Purdue University

Purdue e-Pubs

Department of Computer Science Technical Reports

Department of Computer Science

1986

# Evaluating Storage Management Schemes for Block Structured Languages

Thomas P. Murtagh

Report Number:

86-570

Murtagh, Thomas P., "Evaluating Storage Management Schemes for Block Structured Languages" (1986). *Department of Computer Science Technical Reports.* Paper 489.
https://docs.lib.purdue.edu/cstech/489

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries.
Please contact epubs@purdue.edu for additional information.

Evaluating Storage Management Schemes for Block Structured Langauges

by

Thomas P. Murtagh

Computer Science Department
Purdue University
West Lafayette, IN 47907

*Abstract*: The conventional storage allocation scheme for block structured languages requires the allocation of stack space and the building of a display with each procedure call. Several techniques have been proposed for analyzing the call graph of a program that make it possible to eliminate these operations from many call sequences. In this paper, we compare these techniques, and propose an improved allocation scheme which can substantially reduce allocation overhead, even in the presence of recursion and support for separate compilation.

CSD-TR-570
January 1986

## Introduction

Most compilers for languages that support recursion delay the allocation of storage for a procedure until the point at which it is invoked. In addition, if the language supports nested procedure declarations, instructions to update the display are executed at the time of each call [1]. These actions consume a significant amount of processor time, even on architectures optimized to support procedure calls [13,17]. Current trends in programming methodology, which call for the construction of large programs out of many small procedures [7,26], only promise to make the overhead of procedure calling more significant.

Several schemes have been proposed that would enable compilers to generate more efficient calling sequences for many procedures [8,16,25]. Each of these schemes requires the compiler to build an approximation to the program's call graph which is then analyzed to identify procedures that can be called using more efficient mechanisms.

There are two aspects of the problem of reducing procedure call overhead that have not been adequately considered by previous efforts. First, while it appears likely that any of the schemes proposed would improve the performance of the code generated by a compiler, the magnitude of the improvement has not been experimentally measured. Second, none of the previous proposals have carefully considered the effect of separate compilation on the schemes they proposed. Because all the proposed schemes depend on information about the entire program's call graph, it is not clear that any of them could be used effectively in a translator that supported separate compilation. Support for separate compilation, however, is essential in a production quality compiler. The definitions of many languages require support for separate compilation [9,15,23]. In cases where the language being translated does not include mechanisms for separate compilation in its definition, the importance of separate compilation for the development of large software projects often leads to the design and implementation of separate compilation mechanisms as extensions to the language [4,11,14].

In this report we present work which addresses these issues. First, in section 1 we discuss the problems that arise when one attempts to use one of the previously suggested allocation schemes in a translator that supports separate compilation. Then, in section 2, we present a new scheme and in section 3 explain how it can be extended to function well in the presence of support for separate compilation. Finally, in section 4 we outline the data we have collected to evaluate the expected performance of our scheme and

explain the methods used to collect this data.

## 1. Previous Allocation Schemes and Separate Compilation

The simplest alternative to allocating storage at every procedure call is to statically allocate space for all procedures that the compiler can determine will never be called recursively [25]. Since most programs make little use of recursive procedures, one would expect this simple scheme to significantly reduce procedure call overhead in most programs. In fact, as we will discuss later, this scheme works almost as well as significantly more complex schemes even on programs that use recursion extensively.

The two other previously proposed schemes we wish to consider rely on finding intervals in the call graph of the program [8,16]. An interval is a subgraph of a directed graph with a unique entry point called its header such that every cycle in the subgraph includes the header [2]. If the call graph of a program is partitioned into intervals, it is possible to eliminate allocation operations from the call sequences of all procedures that are not headers by preallocating space for all of the procedures in an interval each time its header is invoked.

