Code Optimization Considerations in List Processing Systems

HANAN SAMET, MEMBER, IEEE

Abstract—Code optimization is characterized as a time versus space tradeoff. Space optimizations are further decomposed into static and dynamic categories. Using this characterization, the optimization requirements of a list processing language such as LISP are examined. Scrutiny of the structure of programs written in such a language reveals that traditional code optimization techniques have little benefit. Instead, a collection of low-level time and static space optimizations is seen to lead to a potential decrease in space and execution time. Dynamic space optimization is also examined in the context of reducing the frequency of occurrence of garbage collection. Alternatively, some language extensions are proposed which reduce the amount of storage that needs to be allocated, and hence may result in a decrease in the frequency of garbage collection.

Index Terms—Code optimization, compilers, garbage collection, LISP, list processing, programming language design.

I. Introduction

With the growing interest in the field of artificial intelligence has come an increase in computing involving symbolic expressions. This interest has been coupled with the development of a variety of programming languages (e.g., [4], [19]). Many of these languages are interpreter-driven and are characterized by a small set of basic primitives. The use of an interpreter has been tolerated by most users by virtue of the smallness of their application. Today, the increasing use of these languages in knowledge-based systems (e.g., MYCIN [23]) has led to the resurfacing of the efficiency problem. The most obvious solution to this problem is to use a compiler. However, the nature of the programs written in such languages (e.g., the use of EVAL in LISP [16]) has not traditionally lent itself to very efficient code. But see MDL [10], [15] where type declarations are used to generate improved code.

Work in code optimization can be characterized, in part, as a space versus time tradeoff. At one extreme is loop unrolling [1], where loops are eliminated in favor of code repetition. At another extreme are techniques such as [11] which are aimed at finding subroutines. This is a logical extension of work done in common subexpression elimination. Another example of the space versus time tradeoff is strength reduction [2].

Optimizations in space can be further broken down into static and dynamic categories. In conventional algebraic languages most of the attention has been focused on static space optimizations. In such cases reducing the size of the final object program (i.e., the space occupied by it) has been the prime consideration. Dynamic space optimization is more concerned with the run-time behavior of a program. Thus, it is the true analog of traditional time optimization since they both share the common goal of affecting execution time behavior. An example of a dynamic space optimization is a reduction in the amount of stack space required by the object program. As another example, consider parallel blocks in an ALGOL 60 [17] program which use the same local storage.

In this paper we do not describe a specific implementation; instead we use the above characterizations of optimization to examine some of the considerations that must be taken into account when attempting to obtain a code optimizer for a list processing language. Our presentation consists of two parts. First, we discuss some time and static space optimizations. These are low-level optimization techniques which are seen to have a direct effect on the amount of space occupied by the program and on the execution time of the program. Next, we explore dynamic space optimizations whose effect is not directly discernible. In the domain of list processing, these optimizations deal with reducing the frequency of garbage collection.

In order to illustrate our ideas we use LISP1.6 [18] (a variant of LISP) as the high-level list processing language and LAP [18] (a variant of the PDP-10 [8] assembly language) as the object language. The format of a LAP instruction is (OPCODE AC ADDR INDEX) where INDEX and ADDR are optional. OPCODE is a PDP-10 instruction. The AC and INDEX fields contain numbers between 0 and 15 and denote accumulators. ADDR denotes the address field. A list of the form (C 0 0 num1 num2) appearing in the address field of an instruction is interpreted as an address of a word containing num1 and num2 in its left and right halves, respectively (assuming that num1 and num2 are less than or equal to 15). Our implementation assumes that NIL is represented by zero and that a LISP cell occupies one full word where the left half contains CAR and the right half contains CDR. The stack is used for passing control between functions. A function of n variables expects to find its parameters in accumulators 1 - n. Accumulator 1 is used to return the result of the function. Accumulator 12 contains a stack pointer. The meanings of the instructions should be clear from the adjoining text; nevertheless, the Appendix contains descriptions of all instructions used in the text.

Manuscript received December 22, 1978; revised November 10, 1980. A preliminary version of this paper was presented in part at the Fifth International Conference on the Implementation and Design of Algorithmic Languages, Rennes, France, 1977.

The author is with the Department of Computer Science, University of Maryland, College Park, MD 20742.

