Aspects of Applicative Programming For File Systems*

Daniel P. Friedman

David S. Wise

Computer Science Department

Indiana University

Bloomington, Indiana 47401

TECHNICAL REPORT NO. 56

ASPECTS OF APPLICATIVE PROGRAMMING FOR FILE SYSTEMS*

DANIEL P. FRIEDMAN

DAVID S. WISE

JANUARY, 1977

*To be presented at ACM Conference on Language Design for Reliable Software, Raleigh, North Carolina, March, 1977.

Research reported herein was supported (in part) by the National Science Foundation under grants numbered DCR75-06678 and MCS75-08145.

Aspects of Applicative Programming for File Systems*

Daniel P. Friedman

David S. Wise

Computer Science Department

Indiana University

Bloomington, Indiana 47401

Abstract - This paper develops the implications of recent results in semantics for applicative programming. Applying suspended evaluation (call-by-need) to the arguments of file construction functions results in an implicit synchronization of computation and output. The programmer need not participate in the determination of the pace and the extent of the evaluation of his program. Problems concerning multiple input and multiple output files are considered; typical behavior is illustrated with an example of a rudimentary text editor written applicatively. As shown in the trace of this program, the driver of the program is the sequential output device(s). Implications of applicative languages for I/O bound operating systems are briefly considered.

Keywords and Phrases - referential transparency, recursive programming, real time, shared file, functional combination, suspension, text editor.

CR Categories - 4.22, 4.32, 4.13, 5.24

*Research reported herein was supported (in part) by the National Science Foundation under grants numbered DCR75-06678 and MCS75-08145.

Introduction

In this paper we develop the techniques of applicative programming by extending their domain of application to the global environment of a computer system. Specifically, the standard parameter and result type is presumed to be a file. The structure of a file roughly corresponds to a list structure with the most commonly occurring file being a string of characters. We demonstrate how the structure of a file and the demands of its physical manifestation determine the macroscopic behavior of the entire system. Our concepts are illustrated by a text editor written as a function.

Applicative programming is a style of programming characterized by referential transparency* with functions as the only control structure. There are several languages that embody this principle: pure LISP [22, 23, 24], ISWIM [3, 20], GEDANKEN [26], RED [2], and PLASMA [16]. The definitions of these languages are characterized by mathematically precise semantics which makes them attractive tools for developing reliable software. This same precision has made possible the development of software verification techniques [3, 6, 7] which continue to improve the utility of such languages for establishing the correctness of programs. Furthermore, experience

*"The characteristic semantic feature of expressions is that they are "evaluated"; that is, the semantic interpretation of an expression ultimately defines its "value." Furthermore, for "pure" expressions, it is exclusively the value that has semantic importance. This linguistic property is termed referential transparency [Quine, 1960], for it allows a subexpression to be replaced by any other expression having the same value without any effect on the value of the whole. Languages or language subsets having the property of referential transparency are termed applicative; other adjectives that have been used include declarative, denotative, descriptive, and functional." [27]

[8, 16, 25, 30] demonstrates that such languages are an extremely convenient way of expressing algorithms.

This last point deserves some historical perspective: since the design of classic languages like FORTRAN followed hardware design, which in turn followed Turing's state transition model of a computing device, nearly all programmers are fluent in that style of programming. Applicative programming traces its style back to Church, Gödel, and earlier, but it does not enjoy a similar popularity today. Its alienation is not deserved, however, in light of its precision, of its convenience, and of its power in dealing with parallel programming problems independently of machine models [12].

The results of this paper are based on recent results in the semantics of applicative languages, specifically the "suspending constructor" [10], which is used by the programmer to create data structures but whose execution may postpone manifesting parts of the structure until they are essential to the system. The concept is similar to call-by-need [29], to call-by-delayed-value [28], and to lazy evaluation [15], but for us the only essentially suspended structures are the files themselves. Yet it is the files, specifically the files transmitted from the system to teletype consoles and to line printers, whose display is the visible behavior of the system, and therefore these files determine what other structures will be built -- what computations are actually carried out [11]! Other files, those maintained within the global environment perhaps as part of a data base and never displayed in their entirety on an external device, may remain suspended like internal structures.

(It is important to recognize that there is no standard read or print function implicit in an applicative language. The classic definitions of these functions include side-effects on the status of the associated files and devices with implications for future invocations of these functions. These violate referential transparency. Files must be arguments and results of the "main" function.)

We extend this model of file structure to more complex computing problems by developing conventions for allowing programs written applicatively to have more than one file as output. It is no problem to specify multiple input files for a program in the applicative languages mentioned above; several input files become several input arguments. The problem of output files is difficult for two reasons. First, most interpreters use a recursion stack to evaluate function invocations with the result that an output structure would have to be represented internally in its entirety before output could begin; suspended data structures have lifted this requirement [10, 11]. Second, most functions are perceived as returning a single result so that decomposition of structures of many results is cumbersome; the concept of functional combination [9, 12, 13] (explained later) alleviates this burden so that the programmer can manipulate multiple outputs conveniently.

Our perspective on applicative programming is supported by examples of the style we advocate. The significant example is the text editor. It uses two input files and two output files, where only one of each is an essentially sequential (not suspendable) file.

Those are the files which are keyed and printed on a full duplex terminal. The other files, the input text and the edited text, will be manifested in their final form during invocation of the editor only as much as is necessary to the creation of the essentially sequential files.

Partial evaluation or partial creation of files, as discussed in this paper, is not new. The idea has been used for some time by designers of operating and file systems, but the programs whose execution exhibited such behavior have been notoriously difficult to write. Terminology such as "coroutine" and "differential file" has arisen around such mystical programming efforts. The significant message of this paper is that similar behavior is available without special effort if the problem is approached through applicative programming and one essential tool: the suspending constructor. If file manipulation programs are composed using an elementary constructor, which is implemented as a suspending constructor in spite of the programmer's intent, then this behavior appears by fiat.

The remainder of this paper is divided into five sections. The first develops the elements of our applicative language and introduces functional combination. It then reviews the suspending cons already mentioned. The second extends this cons to file construction, establishing the dependence of the computation process on the need for results to be displayed for the user. It develops the concept of files as functional input and output by considering the cases in which there are multiple input and multiple output files. The third section presents a simple, though complete,

text editor defined applicatively, and explains how integrity of "shared" files is established through referential transparency of suspended files. The fourth section offers a perspective on a simple I/O bound multiprocessing system. Finally we present a few conclusions.

