SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 8, 407-416 (1978)

Unbounded Computational Structures*

DANIEL P. FRIEDMAN AND DAVID S. WISE
Computer Science Department, 101 Lindley Hall, Indiana University, Bloomington, Indiana 47401, U.S.A.

SUMMARY

The concept of suspended evaluation is used as an approach to co-routines. Problems from the literature involving infinite data structures are solved in a LISP-like applicative language to demonstrate that simple new semantics can enrich old and 'friendly' control structures. It appears that the very nature of these problems draws control structure and data structure together, so that issues of style may be studied at once for both.

KEY WORDS Co-routines LISP Data structures Control structures Programming style

INTRODUCTION

The advocates of structured programming (a term meant to include issues of control as well as data structures) have raised serious questions of programming style. The desire to work with a small, well-organized set of control and data structures has attracted much attention to revealing this small set. Function invocation has long been regarded as essential to structured programming, but the restrictions on function invocation inherent in structured programming have been sorely neglected by those advocating extensions, such as co-routining and multitasking, to the concept of function.

Co-routines can be thought of as functions which can stop in the middle of processing and be restarted at the whim of the programmer. The difficulty associated with co-routines and other structures of this type (frames,[1] objects,[2] etc.) is not one of power, but one of protection. The opportunity for mischief with such models is astounding. In this paper we propose a different approach to co-routines, wherein resumable structures are created by the system but may not be directly manipulated by the user. Thus, the issue of protection disappears while the useful features of such structures are retained.

We have proposed semantics for building data structures which subsumes this sort of control structure into data manipulation.[3] Embedded in LISP, this semantic requires only a slight modification of the user's perspective on the function *cons* which builds a data structure, and on the functions *first* and *rest* which access it. As a result, the program structures which parallel co-routines are hidden from the user, who is unable to manipulate them directly. The only way his programming is affected is by ***knowledge of their existence.*** We argue that this knowledge alone is sufficient to permit clear, structured applicative programs for many classic examples of co-routines, and that the simplicity of such programs will make this program strategy accessible to more users. These constructs permit an approach to infinite structures treated as objects rather than as processes. Older conceptualizations of this representational issue utilize various programming structures (like functional closures,[4] objects,[2] co-routines,[5] dynamic lists,[6] streams,[7] frames,[1,8] generators[8,9] and actors[10]). We shall skirt these ideas while explaining the new approach. The goal of this

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

0038-6644/78/0408-0407$01.00 *Received 22 March 1977*

paper is threefold: it surveys, through examples, recent results on suspended[3] or lazy[11] evaluation of applicatively written programs; it reviews divers esoteric control structures as they are commonly applied to infinite structures; and it draws together aspects of data structures and of control structures by treating difficult problems from both perspectives.

The remainder of this paper is divided into four parts. The first reviews the elements of a stylized applicative programming language[12] we use in expressing programs (we adopt some simplifications) with particular emphasis on the suspended *cons*truction of data structures. In the second section an example, the construction of the list of *all* natural numbers, is explained in some detail from the perspective of co-routines. The third section centres on another example which, written as a sequential (as opposed to applicative) program, requires an unbounded number of processes: a closed form solution for the sieve of Eratosthenes for the list of *all* prime numbers. The fourth section presents several solutions for problems which have been solved using some flavour of co-routines: comparing the contents of a list, merging binary trees, Hamming's problem on composites of three primes and, from the calculus, power sequences and series including Newton's approximation to square roots. The conclusion includes observations on the implications of suspended evaluation as a control structure and as a data structure.

THE LANGUAGE

The language we shall use here is a derivative of LISP, Reference 13, Chapter 1. It has been implemented over LISP on the DEC-10 and then in PASCAL and runs on the CDC 6600.[30] The only data structure is a parenthesized list. Lists are sequential structures of elementary items (numbers or identifiers), or other lists. A system predicate, *atom*, is provided for identifying an elementary item; others test equality of structures. In particular, *null* tests for the empty list: ( ). Most important are the list manipulation primitives: *cons*, *first* and *rest*. The list constructor *cons* takes two arguments, the second of which is always a list; it returns the list which results from adding the first parameter at the left of the list. Just as *cons* builds a list, *first* and *rest* access it; *first* extracts the leftmost item from a non-null list and *rest* returns the remainder of such a list with the first item omitted.