0098-5589/82/0300-0107$00.75

II. Time and Static Space Optimizations

Traditional approaches to code optimization work [6] are primarily oriented towards making use of flow analysis to yield common subexpression elimination and reduction in strength of operators. In most list processing systems such considerations are not as important. Examination of the structure of typical programs reveals that they consist of a large number of small, often recursive, modules. Analysis of the actual programs shows that most of the execution time is spent in setting up the linkages between functions as well as for recursion (e.g., an INTERLISP [26] function call may take up to 350 μs in some cases). Thus, it seems that a substantial payoff can result from optimizing the linkages and making the recursive step as fast as possible. The latter means that the execution path which corresponds to an occurrence of a recursive call is optimized at the possible expense of other execution paths.

Significant reductions in execution time can be obtained by making use of an adaptive calling sequence. For example, consider a calling sequence convention which requires that all arguments to a function are found in the accumulators. Clearly, this convention must be adhered to when there is communication between two different functions. However, in the case of recursion, this is unnecessary. Observation of a large number of programs reveals that while preparing for a function call, a stack is often used for saving values of arguments that have already been computed. Once all of the arguments have been computed, the accumulators are loaded with the appropriate values and the function call occurs. However, in most instances the first task performed by the function is to save its arguments on the stack. The amount of extraneous data shuffling should be obvious.

A calling sequence which uses the stack exclusively has its own share of problems. There is a need for more memory to hold the stack and extra memory operations are necessary when items are accessed from the stack. Data shuffling remains a problem when functions invoke other functions with many of the same variables in the same argument positions. The difficulty is that room must be made for the return address. One solution is to have two stacks—a parameter stack and a control stack [12]. However, this solution has the disadvantage that much space must be used (i.e., two stacks rather than one) as well as requiring that the parameters be on the stack, whereas in a calling sequence which makes use of accumulators, only the parameters required for future reference need to be saved on the stack. The reason a calling sequence which makes use of accumulators appears so poor is that only rarely is there compliance with the previous criterion. Most compilers fail to make the distinction between what should and should not be saved on the stack.

We propose that for an internal recursive call a mixed calling sequence might be appropriate. In this case some of the parameters are found on the stack and others are found in the accumulators. In such a case, if there is more computing to be done within the function after the recursive call, then it is necessary to place the return address on the stack prior to the placement of the arguments on the stack. For example, consider the function START of four arguments in Fig. 1. On the left, the normal LAP encoding is given, while on the right an encoding is given which makes use of a mixed calling sequence. Note that the order of computing arguments was rearranged. Such rearranging must be capable of being proved to yield equivalent results with the original encoding (see, e.g., [22]). Also observe a shift in the location of the parameters to the function at function entry. In the case of entry from outside of the function, accumulators 1-4 will contain the parameters; whereas if the entry was from within the function, then only accumulator 1 contains a parameter. This again requires a proof that accumulators 2-4 are never referenced past the label NEWSTRT with the assumption that they contain the parameters to the function.

```
START (PUSH 12 1)
      (PUSH 12 2)
      (PUSH 12 3)
      (PUSH 12 4)
      = save arguments on stack
        .
        .
        .
      Compute argument for acc.1
      (PUSH 12 1)
      Compute argument for acc.2
      (PUSH 12 1)
      Compute argument for acc.3
      (PUSH 12 1)
      Compute argument for acc.4
      (MOVE 4 1)
      (MOVE 1 -2 12)
      (MOVE 2 -1 12)
      (MOVE 3 0 12)
      (SUB 12 (C 0 0 3 3))
      = set up calling sequence
        by restoring arguments
        from the stack
      (PUSHJ 12 START)
      = perform the recursion
      Continue with rest of
        computation
        .
        .
        .
END   (SUB 12 (C 0 0 4 4))
      (POPJ 12)
```

```
START   (PUSH 12 4)
        (PUSH 12 3)
        (PUSH 12 2)
NEWSTRT (PUSH 12 1)
        = save arguments on stack
          .
          .
          .
        (PUSH 12 (C 0 0 RESUME))
        = save the return address
        Compute argument for acc.4
        (PUSH 12 1)
        Compute argument for acc.3
        (PUSH 12 1)
        Compute argument for acc.2
        (PUSH 12 1)
        Compute argument for acc.1
        (JRST 0 NEWSTRT)
        = jump to NEWSTRT
RESUME  Continue with rest of
          computation
          .
          .
          .
END     (SUB 12 (C 0 0 4 4))
        (POPJ 12)
```

Fig. 1. Comparison of accumulator and mixed calling sequences.