Section I

An Applicative Language

The language we describe is similar to classic list processing languages like LISP [22] with respect to its data type. Since we are emphasizing files in this paper we shall be using that term more frequently although conceptually a file is nothing more than a list stored externally. The following definitions may be more easily read as definitions of lists; no confusion results if the terms are taken to be interchangeable.

A file is a sequence of elementary items or other files. This recursive definition provides for sequential homogeneous files of elementary items or for nested (tree) structured files. Elementary items are often restricted to the character set of one's hardware, although many languages permit items of other types: e.g. integer and real. For the discussion here we restrict ourselves to a standard character set but reserve the parentheses as meta-characters to delimit files. Thus, in the following examples of homogeneous files the right parentheses acts as an end-of-file mark:

```
(FR8ED)
(234567)
()
(FRED(8)(2MANYGREEN)(BANANAS))    .
```

A program is defined as a function which takes as arguments one or more files and returns as its value another file (which is interpreted to be composed of several files in certain examples below).

Our first programming notation is square brackets. (Parentheses are notation for data.) A bracketed sequence evaluates to a file of the evaluated items of that sequence in order. For example, [6 5 4 3] evaluates to (6543).

Let x have the value

(2123)

and let y have the value

(BANANAS);

then [x y] evaluates to

((2123)(BANANAS)).

Bracketed sequences provide only for creating files of fixed size and therefore they can be associated with record structures of other languages. There is also a file building function, cons, for building files of undetermined length; but before introducing it we must describe the syntax for function invocation.

Function invocations are represented by a pair of items enclosed by angle brackets: <f ℓ> . The function position, here denoted by f, indicates the operation to be performed upon the argument list ℓ. Combined with square brackets this functional syntax is very suggestive of standard mathematical notation. Instead of min(i,j) we write <min [i j]> , and <sum [2 1 2 3 1]> evaluates to 9. (See also [2] and [16] for similar applicative expressions.) With the binding of x from above, <sum x> evaluates to 8; this case illustrates that the argument list need not be explicitly bracketed although it usually is.

A most important primitive is cons; it takes two arguments, an item or a file and a file, and returns a file whose first element is that item and whose remainder is the original file. Thus <cons [2 y]> evaluates to (2BANANAS). Two complementary operations, first and rest, return the first component of a file and the file

without its first component, respectively. Thus <first[x]> evaluates to 2 and <rest[y]> evaluates to (ANANAS). These three functions are particularly interesting [10], and we shall return to them in the next section.

We shall use other elementary functions without definition; their meaning is obvious from context. These are often arithmetic, like sum, and include simple predicates: null tests whether its argument is an empty file, and less tests the order of its two arguments. Example functions are presented by relating a prototype invocation to its definition in terms of a conditional expression. This definition is presented as an alternating sequence of tests and values, whose interpretation is assisted by the insertion of the "commenting words" if, then, elseif, and else. For example,

```
<min[i j]> ≡
  if <less[i j]> then i
  else j          .
```

The tests are evaluated in sequence until one succeeds; the value immediately following that test is the value of the function. If no test succeeds then the value of the function is the value of the last expression in the sequence if the sequence is of odd length (the else part).

As an example we present a recursive definition of the function compress which removes all blanks from the file which is its argument.

```
<compress[string]> ≡
  if <null[string]> then []                          ; nothing left
  elseif <blank?[<first[string]> ]>                  ; drop a blank
         then <compress[<rest[string]> ]>
  else <cons[<first[string]>
             <compress[<rest[string]> ]> ]>          ; copy a
                                                       character
```

It is also possible to define functions that take an arbitrary number of arguments in the same manner. An example is the function concat which returns a file that is the concatenation of all its arguments (each of which is a file). An auxiliary function, append, is required to concatenate just two files.

```
<concat strings> ≡
  if <null[strings]> then []                         ; none left
  else <append[<first[strings]>
               <concat <rest[strings]>> ]>           ; binary
                                                       append
<append[stringa stringb]> ≡
  if <null[stringa]> then stringb                    ; nothing left
  else <cons[<first[stringa]>
             <append[<rest[stringa]>                 ; copy a
                     stringb]> ]> .                  ; character
```

Integers may be used as functions; as a function the integer i simply returns its i^{th} argument. One use of this notation provides for array subscripting: If c is bound to a file of files (a matrix) then <3<5 c>> evaluates to the third item in the fifth file (or the entry in the third column of the fifth row). The function 1 is often used with the "invisible argument marker" symbol # as an identity function.

The symbol # evaluates to a token which is ignored as a parameter to a function. Its evaluation is therefore useless except as an eventual argument to some function; in that role it acts much like the numeral zero: as a placeholder in argument structures with no ultimate meaning itself. For example, if d is bound to the

evaluation of [# # 9 # 7 # #] then <1 d> evaluates to 9, while <3 d> diverges since there is no third item in d taken as a parameter list. A list like d is often used in conjunction with functional combination.

Functional Combination

Functional combination is described elsewhere in detail [9, 12, 13] and we shall give it only a cursory definition by example here. It provides a convenient syntax for expressing a recursion which accumulates multiple results in the same way that a single iterative traversal of data may yield several summary results. In particular, we use it in this paper to describe a recursive text editor which yields two output files where a single computational step may affect both.

The hallmark of functional combination is the occurrence of a list in the function position. Such a list is called a combination and must be composed of other allowable functions. For example, [sum product quotient difference] is a combination; we assume their normal arithmetic definitions. Each argument to a combination is required to evaluate to a file, which we take to be a row of a "parameter matrix." (Typically one of these rows results from a recursive call on the function being defined to accumulate multiple results.) The columns of the parameter matrix become argument lists to respective elements of the combination. The result of such an invocation is always a file whose elements correspond to the respective columnar applications.

In order to facilitate the matrix interpretation of functional combination its invocation will always appear with the arguments on separate lines and vertically aligned wherever possible to suggest the columnar relationship. For example:

```
<[sum product quotient difference] [
 [0    1       7          9      ]
 [1    3       #          2      ]
 [0    3       1          #      ]]>
```

evaluates to (1977). The # symbol is used here as a placeholder in expressing a row and as an ignored argument in interpreting a column of the argument matrix.