The syntax we use for function invocation is also the S-expression of LISP. The form (*f a b c*) is interpreted as the function *f* invoked on three arguments, *a*, *b* and *c*, which are each evaluated before being used. In defining functions we shall equate an invocation pattern with a conditional expression represented by a sequence of forms separated by the commenting keywords **if**, **then**, **else** and **elseif**. For instance, the definition of *append*, which concatenates two lists, appears as

```
(append a b) ≡
  if (null a) then b
  else (cons (first a) (append (rest a) b)).
```

To enhance readability we use bracketed expressions in place of LISP's *list*. [ ] evaluates to ( ) and [*a b c*] evaluates to (1 2 3) assuming that *a* is bound to 1, *b* to 2 and *c* to 3 in the active environment. We use angle brackets (like Backus[14] and Hewitt[10]) in place of LISP's *apply*. The two items within the brackets are a function and its list of arguments. Thus, (*f a b c*) is synonomous with ⟨*f*[*a b c*]⟩.

A new syntactic structure we introduce is the 'starred' list. A starred structure in parentheses represents an infinite homogeneous list of the indicated item.[12,15] For instance (0*) is the zero vector of unbounded dimension. Since numbers always evaluate to themselves, [1*] evaluates to (1*) or (1 1 1 1 ...). Such infinite structures are most interesting when we extend them to starred functions. In our scheme structures in the functional position indicate instances of functional combination.[15] A starred function is 'spread across' its arguments much like a multiple-argument MAPCAR in LISP. If f is a function of three arguments and a, b and c are infinite lists, bound to the values of $[a_1\ a_2\ a_3 \ldots]$, $[b_1\ b_2\ b_3 \ldots]$ and $[c_1\ c_2\ c_3 \ldots]$ respectively, then $([f^*]\ a\ b\ c)$ gives the value of

$$[(f\ a_1\ b_1\ c_1)\ (f\ a_2\ b_2\ c_2)\ (f\ a_3\ b_3\ c_3) \ldots].$$

For example, if *nn* is bound to the list of integers (1 2 3 4 ...), then ([sum*] *nn nn*) evaluates to the infinite list of positive even integers and ([sum*] *nn* ([sub1*] *nn*)) evaluates to the odds.

The essence of our approach to infinite structures is implicit in the concept of suspended evaluation. It is easily implemented by changing the semantics of the elementary functions *first*, *rest*, and *cons* from the classic call-by-value to call-by-need[16] or call-by-delayed-value.[17] Instead of an invocation of *cons* placing the ultimate values of its two arguments in the node allocated for its result, each field is filled with a *suspension*, a distinguishable structure containing the unevaluated argument and the current environment. Conceptually, the value of a call on *cons* is unchanged. When either of the two projection functions, *first* and *rest*, is invoked on that value the appropriate argument will be evaluated in the preserved environment. The resulting value is returned and takes the place of the suspension within the extant node. Future accesses will find the value in place of the suspension, so the net effect is that evaluation is postponed but not repeated. As a result, the LISP evaluator is invoked no more often than it is under McCarthy's classic interpretation of these elementary functions, and in many instances it is called significantly less. When the language interpreter itself is implemented using these functions the least fixed-point semantics for pure LISP result. More important to the ideas presented here, no user data structure is built until its existence is essential to the course of the computation; even if the user 'constructs' an infinite structure within his program only the finite part he uses will be manifested.