Another variation of calling sequence rearrangement can be seen in Fig. 2 (note the use of MLISP [24]—an ALGOL-like version of LISP), where the function, REVERSE of [22], to reverse the links of a list is encoded with the aid of an auxiliary function REVERS1. However, instead of the customary formulation of the auxiliary function, we have interchanged the first and second arguments. Thus, the accumulating variable is the first argument rather than the second. The LAP encoding, obtained by a hand coding process, demonstrates an internal calling sequence where L is in accumulator 3 and RL is in accumulator 1. This is useful because XCONS, an equivalent of CONS with the arguments reversed, i.e., XCONS(B,A)=CONS(A,B), is known to leave all accumulators but 1 and 2 unchanged. Thus, there is no need to save L or CDR(L) while computing CONS(CAR(L),RL). Note that to all external functions, REVERS1 still appears to require two arguments in accumulators 1 and 2.

The LAP encoding in Fig. 3 corresponds to the one given in [22] and serves to illustrate the notions expressed earlier with respect to optimizing the execution path corresponding to recursion.

1) Use is made of known values of predicates in order to enable bypassing the start of the program when recursion occurs.

2) Instructions are used which accomplish two tasks at once. (SKIPN 3 2) results in a test of the nullness of L as well as a

```
REVERSE(L) = REVERS1(NIL,L)
REVERS1(RL,L) = if NULL(L) then RL
                else REVERS1(CONS(CAR(L),RL),CDR(L))
```

Fig. 2. MLISP definition of REVERSE.

```
(LAP REVERS1 SUBR)
        (SKIPN 3 2 )        load accumulator 3 with L and
                            skip if not NIL
        (POPJ 12)           return NIL
REV     (HLRZ 2 0 3)        load accumulator 2 with CAR(L)
        (CALL 2(E XCONS))   compute CONS(CAR(L),RL)
        (HRRZ 3 0 3)        load accumulator 3 with CDR(L)
        (JUMPN 3 REV)       if CDR(L) is not NIL then compute
                            REVERS1(CONS(CAR(L),RL),CDR(L))
TAG1    (POPJ 12)           return
```

Fig. 3. LAP encoding corresponding to Fig. 2.

load of accumulator 3. (JUMPN 3 REV) results in the test of the nullness of L as well as recursion. Note that in this case the execution path corresponding to recursion is optimized in the sense that (JUMPN 3 REV) is used instead of the sequence (JUMPE 3 TAG1) followed by an unconditional branch to REV.

3) Flow analysis is seen to play an important role in the placement of parameters. In the program at hand, knowledge that XCONS leaves accumulators 1 and 2 unchanged enables a shift of a parameter to accumulator 3, thereby avoiding the data shuffling which would have been inevitable had accumulator 2 been used.

Static space optimizations are also important. One of the principal complaints about INTERLISP, an otherwise excellent LISP system, is that the compiled code is extremely bulky. As an example of a technique useful in static space optimization, recall Fig. 1 where the return address was pushed on the stack prior to the computation of the parameters to the function call. The same technique can be used in the following situation. Suppose that a function tests a number of conditions, and that based on the values of these conditions, other functions are executed (similar to a CASE statement in ALGOL). Upon termination, all of these functions must return to a common point and execute a segment of code. For example, consider Fig. 4 where a function epilogue is illustrated. This can be implemented by executing a branch to the desired location of the common code sequence after each of the function calls. However, a spacewise more efficient approach is to push the address of the common code sequence on the stack prior to testing the first condition, and then to invoke the functions in the various cases via simple branch (nonstack) instructions. When the invoked functions terminate, they will return via the stack. Thus, the size of the program has been reduced by a number of instructions equal to one less than the number of conditions, with virtually no increase in execution time. The difference in execution time is the difference between the time required to execute a PUSH plus an unconditional jump and the time required to execute a recursive jump (PUSHJ) plus an unconditional jump.

