MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

AI Memo 443 October 1977

DEBUNKING THE "EXPENSIVE PROCEDURE CALL" MYTH

or, PROCEDURE CALL IMPLEMENTATIONS CONSIDERED HARMFUL

or, LAMBDA: THE ULTIMATE GOTO

by

Guy Lewis Steele Jr. *

Abstract:
Folklore states that GOTO statements are "cheap", while procedure calls are "expensive". This myth is largely a result of poorly designed language implementations. The historical growth of this myth is considered. Both theoretical ideas and an existing implementation are discussed which debunk this myth. It is shown that the unrestricted use of procedure calls permits great stylistic freedom. In particular, any flowchart can be written as a "structured" program without introducing extra variables. The difficulty with the GOTO statement and the procedure call is characterized as a conflict between abstract programming concepts and concrete language constructs.

This is an annotated version of a paper to be presented at the ACM National Conference (ACM '77), Seattle, Washington, October 1977.

Keywords: procedure calls, subroutine calls, structured programming, programming style, compilers, optimization

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0643.

* NSF Fellow

Introduction

For the past five to ten years (depending on how you count) the computing community has been embroiled in debate over "structured programming" and, as a special case, the GOTO statement. On the one hand, many authors have observed that programs written without GOTOs are often more perspicuous than programs written with GOTOs by the same programmer. That is, the avoidance of GOTOs is a discipline which has been observed to produce programs which are easier to understand and maintain.

On the other hand, a certain portion of the computing community has rebelled at this discipline. There are several reasons for this rebellion. Among these are:

(1) Some programmers fear that their expressive power or style will be cramped if GOTO is taken away from them. This is true in such contemporary languages as FORTRAN, but in more powerful languages such as ALGOL, PL/I, COBOL, and PASCAL the point is somewhat more debatable. Yourdon comments: "Many programmers feel that programming without the GO-TO statement would be awkward, tedious and cumbersome. For the most part, this complaint is due to force of habit." [You75,178] In all fairness, it should be pointed out that GOTO is a basic, universal primitive with which (in conjunction with a conditional) almost any other control construct can be synthesized if the language designer neglected to include it. {Note Omniscient Implementors} A good example of this is the CASE statement.

(2) Some programmers feel that the imposed discipline doesn't make any difference, because one can write incomprehensible programs without GOTO. This piece of logic is slightly false, for while avoiding GOTO does not guarantee comprehensibility, it often does increase the probability of producing comprehensible code, given our current cultural biases in programming style.

(3) There is some feeling that GOTO is a "cheap" statement, while other constructs (DO, CASE, etc.) are more "expensive". Everyone knows that a GOTO gets compiled into one machine instruction (some kind of branch), while DO produces many instructions. The difficulty here is a misplaced sense of what is "efficient"; and despite the fact that current practitioners of programming discipline tell us not to worry about efficiency, we nevertheless do, at some low unconscious level, whenever we write code. Yourdon tells of the plight of programmers "whose managers have specifically forbidden them to use such statements because of the overhead involved in subroutine-calling statements." [You75,177]

These misplaced feelings of efficiency are felt for other programming language constructs as well, and subtly influence our programming style. The example I wish to consider here is the procedure call. Everyone knows that, unlike GOTO statements, procedure calls are "expensive". Indeed, these feeling are not unjustified, given past and current programming language implementations.

Because procedure calls are "expensive", we tend to avoid using them in our code. Unfortunately, this produces a detrimental effect on our programming style, for what Dijkstra said of PL/I holds true of almost any language: "the procedure is one of [the] main vehicles for expressing structure". [Dij76,8] Recently practitioners of programming discipline have advised us not to be afraid to use procedure calls, as the readability they afford is worth the inefficiency. This is like saying that cars with square wheels are all right because transportation is worth a bumpy ride: we really ought instead to concentrate on improving our wheels.

I would like to suggest here that procedure calls have been given a

"bad press" for a number of reasons. There is no reason that procedure calls must be expensive. I intend to demonstrate here the following points:

(A) Procedure calls are, from a theoretical point of view, glorified GOTO statements; this view leads to interesting compilation techniques which can save space and time.

(B) Procedure calls are in fact quite fast when implemented properly.

(C) Procedure calls have received a bad reputation for a variety of historical reasons.

(D) As a result, we have been advising programmers to adjust their programming style to compensate for poor implementations (or, alternatively, not to adjust but to ignore the poorness of the implementation), while giving little thought to improving these implementations.

(E) Procedure calls, implemented properly and used freely, allow a stylistic freedom far greater than (and largely inclusive of) that afforded by GOTO. Any flowchart can be implemented as a "structured" program without introducing any extra variables.

(F) Much of our difficulty with procedure calls and with GOTO statements is that we have a restricted view of the relationship between programming concepts and language constructs.

A. Procedure Calls as GOTO Statements

In studying the nature of procedure calls, it is appropriate to study the LISP language. LISP, more than any other language, uses procedure calls as the primary means of expression; in particular, the arbitrary distinction between built-in operators (like multiplication) and user functions is almost completely nonexistent. What would appear in FORTRAN as

```
FUNCTION QUAD(A,B,C)
QUAD = (-B + SQRT(B**2 - 4*A*C)) / (2*A)
RETURN
END
```

is in (one dialect of) LISP:

```
(DEFINE (QUAD A B C)
        (/ (+ (- B)
              (SQRT (- (** B 2) (* 4 A C))))
           (* 2 A)))
```

In this way most computations (other than conditionals) can easily be expressed in terms of function calls of the form

```
(F X1 X2 ... Xn)
```

LISP is an expression language, in that every program is a function which returns a value (but this value may be ignored - many programs produce interesting side effects and then return NIL). It differs from other expression languages such as BLISS [Wul71] [Wul75], however, in its uniformity of notation.

The usual way to implement function calls in LISP is by using a stack. When a function is called, a return address is pushed onto the stack; when a function is done, it pops a return address and goes there (after stashing the value to be returned in a register). This is in fact how procedure calls are

implemented in many languages.

Let us consider the execution of a call on the LISP function BAR:

```
(DEFINE (BAR X Y)
        (F (G X) (H Y)))
```

BAR calls G on X, H on Y, and then F on the two results, returning the result of calling F.

For concreteness, we will express the compiled code in a modified form of PDP-10 machine language, using these instructions:

JUMP x	Branch to (i.e. GOTO) location x.
PUSHJ x	Push the location of the PUSHJ, plus 1, on the stack, then branch to location x.
POPJ	Pop an address from the stack and branch there.

The code for BAR might look something like this:

```
BAR:    <set up arguments for G>
        PUSHJ G
BAR1:   <save result from G>
        <set up arguments for H>
        PUSHJ H
BAR2:   <use results from G and H for F>
        PUSHJ F
BAR3:   POPJ
```

We have introduced the labels BAR1, BAR2, BAR3 to aid the exposition. Note that there are no instructions between the PUSHJ F and the POPJ; we shall justify this below.

