>	<<hair1 hair2 h3> <h4 h5>>

Predicates:

atom : x ==> if x is an atom then **T** else if x ≠ ? then **F** else ?
null : x ==> if x =<> then **T** else if x ≠ ? then **F** else ?
length : x ==> if x=<> then 0 else if x=<x_1, x_2, ... , x_n> then n else ?

atom tests whether the argument is an atom, **null** tests whether it is the empty list, and **length** returns the length of a list. Examples follow:

atom : firefox	==>	**T**
null : <not empty at all>	==>	**F**
length : <tell how long <am I>>	==>	4

Arithmetic and logical operations:

+ : x ==> if x=<a,b> and a and b are numbers then a+b else ?
(and similarly for **-**, *, /, <, >, ≤, ≥, =, ≠)
xor : x ==> if x=<a,b> and a and b are boolean then xor(a,b)else ?
(and similarly for **and**, **or**, **not**, **nand**, and **nor**)

These operators compute the standard arithmetic and logical functions. Examples follow:

+ : <2 3>	==>	5

```
- : <4  13>                    ==>   - 9
* : <13  11>                   ==>   143
/ : <81  3>                    ==>   27
< : <100  999>                 ==>   T
= : <k  k>                     ==>   T
```

Functional forms, combinators, and conditional:

f_1 @ f_2 : x ==> f_1 : (f_2 : x)

[$expr_1$, $expr_2$, ... , $expr_n$] : x ==> <$expr_1$: x, $expr_2$: x, ... , $expr_n$: x>

& expr : x ==> if x = <> then <> else if x=<x_1, x_2, ... , x_n> and 1≤n
then <expr : x_1, expr : x_2, ... , expr : x_n> else ?

! expr : x ==> if x = <y> then <y> else if x=<x_1, x_2, ... , x_n> and 2≤n
then expr : <x_1, ! expr : <x_2, ... , x_n>> else ?

\ expr : x ==> if x = <y> then <y> else if x=<x_1, x_2, ... , x_n> and 2≤n
then expr : < \ expr : <x_1, ... , x_{n-1}>, x_n> else ?

| expr : x ==> if x = <y> then <y> else if x=<x_1, x_2, ... , x_n> and 2≤n
then expr : < | expr : <x_1, ... , $x_{floor(n/2)}$>, | expr : <$x_{floor(n/2)}$>> else ?

(C -> A ; B) : x ==> if C : x ≠ <> then A : x else if C : x = <> then B : x else ?

@ denotes functional composition and is by far the most common FP operator. Square brackets ([and]) denote list construction. The functor & maps the given function onto each element of the argument list. !, \, and | invoke several kinds of parallel tree evaluation, while the construct (C -> A;B) is the FP conditional operator: if C evaluates to true, A is evaluated and returned, otherwise B is evaluated and returned. Examples follow:

```
* @ tlr : <3  2  1>             ==>   6
[+@[2,2],1,2] : <b  1>          ==>   <2  b  1>
& atom : <yes <no> <<maybe>>>    ==>   <T  F  F>
! * : <1  2  3  4  5  6>        ==>   720
| id : <1  2  3  4  5  6>       ==>   <<<1  2>  3>  <<4  5>  6>>
(atom -> %1  ;  %-1) : oxygen    ==>   1
```