III. Dynamic Space Optimizations

Dynamic space optimization is particularly important in the case of LISP, since unlike most conventional programming languages it does not have an explicit storage allocation mechanism. Storage is allocated whenever a CONS operation is performed, in which case a cell is allocated from a heap (i.e.,

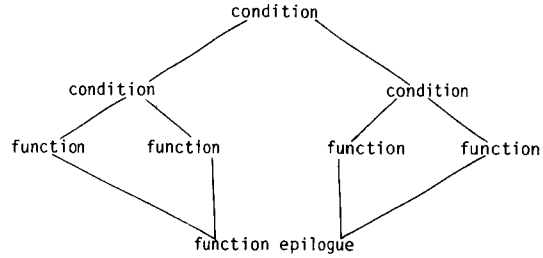


Fig. 4. Function epilogue.

the free storage list). More importantly, there is no mechanism for releasing storage. Thus, once the free storage list is exhausted, there are two possible alternatives. One choice is to quit and emit a message to the effect that storage is exhausted. The second choice entails determining which of the cells that have been previously allocated are no longer accessible. This procedure is known as garbage collection [13] and it is rather time consuming. This is especially true if most of the cells are in use, in which case very little storage is reclaimed and the garbage collection procedure will have to be reinvoked shortly. Such a situation tends to negate any effect of time optimizations (however, savings resulting from static space optimizations are still valid).

The need to do garbage collection is one of the primary deficiencies of LISP and similar list processing languages. This is because when garbage collection takes place, execution of the program is suspended until a sufficient amount of storage is reclaimed. The suspension of execution is extremely undesirable in real-time applications. A number of alternatives exist, one of which is to possibly have a second processor whose sole task is to perform garbage collection. Thus, it is seen that in the ideal situation garbage collection is always occurring in parallel with the main process (for an interesting exposition on the problems associated with parallel garbage collection see [25]). Another alternative has been proposed in [3] which is termed "on the fly" garbage collection. In such a case each time a CONS operation occurs a small segment of the space is garbage collected. The advantage of such a method is that the garbage collection process is distributed over time and requires a constant amount of time per node. The disadvantage is that it requires twice the space that is being garbage collected.

In Section II we discussed a number of low-level time and static space optimizations. Individually, they are not earth-shattering. However, when viewed collectively, significant reductions in total space and time can be observed. Some of these optimizations have additional benefits. Specifically, those optimizations which also have the effect of reducing the size of the stack should reduce the number of active links that will need to be pursued during the marking phase of garbage collection. This means that the number of cells that will be marked has a high probability of decreasing with the result that a greater fraction of the space will be garbage, hence leading to more space being reclaimed. We have no data with respect to the effect of our optimizations on garbage collection. However, one rather complex program HIER [21], which was optimized using the techniques presented in Section II, occupied 72 percent of the space needed by the original

function, was 40 percent faster in execution time, and required 50 percent of the stack space needed by the original function. The last reduction has an immediate benefit in the sense that the amount of stack space necessary can be reduced, thereby decreasing the likelihood of stack overflow. Similar results were also obtained for numerous other functions (see [20] and [22]).

Another approach to reducing the need to do garbage collection at run-time is termed "compile-time garbage collection" [7]. In this method algorithms which use CONS cells are converted by the compiler to "equivalent" algorithms which do not perform CONS operations. This is achieved by changing links in the list structure rather than working on a new copy of the data structure. For example, the algorithm given in Fig. 2 to reverse the links of a list can be encoded using compile-time garbage collection in the following manner:

```
REVERSE(L) = REVERS2(L,NIL);
REVERS2(L,RL) = if NULL L then RL
                else REVERS2(CDR L,RPLACD(L,RL));
```

However, such techniques do not preserve equivalence since if node-sharing is likely to occur, then all lists in which the argument to REVERSE appears, will be effected—an undesirable result.

Until now we have seen how dynamic space optimization can be effected by code optimization. In the case of garbage collection, an alternative method of reducing its frequency is to change the mechanism which is used to obtain storage. A CONS operation is often performed unnecessarily, i.e., a cell containing the appropriate elements of the pair has already been allocated. For example, suppose CONS(A,B) has been performed. Subsequently, it is determined that A is EQ to C. Now, if it is desired to perform CONS(C,B), then there is in fact no need to do so since such a cell already exists. This can be recognized by using a hashing [14] scheme for CONS where the hashing function has as its arguments the addresses corresponding to the arguments of the CONS. Of course, such an implementation means that prior to the allocation of a cell from the free storage list, we must determine if a cell has already been allocated with the same components. Also note that when CONS is hashed, the determination of whether a CONS cell with the same components has already been used is done at run-time and is unrelated to common subexpression elimination—a process performed at compile-time.

There are several factors which must be taken into account when evaluating the hashed CONS method of storage allocation.