On arrival at BAR, the arguments X and Y are presumably in registers or some other appropriate place, and a return address (say FOO5) is on the stack. After we execute the PUSHJ G, the stack will look like this:

```
BAR1
FOO5
...
```

G may call other functions, and the stack will go up and down, but eventually it will put a value in the result register and do a POPJ, returning to BAR1. This leaves the stack like this:

```
FOO5
...
```

In a similar manner, the PUSHJ H will push BAR2 on the stack, and H will eventually pop it when it exits. The same is true of the PUSHJ F and BAR3. When F returns to BAR3 with a value in the result register, the POPJ at BAR3 is performed, returning to FOO5. Since BAR was to return the result of calling F, the correct value is in the result register for FOO5.

Notice that during the execution of F the stack looked like this:

```
BAR3
FOO5
...
```

Suppose that at the end of BAR we changed the sequence

```
        PUSHJ F          to      JUMP F
BAR3:   POPJ
```

Then on arrival at the JUMP the stack would look like this:

```
        FOO5
        ...
```

Instead of a PUSHJ to push BAR3, we have a JUMP to F. Thus on arrival at F, the stack has only FOO5 on top. When F is done, it will return to FOO5 instead of BAR3. But this is all right! F has put the correct value in the result register, and this value will be transmitted to FOO5, with the stack properly adjusted. All that has changed is the useless pushing of BAR3, and the encoding of a procedure call as a JUMP (that is, a GOTO!).

This approach is quite general. The idea is obscured by algebraic syntax, but if we rewrite a program in LISP notation, it is clear that the last thing that program does is a procedure call of some kind. To prove this, we note that the outermost construct in the program body must fall into one of a limited number of cases:

* A call to an "intrinsic" function which is compiled open. In this case that function is simply compiled open. That function may compile open as one or more arithmetic instructions followed by a POPJ, or else it must recursively fall into some other case.
* A call to an external function. In this case, as argued above, we can simply JUMP to the new function, as there is no need to return to the caller.
* A conditional. The argument applies recursively to the branches of the conditional.
* A sequential block. The argument applies recursively to the last component of the block.
* A looping construct. The argument applies recursively to the expression which gives the value of the loop (which may be assumed to fall into the first case if the value is to be ignored).

Other constructs are handled similarly. In this way, if the last thing a procedure does is call another (external) procedure, that call can be compiled as a GOTO. Such a call is called tail-recursive, because the call appears to be recursive, but is not, since it appears at the tail end of the caller.

This approach can be made even more uniform by compiling every procedure call as a GOTO. The idea is that a return address is pushed on commencement of the evaluation of an argument, rather than application of a function. This provides a uniform compilation discipline. For example, consider the BAR function used above:

```
(DEFINE (BAR X Y)
        (F (G X) (H Y)))
```

In order to compile the body, we must first compile the argument forms (G X) and (H Y). Since (G X) is an argument form, we push a return address, then set up the arguments for G, then call G. (H Y) is treated similarly. Now that the arguments for F are available, we set them up and call F:

```
BAR:    PUSH [BAR1]
        <set up arguments for G>
        JUMP G
BAR1:   <save result from G>
        PUSH [BAR2]
        <set up arguments for H>
        JUMP H
BAR2:   <use results from G and H for F>
        JUMP F
```

We did not push a return address for F since the call to F is not an argument form. Notice that this uniform discipline lends itself to passing arguments on the stack, above the return address. The called procedure is then responsible for popping the arguments. On the other hand, if argument passing does not use the stack, we can permute the PUSH of the return address with the argument set-up code:

```
BAR:    <set up arguments for G>
        PUSH [BAR1]
        JUMP G
BAR1:   <save result from G>
        <set up arguments for H>
        PUSH [BAR2]
        JUMP H
BAR2:   <use results from G and H for F>
        JUMP F
```

If our machine has a PUSHJ instruction, then a peephole optimization [McK65] can now transform the PUSH/JUMP sequence into a single PUSHJ instruction. Thus we see that a procedure call can be uniformly treated as a GOTO, with the PUSHJ instruction considered an optimization (rather than vice versa!).

There are a couple of difficulties with this idea. One is that the stack is often used to hold information other than return addresses, such as stack-allocated data structures and intermediate results. This is no problem, as it turns out; it is merely necessary to clean up the stack just before calling F, rather than just after calling F; then the original PUSHJ F and the POPJ will be consecutive, and so can be expressed as a JUMP F. {Note Shuffle Arguments}

This leads to a second difficulty, which is that some languages would allow F to refer globally to stack-allocated structures within BAR, such as the dynamic values of X and Y. In this case we cannot clean up the stack until after calling F. There is, however, some evidence [Wul73] that such global variables are as "harmful" as GOTO statements; in any case it is a good idea to minimize their use, and to define variables to be lexical in scope by default. It turns out that in most programming languages (COBOL, PL/I, FORTRAN, and LISP in particular) the distinction between lexical and dynamic scoping doesn't matter most of the time anyway. We should leave the compiler free to produce the best possible code; only in cases where structures are explicitly declared to be dynamically referenced should the compiler be forced to leave them on the stack in an otherwise tail-recursive situation.

In general, procedure calls may be usefully thought of as GOTO statements which also pass parameters, and can be uniformly encoded as JUMP instructions. This is a simple, universal technique, to be contrasted with the more powerful recursion-removal techniques such as those in [Dar76]. Our approach results in all tail-recursive procedure calls being compiled as

iterative code, almost without even trying, for it is more a matter of the code generation strategy than of a specific attempt to remove recursions. (For more discussion of this, see [Ste76b]. For an interesting comparison between GOTO and the APL execute operator, see [Syk77].)

B. Procedure Calls Can be Fast

The above examples have shown procedure calls encoded as a simple JUMP, or at worst as a PUSHJ. These simple instructions are not time-consuming, even on computers which must simulate PUSHJ because it is not a primitive instruction. What then makes procedure calls so expensive?

The answer seems to be that most implementations are rather thoughtless or careless in this regard. It is usual for a compiler to slap standard "prologue" and "epilogue" code around every procedure; this code typically sets up complex procedure frame structures, allocates all declared variables, and sometimes even saves and restores all registers. Auslander and Strong [Aus76] report that one simple procedure call, compiled by the OS/360 PL/I optimizing compiler, pushes 336 bytes onto the stack! Yourdon [You75,98] reports that on a 360/50 a PL/I procedure call costs 198 microseconds. It is no wonder that programmers feel that procedure calls are slow - they are!

That is, they are slow as currently implemented. Unfortunately, our thinking has generally been colored to the point where we simply assume that all procedure calls are inherently slow. Even SIGSAM Bulletin, a journal contributed to in large part by LISP programmers, said in an editorial [Jo72]:

> "... one might expect CAMAL, SAC-1, ALTRAN, and TRIGMAN to run the fastest, because they make efficient use of special-purpose data structures and because they are written either in FORTRAN or machine language; and present versions of MACSYMA, REDUCE, and SCRATCHPAD to run slower -- because of their more general expression handling ability and because of the frequency and generality of function calling in LISP."

