MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

AI Memo 379

November 1976

LAMBDA

THE ULTIMATE DECLARATIVE

by

Guy Lewis Steele Jr. *

Abstract:

In this paper, a sequel to LAMBDA: The Ultimate Imperative, a new view of LAMBDA as a renaming operator is presented and contrasted with the usual functional view taken by LISP. This view, combined with the view of function invocation as a kind of generalized GOTO, leads to several new insights into the nature of the LISP evaluation mechanism and the symmetry between form and function, evaluation and application, and control and environment. It also complements Hewitt's actors theory nicely, explaining the intent of environment manipulation as cleanly, generally, and intuitively as the actors theory explains control structures. The relationship between functional and continuation-passing styles of programming is also clarified.

This view of LAMBDA leads directly to a number of specific techniques for use by an optimizing compiler:

(1) Temporary locations and user-declared variables may be allocated in a uniform manner.
(2) Procedurally defined data structures may compile into code as good as would be expected for data defined by the more usual declarative means.
(3) Lambda-calculus-theoretic models of such constructs as GOTO, DO loops, call-by-name, etc. may be used directly as macros, the expansion of which may then compile into code as good as that produced by compilers which are designed especially to handle GOTO, DO, etc.

The necessary characteristics of such a compiler designed according to this philosophy are discussed. Such a compiler is to be built in the near future as a testing ground for these ideas.

Keywords: environments, lambda-calculus, procedurally defined data, data types, optimizing compilers, control structures, function invocation, temporary variables, continuation passing, actors, lexical scoping, dynamic binding

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

Contents

1. A Different View of LAMBDA	1
1.1. Primitive Operations in Programming Languages	1
1.2. Function Invocation: The Ultimate Imperative	2
1.3. LAMBDA as a Renaming Operator	7
1.4. An Example: Compiling a Simple Function	8
1.5. Who Pops the Return Address?	11
2. Lexical and Dynamic Binding	12
3. LAMBDA, Actors, and Continuations	16
3.1. Actors = Closures (mod Syntax)	16
3.2. The Procedural View of Data Types	20
4. Some Proposed Organization for a Compiler	25
4.1. Basic Issues	25
4.2. Some Side Issues	27
5. Conclusions	29
Appendix A. Conversion to Continuation-Passing Style	30
Appendix B. Continuation-Passing with Multiple Value Return	36
Notes	39
References	42

Acknowledgements

Thanks are due to Gerald Sussman, Carl Hewitt, Allen Brown, Jon Doyle, Richard Stallman, and Richard Zippel for discussing the issues presented here and for proofreading various drafts of the document.

An earlier version of this document was submitted in April 1976 to the Department of Electrical Engineering and Computer Science at MIT in the form of a proposal for research towards a Master's Thesis.

1. A Different View of LAMBDA

Historically, LAMBDA expressions in LISP have been viewed as functions: objects which, when applied ordered sets of arguments, yield single values. These single values typically then become arguments for yet other functions. The consistent use of functions in LISP leads to what is called the applicative programming style. Here we discuss a more general view, of which the functional view will turn out to be a special case. We will consider a new interpretation of LAMBDA as an environment operator which performs the primitive declarative operation of renaming a quantity, and we will consider a function call to be a primitive unconditional imperative operator which includes GOTO as a special case. (In an earlier paper [Steele 76] we described LAMBDA as "the ultimate imperative". Here we assert that this was unfortunately misleading, for it is function invocation which is imperative.)

1.1. Primitive Operations in Programming Languages

What are the primitive operations common to all high-level programming languages? It is the data manipulation primitives which most clearly differentiate high-level languages: FORTRAN has numbers, characters, and arrays; PL/I has strings and structures as well; LISP has list cells and atomic symbols. All have, however, similar notions of control structures and of variables.

If we ignore the various data types and data manipulation primitives, we find that only a few primitive ideas are left. Some of these are:

Transfer of control
Environment operations
Side effects
Process synchronization

Transfer of control may be subdivided into conditional and unconditional transfers. Environment operations include binding of variables on function entry, declaration of local variables, and so on. Side effects include not only modifications to data structures, but altering of global variables and input/output. Process synchronization includes such issues as resource allocation and passing of information between processes in a consistent manner.

Large numbers of primitive constructs are provided in contemporary programming languages for these purposes. The following short catalog is by no means complete, but only representative:

Transfer of control
- Sequential blocks
- GOTO
- IF-THEN-ELSE
- WHILE-DO, REPEAT-UNTIL, and other loops
- CASE
- SELECT
- EXIT (also known as ESCAPE or CATCH/THROW)
- Decision tables

Environment operations
- Formal procedure parameters
- Declarations within blocks
- Assignments to local variables
- Pattern matching

Side effects
- Assignments to global (or COMMON) variables
- Input/output
- Assignments to array elements
- Assignments to other data structures

Process synchronization
- Semaphores
- Critical regions
- Monitors
- Path expressions

Often attempts are made to reduce the number of operations of each type to some minimal set. Thus, for example, there have been proofs that sequential blocks, IF-THEN-ELSE, and WHILE-DO form a complete set of control operations. One can even do without IF-THEN-ELSE, though the technique for eliminating it seems to produce more rather than less complexity. {Note No IF-THEN-ELSE} A minimal set should contain primitives which are not only universal but also easy to describe, simple to implement, and capable of describing more complex constructs in a straightforward manner. This is why the semaphore is still commonly used; its simplicity makes it is easy to describe as well as implement, and it can be used to describe more complex synchronization operators. The expositors of monitors and path expressions, for example, go to great lengths to describe them in terms of semaphores [Hoare 74] [Campbell 74]; but it would be difficult to describe either of these "high-level" synchronization constructs in terms of the other.

With the criteria of simplicity, universality, and expressive power in mind, let us consider some choices for sets of control and environment operators. Side effects and process synchronization will not be treated further in this paper.

1.2. Function Invocation: The Ultimate Imperative

The essential characteristic of a control operator is that it transfers control. It may do this in a more or less disciplined way, but this discipline is generally more conceptual than actual; to put it another way, "down underneath, DO, CASE, and SELECT all compile into IFs and GOTOs". This is why many people resist the elimination of GOTO from high-level languages; just as the semaphore seems to be a fundamental synchronization primitive, so the GOTO seems to be a fundamental control primitive from which, together with IF, any more complex one can be constructed if necessary. (There has been a recent controversy over the nested IF-THEN-ELSE as well. Alternatives such as repetitions of tests or decision tables have been examined. However, there is no denying that IF-THEN-ELSE seems to be the simplest conditional control operator easily capable of expressing all others.)

One of the difficulties of using GOTO, however, is that to communicate information from the code gone from to the code gone to it is necessary to use global variables. This was a fundamental difficulty with the CONNIVER language [McDermott 74], for example; while CONNIVER allowed great flexibility in its control structures, the passing around of data was so undisciplined as to be completely unmanageable. It would be nice if we had

some primitive which passed some data along while performing a GOTO.

It turns out that almost every high-level programming language already has such a primitive: the function call! This construct is almost always completely ignored by those who catalog control constructs; whether it is because function calling is taken for granted, or because it is not considered a true control construct, I do not know. One might suspect that there is a bias against function calling because it is typically implemented as a complex, slow operation, often involving much saving of registers, allocation of temporary storage, etc. {Note Expensive Procedures}

Let us consider the claim that a function invocation is equivalent to a GOTO which passes some data. But what about the traditional view of a function call which expects a returned value? The standard scenario for a function call runs something like this:

[1] Calculate the arguments and put them where the function expects to find them.
[2] Call the function, saving a return address (on the PDP-10, for example, a PUSHJ instruction is used, which transfers control to the function after saving a return address on a pushdown stack).
[3] The function calculates a value and puts it where its caller can get it.
[4] The function returns to the saved address, throwing the saved address away (on the PDP-10, this is done with a POPJ instruction, which pops an address off the stack and jumps to that address).

It would appear that the saved return address is necessary to the scenario. If we always compile a function invocation as a pure GOTO instead, how can the function know where to return?

To answer this we must consider carefully the steps logically required in order to compute the value of a function applied to a set of arguments. Suppose we have a function BAR defined as:

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

In a typical LISP implementation, when we arrive at the code for BAR we expect to have two computed quantities, the arguments, plus a return address, probably on the control stack. Once we have entered BAR and given the names X and Y to the arguments, we must invoke the three functions denoted by F, G, and H. When we invoke G or H, it is necessary to supply a return address, because we must eventually return to the code in BAR to complete the computation by invoking F. But we do not have to supply a return address to F; we can merely perform a GOTO, and F will inherit the return address originally supplied to BAR.

Let us simulate the behavior of a PDP-10 pushdown stack to see why this is true. If we consistently used PUSHJ for calling a function and POPJ for returning from one, then the code for BAR, F, G, and H would look something like this:

```
BAR:    ...                 F:      ...
        PUSHJ G                     POPJ
BAR1:   ...
        PUSHJ H             G:      ...
BAR2:   ...                         POPJ
        PUSHJ F
BAR3:   POPJ                H:      ...
                                    POPJ
```

We have labeled not only the entry points to the functions, but also a few key points within BAR, for expository purposes. We are justified in putting no ellipsis between the PUSHJ F and the POPJ in BAR, because we assume that no cleanup other than the POPJ is necessary, and because the value returned by F (in the assumed RESULT register) will be returned from BAR also.

Let us depict a pushdown stack as a list growing towards the right. On arrival at BAR, the caller of BAR has left a return address on the stack.

..., <return address for BAR>

On executing the PUSHJ G, we enter the function G after leaving a return address BAR1 on the stack:

..., <return address for BAR>, BAR1

The function G may call other functions in turn, adding other return addresses to the stack, but these other functions will pop them again on exit, and so on arrival at the POPJ in G the stack is the same. The POPJ pops the address BAR1 and jumps there, leaving the stack like this:

..., <return address for BAR>

In a similar manner, the address BAR2 is pushed when H is called, and H pops this address on exit. The same is true of F and BAR3. On return from F, the POPJ in BAR is executed, and the return address supplied by BAR's caller is popped and jumped to.

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

..., <return address for BAR>, BAR3, ...

Suppose that at the end of BAR we replaced the PUSHJ F, POPJ by GOTO F. Then on arrival at the GOTO the stack would look like this:

..., <return address for BAR>

The stack would look this way on arrival at the POPJ in F, and so F would pop this return address and return to BAR's caller. The net effect is as before. The value returned by F has been returned to BAR's caller, and the stack was left the same. The only difference was that one fewer stack slot was consumed during the execution of F, because we did not push the address BAR3.

Thus we see that F may be invoked in a manner different from the way in which G and H are invoked. This fact is somewhat disturbing. We would like our function invocation mechanism to be uniform, not only for aesthetic reasons, but so that functions may be compiled separately and linked up at run time with a minimum of special-case interfacing. Uniformity is achieved in some LISPs by always using PUSHJ and never GOTO, but this is at the expense of using more stack space than logically necessary. At the end of every function

X the sequence "PUSHJ Y; POPJ" will occur, where Y is the last function invoked by X, requiring a logically unnecessary return address pointing to a POPJ. {Note Debugging}

An alternate approach is suggested by the implementation of the SCHEME interpreter. [Sussman 75] We note that the textual difference between the calls on F and G is that the call on G is nested as an argument to another function call, whereas the call to F is not. This suggests that we save a return address on the stack when we begin to evaluate a form (function call) which is to provide an argument for another function, rather than when we invoke the function. (The SCHEME interpreter works in exactly this way.) This discipline produces a rather elegant symmetry: evaluation of forms (function invocation) pushes additional control stack, and application of functions (function entry and the consequent binding of variables) pushes additional environment stack. Thus for BAR we would compile approximately the following code:

```
BAR:    PUSH [BAR1]              ;save return address for (G X)
        <set up arguments for G>
        GOTO G                   ;call function G
BAR1:   <save result of G>
        PUSH [BAR2]              ;save return address for (H Y)
        <set up arguments for H>
        GOTO H                   ;call function H
BAR2:   <set up arguments for F>
        GOTO F                   ;call function F
```

The instruction PUSH [X] pushes the address X on the stack. Note that no code appears in BAR which ever pops a return address off the stack; it pushes return addresses for G and H, but G and H are responsible for popping them, and BAR passes its own return address implicitly to F without popping it. This point is extremely important, and we shall return to it later.