Another example illustrates the use of functional combination to return a file of two files each built up at each step of a single recursion. The function deal takes a file as an argument and splits it into two files each of alternate elements from the argument file; it is essentially the deal of a complete deck of playing cards, as if for a two-handed game of Old Maid.

```
<deal [deck]>
  if <null[deck]> then [[][]]                          ; two empty
  elseif <null[<rest[deck]>]> then [deck []]           ; odd deck
  else <[cons    cons][                                ; deal two
        [<1 deck>  <2 deck>]
        <deal [<rest [<rest[deck]>]> ]> ]>
```

Suspending cons

The function cons is representative of an entire class of functions that build structures by filling in the values of fields within nodes. Operationally it also serves as a space allocator although that characteristic plays a lesser role in the following discussion. We have proposed a new interpretation for cons and its

extractor functions first and rest which avoids the construction of those portions of structures that are never accessed after their creation. The results apply to any operation that assigns a value to a field, provided that it is possible to preserve a record of all relevant bindings. This criterion is difficult to meet in a system in which users can change assigned values, but it is easily satisfied under a regime of applicative programming in which the user can only create and implicitly release such bindings [17].

Using the function cons as a paradigm of structure-creating functions, we briefly explain its implementation. When cons is invoked by the user, the value returned is a pointer to a newly built structure. Rather than evaluate the arguments to cons and create the complete structure, we create a structure consisting of two suspensions. A suspension consists of a reference to the form (expression) whose evaluation was deferred and a reference to the environment of variable bindings in which the suspension was originally created. These two structures must remain intact for the life of the suspension. The reference to the form is a pointer to a piece of program, so the space it occupies usually represents no great overhead. Environments present more of a problem, since we are accustomed to viewing them only as temporary structures. Moreover, use of destructive assignment operations generally requires recreation of the entire environment in order to assure the integrity of references to the environment as it existed before the assignment. Destructive assignments, if not well controlled, become costly; by design they do not exist in our source language.

When either of the "probing" functions first or rest is invoked, the following events occur. A designated field of the argument is checked to determine if it contains a suspension (suspensions are flagged and easily distinguished); if not, then its contents are returned. If a suspension is present, then the evaluator is invoked upon the designated form within the preserved environment. The result is stored back in the designated field in place of the suspension (for next time), and the value is returned as a final result. These events constitute coercion of the suspension. The two functions, first and rest, therefore act as probes into the data structure, with possible effects of a predictable and benign sort, rather than as simple extractor functions.

We introduce the term manifest to describe a structure which is actually extant in its ultimate form. A structure that is not completely manifest is said to be promised. At the implementation level, promised structures are characterized by the presence of some references that are suspended.

A fortunate side-effect of suspending the creation of data structures is the ability to deal with infinite structures. Consider the list defined (but never completely constructed) by the invocation of <terms[0]> where

```
<terms[n]> ≡ <cons[<reciprocal [<square[n]> ]>
                  <terms[<add1[n]> ]> ]>          .
```

That list, the reciprocal of the squares of all the non-negative integers, might be familiar since its sum, excluding the first term, converges to $\pi^2/6$. Suppose that z were bound to the result of

<terms[0]>; in fact, because of the suspending cons, z is initially bound only to a promise of this result. As long as <1 z> is not computed (since it diverges on division by zero) and as long as a complete traversal of the structure is not attempted, the unboundedness of z poses no problem. An access to <6 z>, if demanded by the computation, would find the answer 0.04 even though that number had not been present before that access; it would have been computed had it been of interest earlier. (This use of cons is similar to Landin's prefix* [20, also 4], but it differs precisely in that the rest of the list z may be accessed without computing the divergent first element.) More implications of cons on infinite structures may be found in [14].

The same techniques used for cons may be applied to any record creating (field assignment) function within the system. We have proposed an interpreter [10] in which all field assignments are suspended. This change has a great impact because the construction of environments may be suspended. This means that no argument will be evaluated unless the corresponding formal parameter has been accessed by some operation critical to the execution of the program. This effects the call-by-need argument-passing protocol [29], the call-by-delayed-value [28] and lazy evaluation scheme [15].

As a result of suspending, evaluations are delayed as long as possible. Ultimately, all evaluations take place as a result of the demands of the driver of the output device which tries to move the contents of its file to the external device. As it traverses the structure it is outputting, it invokes first and rest,

causing top-level evaluation, which in turn results in the creation and inspection of more structure, indirectly forcing all of the necessary evaluations. Regardless of the intentions of the programmer, the only structures which are actually built are those that are essential to deciding what information is to be output. Least-fixed-point semantics for the language result [10].

Section II

Promised files

What, then, is the essence of computation which determines which probes must be made? We claim that the essence is the result; in our case the result of the function invoked as a "main" or top-level program. That result is either a file, as in the compress example above, or a structure of files like those we shall discuss later. The output file is handed to a device which consumes the file. If the device is a random-access device similar to main memory (but slower) then there is no reason to manifest the file immediately. It is sufficient to store it in its suspended form. If the file is handed to a sequential device then suspensions could cause trouble.

A suspended file cannot be stored untransformed on a sequential device since further access must transform it. Coercing a promised file on a sequential device is unreasonable. The transformation may stretch or shrink a portion of the file, requiring a repacking of the entire file. It may require access to many parts of the file because of the nested structure in suspensions, requiring that the physical device be slewed repeatedly. Since the effect of all these transformations is to manifest pieces of the file one at a time, all the transformations should be anticipated by coercing all the suspensions in files sent to sequential devices.

When one considers the role of sequential files in a system this convention is even more reasonable. With the exception of back-up copies of random-access storage, the role of sequential devices is communication outside the system: people and other computers. Teletypes, line printers, readers of punched cards all fall into this category, as do magnetic tapes and phone lines when used as a channel to such equipment. (Magnetic tapes used to save and then to restore random access memory do not qualify. The sequence of creation is immaterial to the order in which the data are copied.)

When a file is transmitted to a sequential device it is traversed sequentially as the contents are moved onto the external medium. The traversal of the file uses the functions first and rest which coerce any suspensions uncovered [11]. If the file were entirely suspended then this traversal would determine where and when various parts of the file became essential. The essence of computation then becomes "What character is printed next?" Under this philosophy nothing is computed unless it is directly relevant to what is on an output file sent to a sequential device. One should compare such behavior to execution of an infinite loop with no WRITE statement in FORTRAN; is a poor program a sufficient excuse for such waste?

The classic view of a device driver is a piece of program that looks out from the computer and watches over the device, making sure that its status is readily available when the processor wants it. We view an output device driver as a piece of code which looks

into the processor from the device, continually demanding something from the program. For random-access devices the response to the demand is probably a promised file. For sequential devices the device driver is a traversal algorithm [11]; the demands are for successive characters; the action of the processor is to manifest a file on the external medium as its contents are encountered by the device driver. Given that most files are suspended before they reach a sequential device and given that there is little computation necessary to build a suspended file, practically all computation is invoked by these output drivers.