In that same editorial comparative running times were given for the systems, and indeed the LISP-based systems were five to ten times slower than the others -- except MACSYMA, which was comparable to the FORTRAN and machine-language systems! Clearly this contradicts the cited intuitive belief about procedure calls.

A reply by Fateman [Fat73] further emphasized this point. In actual timing tests conducted on numerical code using the MacLISP compiler and the then current DEC FORTRAN compiler, the MacLISP code was faster! Fateman comments:

> "... 'the frequency and generality of function calling in LISP' is a high cost only in inappropriately designed computers (or poorly designed LISP systems). ... The point we wish to make is that compiled properly, LISP may be as efficient a vehicle for conveying algorithms, even numerical ones, as any other higher-level language, e.g. FORTRAN. An examination of the machine code produced by the two compilations shows that the inner-loop arithmetic compilations are virtually identical, but that LISP subroutine calls are less expensive."

(For a discussion of the techniques used to achieve FORTRAN-like arithmetic ability in LISP, see [Ste77b].)

C. Why Procedure Calls Have a Bad Reputation

The very notion of "procedure call" in a programming context was only worked out painfully after computers had already existed for some time. Indeed, the idea of automatically linked library procedures met with considerable opposition when first proposed. [Hop73] Since procedure calling instructions were not planned ahead of time in the way that arithmetic, conditional, and branch operations were, one would assume they were implemented on early computers in a rather clumsy fashion. While the basic arithmetic and conditional jump instructions have changed little in nature over the years, one can trace an evolution of special instructions used for procedure calls. Machines before 1960 (for example the IBM 1620 and 704) typically had only one instruction (if any) for subroutine calling, which saved the instruction counter in a register or in the first memory location of the called subroutine. The PDP-1 had three instructions which were all variants on this theme. The IBM 360 had only one such instruction (with two addressing modes), but the programmer had the additional choice of which register to store the program counter into. As far as I know offhand, stack instructions did not generally appear in non-stack-oriented machines until the mid-1960's, in such machines as the PDP-6, which in addition to PUSHJ offered three other subroutine-calling instructions; in retrospect, this seems to have been more out of uncertainty as to which would be used than out of necessity for a variety of instructions, for only two of the four are used at all extensively now on the PDP-10 (I am ignoring the "UUO" mechanism as not being a general subroutine-calling device). The PDP-11 (1970 or so) is the earliest machine I know of which was not essentially stack oriented (as some early Burroughs machines were) but which provided only a stack-pushing subroutine call instruction.

The interesting thing is that all of these examples have attempted to compress the procedure call operation into a single instruction. As may be inferred from the discussion above and in [Ste76b], this isn't necessarily the right way to do it. The procedure call may conceptually be divided into three phases: push a return address if necessary, set up arguments, go to called procedure. (Those familiar with spaghetti stacks [Bob73] will recognize this sequence as "create a frame, set up the arguments in the frame, go to called procedure".) It is important to note that the first step naturally comes first, and that it is conditional (but for lexically scoped programs this condition can be determined at compile time for any given procedure call as described above). The mistake that we make is to attempt to combine the first and third steps unconditionally into a single, universal instruction. The result is that the return address is always pushed whether it is necessary or not.

(It is appropriate to note here that the procedure call instruction might not itself push the return address onto the stack. It might put it into a register, in which case that register's previous contents must first be pushed, in general, as there are only a finite number of registers. Another case, often used in FORTRAN implementations, is that every procedure has a location reserved in memory for holding the return address for that procedure. This does not permit recursion, and wastes memory space compared to the use of a stack, because if recursion is not permitted the total stack space could not exceed the number of distinct procedures anyway.)

Compiler writers have often simplified their job at the expense of the procedure call by adopting certain standard protocols. One of these is that the called procedure should save all registers that it uses. This is in turn often simplified to "save all registers". It is seldom the case that all of these registers actually need to be saved; indeed, in statement-oriented

languages such as FORTRAN with little global optimization by the compiler, often no registers need be saved across a procedure call! Thus this convention can only lead to unnecessary extra running time, which gets charged to the poor procedure call. (This convention does have the virtue of helping to isolate the effects of buggy compiler output; but this feature is not without cost.)

The great speed of compiled MacLISP procedure calls is primarily due to its taking the inverse approach: the caller is responsible for saving any registers that it will need after calling another procedure. It might be thought that this would lead to a much greater code size than the other convention, but this is offset by three effects. One is that, as noted above, few registers actually need to be saved in practice. Another is that the compiler can know which registers are not destroyed by built-in functions and avoid saving such registers unnecessarily. (This can be compared with knowing which registers are used by the out-of-line "intrinsic" functions in a FORTRAN implementation; or, for that matter, knowing that certain instructions clobber certain registers, such as DIVIDE producing both a quotient and a remainder.) The third is that the architecture of the PDP-10, while not essentially stack-oriented, does not make references to stack values unduly difficult; thus it is often just as easy to keep a variable on the stack rather than in a register in the first place.

At the source-language level, there are other factors which contribute to the procedure call's poor reputation. Nearly all algebraic computer languages draw a syntactic distinction between operators and user functions, if not also a semantic distinction. Often built-in functions are also distinguished in some silly way from user functions, even though they are used in syntactically similar ways. As an example, you cannot pass "+" as an external function argument in FORTRAN, even though it is mathematically a perfectly good function of two arguments; similarly you cannot pass a statement function, even though there is no syntactic difficulty as there is for "+". [ANS76,8.7/15.4.3] You can pass an intrinsic function as an argument, unless it is one of the MIN/MAX series. [ANS76,8.8/15.3.2] PL/I built-in functions can return array or structure values, but not user-defined functions. [IBM70b,162] Even as enlightened a language as APL does not permit, in current implementations, a user function to be used within the reduction or inner/outer product constructions. Such decisions are occasionally questioned, but most people accept them on the grounds of "efficiency". This is absurd. There is no reason one cannot accept the general case, and handle important special cases specially. For example, if a user should try to pass a statement function or intrinsic function as an argument in FORTRAN, the compiler could jolly well provide a reference to an EXTERNAL version of that routine, while continuing to use the internal version (if it is in fact compiled as a distinct version) where applicable.

Even if we accept such arbitrary semantic distinctions in our languages, there remain the syntactic differences. Most languages require user functions to be referenced in a rather awkward manner, and subroutines (value-less procedures) in even more awkward ways. FORTRAN requires subroutines to be invoked using the keyword "CALL". COBOL is even worse: it uses the longer keyword "PERFORM" for internal subroutines, and two keywords "CALL ... USING" for external subroutines. [IBM70a] Moreover, for many years it took three statements to call an external procedure:

```
ENTER LINKAGE.
CALL FOO USING ARG1 ARG2 ARG3.
ENTER COBOL.
```

[IBM68] [DEC69] (One notable exception to this mess is LISP. There are also some extensible algebraic languages available, such as EL/1 [Weg71] [Weg74]; many of these are in fact implemented in a LISP-like manner beneath a veneer of ALGOL-like syntax.) It is generally true that we tend to believe that the expense of a programming construct is proportional to the amount of writer's cramp it causes us (by a "belief" I mean here an unconscious tendency rather than a fervent conviction). Indeed, this is not a bad psychological principle for language designers to keep in mind. We think of addition as cheap partly because we can notate it with a single character: "+". Even if we believe that a construct is expensive, we will often prefer it to a cheaper one if it will cut our writing effort in half. This is a lesson that practitioners of programming discipline have been trying to sell us, but it is a good one only if our programming languages are designed to cooperate with our natural tendency toward brevity.