Those familiar with the MacLISP compiler will recognize the code of the previous example as being similar to the "LSUBR" calling convention. Under this convention, more than just return addresses are kept on the control stack; a function receives its arguments on the stack, above the return address. Thus, when BAR is entered, there are (at least) three items on the stack: the last argument, Y, is on top; below that, the previous (and in fact first) one, X; and below that, the return address. The complete code for BAR might look like this:

```
BAR:    PUSH [BAR1]              ;save return address for (G X)
        PUSH -2(P)               ;push a copy of X
        GOTO G                   ;call function G
BAR1:   PUSH RESULT              ;result of G is in RESULT register
        PUSH [BAR2]              ;save return address for (H Y)
        PUSH -2(P)               ;push a copy of Y
        GOTO H                   ;call function H
BAR2:   POP -2(P)                ;clobber X with result of G
        MOVEM RESULT,(P)         ;clobber Y with result of H
        GOTO F                   ;call function F
```

(There is some tricky code at point BAR2: on return from H the stack looks like:

..., <return address for BAR>, X, Y, <result from G>

After the POP instruction, the stack looks like:

..., <return address for BAR>, <result from G>, Y

That is, the top item of the stack has replaced the one two below it. After the MOVEM (move to memory) instruction:

..., <return address for BAR>, <result from G>, <result from H>

which is exactly the correct setup for calling F. Let us not here go into the issue of how such clever code might be generated, but merely recognize the fact that it gets the stack into the necesssary condition for calling F.)

Suppose that the saving of a return address and the setting up of arguments were commutative operations. (This is not true of the LSUBR calling convention, because both operations use the stack; but it is true of the SUBR convention, where the arguments are "spread" [McCarthy 62] [Moon 74] in registers, and the return address on the stack.) Then we may permute the code as follows (from the original example):

```
BAR:    <set up arguments for G in registers>
        PUSH [BAR1]             ;save return address for (G X)
        GOTO G                  ;call function G
BAR1:   <save result of G>
        <set up arguments for H in registers>
        PUSH [BAR2]             ;save return address for (H Y)
        GOTO H                  ;call function H
BAR2:   <set up arguments for F in registers>
        GOTO F                  ;call function F
```

As it happens, the PDP-10 provides an instruction, PUSHJ, defined as follows:

```
        PUSH [L1]
        GOTO G          is the same as          PUSHJ G
L1:                                         L1:
```

except that the PUSHJ takes less code. Thus we may write the code as:

```
BAR:    <set up arguments for G in registers>
        PUSHJ G                 ;save return address, call G
        <save result of G>
        <set up arguments for H in registers>
        PUSHJ H                 ;save return address, call H
        <set up arguments for F in registers>
        GOTO F                  ;call function F
```

This is why PUSHJ (and similar instructions on other machines, whether they save the return adress on a stack, in a register, or in a memory location) works as a subroutine call, and, by extension, why up to now many people have thought of pushing the return address at function call time rather than at form evaluation time. The use of GOTO to call a function "tail-recursively" (known around MIT as the "JRST hack", from the PDP-10 instruction for GOTO, though the hack itself it dates back to the PDP-1) is in fact not just a hack, but rather the most uniform method for invoking functions. PUSHJ is not a function calling primitive per se, therefore, but rather an optimization of this general approach.

1.3. LAMBDA as a Renaming Operator

Environment operators also take various forms. The most common are assignment to local variables and binding of arguments to functions, but there are others, such as pattern-matching operators (as in COMIT [MITRLE 62] [Yngve 72], SNOBOL [Forte 67], MICRO-PLANNER [Sussman 71], CONNIVER [McDermott 74], and PLASMA [Smith 75]). It is usual to think of these operators as altering the contents of a named location, or of causing the value associated with a name to be changed.

In understanding the action of an environment operator it may be more fruitful to take a different point of view, which is that the value involved is given a new (additional) name. If the name had previously been used to denote another quantity, then that former use is shadowed; but this is not necessarily an essential property of an environment operator, for we can often use alpha-conversion ("uniquization" of variable names) to avoid such shadowing. It is not the names which are important to the computation, but rather the quantities; hence it is appropriate to focus on the quantities and think of them as having one or more names over time, rather than thinking of a name as having one or more values over time.

Consider our previous example involving BAR. On entry to BAR two quantities are passed, either in registers or on the stack. Within BAR these quantities are known as X and Y, and may be referred to by those names. In other environments these quantities may be known by other names; if the code in BAR's caller were (BAR W (+ X 3)), then the first quantity is known as W and the second has no explicit name. {Note Return Address} On entry to BAR, however, the LAMBDA assigns the names X and Y to these two quantities. The fact that X means something else to BAR's caller is of no significance, since these names are for BAR's use only. Thus the LAMBDA not only assigns names, but determines the extent of their significance (their scope). Note an interesting symmetry here: control constructs determine constraints in time (sequencing) in a program, while environment operators determine constraints in space (textual extent, or scope).

One way in which the renaming view of LAMBDA may be useful is in allocation of temporaries in a compiler. Suppose that we use a targeting and preferencing scheme similar to that described by in [Wulf 75] and [Johnsson 75]. Under such a scheme, the names used in a program are partitioned by the compiler into sets called "preference classes". The grouping of several names into the same set indicates that it is preferable, other things being equal, to have the quantities referred to by those names reside in the same memory location at run time; this may occur because the names refer to the same quantity or to related quantities (such as X and X+1). A set may also have a specified target, a particular memory location which is preferable to any other for holding quantities named by members of the set.

As an example, consider the following code skeleton:

((LAMBDA (A B) <body>) (+ X Y) (* Z W))

Suppose that within the compiler the names T1 and T2 have been assigned to the temporary quantities resulting from the addition and multiplication. Then to process the "binding" of A and B we need only add A to the preference class of T1, and B to the preference class of T2. This will have the effect of causing A and T1 to refer to the same location, wherever that may be; similarly B and T2 will refer to the same location. If T1 is saved on a stack and T2 winds up in a register, fine; references to A and B within the <body> will automatically have this information.

On the other hand, suppose that <body> is (FOO 1 A B), where FOO is a

built-in function which takes its arguments in registers 1, 2, and 3. Then A's preference class will be targeted on register 2, and B's on register 3 (since these are the only uses of A and B within <body>); this will cause T1 and T2 to have the same respective targets, and at the outer level an attempt will be made to perform the addition in register 2 and the multiplication in register 3. This general scheme will produce much better code than a scheme which says that all LAMBDA expressions must, like the function FOO, take their arguments in certain registers. Note too that no code whatsoever is generated for the variable bindings as such; the fact that we assign names to the results of the expressions (+ X Y) and (* Z W) rather than writing

```
(FOO 1 (* Z W) (+ X Y))
```

makes no difference at all, which is as it should be. Thus, compiler temporaries and simple user variables are treated on a completely equal basis. This idea was used in [Johnsson 75], but without any explanation of why such equal treatment is justified. Here we have some indication that there is conceptually no difference between a user variable and a compiler-generated temporary. This claim will be made more explicit later in the discussion of continuation-passing. Names are merely a convenient textual device for indicating the various places in a program where a computed quantity is referred to. If we could, say, draw arrows instead, as in a data flow diagram, we would not need to write names. In any case, names are eliminated at compile time, and so by run time the distinction between user names and the compiler's generated names has been lost.

Thus, at the low level, we may view LAMBDA as a renaming operation which has more to do with the internal workings of the compiler (or the interpreter), and with a notation for indicating where quantities are referred to, than with the semantics as such of the computation to be performed by the program.

1.4. An Example: Compiling a Simple Function

One of the important consequences of the view of LAMBDA and function calls presented above is that programs written in a style based on the lambda-calculus-theoretic models of higher-level constructs such as DO loops (see [Stoy 74] [Steele 76]) will be correctly compiled. As an example, consider this iterative factorial function:

```
(DEFINE FACT
        (LAMBDA (N)
                (LABELS ((FACT1
                          (LAMBDA (M A)
                                  (IF (= M 0) A
                                      (FACT1 (- M 1)
                                             (* M A))))))
                        (FACT1 N 1))))
```

Let us step through a complete compilation process for this function, based on the ideas we have seen. (This scenario is intended only to exemplify certain ideas, and does not reflect entirely accurately the targeting and preferencing techniques described in [Wulf 75] and [Johnsson 75].)

First, let us assign names to all the intermediate quantities (temporaries) which will arise:

```
(DEFINE FACT
        (LAMBDA (N)
             T1≡(LABELS ((FACT1
                          (LAMBDA (M A)
                               T2≡(IF T3≡(= M 0) A
                                      T4≡(FACT1 T5≡(- M 1)
                                                T6≡(* M A))))))
                        T7≡(FACT1 N 1))))
```

We have attached a name T1-T7 to all the function calls in the definition; these names refer to the quantities which will result from these function calls.

Now let us place the names in preference classes. Since N is used only once, as an argument to FACT1, which will call that argument M, N and M belong in the same class; T5 also belongs to this class for the same reason. T1, T2, T4, and T7 belong in the same class because they are all names, in effect, for the result of FACT1 or FACT. T6 and A belong in the same class, because T6 is an argument to FACT1; T2 and A belong in the same class, because A is one possible result of the IF. T3 is in a class by itself.

```
{M, N, T5}
{A, T1, T2, T4, T6, T7}
{T3}
```

A fairly complicated analysis of the "lifetimes" of these quantities shows that M and T5 must coexist simultaneously (while calculating T6), and so they cannot really be assigned the same memory location. Hence we must split T5 off into a class of its own after all.

Let us suppose that we prefer to target the result of a global function into register RESULT, and the single argument to a function into register ARG. (FACT1, which is not a global function, is not subject to these preferences.) Then we have:

```
{M, N}                    target ARG (by virtue of N)
{T5}
{A, T1, T2, T4, T6, T7}   target RESULT (by virtue of T1)
{T3}
```

T3, on the other hand, will need no memory location (a property of the PDP-10 instruction set). Thus we might get this assignment of locations:

```
{M, N}                    ARG
{T5}                      R1
{A, T1, T2, T4, T6, T7}   RESULT
```

where R1 is an arbitrarily chosen register.

We now really have two functions to compile, FACT and FACT1. Up to now we have used the renaming properties of LAMBDA to assign registers; now we use the GOTO property of function calls to construct this code skeleton:

```
FACT:   <set up arguments for FACT1>
        GOTO FACT1              ;call FACT1

FACT1:  <if quantity names M is non-zero go to FACT1A>
        <return quantity named A in register RESULT>
        POPJ

FACT1A: <do subtraction and multiplication>
        GOTO FACT1              ;FACT1 calling itself
```

Filling in the arithmetic operations and register assignments gives:

```
;;; On arrival here, quantity named N is in register ARG.
FACT:   MOVEI RESULT,1          ;N already in ARG; set up 1
        GOTO FACT1              ;call FACT1


;;; On arrival here, quantity named M is in ARG,
;;;  and quantity named A is in RESULT.
FACT1:  JUMPN ARG,FACT1A
        POPJ                    ;A is already in RESULT!

FACT1A: MOVE R1,ARG             ;must do subtraction in R1
        SUBI R1,1
        IMUL RESULT,ARG         ;do multiplication
        MOVE ARG,R1             ;now put result of subtraction in ARG
        GOTO FACT1              ;FACT1 calling itself
```

This code, while not perfect, is not bad. The major deficiency, which is the use of R1, is easily cured if the compiler could know at some level that the subtraction and multiplication can be interchanged (for neither has side effects which would affect the other), producing:

```
FACT1A: IMUL RESULT,ARG
        SUBI ARG,1
        GOTO FACT1
```

Similarly, the sequence:

```
        GOTO FACT1

FACT1:
```

could be optimized by removing the GOTO. These tricks, however, are known by any current reasonably sophisticated optimizing compiler.