The new semantics for *cons* extends its role as the creator of data structures.[18] The significant feature of the semantics for manipulating data structures is that no structure is created until it is ultimately necessary to the computation. We will not carry this to the extreme in which all structures belonging to the system are suspended; here we only propose to suspend construction of structures specified by the user's code. The behaviour of his code is nevertheless drastically altered, since any evaluation, even from within a probe, stops on an application of *cons* as surely as if it were a constant function. Arguments are held for evaluation if and when they are actually used.

Even more significant is the conceptual treatment of structures by the programmer. In applying *cons* to build a structure, and later using *first* and *rest* to access it, he conceives the structure as being manifest in a form little different from the directed graph model of data structures. In fact there may be 'buds' in the physical representation of his structure where he envisions trees; structure which is unused remains ungrown. This property is of particular significance when a suspension represents an infinite structure, which will grow nicely within the user's capacity to use parts of the structure. He cannot use it all! The user is free to perceive the infinite structure as an infinite graph if he wishes.

The idea of a suspension, particularly of a linear list, is closely related to many structures from the literature. Landin[7] proposed the stream model which has been described in some detail by Burge.[19] In this type of list only the *rest* of a list may be suspended, and coercion of that suspension must not return the empty list. Using other terminology, *cons* is strict[17] in its first argument[3] and non-strict in its second. As a result, access into a stream will manifest the prefix of the linear structure up to and including the information sought.

Another perspective on the stream model is available through co-routines. Associated with each stream is a process (which can be activated to generate more of the tail whenever it is needed). That process is very much like the generator of IPL-V[9] or of CONNIVER[8] which is explicitly established and then implicitly invoked by accessing the structure. The dynamic list in POP-2[6] is a similar feature. Henderson and Morris,[11] however, describe a system which is substantially the same as our views of *cons*; the user is not asked to specify for infinite structures any more than is expressed when constructing ordinary finite structures.

THE NATURAL NUMBERS

The first example we shall develop is the definition of the list of all natural numbers. There are two approaches to this problem using suspended *cons*. The first is the definition of a program which is expressed as a recursive function, and the second is the direct definition of the data structure itself. For this example the second is the more concise, but it is less general as we shall see in the next section.

Let us define a function *fromheretoeternity* which takes an integer as its argument and returns the ordered list of all integers greater than or equal to it[6, 11, 19, 20, 21]

(*fromheretoeternity* i) ≡
 (*cons* i (*fromheretoeternity* (*add*1 i))).

Then the natural numbers, *nn*, may be defined as

nn ≡ (*fromheretoeternity* 1).

Because termination tests are not needed, the mental process of writing these definitions is even simpler than that required for writing familiar recursive functions like *factorial* once the programmer stops worrying about how the machine handles infinite structures. It is best to forget about the semantics of *cons*, accepting its ability to build Towers of Babel as long as the blueprints make sense. Then the creation of the function and the correctness of *nn* follow from the inductive definition of the natural numbers:

1 is a natural number;

if n is a natural number **then** so is $n+1$.

The semantics of *nn* may be viewed in two ways. Since it is homogeneous, the view of it as a generator or as the result of a co-routine activated on any call is perfectly satisfactory. In our model the entire structure is represented by the portion that has been used by the program at the time that a snapshot is taken.

Another definitional style treats *nn* as a data structure defined recursively upon itself without any new function definition. This style depends heavily upon the use of functional combination (the use of *'s here) to imply applications of *cons*. LISP allows the use of *MAP*ping functions for similar purposes, avoiding the necessity for additional function definitions through a specific language feature.

This recursion scheme most easily handles recursive definitions of infinite lists in which the value of the $(n+k+1)$th element depends only on the nth through $(n+k)$th elements. This restriction should be compared with the problem considered in the next section which involves a course-of-values recursive definition. For *nn*, however, the definition follows easily with $k = 0$.

nn ≡ (*cons* 1 ([*add*1*] *nn*)).

This definition recurses on the definition of the list *nn* just as the definition of the function *fromheretoeternity* recurses on itself. In both cases the definition is easiest to read under the assumption that the structure (be it data or function) being defined already exists.