1) If side effect operations such as RPLACA and RPLACD are allowed, then there is a potential for disastrous results due to node-sharing.

2) There is a constant need to check if a cell has already been allocated with the said components. This may prove to be time consuming. However, as mentioned earlier, in a real-time application we are more concerned with predictability of length (i.e., in time) of operations. Nevertheless, the fact that the amount of time spent to check if a cell has already been allocated greatly exceeds the average time per cell spent in a garbage collection process must be taken into consideration.

3) There is a need for hash tables and pointers to keep track of the various cells in use. Therefore, unless there is a high factor of duplication (i.e., there are many instances of CONS cells with the same components) we may find that our available storage has been cut down significantly. However, these tables need not be part of the free storage list. This has a significant meaning in a computer with a limited directly addressable address space. For example, on certain versions of a minicomputer such as the PDP-11 [9] which support segments, it is advantageous to keep the tables in a separate segment. In such a case we maximize the amount of space that is capable of being directly addressed with respect to containing LISP free storage while the bookkeeping is done by making use of other segments. This can be understood by noting, for example, that the PDP-11 which has an I (instruction) and (D) data space capability as well as segmentation; however, a program can only directly address 64K bytes of storage. Thus, we use as much of this space as is possible for the free storage list. Hashing is one way of maximizing its use since two LISP cells which have the same CAR and CDR pointers will be represented by the same word in memory. The actual bookkeeping is done using storage in a separate segment.

Dynamic space optimization may also be achieved by augmenting the expressive power of a programming language. In the case of LISP, a CONS operation is often performed in order to circumvent the inability to return more than one result from a LISP function. In such a case a list is formed whose elements are the results of the function rather than by use of a special construct as is available in the POP2 [5] language.

In a LISP system which uses an accumulator to return the value of the function and a stack for the purpose of program control, a return of more than one result can be implemented as follows. Return the first result in an accumulator, and return the remaining results in a contiguous block of storage immediately above the top of the stack (which contains the return address). The only remaining task is to indicate how a multiple result is to be specified in LISP. In our view the most natural way is to add the special form RLAMBDA [20] which we define to be identical to an internal LAMBDA of as many arguments as there are results being returned and only one binding, i.e., the function which has more than one result.

For example, consider Fig. 5 where the function G calls the function H which returns as its result three values. In this case the variables B, C, and D in function G are bound to (H1 A), (H2 A), and (H3 A), respectively.

The only distinction with LAMBDA is that the values of all but the first argument are found on top of the stack. Thus, if any other function is to be called, then the stack must be adjusted to save these values below the stack pointer which points to the top of the stack. A typical solution is to store the value that was returned in the result accumulator (i.e., accumulator 1) in the location pointed at by the stack pointer (i.e., the stack location which contained the previous return address), and then increase the stack pointer by a number equal to the number of values that were returned.

We must also provide a syntactic mechanism for denoting that more than one result is to be returned. The easiest way to achieve this is to have, in addition to the property associ-

```
(DEFPROP G (LAMBDA (A)
                   ((RLAMBDA (B C D)
                             (M B C D))
                    (H A)))
 EXPR 1)
(DEFPROP H (LAMBDA (A)
                   (H1 A)
                   (H2 A)
                   (H3 A))
 EXPR 3)
(DEFPROP SUMDIV (LAMBDA (DIVIDEND DIVISOR)
                        ((RLAMBDA (QUOTIENT REMAINDER)
                                  (*PLUS QUOTIENT REMAINDER))
                         (DIVISION DIVIDEND DIVISOR)))
 EXPR 1)
```

Fig. 5. Examples of functions that return more than one result.

ated with each function denoting its type (e.g., EXPR when the arguments have been evaluated prior to the function call and FEXPR if not), a property that indicates the number of results returned by the function. For example, the function H in Fig. 5 is an EXPR which returns three results. The actual act of returning more than one result, say n, is accomplished by returning the last n values that have been computed. For example, given a LISP function that returns two results, the conditional form [COND (p1 e11 e12) (p2 e21 e22 e23) (T e31 e32 e33)] would be interpreted to return as its value e11 and e12 if p1 is true, e21 and e22 if p2 is true, and e32 and e33 otherwise. We also make the added stipulation that a function returns the same number of results in all cases.

As an additional example, consider the function DIVISION which returns as its result the QUOTIENT and the REMAINDER when integer division is performed on its two arguments, i.e., the first argument is integer-divided by the second argument. Fig. 5 shows the use of DIVISION in the definition of the function SUMDIV of two arguments whose value is the sum of the quotient and the remainder when DIVIDEND is integer-divided by DIVISOR.