In [Dij76,xvii] Dijkstra comments:

> "... it therefore hurts me to see the semantics of the repetitive construct
>
> 'while B do S'
>
> defined as that of the call
>
> 'whiledo(B,S)'
>
> of the recursive procedure (described in ALGOL 60 syntax):
>
> ```
> procedure whiledo(condition,statement);
> begin if condition then begin statement;
>                             whiledo(condition,statement) end
> end  "
> ```

It hurts me too, partly because Dijkstra here uses call-by-name to an indefinite number of levels, but even more because the syntax of ALGOL 60 makes the example twice as frightening as it really is. The LISP version

```
((LABEL LOOP
        (LAMBDA () (IF (NOT B)
                       (PROGN S (LOOP))))))
```

is at least slightly less awesome to me (though of course it is still more awesome than simply "while B do S"). {Note Step Variables}

As an additional defense of the procedure call, it should be pointed out that it constitutes a universal construct when properly implemented. The practitioners of programming discipline point with pride to such theorems as that of Boehm and Jacopini [Boe66], showing that any program can be written using their favored constructs. Such theorems have recently been ballyhooed about to the point of absurdity:

> "Structured programming is based on the mathematically proven Structure Theorem." [Nei76]

It has even been demonstrated, as a mathematical curiosity, that IF-THEN-ELSE can be dispensed with. [Pre75] It ought not be forgotten, however, that procedure calls can easily simulate sequencing, conditionals, and repetition, while the converse is decidedly not true. Even without completely solving the so-called FUNARG problem [Mos70], a surprising variety of tricks can be played with procedure calls, including dynamic binding and iteration. If the FUNARG problem is solved, additional tricks are easily played, such as escape

expressions, call-by-name, procedural data types, etc. The interested reader is referred to [Ste76a] and [Ste76b] for some examples of how this is done.

D. What Are We Doing About It?

Up to now, we have spent more time ignoring or circumventing the problem than solving it. Yourdon says "In most cases, a modular approach should not require more than 5-10% extra CPU time; this seems to be a reasonable price to pay..." [You75,99] I would suggest that this price is totally unreasonable when the technology (or, more accurately, the design philosophy) exists to reduce it!

There has been some mathematical work done on recursion removal [Str71] [Dar76] which is aimed both at converting procedure calls to GOTO statements and at transforming programs into other forms requiring less recursion. Some of this work is both mathematically interesting and practically applicable. Sometimes, however, it has gone up a garden path under the influence of the "expensive procedure call" myth. One example is a paper by Auslander and Strong [Aus76] which describes a technique for removing recursions from PL/I programs. This involves a set of source-to-source transformations which convert PL/I function calls into GOTO statements. Extra stacks are introduced in the form of arrays (though in their example they use an already existing array by means of an extremely clever trick) which are used to contain saved values of variables and return addresses. To put it quite simply (though they do not), they compile the PL/I program into another PL/I program which is more like machine language, and which the real PL/I compiler can therefore process more easily. They report that this technique improves run time by 40%, and space used per level of recursion from 336 bytes to 9, a 97% saving!

This seems impressive until we realize that their transformations are almost entirely straightforward and mechanical and could easily be made a part of the PL/I compiler, and furthermore that they are essentially techniques which have been used by the MacLISP compiler and others for almost a decade: turning procedure calls into GOTO when possible, and avoiding pushing variable values unless necessary! Then we are impressed only by the inefficiency of this so-called "optimizing" PL/I compiler!

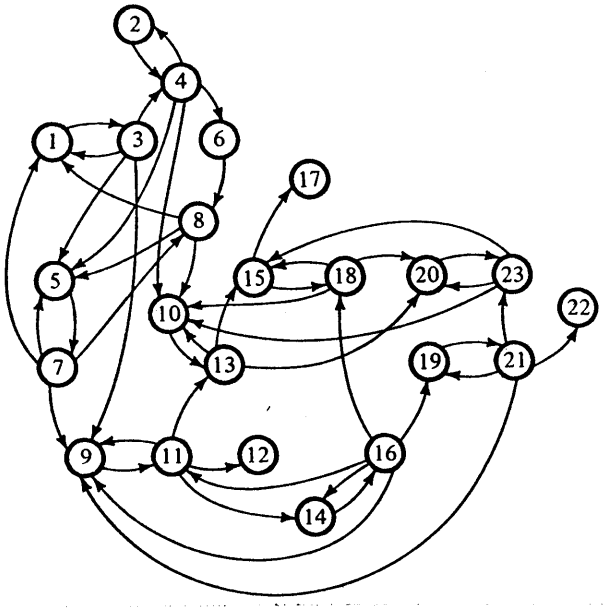
What is more, Auslander and Strong conclude:

> "These techniques can be applied to a program without an understanding of its purpose. However, they are complex enough so that we are inclined to teach them as tools for programmers rather than try to mechanize them as an option in an optimizing compiler."

In other words, we are now so afraid of procedure calls that, rather than fix our compilers, we wish to teach programmers complex techniques for using GOTO to circumvent the shortcomings of our compilers! Such a desire is completely outrageous. Not only does it violate the philosophical principles of clarity in language design and programming style we have slowly been forced to accept, but it is demonstrably ridiculous, because while the complete generality of these techniques has perhaps not been implemented in a compiler, a good part of it has, and has worked for eight to ten years at least. Such is the ludicrous position we have come to.

E. The Expressive Power of Procedure Calls

Here we shall consider an example of how expressive procedure calls can be when used freely. The example is taken from Yourdon [You75,233]; he implies that the program was an actual one in use. He suggests that, rather than using many flag variables to indicate various conditions within a program, one should use a single variable which encodes the state of the program. The motivation behind this is that one should also draw a state diagram showing the valid transitions between states; the programmer is encouraged to think of his program as a finite-state automaton. In this way one can avoid the common error of producing an invalid combination of flag values. The state diagram for Yourdon's example is reproduced here.



The 128 possible combinations of seven independent flags have been reduced to 23 permissible states and the legal transitions among them.

The program itself is to be implemented using two variables OLD-STATE and NEW-STATE and a computed-GOTO or CASE statement. The main program dispatches to the module designated by NEW-STATE. The module can then check OLD-STATE to be sure the transition was valid. For example, module 14 would ensure that OLD-STATE held 11 or 16. When module 14 is done, it must store 14 into OLD-STATE and set NEW-STATE to the next state, and then branch back to the computed GOTO or CASE. It is assumed, though not stated, that all variables common to several modules are declared in an outer environment accessible to the modules.