The term **letrec**[22] has been used in the literature to establish environments for such recursive definitions. They have classically been used to define functions in an environment within which they were already assumed to be defined. The identical techniques work for defining data structures. The definition of *nn* immediately above could be written as

```
(label nn (cons 1 ([add1*] nn)))
```

and run under the interpreter for suspending *cons* given in the appendix of Reference 3. Thus, the expressiveness of data structures and program structures is nearly the same.

The ultimate user of structures is the output portion of the program. In traversing the structure which represents the result of a program (the top level function call) precisely those parts of intermediate structures are created which are essential to the ultimate result. Elsewhere[20] we consider a specific output driver which runs the system evaluator to print out the value of *nn*, under either definition given here, within the resources of the 'best' sequential program: finite memory and a lot of paper.

SIEVE OF ERATOSTHENES

The example presented in this section is a closed form expression of the sieve of Eratosthenes for generating *all* prime numbers. This algorithm is often expressed in sequential languages as generating a finite number of primes.

'Being interested only in terminating programs, we shall make a program generating only the, say, first 1000 values of the sequence.'

Dijkstra (Reference 23, p. 129).

In either context the algorithm may be expressed as sifting primes from the list of integers by repeatedly deleting the first element, a prime, from the list and eliminating all its multiples from the list. When the list is bounded then the list transformation is a finite process and may be completed as each prime is discovered.

Our code is similar to that attributed to Quarendon,[11] but the only arithmetic operation we use is addition. The structure of the applicative expression of the program is similar to the algorithm outlined above but the data structures used are perceived as infinite[11, 21, 24]

```
primes ≡ (sieve (fromheretoeternity 2)).
```

The integers strictly greater than one will be sifted, as one is not a prime.

```
(sieve nums) ≡
    (cons (first nums)
          (sieve (removemult (first nums)
                             (twice (first nums))
                             (rest nums)))).
```

One call of the function *sieve* identifies the first element of the list as a new prime which is *cons*tructed onto the rest of the primes, which may be obtained by sifting all multiples of the newly discovered prime, starting with its double, from the list of candidates at this point. We intend (twice *n*) to mean (sum *n n*); a programmer tolerant of multiplication and familiar with elementary number theory would use (square *n*) instead.

```
(removemult inc mult lis) ≡
  if (less? (first lis) mult)
    then (cons (first lis) (removemult inc mult (rest lis)))
  elseif (greater? (first lis) mult)
    then (removemult inc (sum mult inc) lis)
  else (removemult inc mult (rest lis)).
```

In order to remove all multiples of a number, *inc*, greater than or equal to a candidate multiple, *mult*, from a list, *lis*, three cases must be considered. In the first case the first item on the list is too small so it is prefixed to what results from processing the rest of the list. Secondly, if that first item is larger than *mult*, then the process should run on the same list with the next higher multiple. Finally, if the first item is the multiple sought, then the answer is what results from processing only the rest of the list.

That is, for each prime discovered, another sifting incantation of *removemult* is created to filter the initial list of integers. Each one becomes a process which lines up on the path from the generator of all the integers to the printing device. Under our semantics, however, the user need not be conscious of the creation of all these processes; that is implicit in his use of *cons*. He need not be aware of their activation and interaction; that is implicit in his use of *first* and *rest*.

This example, then, demonstrates how *multiple* co-routines may be created without appeal to generators or explicit process creation. However, since the next item on the list of primes depends on all its predecessors (course of values recursion) the facility of functional combination to define data structures directly does not appear to be useful. That control structure does not extend to models which mimic an unbounded number of processes.

In closing this section we must make one negative comment. Of course no implementation can generate primes forever in the same way that integers may be listed to the limit of one's paper supply. In printing *nn* the number of extant suspensions is bounded.[20] In printing *primes* the number of suspensions grows with the output until storage capacity is exceeded. Therefore, the list of primes actually printed may be comparable in length to the results from bounded sequential implementations of the sort mentioned at the beginning of the section. The difference is that the bound is not explicit in the algorithm itself.