What is more important is the philosophy taken in interpreting the meaning of the program during the compilation process. The structure of this compiled code is a loop, not a nested sequence of stack-pushing function calls. Like the SCHEME interpreter or the various PLASMA implementations, a compiler based on these ideas would correctly reflect the semantics of lambda-calculus-based models of high-level constructs.

1.5. Who Pops the Return Address?

Earlier we showed a translation of BAR into "machine language", and noted that there was no code which explicitly popped a return address; the buck was always passed to another function (F, G, or H). This may seem surprising at first, but it is in fact a necessary consequence of our view of function calls as "GOTOs with a message". We will show by induction that only primitive functions not expressible in our language (SCHEME) perform POPJ; indeed, only this nature of the primitives determines the fact that our language is functionally oriented!

What is the last thing performed by a function? Consider the definition of one:

(DEFINE FUN (LAMBDA (X1 X2 ... XN) <body>))

Now <body> must be a form in our language. There are several cases:

[1] Constant, variable, or closure. In this case we actually compiled a POPJ in the case of FACT above, but we could view constants, variables, and closures (in general, things which "evaluate trivially" in the sense described in [Steele 76]) as functions of zero arguments if we wished, and so GOTO a place which would get the value of the constant, variable, or closure into RESULT. This place would inherit the return address, and so our function need not pop it. Alternatively, we may view constants, etc. as primitives, the same way we regard integer addition as a primitive (note that CTA2 above required a POPJ, since we had "open-coded" the addition primitive).

[2] (IF <pred> <exp1> <exp2>). In this case the last thing our function does is the last thing <exp1> or <exp2> does, and so we appeal to this analysis inductively.

[3] (LABELS <defns> <exp>). In this case the last thing our function does is the last thing <exp> does. This may involve invoking a function defined in the LABELS, but we can consider them to be separate functions for our purposes here.

[4] A function call. In this case the function called will inherit the return address.

Since these are all the cases, we must conclude that our function never pops its return address! But it must get popped at some point so that the final value may be returned.

Or must it? If we examine the four cases again and analyze the recursive argument, it becomes clear that the last thing a function that we define in SCHEME eventually does is invoke another function. The functions we define therefore cannot cause a return address to be popped. It is, rather, the primitive, built-in operators of the language which pop return addresses. These primitives cannot be directly expressed in the language itself (or, more accurately, there is some basis set of them which cannot be expressed). It is the constants (which we may temporarily regard as zero-argument functions), the arithmetic operators, and so forth which pop the return address. (One might note that in the compilation of CURRIED-TRIPLE-ADD above, a POPJ appeared only at the point the primitive "+" function was open-coded as ADD instructions.)

2. Lexical and Dynamic Binding

The examples of the previous section, by using only local variables, avoided the question of whether variables are lexically or dynamically scoped. In this section we will see that lexical scoping is necessary in order to reflect the semantics of lambda-calculus-based models. We might well ask, then, if LISP was originally based on lambda calculus, why do most current LISP systems employ dynamic binding rather than lexical?

The primary reason seems to be the introduction of stack hardware at about the time of early LISP development. (This was not pure cause and effect; rather, each phenomenon influenced the other.) The point is that a dynamic bindings stack parallels the control stack in structure. If one has an escape operator [Reynolds 72] (also known as CATCH [Moon 74] or EXIT [Wulf 71] [Wulf 72]) then the "control stack" may be, in general, a tree structure, just as the introduction of FUNARGs requires that the environment be tree-structured. [Moses 70] If these operators are forbidden, or only implemented in the "downward" sense (in the same way that ALGOL provides "downward funarg" (procedure arguments to functions) but not "upward funarg" (procedure-valued functions)) as they historically have been in most non-toy LISP systems, then hardware stack instructions can always be used for function calling and environment binding. Since the introduction of stack hardware (e.g. in the PDP-6), most improvements to LISP's variable binding methods have therefore started with dynamic binding and then tried to patch it up.

MacLISP [Moon 74] uses the so-called shallow access scheme, in which the current value of a variable is in a fixed location, and old values are on a stack. The advantage of this technique is that variables can be accessed using only a single memory reference. When code is compiled, variables are divided into two classes: special variables are kept in their usual fixed locations, while local variables are kept wherever convenient, at the compiler's discretion, saving time over the relatively expensive special binding mechanism.

InterLISP [Teitelman 74] (before spaghetti stacks) used a deep access scheme, in which it was necessary to look up on the bindings stack to find variable bindings; if a variable was not bound on the stack, the its global value cell was checked. The cost of the stack search was ameliorated by looking up, on entry to a function, the locations of variables needed by that function. The advantage of this scheme is that the "top level" value of a variable is easily accessed, since it is always in the variable's value cell. (InterLISP also divides variables into two classes for purposes of compilation; only special variables need be looked up on the bindings stack.)

Two other notable techniques are the use of value cells as a cache for a deep dynamic access scheme, and "spaghetti stacks" [Bobrow 73], which attempt to allow the user to choose between static and dynamic binding. The problem with the latter is that they are so general that it is difficult for the compiler to optimize anything; also, they do not completely solve the problem of choosing between static and dynamic binding. For example, the GEN-SQRT-OF-GIVEN-EXTRA-TOLERANCE function given in [Steele 76] cannot be handled properly with spaghetti stacks in the straightforward way. The difficulty is that there is only one access link for each frame, while there are conceptually two distinct access methods, namely lexical and dynamic.

Unfortunately, dynamic binding creates two difficulties. One is the well-known "FUNARG" problem [Moses 70]; the essence of this problem is that lexical scoping is desired for functional arguments. The other is more subtle. Consider the FACT example above. If we were to use dynamic binding, then every time around the FACT1 loop it would be necessary to bind M and A on a stack. Thus the binding stack would grow arbitrarily deep as we went around

the loop many times.

It might be argued that a compiler might notice that the old values of M and A can never be referenced, and so might avoid pushing M and A onto a stack. This is true of this special case, but is undecidable in general, given that the compiler may not be in a position to examine all the functions called by the function being compiled. Let us consider our BAR example above:

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

Under dynamic binding, F might refer to the variables X and Y bound by BAR. Hence we must push X and Y onto the bindings stack before calling F, and we must also pop them back off when F returns. It is the latter operation that causes difficulties. We cannot merely GOTO F any more; we must provide to F the return address of a routine which will pop X and Y and then return from BAR. F cannot inherit BAR's return address, because the unbinding operation must occur between the return from F and the return from BAR.

Thus, if we are to adhere to the view proposed earlier of LAMBDA and function calls, we are compelled to accept lexical scoping of variables. This will solve our two objections to dynamic binding, but there are two objections to lexical scoping to be answered. The first is whether it will be inherently less efficient than dynamic binding (particularly given that we know so much about how to implement the latter!); the second is whether we should abandon dynamic binding, inasmuch as it has certain useful applications.

ALGOL implementors have used lexical scoping for many years, and have evolved techniques for handling it efficiently, in particular the device known as the display. [Dijkstra 67] Some machines have even had special hardware for this purpose [Hauck 68], just as PDP-6's and PDP-10's have special hardware which aids dynamic binding. The important point is that even if deep access is used, it is not necessary to search for a variable's binding as it is for dynamic binding, since the binding must occur at a fixed place relative to the current environment. The display is in fact simply a double-indexing scheme for accessing a binding in constant time. It is not difficult to see that search is unnecessary if we consider that the binding appears lexically in a fixed place relative to the reference to the variable; a compiler can determine the appropriate offset at compile time. Furthermore, the "access depth" of a lexical variable is equal to the number of closures which contain it, and in typical programs this depth is small (less than 5).

In an optimizing compiler for lexically scoped LISP it would not necessary to create environment structures in a standard form. Local variables could be kept in any available registers if desired. It would not necessary to interface these environment structures to the interpreter. Because the scoping would be strictly lexical, a reference to a variable in a compiled environment structure must occur in compiled code appearing within the LAMBDA that bound the variable, and so no interpreted reference could refer to such a variable. Similarly, no compiled variable reference could refer to an environment structure created by the interpreter. (An exception to this argument is the case of writing an interactive debugging package, but that will be discussed later. This problem can be fixed in any case if the compiler outputs an appropriate map of variable locations for use by the debugger.)

Consider this extension of a classic example of the use of closures:

```
(DEFINE CURRIED-TRIPLE-ADD
        (LAMBDA (X)
```

```
(LAMBDA (Y)
        (LAMBDA (Z) (+ X Y Z))))
```

Using a very simple-minded approach, let us represent a closure as a vector whose first element is a pointer to the code and whose succeeding elements are all the quantities needed by that closure. We will write a vector as [x0, x1, ..., x<n-1>]. Let us also assume that when a closed function is called the closure itself is in register CLOSURE. (This is convenient anyway on a PDP-10, since one can call the closure by doing an indexed GOTO, such as GOTO @(CLOSURE), where @ means indirection through the first element of the vector.) Let us use the LSUBR calling convention described earlier for passing arguments. Finally, let there be a series of functions nCLOSE which create closure vectors of n elements, each taking its arguments in reverse order for convenience (the argument on top of the stack becomes element 0 of the vector.) Then the code might look like this:

```
CTA:    PUSH [CTA1]       ;X is on stack; add address of code
        GOTO 2CLOSE       ;create closure [CTA1, X]

CTA1:   PUSH CLOSURE      ;now address of [CTA1, X] is in CLOSURE
        PUSH [CTA2]       ;Y was on stack on entry
        GOTO 3CLOSE       ;return closure [CTA2, [CTA1, X], Y]

CTA2:   POP RESULT                ;pop Z into result
        ADD RESULT,2(CLOSURE)     ;add in Y (using commutativity, etc.)
        MOVE TEMP,1(CLOSURE)      ;fetch pointer to outer closure
        ADD RESULT,1(TEMP)        ;add in X
        POPJ                      ;return sum in RESULT
```

Admittedly this does not compare favorably with uncurried addition, but the point is to illustrate how easily closures can be produced and accessed. If several variables had been closed in the outer closure rather than just X, then one might endeavor in CTA2 to fetch the outer closure pointer only once, just as in ALGOL one loads a display slot only once and then uses it many times to access the variables in that contour.

A point to note is that it is not necessary to divide lexically scoped variables into two classes for compilation purposes; the compiler can always determine whether a variable is referred to globally or not. Furthermore, when creating a closure (i.e. a FUNARG), the compiler can determine precisely what variables are needed by the closure and include only those variables in the data structure for the closure, if it thinks that would be more efficient.

For example, consider the following code skeleton:

```
(LAMBDA (A B C D E)
        ...
        (LAMBDA (F G) ... B ... E ... H ...) ...)
```

It is quite clear that H is a global variable and so must be "special", whereas B and E are local (though global to the inner LAMBDA). When the compiler creates code to close the inner LAMBDA expression, the closure need only include the variables B and E, and not A, C, or D. The latter variables in fact can be kept in registers; only B and E need be kept in a semi-permanent data structure, and even then only if the inner closure is actually created.

Hewitt [Hewitt 76] has mentioned this idea repeatedly, saying actors are distinguished from LISP closures in that actor closures contain precisely

those "acquaintances" which are necessary for the actor closure to run, whereas LISP closures may contain arbitrary numbers of unnecessary variable bindings. This indeed is an extremely important point to us here, but he failed to discuss two aspects of this idea:

(1) Hewitt spoke in the context of interpreters and other "incremental" implementations rather than of full-blown compilers. In an interpreter it is much more convenient to use a uniform closure method than to run around determining which variables are actually needed for the closure. In fact, to do this efficiently in PLASMA, it is necessary to perform a "reduction" pre-pass on the expression, which is essentially a semi-compilation of the code; it is perhaps unfair to compare a compiler to an interpreter. {Note PLASMA Reduction} In any case, the semantics of the language are unaffected; it doesn't matter that extra variable bindings are present if they are not referred to. Thus this is an efficiency question only, a question of what a compiler can do to save storage, and not a question of semantics.