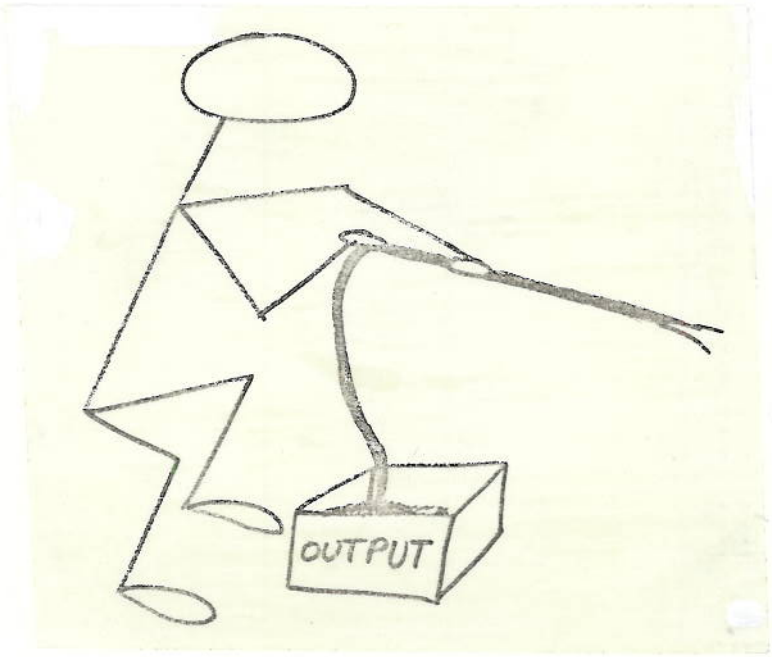

Figure 1. A device driver drawing output from a program.

A minor but pleasing effect is that computation is naturally overlapped with output since only the computation necessary to discover which character is to be printed next is performed before printing it.

The device driver for an input device is a simpler program, which looks from the processor out to the device, satisfying the demands for a next character when the processor requests one from the input file. Only the work necessary to satisfy the immediate demand is performed. If the input device is random-access then the item actually read could be a previously created suspension. In this case the request to the input driver would be translated into a coercion on the promised file structure. A part of the creating program would actually run yielding the desired character and transforming the file structure to avoid repeating the same computation on future access.

As an example we follow the behavior of the compress algorithm, taking into account the effect of the suspending cons. Letting the symbol Δ stand for the blank character, consider that function compressing the file (COΔOP). We assume that COΔOP is the contents of a file on some sequential device. None of it has been read yet. The program compress is run until the first application of cons, which is suspended, and the result is handed to the output driver. In order to get to that application of cons the predicates <null[string]> and <blank?[<first[string]>]> have been evaluated; they have forced the file to be opened and the "C" to be read, respectively. No more computation has been invested in building the

"output file" that the driver gets. If the output device is random-access then no more computation is done at this time: the output file is stored as the suspension of the argument to that first application of cons, including appropriate bindings back to the input file (which cannot be destroyed as long as this promised file survives).

If, however, the output file is sequential then the driver will traverse it to transfer it to the physical device immediately. This traversal proceeds one character at a time with computation halting at a basis (terminal) condition or halting at a cons until the traversal algorithm forces it to proceed further. Beginning with the original suspended output file with the argument string bound to (COΔOP) the traversal and writing proceeds as follows:

The output file is opened.	(
The first character of string is written.	C
The traversal proceeds to a new invocation of compress with string rebound to the rest of string: (OΔOP).	
The string is tested by null.	
The first character of string is tested by blank?	
The first character of string is written.	O
The traversal proceeds to a new invocation of compress with string rebound to the rest of string: (ΔOP).	
The string is tested by null.	
The first character of string is tested by blank? yielding success.	
The traversal proceeds to a new invocation of compress with string rebound to the rest of string: (OP).	

The string is tested by null.

The first character of string is tested by blank?

The first character of string is written. O

The traversal proceeds to a new invocation of compress with string rebound to the rest of string: (P).

The string is tested by null.

The first character of string is tested by blank?

The first character of string is written. P

The traversal proceeds to a new invocation of compress with string rebound to the rest of string: ( ).

The string is tested by null yielding success.

The output file is closed.)

On the surface the summary above appears to be a simple trace of a program. It may be regarded either as a trace of an iterative algorithm printing a structure or as a trace of the evaluation of a recursive function. Remarkably, it is both. The basic algorithm is iterative [11]; since once a variable is rebound its former value is never required so only one environment (set of bindings) at a time is needed. Space required is bounded and no stack is used; running time is proportional to the length of the original input file as one would expect of good iterative code. Evaluation proceeds at the pace of the final output and because of suspending cons the interpreter does not consume space for intermediate results as does a classic interpreter for recursive programs.

We have pointed out elsewhere [14] a rather startling difference between our semantics for such recursive function definitions and the classic semantics. The output "driven" evaluation allows at least the prefix of infinite structures to be printed, up to the resource limit (time or line limit), whereas classic semantics lead to divergence with an overflow of the recursion stack. Our approach does not use a stack so that this artificial constraint does not concern the user.

Suppose then that a user were typing at a full duplex teletype terminal where the keyboard was the input file to the function compress and the printer was the output file. Initially the input file is treated as if it were promised and could only be manifested at the typing speed of the user. As the user types "(" the printer echoes "(". As he types "C" the echo is "C". As he types "OΔO" the echo is only "OO", and "P)" is echoed "P)". The computation appears to be synchronized cleverly with the input when in fact the output driver is "gobbling up" the characters as soon as the function and, indirectly, the input driver make them available. As the user keys "(COΔOP)" the printer types "(COOP)".

Multiple input files

Programs that have many files as input and only one file as output are only a slight generalization of single-input/output-file programs. If there are to be precisely N input files then the program is written as a function which takes N parameters. Our model also allows functions to be written which take an arbitrary number of files; such functions have but one formal parameter to represent the argument list.

As an example of this feature we exhibit the definition of a multi-way merge using a selection tree [19]. The function merge takes an arbitrary number of sorted files as arguments and merges them into a single one. The last case in the definition of mergen deserves a warning: the argument list for merge2 is the result of functional combination.