FURTHER EXAMPLES

In this section we present several more examples demonstrating the creation of infinite data structures. Several of the problems are taken from the literature, but we have rewritten the solutions to take advantage of suspensions created by *cons*. In reviewing these programs it is useful to notice that syntactically there is no difference from the classic use of *cons*[3, 13] and that the only semantic change is the existence of suspensions. Since the user is unable to manipulate suspensions, they have no effect on his programming style except to give him confidence that no structure will be built prematurely.

The first example is due to Hewitt[25] by way of Henderson and Morris,[11] who showed that the obvious solution in LISP exhibits the desired behaviour under their new semantics. The problem is to determine whether the elementary items of two list structures are equal when compared by simultaneous traversals. A classic solution might read

```
    (samefringe? a b) ≡
      (eqlis (collapse a) (collapse b)).
    (collapse lis) ≡
      if (null lis) then [ ]
      elseif (atom lis) then [lis]
      else (append (collapse (first lis))
                   (collapse (rest lis))).
```

where *append* is taken from the language description section and *eqlis* is true if its arguments are two identical lists of atoms. If it is to perform as desired, the function must traverse the lists *a* and *b* no further than absolutely necessary; when two different elementary items are found in corresponding positions then the rest of the list should not be processed at all. Under McCarthy's semantics the *collapse* of both lists is completed before any comparison is made; under the semantics of suspensions (used by *cons* in *append*) no more of the collapsed lists are built than are required by *eqlis*, which requires no traversal beyond the first difference. Therefore, the user need not change his programming habits in order to achieve the multitasking performance.

A related example is the merging of the items in two or more binary search trees into a single sorted sequence.[26] The solution requires only a merging algorithm for the sorted lists.

```
    ⟨mergetrees listoftrees⟩ ≡
      ⟨mergelists ([collapse*] listoftrees)⟩.
    ⟨mergelists lists⟩ ≡ ⟨merge (sort lists)⟩.
```

The sort is in the order of the first elements.[27]

```
    (sort lists) ≡
      if (null lists) then [ ]
      else (insert (first lists) (sort (rest lists))).
    (insert alist lists) ≡
      if (null lists) then [ ]
      elseif (less? (first alist) (first (first lists)))
        then (cons alist lists)
      else (cons (first lists) (insert alist (rest lists))).
    ⟨merge sortedlists⟩ ≡
      if (null sortedlists) then [ ]
      elseif (singleton? (first sortedlists))
        then (cons (first (first sortedlists))
                   ⟨merge (rest sortedlists)⟩)
      else (cons (first(first sortedlists))
                 ⟨merge (insert (rest (first sortedlists))
                                (rest sortedlists))⟩).
```

As in the previous example, the use of suspended *cons*, explicitly here and implicitly in *append*, prevents the intermediate results of *collapse* from ever being entirely present in the system.[20]

The next example is credited by Dijkstra (Reference 23, p. 129), to Hamming:

To generate in increasing order the sequence 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, ... of all numbers divisible by no primes other than 2, 3, or 5.

The solution (due to G. Kahn and D. MacQueen[28]) is fairly easy following the discussion above:

```
composites235 ≡
  (cons 1 (remdup (merge ([product*] [2*] composites235)
                         ([product*] [3*] composites235)
                         ([product*] [5*] composites235)))).
(remdup lis) ≡
  if (same (first lis) (first (rest lis)))
    then (remdup (rest lis))
  else (cons (first lis) (remdup (rest lis))).
```

This solution is remarkable partly because of the way in which it uses the function *merge*, which we just finished defining as a help function to operate on an argument list of finite lists. In this application the arguments are infinite so that the first two conditions in merge *never* succeed. The *else* condition is the only relevant one, and it applies as well to the infinite case. As before, the correct program for infinite structures turns out to be less difficult than the corresponding problem solved for finite ones!