(2) It is not always more efficient to create minimal closures! Consider the following case:

```
(LAMBDA (A B C D)
        ...
        (LAMBDA () ... A ... B ...)
        (LAMBDA () ... A ... C ...)
        (LAMBDA () ... A ... D ...)
        (LAMBDA () ... B ... C ...)
        (LAMBDA () ... B ... D ...)
        (LAMBDA () ... C ... D ...)  ...)
```

The six closures, if each created minimally, will together contain twelve variable bindings; but if they shared the single environment containing A, B, C, and D as in a LISP interpreter, there would be only four bindings. Thus PLASMA may in certain cases take more storage with its closure strategy rather than less. On the other hand, suppose five of the closures are used immediately and then discarded, and only the sixth survives indefinitely. Then in the long run, PLASMA's strategy would do better!

The moral is that neither strategy is guaranteed to be the more efficient in any absolute sense, since the efficiency can be made a function of the behavior of the user's program, not just of the textual form of the program. The compiler should be prepared to make a decision as to which is more efficient (and in some simple and common cases such a choice can be made correctly), and perhaps to accept advice from the user in the form of declarations.

It seems, then, that if these ideas are brought to bear, lexical binding need not be expensive. This leaves the question of whether to abandon dynamic binding completely. Steele and Sussman [Steele 76] demonstrate clearly the technique for simulating dynamic binding in a lexically scoped language; they also make a case for separating the two kinds of variables and having two completely distinct binding mechanisms, exhibiting a programming example which cannot be coded easily using only dynamic binding or only lexical scoping. The two mechanisms naturally require different compilation techniques (one difference is that fluid variables, unlike static ones, are somewhat tied down to particular locations or search mechanisms because it cannot generally be determined at compile time who will reference a variable when), but they are each so valuable in certain contexts that in a general-

purpose programming language it would be foolish to abandon either.

3. LAMBDA, Actors, and Continuations

Suppose that we choose a set of primitive operators which are not functions. This will surely produce a radical change in our style of programming, but, by the argument of the previous section, it will not change our interpretation of LAMBDA and function calling. A comparison between our view of LAMBDA and the notion of actors as presented by Hewitt will motivate the choice of a certain set of non-functional primitives which lead to the so-called "continuation-passing" style.

3.1. Actors ≡ Closures (mod Syntax)

In [Sussman 75] Sussman and Steele note that actors (other than those which embody side effects and synchronization) and closures of LAMBDA expressions are isomorphic in their behavior. Smith and Hewitt [Smith 75] describe an actor as a combination of a script (code to be executed) and a set of acquaintances (computational quantities available to the code). A LISP closure in like manner is a combination of a body of code and a set of variable bindings (or, using our idea of renaming, a set of computational quantities with (possibly implicitly) associated names). Hewitt [Hewitt 76] has challenged this isomorphism, saying that closures may contain unnecessary quantities, but I have already dealt with this issue above.

Let us therefore examine this isomorphism more closely. We have noted above that it is more accurate to think of the caller of a LAMBDA as performing a GOTO rather than the LAMBDA itself. It is the operation of invocation that is the transfer of control. This transfer of control is similar to the transfer of control from one actor to another.

In the actors model, when control is passed from one actor to another, more than a GOTO is performed. A computed quantity, the message, is passed to the invoked actor. This corresponds to the set of arguments passed to a LAMBDA expression. Now if we wish to regard the actor/LAMBDA expression as a black box, then we need not be concerned with the renaming operation; all we care about is that an answer eventually comes out. We do not care that the LAMBDA expression will "spread" the set of arguments out and assign names to various quantities. In fact, there are times when the caller may not wish to think of the argument set as a set of distinct values; this attitude is reflected in the APPLY primitive of LISP, and in the FEXPR calling convention. The actors model points out that, at the interface of caller and callee, we may usefully think of the argument set as a single entity.

In the actors model, one important element of the standard message is the continuation. This is equivalent to the notion of return address in a LISP system (more accurately, the continuation is equivalent to the return address plus all the quantities which will be needed by the code at that address). We do not normally think of the return address as an argument to a LAMBDA expression, because standard LISP notation suppresses that fact.

On the one hand, though, Steele and Sussman [Steele 76] point out that it is possible to write LISP code in such a manner that return addresses are passed explicitly. (This style corresponds to the use in PLASMA of ≡≡> and <== to the exclusion of ≡>, <≡, and functional notation.) When code is written in this "continuation-passing style", no implicit return addresses are ever created on the control stack. All that is necessary to write code entirely in this style is that continuation-passing primitives be available.

The reason LISP is so function-oriented is that all the primitives (CAR, CONS, +, etc.) are functions, expecting return addresses on the stack. The stack is simply a conventional place to pass some (or all) of the arguments. If, for example, we consider the LSUBR argument-passing convention described earlier, it is easy to think of the return address as being the "zeroth" argument, for it is passed on the stack just below arguments 1 through n.

On the other hand, the PLASMA language, while based on actor semantics, has a number of abbreviations which allow the user to ignore the continuation portion of a message in the same way he would in LISP. When the user writes in PLASMA what would be a function invocation in LISP, the PLASMA interpreter automatically supplies an "underlying continuation" which is passed in a standard component of the message packet. This is analogous to the way the LISP system automatically supplies a return address in a standard place (the control stack). (Hewitt [Hewitt 76] has expressed doubt as to whether these underlying continuations can themselves be represented explicitly as LAMBDA expressions. My impression is that he sees a potential infinite regression of underlying continuations. If the underlying continuations are written in pure continuation-passing style as defined in [Steele 76], however, this problem does not arise.)

Let us define a convenient set of continuation-passing primitives. Following the convention used in [Steele 76], we will let the last argument(s) to such a primitive be the continuation(s), where a continuation is simply a "function" of values delivered by the primitive.

(++ a b c)	delivers the sum of a and b to the continuation c.
(-- a b c)	delivers the difference of a and b to the continuation c.
(** a b c)	delivers the product of a and b to the continuation c.
(^^ a b c)	delivers a raised to the power b to the continuation c.
(%= a b c)	delivers T to continuation c if a and b are arithmetically equal, and otherwise NIL.
(== a b c d)	invokes continuation c if a and b are arithmetically equal, and otherwise continuation d (c and d receive no values from ==).

Note that predicates may usefully be defined in at least two ways. The predicate %= is analogous to a functional predicate in LISP, in that it delivers a truth value to its continuation, while == actually implements a conditional control primitive.

Thus far in our comparison of closures and actors we have focused on aspects of control. Now let us consider the manipulation of environments. When an actor is invoked, it receives a message. It is convenient to assign names to parts of this message for future reference within the script. This is done by pattern matching in PLASMA, and by "spreading" in LISP. This assignment of names is a matter purely internal to the workings of the actor/LAMBDA expression; the outside world should not be affected by which names are used. (This corresponds indirectly to the notion of referential transparency.)

In discussing control we noted that on invoking a function the LISP and PLASMA interpreters create an implicit underlying continuation, a return address. This is a hidden creation of a control operation. Are there any hidden environment operations?

The hidden control operation occurs just before invocation of a function. We might expect, by symmetry, a hidden environment operation to occur on return from the function. This is in fact the case. The underlying continuation, itself an actor, will assign one or more hidden names to the contents of the message it receives. If the actor originally invoked was a

function, then the message is the returned value of the function. Consider this example, taken from [Steele 76]:

```
(- (^ B 2) (* 4 A C))
```

When the function "^" is invoked, the message to it contains three items: the value of B, the value of 2, and the implicit continuation. When it returns, the implicit continuation saves the returned value in some internal named place, then invokes the function "*". The message to "*" contains four items: the values of 4, A, and C, and a second implicit continuation. This continuation contains the internal place where the value from "^" was saved! When "*" returns, the continuation then calls "-", giving it the saved result from "^", the result freshly obtained from "*", and whatever continuation was given for the evaluation of the entire expression; "-" will deliver its result to that (inherited) continuation.

This is all made more clear by writing the example out in pure continuation-passing style, using our continuation-passing primitives:

```
(^^ B 2
    (LAMBDA (X)
            (** 4 A C
                (LAMBDA (Y)
                        (-- X Y <the-inherited-continuation>)))))
```

Here X and Y are the explicit names for intermediate quantities which were hidden before by the function-calling syntax. We use LAMBDA to express the continuations passed to "^^" and "**". These LAMBDA expressions are not functions; they never return values, but rather merely invoke more continuation-passing primitives. However, the interpretation of a LAMBDA expression is as before: when invoked, the arguments are assigned the additional names specified in the LAMBDA variables list, and then the body of the LAMBDA expression is executed.

In the context of a compiler, the intermediate quantities passed to the continuations are usually known as "temporaries", or are kept in places called temporaries. Usually the temporaries are mentioned in the context of the problem of allocating them. The present analysis indicates that they are just like names assigned by the user; they are different only in that the user is relieved by the syntax of a functional language of having to mention their names explicitly. This is made even more clear by considering two extremes. In assembly language, there are no implicitly named quantities; every time one is used, its name (be it a register name, a memory location, or whatever) must be mentioned. On the other hand, in a data flow language (e.g. some of the AMBIT series) it is possible to draw arrows and never mention names at all.

By considering temporaries as just another kind of name (or alternatively, user names to be just another kind of temporary), the example of allocation of temporaries given earlier may be understood on a firmer theoretical level. Furthermore, greater understanding of this uniformity may lead to advances in language design. Let us consider two examples.

First, we may notice that the underlying continuations in LISP and PLASMA take only one argument: the returned value. Why are there not implicit continuations of more than one argument? The answer is that this characteristic is imposed by the syntax and set of primitives provided in functional languages. Suppose we were to augment LISP as follows (this is not a serious proposal for a language extension, but only an example):

(1) Wherever n consecutive arguments might be written in a function call, one may instead write {f x1 ... xm}n, where n is a positive integer. The "function" f must return n values, which are used as n arguments in the function call.
(2) The primitive (values x1 ... xn) returns its n arguments as its n values. Thus writing "{values x1 ... xn}n" is the same as writing "x1 ... xn" as arguments in a function call.

Then we might write code such as:

```
(DEFINE FOO
        (LAMBDA (A)
                (VALUES (^ A 2) (^ A 3))))

(LIST {FOO 5}2 (+ {FOO 4}2) {FOO 3}2)
```

Evaluating the second form produces (25 125 80 9 27). When FOO is invoked, it is provided an implicit continuation which expects two arguments, i.e. two returned values from FOO. We need VALUES as a primitive in order to be able to return several values. (We could imagine syntactic sugar for this, such as (LAMBDA (A) (^ A 2) (^ A 3)), but it is no more than sugar.) When (^ A 2) and (^ A 3) have been evaluated, FOO does a GOTO to VALUES, whereupon VALUES inherits the two-argument continuation given to FOO. Thus the LAMBDA expression for FOO never needs to know how many arguments its continuation takes; that is a matter for the primitives to decide.

All this suggests our second example, namely a way to return multiple values from a function without all the extra syntax and primitives. All that is necessary is to use explicit continuation-passing. Thus the above example might be written:

```
(DEFINE FOO
        (LAMBDA (A CONT)
                (CONT (^ A 2) (^ A 3))))

(FOO 5 (LAMBDA (X1 X2)
           (FOO 4 (LAMBDA (Y1 Y2)
                      ((LAMBDA (X3)
                           (FOO 7 (LAMBDA (X4 X5)
                                      (LIST X1 X2 X3 X4 X5))))
                       (+ Y1 Y2))))))
```

Here we have used a mixture of functional and continuation-passing styles, employing only enough of the latter to express the multiple values returned by FOO. The implicit continuations for the evaluation of the arguments to LIST and "+" (and their implicit temporaries X1, X2, X3, X4, X5, Y1, and Y2) have been made explicit. While one might see how to implement multiple-value-return in an interpreter on the basis of the first example (by augmenting the interpreter to handle the new "primitives"), the second makes it clear how to compile it without introducing new primitives at the low level. (Appendix A presents a program which converts ordinary SCHEME programs to pure continuation-passing style; Appendix B presents a modification to this program which handles the multiple-value-return construct.) Furthermore, by using the same mechanism to compile both function calls and multiple value returns, the multiple values will get returned in registers in the same way arguments might be passed, without the need for any additional machinery. {Note PLASMA Registers}

3.2. The Procedural View of Data Types

Up to now we have concentrated on LAMBDA and function calling as environment and control primitives. Based on the actor approach, we will see that a certain amount of useful data manipulation can be expressed in terms of LAMBDA expressions. If compiled well enough, such data manipulation would be no more costly than code generated by special-case compiler routines. Thus, yet one more programming construct could be handled by this general compilation mechanism, making the compiler yet more uniform.