We shall modify this set-up slightly to make it more foolproof. The first problem is that every module must contain code to manipulate OLD-STATE and NEW-STATE, and it is easy to forget to include this code. We shall perform this work within the CASE statement itself; this collects the transition information into one place. In order to avoid branching to the CASE statement, we shall embed the CASE statement in a loop. To terminate the loop, we will allow state 0 to represent the exit state. Finally, we shall rename the variables NEW-STATE and OLD-STATE to PROGRAM-COUNTER and OLD-PC. The resulting code looks like this:

```
UNTIL PROGRAM-COUNTER = 0 DO
    CASE PROGRAM-COUNTER OF
        1:   BEGIN
               IF NOT (OLD-PC = 3
                       OR OLD-PC = 7
                       OR OLD-PC = 8)
                  THEN ERROR;
               OLD-PC := PROGRAM-COUNTER;
               PERFORM MODULE1;
             END;
        2:   ...
        ...
        23:  BEGIN
               IF NOT (OLD-PC = 20
                       OR OLD-PC = 21)
                  THEN ERROR;
               OLD-PC := PROGRAM-COUNTER;
               PERFORM MODULE23;
             END;
    ENDCASE;
```

The code for each module must end with a statement that assigns a new value to PROGRAM-COUNTER. The use of a number to identify the next module to execute obscures the code, so let us assume that we can use symbolic names (defined by a PARAMETER statement, for example). Let us also assume the trivial syntactic sugar:

```
JUMP x    means    PROGRAM-COUNTER := x
```

Then at the end of each module we can write a JUMP statement to the next module. For example:

```
PROCEDURE MODULE23;
    BEGIN
        <do processing>;
        IF <test1> THEN JUMP MODULE10
            ELSE IF <test2> THEN JUMP MODULE15
            ELSE JUMP MODULE20;
    END;
```

Let us pause for a few observations. First, the outer loop of our program may be compared to a hardware CPU, and the modules to the instructions of that CPU. It has a program counter which takes us from instruction to instruction. (For an example of this style of programming in LISP, see [Sus75].) Second, our nice program has become a rat's nest of JUMP statements (which might look more familiar had we used the keyword GOTO instead of JUMP). This is not at all surprising, since the structure of our program merely reflects the structure of our problem as exhibited in the state-transition diagram, and that too is a rat's nest. Third, our attempt to improve the program by using a nice, structured approach has resulted in the effective use of GOTO all over the place!

We note that the use of symbolic names in JUMP statement reduces the probability of errors in describing state transitions, and so we may feel confident in removing the error-checking code from the main loop. (Moreover, a cross-indexing program can find all the references to each module for us, which could not easily be done when numbers were used.) We furthermore can

eliminate the outer loop and CASE statement entirely; all that would be needed is GOTO statements at the end of each module, linking them together. This will speed up the program by eliminating the PROGRAM-COUNTER variable, without appreciably altering the structure of the program. Unfortunately, this produces a rat's nest of true GOTO statements, which is not structured.

This points up to some extent the ultimate futility of the Boehm-Jacopini theorem. We can certainly express all programs in a structured form by introducing flag variables, but the preceding series of reasonable transformations merely renders it unstructured again, demonstrating that we had not really made the program structured in nature, but only in form.

There is another way out. Let us not use GOTO statements to jump between modules, and let us not use flag variables to signal what are effectively GOTO statements to an outer control loop. Instead, let us consider what a single module does: it performs its bit of processing, and then invokes or otherwise designates another module to complete processing. Suppose therefore that at the end of every module we had a procedure call to the next module:

```
PROCEDURE MODULE23;
    BEGIN
        <do processing>;
        IF <test1> THEN PERFORM MODULE10
            ELSE IF <test2> THEN PERFORM MODULE15
            ELSE PERFORM MODULE20;
    END;
```

where we might as well have written "CALL" for "PERFORM".

This will certainly do what we want; if MODULE23 calls MODULE10, then MODULE10 will carry on, calling other modules in the process, and eventually the program will complete. It is also "structured"; the only constructs used are sequencing, possibly conditionals, and procedure calls. It uses no GOTO statements. There is also an additional bonus: if MODULE23 wants to pass some information to MODULE10, it can pass them as parameters rather than having to use global variables. This can help prevent conflicting usages of global variables among disjoint module sets.

From this example we can induce the following theorem:

> Any flowchart can be written as a program which uses only sequencing, conditionals, and procedure calls.

This theorem is not new; it appears in McCarthy's 1960 paper. [McC60] It is quite easy to prove. We assume without loss of generality that the boxes of the flowchart are of two sorts: process boxes with one exit, and conditional boxes with two. A process box may contain any single-exit computation whatsoever (which may be built up from sequencing, conditional, and while-do loops, for example). A conditional may contain a predicate which decides which of two exits to take.

For each box in the flowchart we construct a procedure. If a box A is a process box whose exit branch leads to box B, then the procedure for A is:

```
PROCEDURE A; BEGIN <processing>; CALL B END;
```

If a box C is a conditional box whose exit branches lead to boxes D and E, the the procedure for C is:

```
PROCEDURE C; IF <predicate> THEN CALL D ELSE CALL E;
```

Every module is "structured" in form; moreover, the modules are not necessarily as tiny in size as the examples indicate, since a process box may contain an arbitrarily large one-in/one-out program.

There are several possible objections to such a style of programming:

(1) It requires recursion to implement loops in the flowchart.
* That's right. If your programming language won't support recursion (e.g. most FORTRAN implementations), you can't use this particular "structured, modular approach".

(2) Procedure calls are expensive.
* They shouldn't be!

(3) The chain of procedure calls will keep pushing stack, and the stack will overflow.
* This is true of current programming language implementations, but it has been shown above that such implementations use far more stack than necessary. If tail-recursive procedure calls are compiled as JUMP instructions, then this problem does not arise.

(4) This style of programming is unnatural: "That's not what procedures are for!"
* This is largely a matter of taste. I have written a number of medium-sized programs in this style (using a dialect of LISP) and find it quite natural for certain purposes. It accomodates itself well to "state-transition" kinds of programs. It also permits one to create non-standard looping constructs which are one-in/one-out, but which have complex relationships among the variables being stepped.

An additional observation should be made, of course: in the example above, the use of procedure calls hasn't endowed the program with any more structure than the use of flag variables or PROGRAM-COUNTER did, compared with the GOTO version. All it has done is possibly make the code more palatable. This may be a useful psychological illusion, but it is as much a myth as the belief that procedure calls are inherently expensive.

F. Procedure Calls and Modularity

The primary role which the procedure call plays in the current philosophy of programming discipline is as the agent of modularity. Similarly, the primary role played by GOTO is as the agent of tangled flowgraphs.

I would like to suggest that our difficulties in dealing with programming and programming languages stem in part from a confusion between the abstract notions of program organization and the concrete embodiment of these notions as programming language constructs. In order to simplify our thinking we have attempted to enforce a one-to-one mapping between these two sets, and it doesn't work very well. For example, we decree that procedures are the method of producing modularity; that WHILE-DO loops are the way to iterate; that IF-THEN-ELSE is the way to produce conditionals; that GOTO is the way to produce peculiar program structures.