We close with two examples which point the way toward applications of these concepts in the calculus. The first is a simple power sequence based on an arbitrary function f. Given an argument x we would like the sequence

$$(x, (f\,x), (f\,(f\,x)), (f\,(f\,(f\,x))), \ldots).$$

The solution is little more difficult than the concept:

```
sequence ≡ (cons x ([f*] sequence)).
```

which might well be parameterized (Reference 19, p. 134):

```
(sequence f x) ≡ (label sss (cons x ([f*] sss))).
```

This sort of problem is easily generalized into a series, where a series is the list of partial sums of a sequence. First we express the list of all (reversed) initial segments of the sequence:

```
segments ≡ ([cons*] sequence (cons [ ] segments)).
```

If *sequence* were *nn* then segments would be ((1) (2 1) (3 2 1) ...). We apply addition and parameterize to get the series function:

```
(series seq) ≡
  ([total*] (label ss ([cons*] seq (cons [ ] ss)))).
(total addends) ≡ ⟨sum addends⟩.
```

Thus infinite sums are programmed from infinite sequences just as the analysts define series from sequences.

Finally, we consider the infinite list of approximations to the square root of y, as generated by Newton's method. If we take f to be defined by $(f\,x) = (y/x+x)/2$ with x initially 1, then the solution is given by the sequence generator above. Perhaps a clearer solution for the same problem is (*Newton y* 1) where

```
(Newton y app) ≡
  (cons app (Newton y (half (sum (quotient y app) app)))).
```

Infinitesimals and convergence are fundamental to the analysts and grief to the programmer. Given these structures, however, the problems take on a new and pleasant flavour. In dealing with an approximation, say to the square root of 2, the programmer may select some element from a series generated by *Newton* and carry out a computation with the knowledge that more precision is available by choosing the next element.[29]

CONCLUSION

We are not the first advocates of applicative programming, nor are we the first to use co-routines to construct infinite data structures. We have simply surveyed the use of infinite data structures from the perspective of suspensions, which become part of the user's data structure whether he likes it or not. It appears that the mystery of co-routining vanishes when the control structure is thereby removed from the user's 'tool kit'. Without the decision on whether or not to use partial evaluation he is able to write straightforward source code which runs under new, but consistent, semantics.

Program structure and data structure merge in our view, so that issues of 'style' or 'structure' may be handled at once for both program and data. In order to express time-dependent structures, like the 'infinite' structures we consider, one must only *know* of the existence of suspensions and use the classic language (e.g. LISP). Well worn axioms of style and structure are then available for new classes of problems, and the quirks of these new problems will help refine old conventions by identifying important generalizations.