The two interval based schemes partition the call graph into intervals in different ways. The first of these schemes [8] uses a partition in which every recursive procedure is treated as a header, while the other interval based scheme [16] simply applies a well known algorithm that partitions the call graph into maximal intervals [2]. Based on this difference, we will refer to the first scheme as the non-recursive interval scheme and the second as the maximal interval scheme. In many cases the maximal interval scheme can eliminate more allocation operations. For example, if A and B are mutually recursive procedures such that B is only called by A, A and B would both be headers in the non-recursive interval scheme, but only A would be a header in the maximal interval scheme.

There are several other respects in which the two interval schemes differ, two of which are particularly significant to our concerns. First, while no provisions for separate compilation were discussed in the presentation of the maximal interval scheme, the non-recursive interval scheme was used in a compiler that supported separate compilation. This was accomplished by simply requiring that all procedures that could be called externally be included in the set of interval headers. Unfortunately, this approach is extremely sensitive to the number of procedures that can be called externally. Suppose, for example, that Fig. 1

shows a part of the call graph of a program containing procedures A, B, C, D and E and that these procedures form a module that is compiled separately from the rest of the program. When applied to this module, the approach used by the non-recursive interval scheme works well. Only A is treated as a header. If, however, one additional procedure, B, were accessible externally, all of the procedures with the exception of E would become interval headers. In the worse case, where the compiler must assume that any procedure could be called externally, every procedure in every module would be treated as a header.

The other important difference between the two schemes is that the maximal interval scheme statically allocates space for all procedures in intervals whose headers are non-recursive. To understand the importance of this feature consider the behavior of the static allocation scheme compared to the behavior of a scheme that uses maximal intervals but does no static allocation when both schemes are applied to the call graph shown in Fig 2. Both would allocate space for A and B at program initiation. The static scheme would also allocate space for E at this time, while the maximal interval scheme would be forced to treat E as a header and allocate space for E every time it was called. On the other hand, the maximal interval scheme would avoid allocating space for D when it was called by preallocating space for it whenever C, the header of its interval, was called. The static scheme would have to allocate space for D on each call.

At first, the results appear to be a tie. Both schemes reduce the number of procedures whose calls involve allocation by two. In real programs, however, calls to procedures similar to E, which provide services to several procedures that appear 'higher' in the call graph, tend to be much more common than calls to procedures like D. For example, a compiler might have a call graph similar to the one shown with D being a procedure corresponding to a non-terminal in a recursive descent parser and E being the get-token routine. As a result, on most programs containing recursive procedures, the simple maximal interval

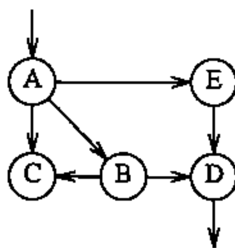


*Figure 1.*

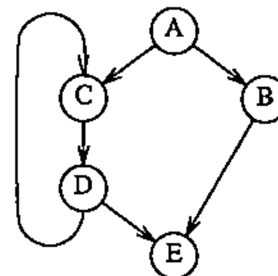


*Figure 2.*

scheme must perform storage allocation much more frequently that the static scheme. Adding static allocation to the interval scheme, however, provides a scheme that is better than the simple static allocation scheme, since the interval scheme can also eliminate allocation from calls to recursive procedures.

The significance of this aspect of the maximal interval scheme is that it implies that all three of the schemes we have described require the ability to identify procedures which cannot be called recursively and the two best schemes require the ability to statically allocate space for such procedures. Unfortunately, both of these requirements are difficult to meet in a translator that supports separate compilation.

If a program is broken into many modules that are compiled separately, there are generally very few procedures that the compiler can recognize as non-recursive. If a procedure in some module calls any procedure not in the module, the compiler must assume that a path back to the original procedure exists. Therefore, only those procedures in a module from which there is no sequence of calls that leads to a procedure outside the module can be treated as non-recursive procedures. For example, when processing the module whose call graph is shown in Fig. 1 above, a translator must assume that there may be a path from D back to A in the call graph of the remainder of the program. As a result, the translator can only assume that C is non-recursive, even though all five procedures may be non-recursive.