The fact is that this just isn't so. Consider the notion of modularity, which is indeed a useful concept for organizing programs. While procedure calls are indeed a method of modularizing programs, there are other methods. The PL/I %INCLUDE construct or the COBOL COPY construct are one alternative. Another is the "PRINLEVEL" feature in some LISP systems, which allows you to print the overall structure of a program while suppressing the detail of computations below a certain level of nesting. A third example (due to R. Zippel) is the common FORTRAN trick of breaking up a single complex assignment statement into several smaller ones:

```
      FOO = F(G(A,B,C,D) + H(A,B,C), G(C,D,A,B)
     1          - H(C,D,A), G(D,C,B,A) * H(D,C,B))
```

becomes

```
      FOO1 = G(A,B,C,D) + H(A,B,C)
      FOO2 = G(C,D,A,B) - H(C,D,A)
      FOO3 = G(D,C,B,A) * H(D,C,B)
      FOO = F(FOO1, FOO2, FOO3)
```

If someone had asked me whether assignment statements were an agent of modularity in programming languages, I should certainly have replied in the negative before seeing this example.

Similarly, consider the notion of iteration, another useful concept in organizing programs. We are all familiar with the use of WHILE-DO and its variants, and also with the use of IF-THEN-ELSE and GOTO to achieve the same purpose. But, as shown earlier, procedure calls can be an agent of iteration. While I would hesitate to write

```
procedure whiledo(condition,statement);
   begin if condition then
      begin statement;
            whiledo(condition,statement)
      end
   end;
whiledo(B,S);
```

for "WHILE B DO S", I would not hesitate to write

```
real procedure sqrt(arg);
   value arg;  real arg;
   begin
      real procedure sqrt1(approx);
         value approx; real approx;
         if abs(arg - approx*approx) < epsilon
            then approx
            else sqrt1((approx + arg/approx)/2);
      sqrt1(arg/2);
   end;
```

knowing, if necessary, that it could be compiled as an iteration rather than as a stack-pushing recursion. Of course, I would prefer not to have to write the "value" declarations, and might prefer some other notation, such as LISP notation, but the essential idea is the same.

It is even possible to use procedure calls to implement conditional expressions, though this has heretofore been largely a curiosity for students of lambda calculus. [Chu41] Similarly, many assignment statements can be modelled and even implemented through the use of procedure calls. I have written two LISP compilers which use procedure calls to implement all assignments and iterations within the compiler. [Ste77a] I have used these compilers to compile themselves, and there seems to have been no demonstrable sacrifice of speed due to the use of procedure calls. Moreover, the code, totalling some seventy pages of source text, has been extremely easy to modify and maintain.

The important point is that for each important programming concept which we have found useful -- modularity, iteration, assignment, conditionals

-- there may exist more than one programming language construct which can embody that concept in a reasonably natural manner. Furthermore, it sometimes requires more than one construct to properly embody a given concept. For example, WHILE-DO would be utterly useless in expressing iteration if some form of assignment statement (or other side effect) were not also used!

In understanding (a piece of) a program it is necessary to determine not only what construct is being used, but the concept it is intended to express. While we may prefer to think of a procedure call as meaning "go do this other module and then come back", this is only one possible interpretation. We must realize that there are situations where coming back doesn't matter (i.e. the tail-recursive cases), and these situations may be exploited. Just as a concept such as modularity may be expressed by diverse constructs, so may a language construct be interpreted in various ways, some of which may lead to superior compilation techniques. {Note Various Optimizations} One example of this is the tail-recursive procedure call; another is the logical expression occurring in the predicate of a conditional, which does not actually have to produce a Boolean value when compiled (this is called "anchor pointing" in [Allt72]).

It is not unreasonable to want to be able to infer the intent of a piece of code from the particular construct used to express it. If only GOTO is used to express all control structure, it is hard to identify the conceptually important notions of conditional, iteration, and escape occurring in the program. It is important that alternative modes of expression exist; but the mere banishing of one abused alternative is unlikely of itself to cause correct usage of the others. Instead, a language should be so designed that one is encouraged to use a construct if, and only if, it is appropriate; it most also provide enough constructs to cover all reasonable programming concepts. Otherwise, programmers will merely misuse the constructs they are given (and most programmers are very good at this!). The structure of a program is dictated largely by the structure of the problem. If the problem solution is best expressed as a finite-state automaton, for example, then no amount of structured camouflage will help that fact.

This is the essential frustration we have experienced with GOTO. We discovered that GOTO was often being used even in situations where more appropriate and expressive constructs were available, and that it was being used for all sorts of purposes we hadn't anticipated, and so we sought to eliminate unwanted concepts and programming styles by banning a construct. This is as fruitless as eliminating iteration by banning DO-loops would be (people would just use GOTO or procedure calls), or eliminating recursion by banning procedure calls (people would, and do, simulate it by using an array as a stack). We need to get a better grasp on organizational concepts and their relationship to the individual constructs which make up our languages.

Conclusions

Procedure calls are demonstrably not inherently as inefficient as computing folklore would lead us to believe. There are implementations of higher-level programming languages in which procedure calls are almost as cheap as GOTO statements.

Not all procedure calls need push a return address. "Tail-recursive" procedure calls (those occurring at the end of a procedure) can be compiled almost as if they were simple GOTO statements. In fact, procedure calls can uniformly be treated as GOTO statements which pass parameters, with return addresses being pushed at a conceptually different point (the commencement of argument evaluation). Such a technique reduces the amount of stack space

needed, provided lexical scoping is used (as in ALGOL) or a subset of lexical scoping (as is largely true of FORTRAN and COBOL). Even languages with dynamic scoping rules, such as APL and some LISP dialects, can use this technique in situations where dynamic references are not involved.

Procedure calls permit an extraordinarily powerful style of programming which, even though it is completely "structured", includes most rat's nests of GOTO statements as a subset. This style merely involves writing a procedure call where one would ordinarily write a GOTO at the end of a procedure. (The technique will not reproduce the "escape expression" effect of writing a GOTO from inside a loop to outside the loop, however.) This style is sufficiently powerful to represent any flowchart without introducing flag variables or GOTO statements. Furthermore, this style of programming is a natural way to write certain kinds of commonly occurring programs. The use of this style does not depend on procedure calls being cheap or being compiled as tail-recursive branches, though if they are so compiled running time is reduced and less stack is consumed, which are desirable characteristics apart from the issue of style.

We might wonder why such rat's nests are not written using procedure calls in practice, when they are certainly possible and violate no rules of "structured" programming. The answer is probably that GOTO statements, being "cheap", are used freely enough to produce rat's nests, while procedure calls, being "expensive", are used sparingly. We therefore come to the paradoxical conclusion that improving the implementation of procedure calls is a mixed blessing; we can improve our programs both in time and space, but we may bring on the same problems we had with GOTO by encouraging the use of procedure calls in stylistically diverse ways. We could simply ignore the whole thing, and go on letting procedure calls be expensive, in order to discourage their use; but this would not be intellectually honest. It is appropriate that we should have a healthy respect for procedure calls, and use them sparingly; but we should respect them because they are powerful, and not because they are "expensive".