Knuth comments on his solution that "from one standpoint it is equivalent to ... two-way merges performed concurrently as co-routines." Our code is written as two-way merges which perform as coroutines only under the suspending implementation for the cons within merge2. This code is easily composed using only the idea of two-way merges; afterwards one pleasantly discovers the coroutine behavior possible if cons suspends.

```
<merge files> ≡ <mergen[files]>

<mergen[files]> ≡                                   ; n-way merge
   if <null[files]> then []                         ; empty
   elseif <null[<rest[files]>]>
      then <first[files]>                           ; one file
   else <merge2 <[mergen mergen][
                  <deal[files]>  ]>>                ; split them

<merge2[f1 f2]> ≡                                   ; 2-way merge
   if <null[f2]> then f1                            ; one empty
   elseif <null[f1]> then f2                        ; other empty
   elseif <less[<first[f1]> <first[f2]>]>           ; choose smaller
      then <cons[<first[f1]>                        ; and recurse on
                  <merge2[<rest[f1]> f2]>]>         ; the remainder
   else <cons[<first[f2]>
              <merge2[f1 <rest[f2]>]>]>
```

As before, the application of merge to promised files may not cause any computation at all if the output file can itself be promised. If on the other hand it is sent to a sequential device, then the device driver for that device will manifest the promised input files in the order that the merge function requires their contents. Still, as the algorithm is run only a finite amount of information about each input file need be carried within the main memory of the machine; all but the current elements of these files remain on external devices.

Multiple output files

We now turn to the case in which a program has multiple output files. While iterative programming languages ordinarily provide for multiple output files it is not obvious how a function, which usually returns a single result, can return instead three or four results, each one an output file. To deal with this problem we adopt the convention that the normal result of any top level function is a file of files. When there is only one output file the result would be a file of one file. The result for compress developed above must then be ((COOP)).

Often multiple output files are generated using functional combinations. As an example, consider the "inverse merge" operation performed by the receiver of a simplex ANSI 8-bit encoded message. We present a function which takes a file of characters as input and separates it into two files: the file excluding all occurrences of the ten communication control characters and the file of these control characters and text separators (e.g. unit, record, group, and file separators) only. Only text separators appear in both files.

```
<separate[file]> ≡
  if <null[file]> then [[][]]                          ; empty
  elseif <separator[<first[file]>]>
     then <[cons          cons          ][              ; separator to both
           [<first[file]> <first[file]>]                ; files
           <separate[<rest[file]>]>     ]>
  elseif <comcontrol[<first[file]>]>
     then <[  1  cons                  ][               ; control to second
           [  #  <first[file]>         ]                ; file
           <separate[<rest[file]>]> ]>
  else <[cons              1       ][                   ; text to first file
        [<first[file]>     #       ]
        <separate[<rest[file]>]> ]>
```

If more output files are required then the result is a file of corresponding length. Each element in the result file is turned over to a single output driver. The characteristics of that device (random-access/sequential and its relative speed) will determine the actual behavior of the program and the pattern in which the various components of the result are coerced; each of these "brother" drivers may behave independently even though structures might be shared. However, it is more appropriate to consider multiple output files as being handed to drivers that are synchronized in some way. If one device driver manifests its file completely while a brother device (say a random-access one) leaves its file entirely promised, then the environments that were released by coercion of the first file may remain in the system because they are referenced indirectly through the suspensions of the second. After the first file is entirely manifest on the output device, many of the environments generated in the process of coercing may remain. This causes a

large space overhead in the representation of the second file. It is generally better to traverse the second file concurrently with the first file so that the intermediate environments need not be maintained. There are two cases in which it would be wrong to coerce the second file along with the first: if the second file diverged despite the first file's convergence†, or if the manifestation of the second file resulted in the creation of a set of environments disjoint from those used in the manifestation of the first.

Synchronization of multiple files may be achieved by pacing the manifestation of all files to the fastest of the brother drivers. Another strategy is more selective in coercing suspensions before the structure is actually needed: when an environment is first abandoned by the coercion of any suspension all other suspensions referencing that environment are also coerced. We call this selective coercion dragging because the traversal of one file drags along the evaluation of some others. Any environment released by manifestation of the dragging file is then dereferenced by all other suspensions and released, but the suspensions of the slower, dragged files are not necessarily coerced to their ultimate value. Either of these strategies may be implemented by logging suspensions for later coercion off-line, just as computations ordered from an interactive computing session may be completed in background batch mode.

---

†The term convergent (divergent) normally applies to functions which (do not) terminate with an answer. Our use of these terms applied to files reflects the fact that these data structures may exhibit similar behavior since they may be evaluated through a coercion.

Section III

A text editor

We now present a rather complex function that takes two files as input and produces two files as output. One input file and one output file are tied to a full duplex teletype. The user creates the teletype input file as a sequential file while he causes the

other to be read from a mass storage unit (which may be a random access device -- we assume it is); one output file is routed back to that unit.

The function is a text editor. The user-constructed (editing) input file contains the editing commands to be performed on the second input file (the text). These commands are echoed on the teletype printer along with interleaving responses from the editing process. The edited file output to the mass storage unit may be assumed to be suspended, although much of it may be dragged into existence by the traversal required to manifest the teletype output file. In the case that the user is editing only the first third of his text, only the first third of the input file from mass storage is necessary to create the responses on his teletype, and so no more than the first third of the edited file will be dragged onto mass storage. The remainder will be represented as a promise referencing the latter two thirds of the text file indirectly. It is entirely possible that the balance of the original text (the first third) may have been abandoned after the editing session. Under this assumption the space required on mass storage to represent the text before and after editing is roughly comparable, although the better part of the text is as yet unaltered. Future use of the edited file will determine if it is ever to be manifested in its entirety.

The text is an ordinary file of characters, one of which is distinguished in its treatment by the editor. This is the newline character NL which is used as a delimiter for the t-command explained below. Other commands treat NL as an ordinary character.

The editor has only a basic set of commands; the implicit cursor moves in a forward direction through the text and all changes are made at the cursor position. The commands are summarized below:

(t) types a line from the cursor position forward to and including the next NL;

(r) repositions the cursor at the beginning of the text;

(d) deletes the character to the right of the cursor;

(i target) inserts the target string to the left of the cursor;

(f target) finds the first occurrence of the target string to the right of the cursor repositioning the cursor just after it. Success is reported if the target string is found or else the cursor is moved to the end of the text and failure is reported;

(s target replacement) locates target string just as the f-command does and reports success or failure, but replacement string is substituted for target string if found and the cursor is moved past it.

We require the system to send a prompt character (denoted :) to the teletype after every command line (terminated by NL).