Even if recursive procedures could be easily identified, the static allocation of non-recursive procedures is complicated by separate compilation. When all of the procedures in a program are compiled together, the compiler can allocate space for the activation records of all non-recursive procedures in one block. The displacements from the beginning of this block to the words allocated for variables declared in any of these procedures is determined at compile time. As a result, all references to these variables can be made relative to a single base address that can be maintained in a register. With separate compilation, the compiler can only allocate space for the non-recursive procedures contained in the module it sees. Thus, instead of having one block of storage containing all variables declared in non-recursive procedures, there will be one block of storage for each separately compiled module. The displacements used by the compiler to generate code must be computed relative to these independent blocks of storage. This implies either that the run-time system must maintain one base address pointer for each separately compiled module that contained non-recursive procedures or that the linker must coalesce the storage blocks and relocate the displacements generated by the compiler.

## 2. Cutset Based Activation Record Allocation

In this section we present an activation record allocation scheme based on cycle cutsets that improves upon the other schemes described and adapts more easily to separate compilation than these other schemes. Given a directed graph G, a set of nodes S such that any cycle in G must contain at least one node in S is called a cycle cutset or a feedback vertex set. For example, the header sets in both the non-recursive interval scheme and the maximal interval scheme form cutsets. These cutsets, however, possess an important additional property. For any node, x, in a graph, there is exactly one interval header from which x can be reached without passing through another interval header. This property makes it possible to assign responsibility for the allocation of each procedure's activation record to exactly one other procedure. This somewhat simplifies storage management, but it is not essential.

Consider the call graph in Fig. 3. An interval based scheme could not safely assign responsibility for the allocation of storage for procedure D to either B or C alone because control might reach D without passing through whichever procedure was picked. Instead, the interval based schemes are forced to assign responsibility for D's allocation to D. As a result, the maximal interval scheme resorts to static allocation to avoid repeatedly allocating space for D. The problem may be solved without static allocation, however, by making both B and C responsible for allocating space for D. That is, each time B is called it could allocate space for itself and D and similarly for C. Then, when either B or C called D, the new instance of D could use the storage provided by its caller.

In general, given a cutset for the call graph of a program that includes the main program as an element, if whenever a call is made to a cutset element storage is allocated for each procedure that can be reached from that cutset element without passing through any other member of the cutset then every procedure called can safely use the space allocated for it by the last cutset element called.

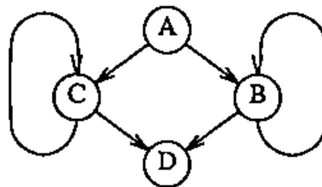


*Figure 3.*

Given this fact, we can consider the possibility of using cutsets other than those composed of interval headers to reduce storage allocation overhead. An obvious approach is to seek a cutset of minimal size. Finding such a cutset in an arbitrary directed graph, however, is known to be an NP-complete problem [12]. Fortunately, there are algorithms that work well on most graphs. In particular, Rosen's extension [20] of Shamir's algorithm [21] provides a means of finding minimal cutsets for reducible graphs and close to minimal cutsets for many other graphs in linear time with respect to the number of edges in the graph. In our work, we have chosen to use the cutsets produced by this algorithm.

Given that a good cutset can be found, we still must consider how storage will be allocated when an element of the cutset is called. There are two constraints on the allocation performed. First, if some procedure is allocated by more than one cutset element, we must ensure that the displacement from the beginning of the block allocated to the beginning of the frame for the procedure is the same in all cases. If this is done, then the code for all procedures allocated by a cutset element can use a single pointer to the beginning of the area allocated to reference local variables. If not, then we will have to provide each procedure with a pointer to its activation record within the area allocated by the cutset element. The overhead of maintaining these pointers would replace much of the storage allocation overhead that had been eliminated.