Acknowledgements

Discussions with Mike Genesereth, Richard Stallman, and Richard Zippel illuminated many key points. Johan DeKleer, Jon Doyle, Tom Knight, and Richard Greenblatt also provided useful comments. Gerry Sussman was, as always, a great source of enlightenment.

Carl Hewitt and Richard Stallman provided additional useful comments which led to the notes, which were added after final submission of the paper to ACM '77.

Notes

{Note Omniscient Implementors}

One can argue quite strongly that there are so large a number (possibly infinite) of distinct useful control constructs that no one language could embody them all, and that therefore no language designer should be so conceited as to think he has encompassed all desirable constructions in a given language. By this reasoning, the omission of CASE, or Dahl loops, or event constructions, or whatever else is not a matter of neglect, but of necessity: you just can't foresee them all.

(This brings out a serious flaw in the present theory of structured programming; by assuming that all programs can conveniently be written using only certain structures, it implicitly assumes that the problems to be solved by these programs have solutions which can be decomposed using these structures. We have never seen any justification advanced for this latter assumption; and indeed, there are many counterexamples, such as Yourdon's "finite-state machine" program mentioned in the text.)

Until a much more advanced theory of programming is devised, designers of practical languages are well advised to leave in "ugly hooks" like GOTO, even if also discouraging their use except in emergencies. After all, using GOTO to simulate a peculiar control construct is probably preferable to a convoluted perversion of a more specialized construct.

{Note Shuffle Arguments}

To elucidate this point further, suppose that function arguments are passed on the stack (above the return address). Then, using a true stack discipline plus tail-recursion, if there are intermediate results or other data above that return address, the arguments to be passed must be moved down over this other data so that they will be in the correct position. This is particularly tricky because these positions are probably also where the arguments passed to you were stored. For example, suppose A calls B, and B calls C tail-recursively. Then A passes a return address R and arguments to B, and B wishes to pass R and different arguments to C. B must replace its arguments from A with the new ones for C. This entails some shuffling of the stack positions.

The need to shuffle stack positions can be alleviated by passing arguments through registers, but this in turn usually requires shuffling of registers. Another way out is to use a more general form of stack, such as the so-called "spaghetti" [Bob73] or "macaroni" [Ste77c] stacks. Under such a scheme there is no need to shuffle old arguments away so that new arguments to be passed may occupy their positions; instead, each stack frame has a pointer to the next one, and two stack frames may both point to a third. Thus B would build a new stack frame F' pointing to the one G containing R; B's arguments remain in frame F, which also points to G. On calling C, F' is passed to C and F is discarded.

{Note Step Variables}

A far more important point not mentioned in the main text is that procedures not only can easily express the control structure of various kinds of loops, but also provide a natural way to express the stepping of the variables. Consider the loop for "iterative factorial" written in terms of a LISP LABEL construct:

```
((LABEL LOOP
        (LAMBDA (M A)
                (IF (= M 0)
                    A
                    (LOOP (- M 1) (* M A)))))
 N 1)
```

Compare this with the Algol version:

```
begin
   A := 1;
   for M := N step -1 until 0 do
      A := A * M;
end
```

As it happens, in the Algol version we could absorb the stepping of one of the variables into a for construction. However, the nature of the loop is that two variables are stepped, and the Algol version makes one of them very hard to see! The stepping must be expressed through a side-effect (assignment) to a variable global to the loop. The LISP version has identical semantics but proceeds without explicit side-effect, and expresses the stepping of the two variables in the same manner. (Cf. the PLASMA version given in [Hew77].)

Procedure calls also allow one to express escapes and non-standard loops. Consider the table-search example from [Knu74], expressed here in terms of procedure calls:

```
procedure search(a,b,n,key);
    begin
        procedure search loop(i);
            if i>n then
                begin
                    n:=n+1;
                    a[n]:=key;
                    b[n]:=1
                end
            else if a[i]=key then
                b[i]:=b[i]+1
            else search loop(i+1);
        search loop(1)
    end
```

The structure of the algorithm to be performed is a loop with two possible exit points. This is easily expressed by procedure calls, because we can specify for each branch of the if-then-else whether or not to take another cycle of the loop. (In fact, we can argue for this style on the basis of an important primitive principle: any notation should accentuate the unusual and make unobtrusive the usual. Now for a loop there are two cases when the body is done: take another cycle, or exit the loop. Now exiting the loop is the unusual case at run time, because we expect to iterate many times for each time a loop is exited; this leads us to design loop syntaxes which accentuate exit conditions. We may ponder, however, the fact that textually there will be one or more iteration points and one or more exit points. In the case of while-do, there is one of each. If we want to have while-do with multiple exits, then we should accentuate the looping action, and de-emphasize the exiting action. This occurs in the version of "search loop" given above.)

An additional advantage of the procedural mode of expression is that

procedure entry points are ideal places to make assertions about the state of the process. The procedure header lists the variable quantities of interest; in the case of a loop expressed in terms of procedure calls (as above), the procediure header mentions explicitly all the variables to be stepped by each cycle of the loop.

{Note Various Optimizations}

Other research has attacked the "expense" of procedure calls from other directions; notable successes have been achieved with the techniques of procedure integration ([All72] [Atk76] [Sch77] and many others) and recursion removal ([Str71] [Aus76] [Dar76]). Much of this effort has been apparently motivated by the notion that procedure calls are expensive and should be done away with in some way. Complementing this is the idea that procedure calls are indeed valuable for their expressive power, and they should be retained and compensated for rather than banned entirely.

We take the slightly different point of view that procedure calls are valuable, but that they do not map one-to-one to the various low-level primitives made available on existing hardware. A given procedure call may, depending on context, be mapped to any one of a number of low-level implementations, some of which are markedly more efficient than others. Up to now, most "optimizing" compilers have had knowledge about the many equivalent ways of compiling arithmetic or array-indexing expressions and how to choose the most efficient, but have had only a single, most general method of compiling procedure calls per se.

What we have tried to stress in the first half of this paper is that this most general method is often much more general than necessary, even for a universal method. We have described another method which is also semantically universal, but which is more efficient and just as easy for a compiler to deal with. We believe that the psychological effect of this new method will be the most important, for it does away with the automatic reflexive thought that "procedure calls always return". (Imagine, for example, that we had always thought of GOTO as branching forward and never backward, under the influence of old paper-tape machines. Until we had dispelled this notion, could we ever have seen the abstract pattern of _while_-_do_?) Once we realize that procedure calls are semantically a superset of GOTO, we are freed to exploit a far more expressive style. It then becomes our task to analyze this style, and to isolate from it important special cases, just as from the maze of GOTO patterns we have isolated such important structures as _while_-_do_.