The procedural approach to data type behavior has been developing for many years. Typical of languages of the early 1970's with such ideas are ECL [Wegbreit 74] and MUDDLE [Galley 75]. Each allows certain characteristics of a data type to be expressed as an arbitrary procedure. Specifically, ECL allows the behavior of the creation, assignment, coercion, subscripting, and printing operations to be procedurally specified; MUDDLE allows procedural specification of the methods for evaluation and application of objects. The pieces of data are still thought of as objects, however, and there are mechanisms for defeating the procedural specifications by "lowering" a data type to a more primitive type (such mechanisms are necessary for use by the behavior specification procedures themselves). One problem with (feature of??) the lowering mechanism is that any procedure, not just one controlling the behavior of a data type, can use the lowering primitive and so defeat the data type functions.

The next step in this direction is the idea that the notion of a data type is meaningful only in terms of the operations that can be performed on it; that is, all that one can do with an object is give it to one of a defined set of procedures which know how to operate on that data type. This notion is exemplified in the CLU language. [Liskov 74] [Liskov 76] Associated with a data type is a cluster of procedures; only those procedures can manipulate the data type. Unfortunately, a "lowering" ("rep") mechanism is still needed within the cluster so that cluster procedures can get at the "underlying representation" of a data object. The definition of this mechanism causes certain problems. CLU at least solves the problem of indiscriminate use of the mechanism, by restricting its use to the cluster procedures.

Carrying this notion still further, Hewitt has proposed the notion of "actors". [Hewitt 73] Rather than dichotomizing the world into data objects and procedures, he suggests that only procedures are meaningful; each "data object" actually embodies all the operations on itself as a procedure. The only operation one can perform on an object is to invoke its procedure. There is no problem of "lowering" the data type to an underlying representation, because an object of the data type does not exist as such to be lowered. In this way the integrity of a data type is much more easily preserved. (While some people object to having to use this model of data types in writing their programs, there is no reason it cannot be hidden with syntactic sugar. {Note PLASMA Sugar} There is much to be said for it as a formal model of data type behavior, but as a practical programming tool it is not always conceptually convenient.)

Let us consider abstractly (though not rigorously) the motivation for the notion of data types. Typically we have an object X and want to perform some operation F on it. Suppose that F is a non-primitive operator; then it must decide what set of primitive actions to perform. Let such decisions made by F partition its domain into classes, such that all objects in a class cause the same set of primitive actions to occur when given to F. One may then

define the data type of X with respect to F to be the class into which F's decisions place X. By extension, one may let F range over some set of operations with similar domains, and let the union of their decisions determine the classes. Loosely speaking, then, a data type is a class of objects which may be operated on in a uniform manner. The notion of data types provides a simple conceptual way to classify an object for the purposes of deciding how to operate on it.

Now let us approach the problem from another direction. Consider a prototypical function call (F X). (We may consider a function call of more than one argument to be equivalent to (F (LIST X1 ... Xn)) for our purposes here.) When executed, this function call is to be elaborated into some series of more primitive operations. It may help to think of execution as a mapping from the product space of the sets of operators and operands to the space of sequences of more primitive operations. Such a mapping can be expressed as a matrix. For example:

	Operation					
	TYPE	PRINT	ATOM	FIRST	CAR	...
Operand						
0	RET(FIXNUM)	TYO("0")	RET(T)	ERROR	ERROR	...
43	RET(FIXNUM)	TYO("4") TYO("3")	RET(T)	ERROR	ERROR	...
(A B)	RET(LIST)	TYO("(") TYO("A") TYO(" ") TYO("B") TYO(")")	RET(NIL)	RET(A)	RET(A)	...
[1]	RET(VECTOR)	TYO("[") TYO("1") TYO("]")	RET(NIL)	RET(1)	RET(1)	...
	...	...	...	...	...	...

Legend: TYO outputs a character; RET returns a value.

Now it would be completely impractical to specify this matrix explicitly in its entirety. It is convenient to lump all FIXNUM objects, for example, into one class, and provide a matrix entry under PRINT which is less efficient for any one application but which works for all such objects. That is, rather than having a separate entry for each FIXNUM under PRINT which knows exactly what characters to output, we have some algorithm which generates digits arithmetically. (Similarly, for lists and vectors we have an algorithm which knows how to print subcomponents in a general manner.) We say that 0 and 43, or [1 2] and [4 5 6], have the same type because almost all operations which apply to both can use the same set of primitive actions, appropriately chosen. Operators may similarly lumped together; for example, in many LISP implementations CAR will get the first element of any composite data object, lumping in such operators as FIRST of a vector or an array.

It is hard to think of natural examples of operator lumping since it seldom occurs in practice. Historically, the tendency has been to break up the matrix by columns. All the entries for TYPE are lumped together, all those for PRINT, and so on. When one invents a new operator, one merely writes a routine encoding the new column of entries; such a routine typically

begins with a dispatch on the data type of its argument. When one invents a new data type, however, it is necessary to change every routine a little bit to incorporate the new row entry.

The procedural approach to data types includes, in effect, a suggestion that the matrix be sliced up by rows instead of columns. The result of this is to group all the operations for a single data type together. This of course yields the inverse problem: adding a new data type is easy, but adding a new generic operator is difficult because many data type routines must be changed. {Note Slice Both Ways}

The important point, however, is that the data type of an object provides a way of selecting a row of the operations matrix. Whether this selection is represented as a procedure, a symbol, or a set of bits does not concern us. When combined with a column selector (choice of operator), it determines what set of actions to undertake. {Note Turing Machines}

Hewitt has pointed out that non-primitive actors can be made to implement data structures such as queues and list cells. It is shown in [Sussman 75] that the PLASMA expression (from [Smith 75]):

```
[CONS ≡
    (≡> [=A =B]
        (CASES
           (≡> FIRST?
               A)
           (≡> REST?
               B)
           (≡> LIST?
               YES)))]
```

may be written in terms of LAMBDA expressions:

```
(DEFINE CONS
        (LAMBDA (A B)
                (LAMBDA (M)
                        (IF (EQ M 'FIRST?) A
                            (IF (EQ M 'REST?) B
                                (IF (EQ M 'LIST?) 'YES
                                    (ERROR ...)))))))
```

(For some reason, Hewitt seems to prefer FIRST? and REST? to CAR and CDR.)

There are two points to note here. One is that what we normally think of as a data structure (a list cell) has been implemented by means of a closure; the result of CONS is a closure of the piece of code (LAMBDA (M) ...) and the environment containing A and B. The other is that the body of the code is essentially a decision proecedure for selecting a column of our operations matrix. This suggests a pretty symmetry: we may either first determine an operator and then submit an object-specifier to the row-selection procedure for that operator, or first determine an operand and submit an operator-specifier to the column-selection procedure for that operand.

This kind of definition has been well known to lambda-calculus theoreticians for years; examples of it occur in Church's monograph. [Church 41] It has generally not been used as a practical definition technique in optimizing compilers, however. Hewitt has promoted this idea in PLASMA, but he has only described an interpreter implementation with no clues as to how to compile it. Moreover, no one seems to have stated the inverse implication, namely, that the way to approach the problem of compiling

closures is to think of them as data structures, with all structures produced by closing a given LAMBDA expression being thought of as having the same data type. Up to now closures have generally been thought of as expensive beasts to implement in a programming language; however, thinking of them in terms of data types should make them appear much less frightening. Consider this definition of CAR:

```
(DEFINE CAR
        (LAMBDA (CELL)
                (CELL 'FIRST?)))
```

Now consider this code fragment:

```
((LAMBDA (FOO)
         ...
         (CAR FOO)
         ...)
 (CONS 'ZIP 'ZAP))
```

It may appear that this must compile into extremely poor code if we use the procedural definition of a list cell given above. However, at the point where the result of the CONS is given to CAR, the compiler can be made to output a HLRZ instruction and no more, just as if the MacLISP NCOMPLR optimizing compiler [Moon 74] had seen (CAR FOO) or the ECL compiler [Wegbreit 74] had seen "FOO.LEFT". All that is required is some knowledge that FOO was created by CONS (that is, we must know FOO's "data type"), plus standard optimization techniques such as procedure integration, constants folding, and dead code elimination. The idea is to recognize that FOO names a closure of two data items with the code of the inner LAMBDA expression in CONS; this could be done either by declaration or by flow analysis. Integrating this LAMBDA expression as well as the definition of CAR into our code fragment yields:

```
((LAMBDA (FOO)
         ...
         ((LAMBDA (CELL) (CELL 'FIRST?))
          (LAMBDA (M)                              ;in FOO
                  (IF (EQ M 'FIRST?) A
                      (IF (EQ M 'REST?) B
                          (IF (EQ M 'LIST?) 'YES
                              (ERROR ...))))))
         ...)
 (CONS 'ZIP 'ZAP))
```

The comment "in FOO" means that any free variables in the expression are meant to refer to quantities in the closure FOO. Notice that we do not take advantage of the explicit appearance of 'ZIP and 'ZAP as arguments to CONS, though we might do so in practice; our purpose here is to illustrate the more general case where we know that FOO names some result of CONS but we don't know which one.

Integrating (LAMBDA (M) ...) into (LAMBDA (CELL) ...) and then substituting through the argument FIRST? yields:

```
((LAMBDA (FOO)
         ...
         (IF (EQ 'FIRST? 'FIRST?) A          ;in FOO
             ...)
         ...)
 (CONS 'ZIP 'ZAP))
```

Standard constants folding and dead code elimination leads to:

```
((LAMBDA (FOO)
         ...
         A              ;in FOO
         ...)
 (CONS 'ZIP 'ZAP))
```

Now the compiler presumably knows the format of the closures produced by CONS; all it needs to do is generate the instruction(s) to fetch the quantity named A out of the closure FOO. If, for example, closures are represented as vectors (as assumed above in the CURRIED-TRIPLE-ADD example) it would in fact take only one instruction on a PDP-10, just as it would for MacLISP.

All this may sound rather complicated, but these are all well-known optimization techniques (see [Allen 72], for example), which happen not to have been applied before in this context. The one tricky point is keeping track of environments correctly (as with the "in FOO" comment above). All kinds of heterogeneous data structures may be created in this way in terms of LAMBDA expressions; no separate primitive creation or selection operators are necessary. A certain amount of data type analysis will be necessary to carry this out. In this context, data type analysis would consist of determining of what LAMBDA expression a given data object is the closure. This may be determined by global data flow analysis (for example, the recent Allen and Cocke algorithm [Allen 76] might be applicable here), or by user-supplied declarations.

If data structures are specified in these terms, it is left up to the compiler to determine a good representation for these structures. If done properly, there is no reason why the creation of a list cell using CONS as above should not actually perform precisely the same storage allocation as might occur in ECL at the low level. In any case, the compiler should know something about designing and packing data structures. (Some work has been done on this already in the ECL system [Wegbreit 74], for example.)

One might object that this technique cannot quite produce the efficiency of MacLISP in performing CONS, since a standard MacLISP list cell contains only two pointers, while the LAMBDA version would produce a cell containing the two pointers plus a pointer to the code for the LAMBDA expression. In a sense this is true; it is necessary to have a pointer to the code. However, we need not actually have a pointer to the code in the closure; all that is necessary is that we be able to locate the code given the object. Standard LISP systems typically encode this information in other ways, calling it the data type. Remembering the operations matrix described earlier, we may think of the code as the data type of the closure; all either does is provide a row selector for the matrix. Current systems such as ECL and MUDDLE which allow definition of arbitrary numbers of data types have indeed found it necessary to store a full pointer, more or less, to describe the data type of an object. In special cases, however, ECL can compress a data type to only a few bits. There is no reason why a sufficiently clever compiler could not use equally clever encodings of the data type, including the technique of encoding the data type in the address of the closure much as

the "Bibop" version of MacLISP does. [Moon 74]

In any case, we can see that the use of closures to define data types need not be expensive. Once again, LAMBDA seems to provide a uniform and general method which is, as always, subject to clever optimizations in special cases.