Second, while it is reasonable to accept some increase in the memory requirements in exchange for the reduction in processing our scheme will provide, memory should not be wasted. In particular, in a program such as that shown in Fig. 4, space for B and C should be overlayed, since they cannot be active simultaneously.

These constraints can be satisfied by using a slight generalization of the allocation procedure used in the maximal interval scheme [16]. We associate a weight with each edge in the graph equal to the size of the activation record of the procedure corresponding to the node from which the edge originates. We define the length of a path as the sum of the weights of the edges used in the path. For each node, P, in the

*Figure 4.*

graph, we compute the maximum of the lengths of the paths from cutset elements to the node P that do not include a second cutset element. This computation can be performed easily during the CUTSCAN phase of Rosen's algorithm for finding cutsets. Then, in the storage area allocated by each cutset element that must allocate space for P, we place the activation record of P at a displacement equal to the maximum length computed.

### 3. Cutset Based Allocation and Separate Compilation

The cutset based allocation scheme we have described can be adapted to a context in which separate compilation is supported using the same approach employed by the non-recursive interval scheme. If all procedures than can be called externally are treated as cutset elements, then all edges into these entry nodes and all edges that leave the module can be ignored when determining which other nodes to include in the cutset. Any cycle involving such edges must already contain a cutset element -- one of the entry procedures.

As with the maximal interval scheme, however, this approach to separate compilation makes the cutset scheme very sensitive to the number of entry points to a module. In particular, if no information about which procedures are entry points is provided, the translator must assume all procedures may be entry points and include all procedures in the cutset. Fortunately, there is a technique which can solve this problem for both the interval and cutset schemes. Suppose that we transform every separately compiled module by adding a new "entry" procedure. This will be the only procedure that can be called externally. That is, we will assume that any call from a procedure in another module to a procedure in the module we are considering will be replaced by a call to the entry procedure. Each such call will pass to the entry procedure the name of the procedure the original call invoked and the parameters to the original procedure. When called the entry procedure will simply call the procedure named by the first parameter its caller passed to it, passing the actual parameters along.

The graph of the module produced by making such transformations can be obtained from the graph of the original module by simply adding a new "entry" node with one edge to every node in the original graph. For example, if we apply this transformation to the module described by Fig. 1, we obtain a module whose subgraph is shown in Fig. 5. While our cutset scheme would be forced to treat every procedure in

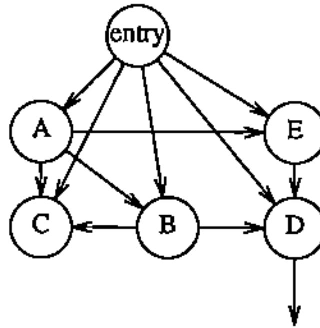


*Figure 5.*

Fig. 1 as a cutset element if no information was available about which procedures could be called externally, only the entry procedure in Fig. 5 needs to be treated as a cutset element. In general, a node in such a transformed graph will only be placed in the cutset if it was a member of a cycle completely contained in the original module.

The program transformation described above can be simulated by providing each procedure which is not a cutset element in the transformed graph with two entry points. The first entry point is used when the procedure is called internally. It does no storage allocation. The other entry point is used when the procedure is called externally. It does all storage allocation required for the fictitious entry procedure and then transfers control to the procedure's first entry point. Using multiple entry points in this way, the cutset based allocation scheme can be used effectively, even in a setting where no information about which procedures can be called externally is provided in the source code of the modules translated.

**4. Performance Data**

To verify the effectiveness of the technique proposed in this report, we constructed a system to analyze the calling patterns of programs and applied it to a collection of programs. A modified version of the 'gprof' execution profiler [10] was used to obtain a description of the call graph of each program analyzed including counts of the number of times each edge in the call graph was traversed during execution. Then, the call graph description and a description of how the procedures of the program were divided into source files was processed by a program which produced the statistics presented below.