Driver functions

```
<editor[commands text]> ≡                                  ; top level
  if <null[commands]> then [ [] text]                      ; for r-restart
  else <editbind <onepass[commands text]>> ;

<editbind[message commands text]> ≡                        ; returns 2
  <[concat  1  ][                                          ; files: tele-
   [message #  ]                                           ; type and
   <editor[commands text]> ]> ;                            ; new text.

<onepass[commands text]> ≡
  if <null[commands]> then [[] [] text]                    ; stop
  elseif <legal[<first[commands]>]>                        ; legal command?
    then <[concat                    1    1   ][
          [<first[commands]>         #    #   ]            ; echo command
          [(NL)                      #    #   ]            ; newline
          <<first[<first[commands]>]>                      ; apply the basic
           <concat[                                        ; editor command to
              <rest[<first[commands]>]>                    ; its explicit and
              [text <rest[commands]>]]>> ]>                ; implicit parameters
  else <[concat                  1     1    ][
        [(ILLEGALΔCOMMAND)       #     #    ]             ; echo error message
        [(NL:)                   #     #    ]             ; newline and prompt
        <onepass[<rest[commands]> text]>]>                 ; continue
```

Command execution

```
<t[text commands]> ≡
    <[concat        1    1  ][          ; type a line
     [<line[text]>  #    #  ]           ; and continue
     <onepass[commands text]>]>         ; to next command

<r[text commands]> ≡
     [(:) commands text]                ; return to top-level

<d[text commands]> ≡
   if <null[text]>
     then <[concat          1    1  ][
           [(EMPTYΔFILE)    #    #  ]
           [(N␣:)           #    #  ]   ; echo error message,
           <onepass[commands text]>]>   ; newline and prompt
   else <onepass[commands <rest[text]> ]> ; and continue ; delete and continue

<i[target text commands]> ≡
     <[concat    1     concat][
      [(:)       #     target]          ; insert target
      <onepass[commands text]>]>        ; and continue

<f[target text commands]> ≡
   if <null[text]>
     then <[concat                1  1][
           [(FAILEDΔTOΔFINDΔ)     #  #]
           [target                #  #]
           [(N␣:)                 #  #]
           <onepass[commands text]>]>   ; continue
   elseif <1<prefix?[target text]>>
     then <[concat                1  1]
           [(FOUNDΔ)              #  #]
           [target                #  #]
           [(N␣:)                 #  #]   ; found it and
           <onepass[commands text]>]>   ; continue
   else <[   1    1    cons          ][
         [   #    #    <first[text]>]   ; search some more
        <f[target <rest[text]> commands]> ]>

<s[target replacement text commands]> ≡
  if <null[text]>
    then <[concat                1 1][
          [(FAILEDΔTOΔFIND )     # #]
          [target                # #]
          [(N␣:)                 # #]
          <onepass[commands text]>]>        ; continue
  elseif <1<prefix?[target text]>>
    then <[concat          1 concat     ]
          [(FOUNDΔ)        # replacement]
          [target          #    #       ]
          [(N␣:)           #    #       ]   ; replace and
          <onepass[commands                 ; continue
                   <2<prefix?[
                      target text]>>]>]>
  else <[   1    1    cons          ][
        [   #    #    <first[text]> ]
       <s[target                            ; search some more
           replacement
             <rest[text]> commands]>]>
```

Auxiliary functions

```
<line[text]> ≡
  if <null[text]> then []                          ; take all
  elseif <NL?[<first[text]>]> then (NL)            ; characters up to
  else <cons[<first[text]>                         ; and including the
             <line[<rest[text]>]> ]>               ; next NL

<prefix?[target text]> ≡
  if <null[target]> then [TRUE text]               ; compare for
  elseif <null[text]> then [FALSE text]            ; pattern match
  elseif <same[<first[target]>
               <first[text]>]>                     ; returning rest
    then <prefix?[<rest[target]>                   ; of text
                  <rest[text]>]>
  else [FALSE text]
```

The outermost call to the function appears as

```
<editor[tty text]>
```

where we understand that the first file on the output list is to be displayed on the user's teletype and the second file is to be the edited text; the program only runs because of the persistence of the output driver that manifests a file on the user's console in spite of the inertia of the other output driver. As soon as there has been sufficient computation to determine the next character on the console output then that character appears. That is, it is possible for this program to echo the commands typed in full duplex (assuming reasonable processor speed) without requiring that the teletype input file be complete. Computation proceeds as each command is entered, allowing more output to appear.

We now trace a short editing session. Suppose the input text (whether manifest or promised) is the file

```
(THEΔQUICKΔBROWN
FOXΔJUMPEDΔOVERΔTHEΔLAZYΔDOG)
```

After the user initiates the editor from his teletype he can begin typing commands. Initially there is no prompt since the program first must establish that the command file is nonempty; this is implicit in the code. Suppose he types

(THEΔSICKΔDOWNFOXΔJUMPEDΔOVERΔTHEΔLAZYΔDOG).

However, the text from FOX forward would not necessarily be recopied as a result of the editing; this would happen only if the remainder of the file were coerced. The prefix up through S should be "dragged" along as a result of the last substitute command, but the phrase ICKΔDOWN may be a suspended copy of the file dragged just before the r-command.

When the text is used in its entirety this fragmentation would be invisible except that the computation necessary to manifest the promised text would have to be performed; if the file is never used it will never be done. If the prefix of the file were again edited, perhaps to insert the needed space after DOWN, then the suffix would still remain uncopied.

File integrity

The purpose of this section is to demonstrate that the referential transparency of an applicative [27] system guarantees the integrity of what are classically perceived as shared files. When such files are rewritten in a dynamic file system, precautions are required to insure the reliability of the processes under changes that may place files in an unstable state. The accessing programs must be cleared through a semaphore [5] or monitor [18] in order to keep other processes away while such changes take place.

Any program that "changes" a file is implemented as a function which returns the transformed file as one of perhaps several results. Future accesses into the transformed file take this output file as an input parameter. (This model may be implemented under any applicative system, although one function would have to be

completely evaluated before its output could be used by another.) Under this model a file may only be "shared" if it is used as an input argument to functions which do not change it. Among functions which do change it there is no provision for simultaneous operation on that file. Conceptually, one function invocation is completed before another is initiated, and the programmer may assume this when he defines the functions that probe and perhaps transform a file. Hence under our model of a system with promised files, internal changes in files may proceed without the necessity of introducing semaphores.

While it is common to conceive of several processes simultaneously, using and possibly changing the same file in a data base system, such processes are never actually simultaneous. Each process, in turn, must pass through a monitor [18] or gatekeeper which has the option of securing a file temporarily in order to effect a change. Although queries appear to enter the system simultaneously, they form a queue at the gate to each file.