4. Some Proposed Organization for a Compiler

In order to test some of the ideas suggested above in a practical context, I propose to construct a working, highly optimizing compiler for a small dialect of LISP. The resulting code should be able to run on a PDP-10 in the MacLISP run-time environment. (The compiler should also be modularized so that code for another machine could be generated, but I do not propose to incorporate any complex and general machine description facility such as that of Snyder [Snyder 75].)

4.1. Basic Issues

The compiler will need to perform a large amount of global data flow and data type analysis; much of this can be based on the approach used by Wulf in the BLISS-11 compiler [Wulf 75], augmented by some general flow-graph analysis scheme. (The BLISS-11 flow analysis is not sufficiently general in that it only analyzes single functions, and within a function the only control constructs are conditionals, loops, and block exits.)

For the allocation of registers and temporaries I propose to use a modification of BLISS-11's preferencing and targeting scheme. This will be tempered by the attitude towards LAMBDA-binding described earlier, namely that it is merely a renaming operation. Thus, no variable is considered to have a specific location or "home"; assignment to a variable should cause no motion of data, but merely reorganize the compiler's idea of where the quantity involved is located at that point in the code. (At any point in the code a quantity may have several names, and may also have several homes, by which I mean physical copies in the runtime machine environment. For example, a quantity may happen to reside in two different registers at some point; there is no a priori reason for either one to be considered THE original copy of that quantity to be preserved for the future.) Data structures are another matter; assignment to a component must actually modify the component. This is the purpose of introducing the ASET primitive, since simple SETQ's as used in LISP PROG statements can be simulated by using LAMBDA expressions (see [Steele 76]). (On the other hand, the modification to the component need not happen immediately, as long as it happens soon enough that some other process, if any cannot detect that the assignment did not happen immediately.)

The essential set of primitives will include the following:

LAMBDA, LABELS, IF
ASET (perhaps restricted to a quoted first argument, i.e. ASETQ)
EQ

These by themselves constitute an extremely rich domain for optimization! As shown above and in [Steele 76], they effectively encompass data structure creation, access, and modification; a host of control structures, including non-local exits; and a variety of parameter-passing disciplines, including call-by-name, call-by-need, and fluid variables. In fact, one of the great assets of this approach is that such constructs can be written as macros and used in both an interpreter and compiler; because the base language is

essentially a lexically scoped LISP, the lambda-calculus-theoretic models of such constructs may be used almost directly to write such macros. Indeed, a library of such macros already exists for the SCHEME interpreter.

If time permits, extensions may be made to handle integers (strictly speaking, integers mod 2^{36}!) and these operations on them:

+	addition
-	subtraction
*	multiplication
//	division
\	remainder
MAX, MIN	maximum and minimum
BOOLE	bit-wise boolean operations
LSH	logical shifting
<, >, =	relationals

The compiler will need to contain the following kinds of knowledge in great detail:

Knowledge about the behavior of LAMBDA expressions and closures, in particular how environments nest and interact, and how procedure integration works. For example, in the situation:

```
(LABELS ((FOO (LAMBDA () ... (BAR)))
         (BAR (LAMBDA () ...)))
        ...)
```

the compiler should be able to realize that FOO and BAR run in the same environment (since each adds no variables to the outer environment), and so the call to BAR in FOO can compile as if it were a GOTO, with no adjustment in environment necessary. If that is the only call on BAR, then no GOTO is needed; the code for BAR may simply follow (or be integrated into) FOO.

Knowledge about how to construct data structures in the run-time environment. In MacLISP, this will imply using the built-in CONS and other low-level storage allocation primitives.

Knowledge about how primitives can be compiled into machine code, or if they are run-time routines, with what conventions they are invoked.

Knowledge about optimization of machine code, for example that a certain combination of PUSH and JRST can be combined into PUSHJ.

Each kind of knowledge should be represented as modularly as possible. It should not be necessary to change fifteen routines in the compiler just to install a new arithmetic primitive.

Although I propose to construct only the lower-level portion of the compiler, plus the necessary macros to provide standard LISP features such as COND and PROG, one could easily imagine constructing an ALGOL compiler, for example, by providing a parser plus the necessary macros as a front end. Indeed, by using CGOL [Pratt 76] for our parser we could create an ALGOL-like language quite easily, which could include such non-LISP features as multiple-value-return and call-by-name.

This possibility indicates that a carefully chosen small dialect of LISP would be a good UNCOL (UNiversal Computer-Oriented Language), that is, a good intermediate compilation language. The reason for trying to develop a good UNCOL is to solve the "m*n" problem compiler-builders face. If one has m programming languages and n machines, then it takes m*n compilers to compile each language for each machine; but it would require only m+n compilers if one had m language-to-UNCOL compilers and n UNCOL-to-machine compilers. Up to

now the UNCOL idea has failed; the usual problem is that proposed UNCOLs are not sufficiently general. I suspect that this is because they tend to be too low-level, too much like machine language. I believe that LAMBDA expressions, combining the most primitive control operator (GOTO) with the most primitive environment operator (renaming) put LISP at a low enough level to make a good UNCOL, yet at a high enough level to be completely machine independent.

It should be noted that a compiler which uses LAMBDA expressions internally does not have to be for a LISP-like language. The features of LISP as an UNCOL which interest me are the environment and control primitives, because they can be used easily to simulate the environment and control structures of most high-level languages.

4.2. Some Side Issues

In this section we discuss briefly some issues which are not directly relevant to LAMBDA expressions, but which will impinge on the design of a compiler. These are:

(1) Order of argument evaluation (as opposed to order of evaluation, which is to be applicative order).
(2) Analysis of side effects and their interactions.
(3) Declarations versus compile-time analysis.
(4) Block compilation (in the InterLISP sense); i.e., inter-function optimization.
(5) Debugging; in particular, the ability to walk around environment structures and examine their contents.
(6) Bootstrapping.

Because these are side issues, we will merely consider an easy way out for each, realizing that the compiler should be designed so as to allow for more complex ways of handling them later.

(1) The two standard choices for order of argument evaluation are "left to right" and "doesn't matter". Most LISP systems use the former convention; most languages with infix syntax, notably BLISS [Wulf 75], use the latter. If the latter is chosen, there is the matter of whether the compiler will enforce it by warning the user if he tries to depend on some ordering; if the former is chosen, there is the matter of determining whether the compiler can perform optimizations which depend on permuting the order. In either of these cases an analysis of side-effect interactions among arguments is necessary, and once we have decided to perform such an analysis, the choice of convention is not too important.

(2) Analysis of side effects is desirable not only between arguments in a single function call, but at all levels. For example, if a copy of a variable is in a register, then it need not be re-fetched unless an assignment has occurred. Similarly, if CONS as above were extended to have an RPLACA message, we would like to know whether sending a given cell an RPLACA message will require re-transmission of a CAR message. The easy approach is to assume that an unknown function changes everything, and not to attempt optimization across calls to unknown functions. Other increasingly clever approaches include:

Declaration of whether a function can produce side effects or not.
Declaration of what kinds of arguments to a function produce side effects.

Declaration of classes of side effects. Thus an RPLACA causes previous CAR operations to become invalid, but not simple variable fetches or GETs.

Declarations of classes of side effects as a function of certain objects and/or arguments. Thus (PUTPROP x y z) invalidates all previous GET operations; (PUTPROP x y 'ZAP) invalidates (GET x 'ZAP) and (GET x y), but not (GET x 'ZIP). Neither one invalidates (CAR FOO), probably. Similarly (RPLACA FOO) does not invalidate (CAR FIE) if it can be determined that FOO and FIE are distinct objects.

Naturally, anything described above as being declared could sometimes also be determined by a clever compile-time analysis.

(3) As for the issue of declarations versus analysis itself, probably both should be available. One might envision first implementing a declaration scheme which can be used to make the thing go, and then adding analysis routines afterwards. The analysis routines should merely create declarations in the same way the user can; this would allow uniformity of processing and extensibility of design.

(4) Block compilation might be necessary for production of very efficient code, though this will of course depend on the style of programming. If many data structures are defined in the actor-like style described above, much procedure integration will be necessary to produce good code. On the other hand, the code should still work if each procedure is compiled separately (which would be desirable for debugging purposes). A middle-of-the-road approach would be to block-compile a set of functions, and compile separate entry points to certain function for interpreted and compiled calls.

(5) While debugging, it may be desirable to be able to examine the environment structures created by compiled code. This probably will have to be a kind of deus ex machina rather than an integral part of the system, but in any case there must be enough information to determine where things are. Environments created by compiled code will not contain the names of the variables, since they are not logically necessary. Instead, the compiler can, for each LAMBDA expression, create a description, suitable for interpretation by a debugging program, of the format of closures created for that LAMBDA expression. This will be enough information to debug with. The compiler could also theoretically output information as to what data is in which registers when, though this would be a mountain of output. This would tie in well with a program-understanding program; one could provide information as to what compiled code correponds to what interpreted code, and how.

(6) One problem with constructing a compiler is deciding what language to code the compiler itself in. As a rule of thumb, one ought to take a very dim view of any supposedly general-purpose language which is not adequate to write its own compiler in. Thus the proposed compiler will be constructed in some superset of the basic LISP described above, one which can easily be transformed by macros into the basic LISP. One advantage of this carefully chosen minimal dialect is that an interpreter for it can be written and debugged in only a day or two. This interpreter can then be used as a development system for writing the machine-dependent portion of the compiler. Thus if necessary the proposed compiler could be bootstrapped onto a new machine easily without requiring the aid of a previously existing implementation.

5. Conclusions

It is appropriate to think of function calls as being a generalization of GOTO, and thus a fundamental unconditional control construct. The traditional attitude towards function calls as expensive operations is unjustified, and can be alleviated by proper compilation techniques.

It is appropriate to think of LAMBDA as an environment operator, which merely attaches new names to computational quantities and defines the extent of the significance of these names. The attitude of assigning names to values rather than values to names leads naturally to a uniform treatment of user and compiler-generated variables.

These results lead naturally to techniques for compiling and optimizing operations on procedurally defined data. This is to be compared with other work, particularly that on actors. Here let us summarize our comparison of actors (as implemented by PLASMA) and closures (as implemented by SCHEME, i.e. LISP):

Closures	Actors
Body of LAMBDA expression	Script
Environment	Set of acquaintances
Variable names	Names (compiled out at reduction time)
Function invocation	Invocation of explicit actors
Function return	Invocation of implicit actors
Return address	Implicit underlying continuation
Continuation-passing style	Exclusive use of =≡> and <≡=
Spreading of arguments	Pattern matching
Temporary (intermediate result)	Name internal to implicit continuation

Let us also summarize some of the symmetries we have seen in the functional style of programming:

Forms (Function Invocations)	Functions (LAMBDA Expressions)
Evaluation	Application (function invocation)
Push control stack before invoking functions which produce argument values	Push environment stack before evaluating form which produces result value
Forms determine sequencing in time	LAMBDA expressions determine extent in space (scope)
Implicit continuation is created when evaluation of a form requires invocation of a function	Implicit temporary is created when return of a function requires further processing of a form

It is important to note that this last symmetry was not known to me ahead of time; I discovered it while writing this document. Starting from the assumption that control and environment structures exhibit great symmetry, plus the knowledge of the existence of implicit continuations, I predicted the existence of hidden temporaries; only then did I notice that such temporaries do occur. I believe this demonstrates that there is something very deep and fundamental about this symmetry, closely tied in to the distinction between form and function. Just as the notion of actors and message-passing has greatly clarified our ideas about control structures, so the notion of renaming has clarified our ideas about environments.