The above examples lead us to conclude that a list processing language (e.g., LISP) should provide a capability for user control of the deallocation of cells. The example of LISP's current handling of multiple results (e.g., by the construction of a list) demonstrates a need for deallocation to be performed by the function to which multiple results are returned. There is also a potential need for deallocation at function exit as is done in ALGOL 60. This could be accomplished by use of specialized CONS operations which leave messages as to the lifetime of the cell that has been allocated. We feel that the determination of cells that can be deallocated in such a manner would be best achieved by a comprehensive flow analysis package.

IV. Conclusion

We have discussed several views of optimization in the context of a list processing system. The main focus of the presentation has been to point out the considerations that should be taken into account in obtaining an efficient compiler-based system. However, several of the optimizations proposed in Section III could also be used in an interpreter-based system.

The majority of the optimizations proposed in Section II have a heuristic flavor associated with them. Many are a result of a trial and error code generation procedure. In such a case there may be several attempts at obtaining an optimal encoding; some of which might be incorrect. The correctness of the translation can be demonstrated by use of a proof system such as [22], which has as its input a high-level language encoding of an algorithm and a low-level language encoding of the same algorithm. Such a proof system is embedded in the translator and is intended to be the final step in the code generation procedure.

Some of the dynamic space optimizations of Section III are more of a language design nature. They must be evaluated in light of their effects on programming style. Clearly, a hashed CONS mechanism implies node-sharing and therefore its use deprives the programmer of the ability to use RPLACA and RPLACD. However, the introduction of a multiple result feature does not seem to have any drawbacks. Another factor to consider is the size of the available directly addressable memory, i.e., a small amount may lead to the desirability of the adoption of a hashed CONS mechanism.

Appendix

PDP-10 Operations

CALL A special LAP instruction which is analogous to a PUSHJ. The difference is that it is used to invoke LISP functions via the property list. This is useful when a trace of the arguments to a function is desired, or when the actual binding of a function changes. (CALL num (E fname)) denotes a CALL to fname where num is the number of arguments.

HLRZ Load the right half of accumulator AC with the left half of the contents of the effective address and clear the left half of AC.

HRRZ Load the right half of accumulator AC with the right half of the contents of the effective address and clear the left half of AC.

JRST Unconditional jump to the effective address.

JUMPE Jump to the effective address if the contents of accumulator AC is zero; otherwise continue execution at the next instruction.

JUMPN Jump to the effective address if the contents of accumulator AC is unequal to zero; otherwise continue execution at the next instruction.

MOVE Load accumulator AC with the contents of the effective address.

POPJ Subtract octal 1 000 001 from accumulator AC to decrement both halves by one. If subtraction causes the count in the left half of AC to reach -1, then set the Pushdown Overflow flag. The next instruction is taken from the location addressed by the right half of the location that was addressed by AC prior to decrementing.

PUSH Add octal 1 000 001 to accumulator AC to increment both halves by one and then move the contents of the effective address to the location now addressed by the right half of AC. If the addition causes the count in the left half of AC to reach zero, then set the Pushdown Overflow flag.

PUSHJ Add octal 1 000 001 to accumulator AC to increment both halves by one. If addition causes the count in the left half of AC to reach zero, then set the Pushdown Overflow flag. Store the contents of the program counter and the processor flags in the right and left halves, respectively, of the location now addressed by the right half of AC, and continue execution at the effective address.

SKIPN Skip the next instruction if the contents of the effective address is not equal to zero. If the AC field specification is nonzero, then load accumulator AC with the contents of the effective address.

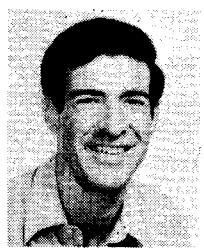
SUB The contents of the effective address is subtracted from the contents of accumulator AC, and the result is left in AC.

Acknowledgment

Special thanks go to R. L. Kirby for his comments.