Special techniques for compiling these special cases are not mere tricks; they reflect the possibility that the programmer had just such a special case in mind when he wrote the code, but was forced (by the "graininess" of the language) to use a more general piece of syntax to express it than he might have liked. While the program as a whole will reflect the originally intended concept, it is unlikely that the syntactic decomposition of the program will correspond in any precise way to the semantic decomposition of the concept. The compiler writer must realize that what the programmer writes is not always precisely what he wants, but only the closest expression thereof permitted by the language. ("I know you believe you understand what you think I said. But I am not sure you realize that what you heard is not what I meant." -- Anon.) The compiler writer must therefore avoid a monistic interpretation of the language definition, and try to determine from a given program the best of all possible intentions and produce code accordingly.

References

[All72] Allen, Frances E., and Cocke, John. "A Catalogue of Optimizing Transformations." In Rustin, Randall (ed.), Design and Optimization of Compilers. Proc. Courant Comp. Sci. Symp. 5. Prentice-Hall (Englewood Cliffs, N.J., 1972).

[ANS76] American National Standards Institute. Draft proposed ANS FORTRAN (BSR X3.9). Reprinted as SIGPLAN Notices 11, 3 (March 1976).

[Atk76] Atkinson, Russell R. Optimization Techniques for a Structured Programming Language. S.M. Thesis. MIT (Cambridge, 1976).

[Aus76] Auslander, M.A., and Strong, H.R. Systematic Recursion Removal. Report RC 5841 (#25283) IBM T.J. Watson Research Center (Yorktown Heights, New York, February 1976).

[Bob73] Bobrow, Daniel G. and Wegbreit, Ben. "A Model and Stack Implementation of Multiple Environments." Comm. ACM 16, 10 (October 1973) pp. 591-603.

[Boe66] Boehm, Corrado, and Jacopini, Guiseppe. "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules." Comm. ACM 9, 5 (May 1966), 366-371.

[Chu41] Church, Alonzo. The Calculi of Lambda Conversion. Annals of Mathematics Studies Number 6. Princeton University Press (Princeton, 1941). Reprinted by Klaus Reprint Corp. (New York, 1965).

[Dar76] Darlington, J., and Burstall, R.M. "A System which Automatically Improves Programs." Acta Informatica 6 (1976), 41-60.

[DEC69] Digital Equipment Corporation. PDP-10 COBOL Language Programmer's Reference Manual. DEC-10-KC1A-D (Maynard, Mass., 1969).

[Dij76] Dijkstra, Edsger W. A Discipline of Programming. Prentice-Hall (Englewood Cliffs, N.J., 1976).

[Fat73] Fateman, Richard J. "Reply to an Editorial." SIGSAM Bulletin 25 (March 1973), 9-11.

[Hew77] Hewitt, Carl. "Viewing Control Structures as Patterns of Passing Messages." AI Journal 8, 3 (June 1977), 323-364.

[Hop73] Hopper, Captain Grace Murray. In "An Interview with Captain Grace Murray Hopper, USNR". Computing (October 10, 1973). Reprinted in SIGPLAN Notices 9, 1 (January 1974), 3-6.

[IBM68] International Business Machines. IBM System/360 Operating System COBOL Language. Form C28-6516-8. Ninth Edition (November 1968).

[IBM70a] International Business Machines. IBM System/360 Operating System American National Standard COBOL. Form GC28-6396-2. Third edition (June 1970).

[IBM70b] International Business Machines. IBM System/360 Operating System PL/I (F) Language Reference Manual. Form GC28-8201-3. Revised (November 1970).

[Jen72] Jenks, Richard D., and Griesmer, James H. "Editor's Comment." SIGSAM Bulletin No. 24 (October 1972), 2-3.

[Knu74] Knuth, Donald E. "Structured Programming with GO TO statements." Computing Surveys 6, 4 (December 1974).

[McC60] McCarthy, John. "Recursive functions of symbolic expressions and their computation by machine - I." Comm. ACM 3, 4 (April 1960), 184-195.

[McK65] McKeeman, W.M. "Peephole optimization." Comm. ACM 8, 7 (July 1965), 443-444.

[Mos70] Moses, Joel. The Function of FUNCTION in LISP. AI Memo 199, MIT AI Lab (Cambridge, June 1970).

[Nei76] Neighbors, Michael A. "Assuring Software Reliability." Computer Decisions 8, 12 (December 1976), 44-46.

[Pre75] Presser, Leon. "Structured Languages." Proc. National Computer Conference 1975. Reprinted in SIGPLAN Notices 10, 7 (July 1975), 22-24.

[Sch77] Scheifler, Robert W. "An Analysis of Inline Substitution for a Structured Programming Language." Comm. ACM 20, 9 (September 1977), 647-654.

[Ste76a] Steele, Guy Lewis Jr., and Sussman, Gerald Jay. LAMBDA: The Ultimate Imperative. AI Memo 353. MIT AI Lab (Cambridge, March 1976).

[Ste76b] Steele, Guy Lewis Jr. LAMBDA: The Ultimate Declarative. AI Memo 379. MIT AI Lab (Cambridge, November 1976).

[Ste77a] Steele, Guy Lewis Jr. Compiler Optimization Based on Viewing LAMBDA as RENAME plus GOTO. S.M. Thesis. MIT AI Lab (Cambridge, May 1977).

[Ste77b] Steele, Guy Lewis Jr. "Fast Arithmetic in MacLISP." Proc. 1977 MACSYMA Users' Conference. NASA Sci. and Tech. Info. Office (Washington, D.C., July 1977), 215-224.

[Ste77c] Steele, Guy Lewis Jr. "Macaroni is Better than Spaghetti." Proc. AI and Programming Languages Conf. (Rochester, New York, August 1977). SIGPLAN Notices 12, 8, SIGART Newsletter 64 (August 1977), 60-66.

[Str71] Strong, H.R., Jr. "Translating Recursion Equations into Flow Charts." Journal of Computer and System Sciences 5, 3 (June 1971), 254-285.

[Sus75] Sussman, Gerald Jay, and Steele, Guy Lewis Jr. SCHEME: An Interpreter for Extended Lambda Calculus. AI Memo 349. MIT AI Lab (Cambridge, December 1975).

[Syk77] Sykes, Roy A., Jr. "Whizbang of the Month: Branching and Iteration." Scientific Time Sharing Corporation News 2, 10 (Bethesda, Maryland, January-February 1977), 5-6.

[Weg71] Wegbreit, Ben. "The ECL Programming System." Proc. AFIPS 1971 FJCC, Vol. 39. AFIPS Press, Montvale, N.J. pp. 253-262.

[Weg74] Wegbreit, Ben, et al. ECL Programmer's Manual. Technical Report 23-74. Center for Research in Computing Technology, Harvard U. (Cambridge, December 1974).

[Wul71] Wulf, W.A., Russell, D.B., and Habermann, A.N. "BLISS: A Language for Systems Programming." Comm. ACM 14, 12 (December 1971), 780-790.

[Wul73] Wulf, William A., and Shaw, Mary. "Global Variable Considered Harmful." SIGPLAN Notices 8, 2 (February 1973), 28-34.

[Wul75] Wulf, William A., et al. The Design of an Optimizing Compiler. American Elsevier (New York, 1975).

[You75] Yourdon, Edward. Techniques of Program Structure and Design. Prentice-Hall (Englewood Cliffs, N.J., 1975).