Appendix A. Conversion to Continuation-Passing Style

Here we present a set of functions, written in SCHEME, which convert a SCHEME expression from functional style to pure continuation-passing style. {Note PLASMA CPS}

```
(ASET' GENTEMPNUM 0)

(DEFINE GENTEMP
        (LAMBDA (X)
                (IMPLODE (CONS X (EXPLODEN (ASET' GENTEMPNUM (+ GENTEMPNUM 1)))))))
```

GENTEMP creates a new unique symbol consisting of a given prefix and a unique number.

```
(DEFINE CPS (LAMBDA (SEXPR) (SPRINTER (CPC SEXPR NIL '#CONT#))))
```

CPS (Continuation-Passing Style) is the main function; its argument is the expression to be converted. It calls CPC (C-P Conversion) to do the real work, and then calls SPRINTER to pretty-print the result, for convenience. The symbol #CONT# is used to represent the implied continuation which is to receive the value of the expression.

```
(DEFINE CPC
        (LAMBDA (SEXPR ENV CONT)
                (COND ((ATOM SEXPR) (CPC-ATOM SEXPR ENV CONT))
                      ((EQ (CAR SEXPR) 'QUOTE)
                       (IF CONT "(,CONT ,SEXPR) SEXPR))
                      ((EQ (CAR SEXPR) 'LAMBDA)
                       (CPC-LAMBDA SEXPR ENV CONT))
                      ((EQ (CAR SEXPR) 'IF)
                       (CPC-IF SEXPR ENV CONT))
                      ((EQ (CAR SEXPR) 'CATCH)
                       (CPC-CATCH SEXPR ENV CONT))
                      ((EQ (CAR SEXPR) 'LABELS)
                       (CPC-LABELS SEXPR ENV CONT))
                      ((AND (ATOM (CAR SEXPR))
                            (GET (CAR SEXPR) 'AMACRO))
                       (CPC (FUNCALL (GET (CAR SEXPR) 'AMACRO) SEXPR) ENV CONT))
                      (T (CPC-FORM SEXPR ENV CONT)))))
```

CPC merely dispatches to one of a number of subsidiary routines based on the form of the expression SEXPR. ENV represents the environment in which SEXPR will be evaluated; it is a list of the variable names. When CPS initially calls CPC, ENV is NIL. CONT is the continuation which will receive the value of SEXPR. The double-quote (") is like a single-quote, except that within the quoted expression any subexpressions preceded by comma (,) are evaluated and substituted in (also, any subexpressions preceded by atsign (@) are substituted in a list segments). One special case handled directly by CPC is a quoted expression; CPC also expands any SCHEME macros encountered.

```
(DEFINE CPC-ATOM
        (LAMBDA (SEXPR ENV CONT)
                ((LAMBDA (AT) (IF CONT "(,CONT ,AT) AT))
                 (COND ((NUMBERP SEXPR) SEXPR)
                       ((MEMQ SEXPR ENV) SEXPR)
                       ((GET SEXPR 'CPS-NAME))
                       (T (IMPLODE (CONS '% (EXPLODEN SEXPR))))))))
```

For convenience, CPC-ATOM will change the name of a global atom. Numbers and atoms in the environment are not changed; otherwise, a specified name on the property list of the given atom is used (properties defined below convert "+" into "++", etc.); otherwise, the name is prefixed with "%". Once the name has been converted, it is converted to a form which invokes the continuation on the atom. (If a null continuation is supplied, the atom itself is returned.)

```
(DEFINE CPC-LAMBDA
        (LAMBDA (SEXPR ENV CONT)
                ((LAMBDA (CN)
                         ((LAMBDA (LX) (IF CONT "(,CONT ,LX) LX))
                          "(LAMBDA (@(CADR SEXPR) ,CN)
                                   ,(CPC (CADDR SEXPR)
                                         (APPEND (CADR SEXPR) (CONS CN ENV))
                                         CN))))
                 (GENTEMP 'C))))
```

A LAMBDA expression must have an additional parameter, the continuation supplied to its body, added to its parameter list. CN holds the name of this generated parameter. A new LAMBDA expression is created, with CN added, and with its body converted in an environment containing the new variables. Then the same test for a null CONT is made as in CPC-ATOM.

```
(DEFINE CPC-IF
        (LAMBDA (SEXPR ENV CONT)
                ((LAMBDA (KN)
                         "((LAMBDA (,KN)
                                   ,(CPC (CADR SEXPR)
                                         ENV
                                         ((LAMBDA (PN)
                                                  "(LAMBDA (,PN)
                                                           (IF ,PN
                                                               ,(CPC (CADDR SEXPR)
                                                                     ENV
                                                                     KN)
                                                               ,(CPC (CADDDR SEXPR)
                                                                     ENV
                                                                     KN))))
                                          (GENTEMP 'P))))
                           ,CONT))
                 (GENTEMP 'K))))
```

First, the continuation for an IF must be given a name KN (rather, the name held in KN; but for convenience, we will continue to use this ambiguity, for the form of the name is indeed Kn for some number n), for it will be referred to in two places and we wish to avoid duplicating the code. Then, the predicate is converted to continuation-passing style, using a continuation

which will receive the result and call it PN. This continuation will then use an IF to decide which converted consequent to invoke. Each consequent is converted using continuation KN.

```
(DEFINE CPC-CATCH
        (LAMBDA (SEXPR ENV CONT)
                ((LAMBDA (EN)
                         `((LAMBDA (,EN)
                                   ((LAMBDA (,(CADR SEXPR))
                                            ,(CPC (CADDR SEXPR)
                                                  (CONS (CADR SEXPR) ENV)
                                                  EN))
                                    (LAMBDA (V C) (,EN V))))
                           ,CONT))
                 (GENTEMP 'E))))
```

This routine handles CATCH as defined in [Sussman 75], and in converting it to continuation-passing style eliminates all occurrences of CATCH. The idea is to give the continuation a name EN, and to bind the CATCH variable to a continuation (LAMBDA (V C) ...) which ignores its continuation and instead exits the catch by calling EN with its argument V. The body of the CATCH is converted using continuation EN.

```
(DEFINE CPC-LABELS
        (LAMBDA (SEXPR ENV CONT)
                (DO ((X (CADR SEXPR) (CDR X))
                     (Y ENV (CONS (CAAR X) Y)))
                    ((NULL X)
                     (DO ((W (CADR SEXPR) (CDR W))
                          (Z NIL (CONS (LIST (CAAR W)
                                             (CPC (CADAR W) Y NIL))
                                       Z)))
                         ((NULL W)
                          `(LABELS ,(REVERSE Z)
                                   ,(CPC (CADDR SEXPR) Y CONT))))))))
```

Here we have used DO loops as defined in MacLISP (DO is implemented as a macro in SCHEME). There are two passes, one performed by each DO. The first pass merely collects in Y the names of all the labelled LAMBDA expressions. The second pass converts all the LAMBDA expressions using a null continuation and an environment augmented by all the collected names in Y, collecting them in Z. At the end, a new LABELS is constructed using the results in Z and a converted LABELS body.

```
(DEFINE CPC-FORM
        (LAMBDA (SEXPR ENV CONT)
                (LABELS ((LOOP1
                          (LAMBDA (X Y Z)
                                  (IF (NULL X)
                                      (DO ((F (REVERSE (CONS CONT Y))
                                              (IF (NULL (CAR Z)) F
                                                  (CPC (CAR Z)
                                                       ENV
                                                       `(LAMBDA (,(CAR Y)) ,F))))
                                           (Y Y (CDR Y))
                                           (Z Z (CDR Z)))
                                          ((NULL Z) F))
                                      (COND ((OR (NULL (CAR X))
                                                 (ATOM (CAR X)))
                                             (LOOP1 (CDR X)
                                                    (CONS (CPC (CAR X) ENV NIL) Y)
                                                    (CONS NIL Z)))
                                            ((EQ (CAAR X) 'QUOTE)
                                             (LOOP1 (CDR X)
                                                    (CONS (CAR X) Y)
                                                    (CONS NIL Z)))
                                            ((EQ (CAAR X) 'LAMBDA)
                                             (LOOP1 (CDR X)
                                                    (CONS (CPC (CAR X) ENV NIL) Y)
                                                    (CONS NIL Z)))
                                            (T (LOOP1 (CDR X)
                                                      (CONS (GENTEMP 'T) Y)
                                                      (CONS (CAR X) Z))))))))
                         (LOOP1 SEXPR NIL NIL))))
```

This, the most complicated routine, converts forms (function calls). This also operates in two passes. The first pass, using LOOP1, uses X to step down the expression, collecting data in Y and Z. At each step, if the next element of X can be evaluated trivially, then it is converted with a null continuation and added to Y, and NIL is added to Z. Otherwise, a temporary name TN for the result of the subexpression is created and put in Y, and the subexpression itself is put in Z. On the second pass (the DO loop), the final continuation-passing form is constructed in F from the inside out. At each step, if the element of Z is non-null, a new continuation must be created. (There is actually a bug in CPC-FORM, which has to do with variables affected by side-effects. This is easily fixed by changing LOOP1 so that it generates temporaries for variables even though variables evaluate trivially. This would only obscure the examples presented below, however, and so this was omitted.)

```
(LABELS ((BAR
          (LAMBDA (DUMMY X Y)
                  (IF (NULL X) '|CPS ready to go!|
                      (BAR (PUTPROP (CAR X) (CAR Y) 'CPS-NAME)
                           (CDR X)
                           (CDR Y))))))
         (BAR NIL
              '(+ - * // ^ T NIL)
              '(++ -- ** //// ^^ 'T 'NIL)))
```

This loop sets up some properties so that "+" will translate into "++" instead of "%+", etc.

Now let us examine some examples of the action of CPS. First, let us try our old friend FACT, the iterative factorial program.

```
(DEFINE FACT
        (LAMBDA (N)
                (LABELS ((FACT1 (LAMBDA (M A)
                                        (IF (= M 0) A
                                            (FACT1 (- M 1) (* M A))))))
                        (FACT1 N 1))))
```

Applying CPS to the LAMBDA expression for FACT yields:

```
(#CONT#
    (LAMBDA (N C7)
            (LABELS ((FACT1
                      (LAMBDA (M A C10)
                          ((LAMBDA (K11)
                                   (%= M 0
                                       (LAMBDA (P12)
                                           (IF P12 (K11 A)
                                               (-- M 1
                                                   (LAMBDA (T13)
                                                       (** M A
                                                           (LAMBDA (T14)
                                                               (FACT1 T13 T14 K11)))))))))
                           C10))))
                     (FACT1 N 1 C7))))
```

As an example of CATCH elimination, here is a routine which is a paraphrase of the SQRT routine from [Sussman 75]:

```
(DEFINE SQRT
        (LAMBDA (X EPS)
                ((LAMBDA (ANS LOOPTAG)
                         (CATCH RETURNTAG
                                (BLOCK (ASET' LOOPTAG (CATCH M M))
                                       (IF ---
                                           (RETURNTAG ANS)
                                           NIL)
                                       (ASET' ANS ===)
                                       (LOOPTAG LOOPTAG))))
                 1.0
                 NIL)))
```

Here we have used "---" and "===" as ellipses for complicated (and relatively uninteresting) arithmetic expressions. Applying CPS to the LAMBDA expression for SQRT yields:

```
(#CONT#
 (LAMBDA (X EPS C33)
     ((LAMBDA (ANS LOOPTAG C34)
          ((LAMBDA (E35)
               ((LAMBDA (RETURNTAG)
                    ((LAMBDA (E52)
                         ((LAMBDA (M) (E52 M))
                          (LAMBDA (V C) (E52 V))))
                     (LAMBDA (T51)
                             (%ASET' LOOPTAG T51
                                     (LAMBDA (T37)
                                         ((LAMBDA (A B C36) (B C36))
                                          T37
                                          (LAMBDA (C40)
                                              ((LAMBDA (K47)
                                                   ((LAMBDA (P50)
                                                        (IF P50
                                                            (RETURNTAG ANS K47)
                                                            (K47 'NIL)))
                                                    %---))
                                               (LAMBDA (T42)
                                                   ((LAMBDA (A B C41) (B C41))
                                                    T42
                                                    (LAMBDA (C43)
                                                        (%ASET' ANS %===
                                                                (LAMBDA (T45)
                                                                    ((LAMBDA (A B C44)
                                                                             (B C44))
                                                                     T45
                                                                     (LAMBDA (C46)
                                                                         (LOOPTAG
                                                                          LOOPTAG
                                                                          C46))
                                                                     C43))))
                                                    C40))))
                                          E35))))))
                (LAMBDA (V C) (E35 V))))
           C34))
      1.0
      'NIL
      C33)))
```

Note that the CATCHes have both been eliminated. It is left as an exercise for the reader to verify that the continuation-passing version correctly reflects the semantics of the original.