References

[1] A. V. Aho and J. D. Ullman, *Principles of Compiler Design*. Reading, MA: Addison-Wesley, 1977.
[2] F. E. Allen, "Program optimization," *Annu. Rev. Automat. Programming*, vol. 5, pp. 239-307, 1969.
[3] H. G. Baker, Jr., "List processing in real time on a serial computer," *Commun. Ass. Comput. Mach.*, pp. 280-294, Apr. 1978.
[4] D. G. Bobrow and B. Raphael, "New programming languages for artificial intelligence," *ACM Comput. Surveys*, pp. 153-174, Sept. 1974.
[5] R. M. Burstall, J. S. Collins, and R. J. Popplestone, *Programming in POP2*. Edinburgh, Scotland: University Press, 1971.
[6] J. Cocke and J. T. Schwartz, *Programming Languages and their Compilers*. New York: New York Univ. Courant Inst., Apr. 1970.
[7] J. Darlington and R. M. Burstall, "A system which automatically improves programs," in *Proc. 3rd Int. Joint Conf. on Artificial Intell.*, 1973, pp. 479-485.
[8] *PDP-10 System Reference Manual*. Maynard, MA: Digital Equipment Corp., 1969.
[9] *PDP-11 Reference Manual*. Maynard, MA: Digital Equipment Corp., 1973.
[10] S. W. Galley and G. Pfister, "The MDL programming language," Massachusetts Inst. of Technol. Lab. for Comput. Sci., 1979.
[11] C. M. Geschke, "Global program optimizations," Ph.D. dissertation, Dep. Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, PA, 1972.
[12] R. L. Kirby, "ULISP for PDP-11s with memory management," Comput. Sci. Cen., Univ. of Maryland, College Park, TR 546, 1977.
[13] D. E. Knuth, *The Art of Computer Programming, Fundamental Algorithms*, vol. 1, 2nd ed. Reading, MA: Addison-Wesley, 1973.
[14] ——, *The Art of Computer Programming, Sorting and Searching*, vol. 3. Reading, MA: Addison-Wesley, 1973.
[15] P. D. Lebling, "The MDL programming environment," Massachusetts Inst. of Technol. Lab. for Comput. Sci., 1980.
[16] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine," *Commun. Ass. Comput. Mach.*, pp. 184-195, Apr. 1960.
[17] P. Naur, Ed., "Revised report on the algorithmic language ALGOL 60," *Commun. Ass. Comput. Mach.*, pp. 299-314, May 1960.
[18] L. H. Quam and W. Diffie, "Stanford LISP 1.6 manual," Dep. Comput. Sci., Stanford Univ., Stanford Artificial Intell. Project Operating Note 28.7, 1972.
[19] C. J. Rieger, J. Rosenberg, and H. Samet, "Artificial intelligence programming languages for computer aided manufacturing," *IEEE Trans. Syst., Man, Cybern.*, pp. 205-226, Apr. 1978.
[20] H. Samet, "Automatically proving the correctness of translations involving optimized code," Ph.D. dissertation, Dep. Comput. Sci., Stanford Univ., Stanford Artificial Intell. Project Memo. AIM-259, 1975.
[21] ——, "A study in automatic debugging of compilers," Dep. Comput. Sci., Univ. of Maryland, College Park, TR 545, 1977.
[22] ——, "Proving the correctness of heuristically optimized code," *Commun. Ass. Comput. Mach.*, pp. 570-582, July 1978.
[23] E. H. Shortliffe, "MYCIN—A rule-based computer program for advising physicians regarding antimicrobial therapy selection," Dep. Comput. Sci., Stanford Univ., Artificial Intell. Project Memo. AIM-251, Oct. 1974.
[24] D. C. Smith, "MLISP," Dep. Comput. Sci., Stanford Univ., Stanford Artificial Intell. Project Memo. AIM-135, Oct. 1970.
[25] G. L. Steele, Jr., "Multiprocessing compactifying garbage collection," *Commun. Ass. Comput. Mach.*, pp. 495-508, Sept. 1975.
[26] W. Teitelman, *INTERLISP Reference Manual*. Palo Alto, CA: Xerox Palo Alto Res. Cen., 1975.



Hanan Samet (S'70-M'75) received the B.S. degree in engineering from the University of California, Los Angeles, and the M.S. degree in operations research and the M.S. and Ph.D. degrees in computer science from Stanford University, Stanford, CA.

In 1975 he joined the University of Maryland, College Park, as an Assistant Professor of Computer Science. In 1980 he became an Associate Professor. His research interests are data structures, image processing, programming languages, artificial intelligence, and database management systems.

Dr. Samet is a member of the Association for Computing Machinery, SIGPLAN, Phi Beta Kappa, and Tau Beta Pi.