The program analyzed were chosen to represent a wide range of program types ranging from completely non-recursive programs maintained in one source file to highly recursive program divided into

many source file. In addition, we attempted to use programs that were used regularly on our system and, where possible, readily available elsewhere. The actual program analyzed are describe below.

grader A Pascal program used locally to process student grade files [6].

sac A syntax analyzer generator which accepts an annotated LL(1) grammar for a language and generates a program that produces abstract syntax trees given source programs in the language [19].

pal An interpreter for an applicative language very similar to SASL used by students in our department's data structures course [22].

spell A spelling checker provided with the Berkeley Unix* system [24].

map A macro preprocessor for Pascal [5].

fp A locally written interpreter for Backus' FP language used by students in our department's programming language course [3].

diff A file comparison program provided with the Berkeley Unix system [24].

make A program used to aid program maintenance provided with the Berkeley Unix system [24].

gprof An execution profiler designed to incorporate information about a program's call graph in the profile data it produces [10].

indent A pretty-printer for C programs provided with the Berkeley Unix system [24].

prolog An interpreter for the prolog language [18].

pc The Pascal compiler provided with the Berkeley Unix system [24].

Table 1 summarizes some simple properties of the programs' source files and gives a count of the number of calls made by each program on our sample data. In this table and the next, the programs are separated into two groups. The first group is composed of programs whose code was kept in one source file. Program in this group are ordered by the percentage of their procedures that are recursive. The second group contains the programs that were divided into several source files. These program are ordered

| program name | procedures | source files | source langauge | calls made |
|---|---|---|---|---|
| grader | 66 | 1 | Pascal | 5415 |
| sac | 54 | 1 | Pascal | 220466 |
| fp | 79 | 1 | Pascal | 231347 |
| pal | 47 | 1 | Pascal | 106753 |
| map | 69 | 1 | Pascal | 127618 |
| spell | 24 | 1 | C | 13331 |
| prolog | 123 | 10 | C | 642204 |
| diff | 41 | 4 | C | 1450 |
| make | 55 | 6 | C | 2190 |
| gprof | 69 | 8 | C | 29154 |
| pc | 310 | 53 | C | 62264 |
| indent | 22 | 5 | C | 10624 |

*Table 1.*

* Unix is a trademark of AT&T Bell Laboratories

by the average number of procedures in each source file.

Table 2 contains some of the statistics computed by our analyzer. Our analyzer could distinguish system library calls from calls to user procedures, but it had no information about how the system library procedures were organized into source files. The analyzer was therefore designed to keep three separate sets of statistics -- one for the entire program assuming that all system library procedures were kept in one source file, one in which calls to system library procedures were ignored and one in which calls to user procedures were ignored. We will concentrate on what we consider the most interesting values -- those obtained when calls to system library procedures are ignored. However, to show that our scheme performs well even when all calls are considered we have included in the table the percentage reductions in the number of calls requiring full allocation overhead when all calls are counted. This figure appears in parentheses after the corresponding figure for the case where calls to system library procedures are ignored.

The columns in Table 2 have the following interpretations. The 'recursive procedures' column gives the percentage of each programs procedure's that belonged to a cycle in the actual call graph. The 'apparently recursive procedures' column indicates the percentage of procedures that a translator would be forced to treat as recursive procedures because of calls made to procedures external to the module being compiled and the 'apparently recursive calls' column gives the percentage of calls that were made to such procedures. We have not included these figure for the first five programs because the compiler could