This queueing is brought about naturally in a functional programming system. If a function is written to search a file then that file will be an argument to that function. If that function (even rarely) might transform the file as a result of the search, as in AVL [1] insertion, then the possibly transformed file will be one of the several outputs of the function. If the transformed file is to be the subject of another search then it will be the input parameter to another invocation of the same function. In the mind of the programmer all these intermediate results are manifest files; if they are on a random access device, in fact they are promised.

The use of a promised file in a subsequent search will probably involve coercing. Coerced suspensions may have originated in several prior invocations of the searching function. Therefore, a snapshot of the state of the file might give the illusion that all these prior invocations were still running, seemingly in parallel but in fact more like coroutines. If the file is not changed by a function (read-only), as long as the most current promise of the file (obtained from a well-monitored file directory) is passed as an argument it is possible for more than one processor at a time to be traversing and coercing suspensions which define different incarnations of the same file. In that way the classic model of a shared, dynamic file may be realized.

The classic model is realized using functional programming depending on referential transparency to protect against ill-defined data structures. The role of the gatekeeper is played by the device drivers which pass the output file from one invocation through the file directory to be the input file of the next.† Parallelism is allowed because files may be promised by building them with suspending cons [12]. Another rather pleasing effect of this approach is that only the minimal work is required to respond to a query into the file system. Much can be left suspended for future queries to perform, if that material is ever needed. If it is needed then the job requesting it may have to bear the entire expense of its creation.

There is justice in charging the computation necessary to maintain a data base according to the inquiries. It is then trivially cheap to add (without searching) to a file but expensive to read such files. This places the cost of a data base on the users.

†Semaphores or monitors are still required to manage the file directory in order to determine unambiguously "Who's next?" The computation of a new entry to the directory is thereby secured, but that computation suspends after its first reference is discovered -- a brief effort compared to a file protected against all use while it is entirely rewritten.

such files. This places the cost of a data base on the users, where it belongs, and encourages sources to insert new information by charging them substantially less than users.

A metaphor, suggested by E.I. Organick, is the driver who must shovel the shared driveway if he is the first one to leave after the blizzard. He either accepts his labor as the price of his unsociably early departure, or he learns to sleep later and let the law of averages spread the winter's work more evenly across his neighbors.

In an active data base a frequent query should encounter rather few suspensions and will be inexpensive. Occasionally, a query might find that many suspensions lie in its path to the answer, increasing the cost to get there. As a result the true costs of sustaining a large dynamic file are equitably distributed among the users; common queries remain cheap but access to obscure or rapidly changing information will incur a computational toll.

Section IV

I/O bound multiprocessing system

In this section we present a brief perspective on an I/O bound operating system; showing how it might interact with a file system. The description is not intended to be complete, but it indicates how applicative programming techniques may relate to another aspect of system programming. We assume the existence of a system evaluator for our language operating at this "top-level." It is not necessary that the evaluator be implemented as a single process. Any mechanism for parallelism consistent with the semantics of our language [12] may be used to enhance its performance by using as many processors as are available.

A file system is postulated in the form of a global environment. Bindings in the global environment provide system functions (cons, null, first, etc.), system constants (NIL, :, TRUE, etc.), and current copies of files. When a file is updated it is necessary to use a modified global environment wherever the updated file is to be available; the new environment is used for all future "top-level" functions. Any suspensions referencing the former copy of the file through the former global environment are preserved; the referential transparency of the system extends to global environments.

The system's input queue is an unbounded (conceptually infinite) list of jobs. Each job is a list of three items that may be defined with references to whatever global environment will be in effect when it is run. The first is a function; the second is a list of input file assignments which will evaluate to input files; the third is a list of output file assignments. The output assignments will be used by the monitor to update its global environment by dispatching the outputs of the function applied to the input files to the specified output devices/destinations. The number of input files for a job should be as large as the number of parameters for its function (since top-level functions are functions on files). Each output file should have a device specified as well.

First we define the function eval* (function definition implicit in the syntax of [12, 13] where the concept of starred combinators is presented.) which applies the system evaluator to a series of expressions in the current global environment.

```
<eval*[forms globalenv]> ≡
   if <null[forms]> then []
   else <cons[<eval[<first[forms]> globalenv]>
              <eval*[<rest[forms]> globalenv]> ]>  .
```

(As in LISP [24] the function eval takes as arguments a form and an environment in which to evaluate it.)

The monitor itself is defined as a function that returns an environment. In ordinary operation it will never have the opportunity to return a value, as it must first complete processing the active (conceptually infinite) input queue. The global environment being created conceptually contains all the results that have been produced as well as the bindings of all file names. We postulate the function forward which distributes a list of files among a list of devices, thereby creating a new environment from an old one (like pairlis [24]). As discussed earlier in the paper, various devices treat these files with various degrees of immediacy; these distinctions affect the apparent performance of the system but not the formal definition of the ultimate environment.

```
<monitor[queue globalenv]> ≡
  if <null[queue]> then globalenv
  else <monitor[<rest[queue]>
                <process <first[queue]>> ]> ;

<process[function infiles outdrivers]> ≡
  <forward[<function <eval*[infiles globalenv]>>
           outdrivers
           globalenv]>
```

The list of jobs is provided by the users of the system. Nothing in our model requires them to be processed completely one at a time. Even if the output device specified by a job is sequential, requiring that the associated file be manifested, nevertheless the device driver may run the programs so slowly that a processor will be idle and have much opportunity to proceed to other jobs before the manifestation is complete. As a result of this behavior, and others allowed by our model [12] the monitor we have

specified above "timeshares" automatically. The only contention within the system is among the device drivers.

We close this section with an example of a job which is an interactive interpreter for our language. The function for the job is interpret. Two input files are needed, a list of forms and a single environment. The first is bound to the user's keyboard; the second is taken as the system's global environment in effect when the job was started. The single output file will be sent to the printer or the teletype. This is just one job; many jobs like this may be specified (for different teletypes) and may be active in a time-shared mix.

```
<interpret[forms env]> ≡
  if <null[forms]> then []
  else <append[ [<first[forms]> NL
                 <eval[<first[forms]> env]> NL PR]
                 <interpret[<rest[forms]> env]> ]>  .
```

Conclusion

We have indicated how the techniques of first-order applicative programming, in combination with suspended evaluation semantics for the constructor function, may be used to implement a file system. In order to provide a convenient facility for multiple output from a file building procedure, the language enhancement known as functional combination is provided. Each file is created and sent to its output device in suspended form; computation therefore proceeds in response to and in a sequence determined by requests for text characters from these devices, and evaluation proceeds only far enough to determine what the next output character will be.