REFERENCES

1. D. G. Bobrow and B. Raphael, 'New programming languages for artificial intelligence research', *Comput. Surv.* **6**, No. 3, 153–174 (1974).
2. O.-J. Dahl and K. Nygaard, 'SIMULA—an ALGOL based simulation language', *Comm. ACM*, **9**, No. 9, 671–678 (1966).
3. 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 University Press, Edinburgh, 1976, pp. 257–284.
4. E. Sandewall, 'A proposed solution to the FUNARG problem', *ACM SIGSAM Bull.* No. 17, 29–42 (1971).
5. M. E. Conway, 'Design of a separable transition-diagram compiler,' *Comm. ACM*, **6**, No. 7, 396–408 (1963).
6. R. M. Burstall, J. S. Collins and R. J. Popplestone, *Programming in POP-2*, Edinburgh University Press, Edinburgh, 1971.
7. P. J. Landin, 'A correspondence between ALGOL 60 and Church's lambda notation, part I', *Comm. ACM*, **8**, No. 2, 89–101 (1965).
8. G. J. Sussman and D. V. McDermott, 'From CONNIVER to PLANNER, a genetic approach', *Proc. FJCC*, AFIPS Press, Montvale, N.J., 1970, pp. 1171–1179.
9. A. Newell, F. M. Tonge, E. A. Feigenbaum, B. F. Green, Jr. and G. H. Mealy, *Information Processing Language V Manual*, Prentice-Hall Inc., Englewood Cliffs, N.J., 1964.
10. C. Hewitt and B. Smith, 'Towards a programming apprentice', *IEEE Trans. on Software Engineering*, **SE-1**, No. 1, 26–45 (1975).
11. P. Henderson and J. Morris, Jr., 'A lazy evaluator', *Proc. 3rd ACM Symp. on Principles of Programming Languages*, pp. 95–103 (1976).
12. D. P. Friedman and D. S. Wise, 'The impact of applicative programming on multiprocessing', *Proc. 1976 Int. Conf. on Parallel Processing* (*IEEE Cat. No. 76CH1127-OC*), pp. 263–272, also *IEEE Trans. Comput.* (to appear).
13. J. McCarthy, P. W. Abrahams, D. J. Edwards, T. P. Hart and M. E. Levin, *LISP 1.5 Programmer's Manual*, M.I.T. Press, Cambridge, Mass., 1963, Chapter 1.
14. J. Backus, 'Programming language semantics and closed applicative languages', *Proc. ACM Symp. on Principles of Programming Languages*, pp. 71–86 (1973).
15. D. P. Friedman and D. S. Wise, 'Functional combination', *Computer Languages*, **3**, No. 1, 31–35 (1978).
16. C. Wadsworth, 'Semantics and progmatics of lambda-calculus', *Ph.D. dissertation*, Oxford University (1971).
17. J. Vuillemin, 'Correct and optimal implementation of recursion in a simple programming language', *J. Comp. Sys. Sci.* **9**, 332–354 (1974).
18. C. A. R. Hoare, 'Recursive data structures', *Int. J. Comput. & Inf. Sci.* **2**, 105–132 (1975).

19. W. H. Burge, *Recursive Programming Techniques*, Addision-Wesley, Reading, Mass., 1975.
20. D. P. Friedman and D. S. Wise, 'Output driven interpretation of recursive programs, or writing creates and destroys data structures', *Information Processing Letters* **5**, No. 6, 155–160 (1976).
21. D. P. Friedman, D. S. Wise and M. Wand, 'Recursive programming through table look-up', *Proc. ACM Symp. on Symbolic and Algebraic Computation*, pp. 85–89 (1976).
22. P. J. Landin, 'The next 700 programming languages', *Comm. ACM*, **9**, No. 3, 157–162 (1966).
23. E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall Inc., Englewood Cliffs, N.J., 1976, Chapter 17.
24. D. R. Hanson, 'Procedure-based linguistic mechanisms in programming languages', *Ph.D. dissertation*, University of Arizona (1976).
25. C. Hewitt, P. Bishop, R. Steiger, I. Greif, B. Smith, T. Matson and R. Hale, 'Behavioral semantics of nonrecursive control structures', in B. Robinet (Ed.), *Programming Symposium*, Springer-Verlag, Berlin, 1974, pp. 385–407.
26. A. Wang and O.-.J. Dahl, 'Coroutine sequencing in a block structured environment', *Nordisk Tidskr. Informationsbehandling* (*BIT*), **11**, 425–559 (1971).
27. D. P. Friedman and D. S. Wise, 'Aspects of applicative programs for file systems (preliminary version)', *Proc. ACM Conf. on Language Design for Reliable Software*, SIGPLAN Notices, **12**, No. 3, 41–55 (1977).
28. G. Kahn and D. MacQueen, 'Coroutines and networks of parallel processes', in B. Gilchrist (Ed.), *Information Processing 77*, North-Holland Publishing Co., Amsterdam, 1977, pp. 993–998.
29. E. A. Ashcroft and W. W. Wadge, 'LUCID—a formal system for writing and proving programs', *SIAM J. Comput.* **5**, No. 3, 336–354 (1976).
30. S. D. Johnson, 'An iterative model for a language based on suspended construction', *M.S. thesis*, Indiana University (1977).