| program name | recursive procedures | apparently recursive procedures | apparently recursive calls | cutset size | entry calls | cutset calls | allocations eliminated | storage ratio |
|---|---|---|---|---|---|---|---|---|
| grader | 2% | | | 2% | 0% | 6% | 94% (69%) | 1.00 |
| sac | 4% | | | 4% | 0% | 0% | 100% (76%) | 1.00 |
| fp | 25% | | | 18% | 0% | 53% | 47% (49%) | 1.11 |
| pal | 36% | | | 15% | 0% | 30% | 70% (59%) | 1.00 |
| map | 50% | | | 7% | 0% | 10% | 90% (52%) | 1.34 |
| spell | 54% | 75% | 55% | 13% | 0% | 5% | 95% (89%) | 1.07 |
| prolog | 11% | 70% | 44% | 9% | 61% | 3% | 36% (36%) | 1.00 |
| diff | 0% | 83% | 98% | 0% | 49% | 0% | 51% (13%) | 1.00 |
| make | 7% | 91% | 83% | 6% | 43% | 10% | 47% (43%) | 1.00 |
| gprof | 3% | 75% | 31% | 3% | 38% | 1% | 61% (59%) | 1.00 |
| pc | 19% | 87% | 84% | 5% | 63% | 8% | 29% (25%) | 1.14 |
| indent | 0% | 77% | 78% | 0% | 70% | 0% | 30% (26%) | 1.00 |

*Table 2.*

assume that none of the system library procedures call user procedures in these cases. The numbers we have included show that it is indeed very difficult to identify non-recursive procedures in a translator that supports separate compilation.

The 'cutset size' figure is the percentage of the procedures that were included in the cutset on which allocation decisions were based, excluding the fictitious entry procedures introduced by the transformation described in section 3. The actual size of the cutset used can be determined by adding the number of source files to this figure. The 'cutset calls' and 'entry calls' columns show what percentage of calls required allocation, breaking this information down to show whether allocation was required to break a cycle internal to a module or because a call crossed modules. The 'allocations eliminated' figure gives the percent of calls from which our scheme could eliminate the need to do storage allocation. As expected, our schemes performance generally declines as the number of procedures per source file declines, but even in the worse case, a significant reduction in allocation is obtained.

The last column gives a figure intended to measure the increase in the amount of storage used by programs translated using our scheme. Our scheme wastes storage in cases where there are paths in a call graph of differing lengths to a given node. For example, for the program fragment shown in Fig. 5 our scheme would allocate space for B and C whenever A was called with the space for C following the space for B. If A called C directly, the space for B would be wasted. In the absence of recursion, there is a somewhat acceptable upper bound on the space wasted. Basically, no more than one copy of each procedure's activation record can be wasted at any time. With recursion, however, the risk of wasting a large amount of space with each call to a recursive routine exists.

To evaluate the significance of this problem, we designed our analyzer to enumerate all cycles in the original call graph and then to compare the amount of storage allocated by our scheme each time a given cycle is traversed to that allocated by the normal allocation scheme. For this computation we assumed each activation record took one unit of storage. The 'storage ratio' column shows the average value of the ratio of these two storage figures for all of the cycles in each program. In most cases, there appears to be no significant increase. The ratio associated with the 'map' program, however, might be considered unacceptable. We are investigating techniques based on the addition of additional cutset elements that would enable a translator to enforce a bound on this ratio.

## 5. Conclusion

In this report, we have presented a practical scheme for optimizing the calling sequences used in compilers for block structured languages. It can be used in compilers that support separate compilation. The algorithms required to implement this scheme all run in linear time with respect to the number of edges in the call graph. The experimental results we have presented verify that this scheme can yield a significant reduction in the number of activation record allocations required.

In this report, we have concentrated on reducing overhead associated directly with storage allocation. It is also possible to integrate the display compaction scheme used in the maximal interval scheme [16] with the techniques we have presented. This will be discussed in a forthcoming report.