A randomly accessed file may never exist in its manifest form, only as promises which get updated from time to time. This observation holds regardless of whether the file is built as a linear or a tree-like structure and it may intimidate the user who grew up with manifest files. But just as the user who grew up with punched cards learned to entrust his file to magnetic storage media when he was convinced it would be there when he needed it, so also will the user accustomed to manifest files learn to trust promised ones whenever he becomes convinced that his information will be there when he uses it.

The new file semantics permits the text editor program to exhibit the desired sequence of behavior while remaining applicative in form; without suspended evaluation an applicative program could not be made to pause with a partial result. The use of random access storage devices makes it possible to postpone the explicit creation of the contents of a file until they are needed by another program; at that point only those portions accessed by the other program will be manifested.

We have indicated that, in an I/O bound environment, an operating system could be written to take advantage of the fact that the top-level programs are output-driven to help determine how to allocate computational resources. As with the text editor the possibility of an applicative formulation of a system is a result of the suspensions since they permit the system to produce output before receiving all its input.

Applicative formulations of algorithms have the advantages of expressiveness and reliability. Formerly, the class of procedures characterized by an interactive process requiring intermittent input, with output produced in response to each input segment, was beyond the scope of this approach. As we have shown with the example of the text editor, this is no longer the case; proponents of applicative programming styles now have the capacity to employ their method in a broader area.

The language we have used is described elsewhere in more complete detail [9, 10, 12]. It has been implemented within LISP and as a separate interpreter written entirely in PASCAL [31]. Much development remains before the possibilities for parallelism [12] and promised (random-access) files is realized. The implementation in its present state, however, produces all of the syntactic behavior, all of the semantic behavior, and some of the pragmatic behavior described in this paper.

Acknowledgement

We thank Elliott Organick, Cynthia Brown, Eric Norman and Mitchell Wand for their careful reading of the manuscript and useful suggestions.

REFERENCES

[1] G.M. Adel'son-Vel'skii and Y.M. Landis. An algorithm for the organization of information. Dokl. Akad. Nauk SSR 146, 263-266. Also Soviet Math. Dokl. 3 (1962), 1259-1262.

[2] J. Backus. Programming language semantics and closed applicative languages. Proc. ACM Symp. on Principles of Programming Languages (1973), 71-86.

[3] R. Boyer and J.S. Moore. Proving theorems about LISP functions. J.ACM. 22, 1 (January, 1975), 129-144.

[4] W.H. Burge. Recursive Programming Techniques, Addison-Wesley, Reading, MA (1975).

[5] E.W. Dijkstra. Co-operating sequential processes. In F. Genuys (ed.), Programming Languages, Academic Press, London (1968), 43-112.

[6] D.I. Good, R.L. London and W.W. Bledsoe. An interactive program verification system. IEEE Trans. on Software Engineering SE-1, 1 (March, 1975), 56-67.

[7] J.V. Guttag, E. Horowitz and D.R. Musser. Abstract data types and software validation, Report ISI/RR-76-48, Information Sciences Institute, Univ. of Southern California (August, 1976).

[8] D.P. Friedman. The Little LISPer, Science Research Associates, Palo Alto (1974).

[9] D.P. Friedman and D.S. Wise. An environment for multiple-valued recursive procedures. 2me Colloq. Intl. sur la Programmation, Springer-Verlag, Berlin (to appear).

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

[11] D.P. Friedman and D.S. Wise. Output driven interpretation of recursive programs, or writing creates and destroys data structures. Information Processing Letters (to appear).

[12] D.P. Friedman and D.S. Wise. The impact of applicative programming on parallel processing. Proc. Intl. Conf. on Parallel Processing, IEEE cat. no. 76CH1127-0C (1976), 263-272.

[13] D.P. Friedman and D.S. Wise. Functional combination. Technical Report 27, Computer Science Dept., Indiana University (1976).

[14] D.P. Friedman, D.S. Wise and M. Wand. Recursive programming through table look-up. Proc. ACM Symp. on Symbolic and Algebraic Computation (1976), 85-89.

[15] P. Henderson and J. Morris, Jr. A lazy evaluator. Proc. 3rd ACM Symp. on Principles of Programming Languages (1976), 95-103.

[16] C.E. Hewitt and B. Smith. Towards a programming apprentice. IEEE Trans. on Software Engineering SE-1, 1 (1975), 26-45.

[17] C.A.R. Hoare. Towards a theory of parallel programming. In C.A.R. Hoare and R.H. Perrott (eds.), Operating Systems Techniques, Academic Press, London (1972), 61-71.

[18] C.A.R. Hoare. Monitors: an operating system structuring concept. Comm. ACM. 17, 10 (October, 1974), 549-557.

[19] D.E. Knuth. Sorting and Searching, Addison-Wesley, Reading, MA (1973), 251-252.

[20] P.J. Landin. A correspondence between ALGOL 60 and Church's lambda notation. Comm. ACM. 8, 2 (August, 1965), 89-101.

[21] P.J. Landin. The next 700 programming languages. Comm. ACM. 9, 3 (March, 1966), 157-162.

[22] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part I. Comm. ACM. 3, 4 (April, 1960), 184-195. Also in S. Rosen (ed.), Programming Systems and Languages, McGraw-Hill, New York (1967), 455-480.

[23] J. McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg (eds.), Computer Programming and Formal Systems, North Holland, Amsterdam (1963), 33-70.

[24] J. McCarthy, P.W. Abrahams, D.J. Edwards, T.P. Hart, and M.E. Levin. LISP 1.5 Programmer's Manual, MIT Press, Cambridge, MA (1962).

[25] S.A. Papert. Teaching children to be mathematicians versus teaching about mathematics. Int. J. Math. Educ. Sci. Tech. 3 (1972), 249-262.

[26] J.C. Reynolds. GEDANKEN -- a simple typeless language based on the principle of completeness and the reference concept. Comm. ACM. 13, 5 (May, 1970), 308-319.

[27] R.D. Tennent. The denotational semantics of programming languages. Comm. ACM. 19, 8 (August, 1976), 437-453.

[28] J. Vuillemin. Correct and optimal implementation of recursion in a simple programming language. J. Comp. Sys. Sci. 9, 3 (1974), 332-354.

[29] C. Wadsworth. Semantics and Pragmatics of Lambda-calculus, Ph.D. dissertation, Oxford (1971).

[30] B. Wegbreit. Mechanical program analysis. Comm. ACM. 18, 9 (September, 1975), 528-539.

[31] N. Wirth. The programming language PASCAL. Acta Informatica 1, 1 (1971), 35-63.