References


1. Aho, A. V. and J. D. Ullman, *Principles of Compiler Design*, Addison-Wesley Publishing Co., Reading, Massachusetts, 1979.
2. Allen, F. E. and J. Cocke, A Program Data Flow Analysis Procedure, *Communications of the ACM 19*, 3 (March 1976), 137-147.
3. Backus, J., Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs, *Comm. ACM 21*, 8 (August 1978), 613-641.
4. Bourne, S. R., A. Birrell and I. Walker, *Algol 68C Reference Manual*, Cambridge University, 1975.
5. Comer, D., MAP: A Pascal Macro Preprocessor for Large Program Development, *Software—Practice & Experience 9*, 3 (March 1979), 203-209.
6. Comer, D., Principles of Program Design Induced from Experience with Small Public Programs, *IEEE Trans. on Software Eng. SE-7*, 2 (March 1981), 169-173.
7. Dijkstra, E. W., Notes on Structured Programming, in *Structured Programming*, O-j. Dahl, E.W. Dijkstra, and C.A.R. Hoare,, *Academic Press, 1972.*
8. Faiman, R. N. and A. A. Kortesoja, An Optimizing Pascal Compiler, *IEEE Trans. on Software Eng. SE-6*, 6 (November 1980), 512-518.

9. Geschke, C. M., J. H. Morris and E. H. Satterthwaite, Early Experience with Mesa, *Communications of the ACM 20*, 8 (August 1977), 540-553.

10. Graham, S. L., P. B. Kessler and M. K. McKusick, gprof: A Call Graph Execution Profiler, *Proceedings of the SIGPLAN Notices '82 Symposium on Compiler Construction, SIGPLAN Notices 17*, 6 (June 1982), 120-126.

11. Joy, W. N., S. L. Graham and C. B. Haley, *Berkeley Pascal User's Manual, Version 2.0*, Department of Electrical Engineering and Computer Science, University of California, Berkeley, Berkeley, California, October 1980.

12. Karp, R. M., Reducibility Among Combinatorial Problems, in *Complexity of Computer Computations*, R. E. Miller and J. W. Thatcher (ed.), Plenum Press, New York, , 85-104.

13. Lampson, B. W., Fast Procedure Calls, *Proceedings of a Symposium on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, California, March 1982, 66-76.

14. Le Blanc, R. J. and C. N. Fisher, A Simple Separate Compilation Mechanism for Block-Structured Languages, *IEEE Trans. on Software Eng. SE-10*, 3 (May 1984), 221-226.

15. Liskov, B., E. Moss, C. Schaffert, R. Scheifler and A. Snyder, CLU Reference Manual, Computation Structures Group Memo 161, Laboratory for Computer Science, MIT, July, 1978.

16. Murtagh, T. P., A Less Dynamic Memory Allocation Scheme for Algol-like Languages, *Conf. Rec. of the 11th ACM Symp. on Principles of Programming Languages*, Salt Lake City, Utah, January 1984, 283-289.

17. Patterson, D. A. and C. H. Sequin, RISC I: A Reduced Instruction Set VLSI Computer, *Eigth Symposium on Computer Architecture*, , May 1981, 443-457.

18. Pereira, F., D. Warren, D. Bowen, L. Byrd and L. Pereira, *CProlog User's Manual Version 1.2*, SRI International, Menlo Park, California.

19. Rollins, E. J., A Syntax Analyzer Constructor, CS/E-82-05, Oregon Graduate Center; Beaverton, Oregon, August 1982.

20. Rosen, B. K., Robust Linear Algorithms for Cutsets, *Journal of Algorithms 3*, (1982), 205-213.

21. Shamir, A., A Linear Time Algorithm for Finding Minimun Cutsets in Reducible Graphs, *Siam J. on Computing 8*, 4 (November 1979), 645-655.

22. Turner, D. A. and R. W. Campbell, SASL Language Reference Manual, CS/-/79, Computational Science, St. Andrews University, 1979.

23. Reference Manual for the Ada Programming Language., ANSI/MIL-STD-1815A-1983, U.S. Department of Defense, February 1983.

24. UCB, *UNIX Programmer's Manual, 4.2 BSD*, Department of Electrical Engineering and Computer Science, University of California, Berkeley, Berkeley, California, August 1983.

25. Walter, K. G., Recursion Analysis for Compiler Optimization, *Communications of the ACM 19*, 9 (September 1976), 514-516.

26. Wirth, N., Program Development by Stepwise Refinement, *Communications of the ACM 14*, 4 (April 1971), 221-227.