Appendix B. Continuation-Passing with Multiple Value Return

The program of Appendix A can easily be modified to handle the multiple-value-return construct. Here we present only the functions which must be changed; all others are as in Appendix A.

```
(SETSYNTAX '/{ 'MACRO
          '(LAMBDA ()
                  (DO ((L NIL (CONS (READ) L)))
                      ((= (TYIPEEK T) 175)          ;ASCII 175 is ")"
                       (TYI)
                       (LIST 'MULTI-RETURN
                             (READ)
                             (REVERSE L))))))

(SETSYNTAX '/) 600500 NIL)
```

This defines the syntactic rule for reading in "{...}n" construct. A call of the form {f x1 ... xm}n is converted into the piece of list structure:

```
(MULTI-RETURN n f x1 ... xm)
```

It is this which is processed by CPC-FORM below.

```
(DEFINE CPC
        (LAMBDA (SEXPR ENV CONT)
                (COND ((ATOM SEXPR) (CPC-ATOM SEXPR ENV CONT))
                      ((EQ (CAR SEXPR) 'QUOTE)
                       (IF CONT "(,CONT ,SEXPR) SEXPR))
                      ((EQ (CAR SEXPR) 'LAMBDA)
                       (CPC-LAMBDA SEXPR ENV CONT))
                      ((EQ (CAR SEXPR) 'IF)
                       (CPC-IF SEXPR ENV CONT))
                      ((EQ (CAR SEXPR) 'CATCH)
                       (CPC-CATCH SEXPR ENV CONT))
                      ((EQ (CAR SEXPR) 'VALUES)                          ;new
                       (CPC-FORM (CONS CONT (CDR SEXPR)) ENV NIL))       ; clause
                      ((EQ (CAR SEXPR) 'LABELS)
                       (CPC-LABELS SEXPR ENV CONT))
                      ((AND (ATOM (CAR SEXPR))
                            (GET (CAR SEXPR) 'AMACRO))
                       (CPC (FUNCALL (GET (CAR SEXPR) 'AMACRO) SEXPR) ENV CONT))
                      (T (CPC-FORM SEXPR ENV CONT)))))
```

The only change here is the test marked "new clause"; it checks for the VALUES construct. It calls CPC-FORM in such a way that the continuation is given all the specified values as its arguments. The third argument of NIL to CPC-FORM means that the first argument has no extra implicit continuation.

```
(DEFINE CPC-FORM
        (LAMBDA (SEXPR ENV CONT)
                (LABELS ((LOOP1
                          (LAMBDA (X Y Z)
                                  (IF (NULL X)
                                      (DO ((F (REVERSE ((LAMBDA (Q)
                                                                (IF CONT (CONS CONT Q) Q))
                                                        (APPLY 'APPEND Y)))
                                               (IF (NULL (CAR Z))
                                                   F
                                                   (CPC (CAR Z)
                                                        ENV
                                                        "(LAMBDA ,(REVERSE (CAR Y)) ,F))))
                                           (Y Y (CDR Y))
                                           (Z Z (CDR Z)))
                                          ((NULL Z) F))
                                      (COND ((OR (NULL (CAR X))
                                                 (ATOM (CAR X)))
                                             (LOOP1 (CDR X)
                                                    (CONS (LIST (CPC (CAR X) ENV NIL)) Y)
                                                    (CONS NIL Z)))
                                            ((EQ (CAAR X) 'QUOTE)
                                             (LOOP1 (CDR X)
                                                    (CONS (LIST (CAR X)) Y)
                                                    (CONS NIL Z)))
                                            ((EQ (CAAR X) 'LAMBDA)
                                             (LOOP1 (CDR X)
                                                    (CONS (LIST (CPC (CAR X) ENV NIL)) Y)
                                                    (CONS NIL Z)))
                                            ((EQ (CAAR X) 'MULTI-RETURN)
                                             (DO ((V NIL (CONS (GENTEMP 'V) V))
                                                  (J (CADAR X) (- J 1)))
                                                 ((= J 0)
                                                  (LOOP1 (CDR X)
                                                         (CONS V Y)
                                                         (CONS (CADDAR X) Z)))))
                                            (T (LOOP1 (CDR X)
                                                      (CONS (LIST (GENTEMP 'T)) Y)
                                                      (CONS (CAR X) Z))))))))
                         (LOOP1 SEXPR NIL NIL))))
```

This function has been changed radically to accomodate MULTI-RETURN. The conceptual alterations are that CONT may be NIL (meaning no explicit continuation, because SEXPR already contains one), and that each element of Y is now a list of temporary names or constants, and not just a single element (hence the use of APPEND). There is also a new case in the COND for MULTI-RETURN.

As an example, here is the example used in the text, processed by this codified version of CPS:

```
(CPS '(LIST (FOO 5)2 (+ (FOO 4)2) (FOO 3)2))

(%FOO 5 (LAMBDA (V1 V2)
          (%FOO 4 (LAMBDA (V6 V7)
                    (++ V6 V7
                        (LAMBDA (T3)
                                (%FOO 3 (LAMBDA (V4 V5)
                                          (%LIST V1 V2 T3 V4 V5 #CONT#)))))))))
```

The only differences between this result and the one in the text is that the continuation-passing versions of LIST and "+" were used, and that the variable names were chosen differently.

Notes

{Note Debugging}

As every machine-language programmer of a stack machine knows, the extra address on the stack is not entirely useless because it contains some redundant information about the history of the process. This information is provided by standard LISP systems in the form of a "backtrace". [McCarthy 62] [Moon 74] [Teitelman 74] This information may give some clues as to "how it got there". One may compare this to the "jump program counter" hardware feature supplied on some machines (notably the PDP-10's at MIT [Holloway 70]) which saves the contents of the program counter before each jump instruction.

{Note Expensive Procedures}

Fateman comments on this difficulty in [Fateman 73]: "...'the frequency and generality of function calling in LISP' is a high cost only in inappropriately designed computers (or poorly designed LISP systems). To illustrate this, we ran the following program in FORTRAN ... [execution time 2.22 sec] ... We then transcribed it into LISP, and achieved the following results: ... [execution time 1.81 sec] ...

"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."

Auslander and Strong discuss in [Auslander 76] a technique for "removing recursion" from PL/I programs which LISP programmers will recognize as a source-to-source semi-compilation. The technique essentially consists of of introducing an auxiliary array to serve as a stack (though the cited paper manages in the example to use an already existing array by means of a non-trivial subterfuge), and transforming procedure calls into GOTO's plus appropriate stack manipulations to simulate return addresses. What is astounding is that this simple trick shortened the size of the example code by 8% and shortened the run time by a whopping 40%! They make the reason clear: "The implementation of the recursive stack costs PL/I 336 bytes per level of recursive call ..." The GOTO's, on the other hand, presumably compile into single branch instructions, and the stack manipulations are just a few arithmetic instructions.

Even more astounding, particularly in the light of existing compiler technology for LISP and other languages, is that Auslander and Strong do not advocate fixing the PL/I compiler to compile procedure calls using their techniques (as LISP compilers have, to some extent, for years). Instead, they say: "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." The bulk of their tranformations are well within the capability of an optimizing compiler. The problem is that historically procedure calls have received little attention from those who design optimizing compilers; Auslander and Strong now suggest that, since this is the case, we should rewrite all procedure calls into other constructs that the compiler understands better! This seems to defeat the entire purpose of having a high-level language.

On pages 8-9 of Dijkstra's excellent book [Dijkstra 76] he says: "In a recent educational text addressed to the PL/I programmer one can find the strong advice to avoid procedure calls as much as possible 'because they make the program so inefficient'. In view of the fact that the procedure is one of PL/I's main vehicles for expressing structure, this is a terrible advice, so terrible that I can hardly call the text in question 'educational'. If you are convinced of the usefulness of the procedure concept and are surrounded by implementations in which the overhead of the procedure mechanism imposes too great a penalty, then blame these inadequate implementations instead of raising them to the level of standards!"

{Note GCD(111,259)}

This marvelous passage occurs on page 4 of [Dijkstra 76]:

"Instead of considering the single problem of how to compute the GCD(111,259), we have generalized the problem and have regarded this as a specific instance of the wider class of problems of how to compute GCD(X,Y). It is worthwhile to point out that we could have generalized the problem of computing GCD(111,259) in a different way: we could have regarded the task as a specific instance of a wider class of tasks, such as the computation of GCD(111,259), SCM(111,259), 111*259, 111+259, 111/259, 111-259, 111^{259}, the day of the week of the 111th day of the 259th year B.C., etc. This would have given rise to a '111-and-259 processor' and in order to let that produce the originally desired answer, we should have had to give the request 'GCD, please' as its input! ...

"In other words, when asked to produce one or more results, it is usual to generalize the problem and to consider these results as specific instances of a wider class. But it is no good just to say that everything is special case of something more general! If we want to follow such an approach we have two obligations:

1. We have to be quite specific as to how we generalize ...
2. We have to choose ('invent' if you wish) a generalization which is helpful to our purpose."

{Note No IF-THEN-ELSE}

The IF-THEN-ELSE construct can be expressed rather clumsily in terms of sequencing and WHILE-DO by introducing a control variable; this is described by Bob Haas in [Presser 75]. A general discussion of the relative complexities of various sets of control structures appears in [Lipton 76].

{Note PLASMA CPS}

Hewitt has performed similar experiments on PLASMA programs [Hewitt 76], by converting PLASMA programs to a form which uses only ==> and <== transmission arrows. A subsequent uniform replacement of these arrows by => and <= preserves the semantics of the programs.

{Note PLASMA Reduction}

Since this was written, there were two changes to the PLASMA implementation. The first, in mid-summer, was a change in terminology, in

which the "reduction" prepass began to be referred to as a "compilation". The second, in August, was the excision of reduction from the PLASMA implementation, evidently because the size of the code was becoming unmanageable. [Hewitt 76] [McLennan 76] It is unfortunate that this experiment in semi-incremental compilation could not be continued.

{Note PLASMA Registers}

In fact, the current implementation of PLASMA happens to work in this way, since the implicit continuations are handled just like any other actor. However, it does not presently take much advantage of this fact since there are no constructs defined to create multiple-argument continuations.

{Note PLASMA Sugar}

PLASMA, for example, provides such sugar in abundance. Many "standard" control and data operations are provided and defined in terms of actor transmissions. Indeed, the user need not be aware of the semantics of actors at all; there is enough sugar to hide completely what is really going on.

{Note Return Address}

There is actually a third quantity passed to BAR, namely the return address; this is not given an explicit name by either BAR or its caller. Instead, the functional notation of LISP leaves the handling of the return address entirely implicit. Later, in the discussion of continuations, the return address will be given an explicit name just like any other passed parameter.

{Note Slice Both Ways}

One may also try slicing the matrix up in both directions, so that each entry may be specified as a separate module. This has been tried in REDUCE, for example. [Griss 76] It can lead to a rather disjointed style of programnming, however; in practice, one tends to group routines which all fall in a single row or column of the operations matrix.

{Note Turing Machines}

Compare this with the basic action of a Turing machine, which is to use two parameters (the current state and the symbol under the tape head) to index a matrix of actions to take.

References

[Allen 72]
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).

[Allen 76]
Allen, Frances E., and Cocke, John. "A Program Data Flow Analysis Procedure." Comm. ACM 19, 3 (March 1976), 137-147.

[Auslander 76]
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).

[Bobrow 73]
Bobrow, Daniel G. and Wegbreit, Ben. "A Model and Stack Implementation of Multiple Environments." CACM 16, 10 (October 1973) pp. 591-603.

[Campbell 74]
Campbell, R.H., and Habermann, A.N. The Specification of Process Synchronization by Path Expressions. Technical Report 55. Comp. Lab., U. Newcastle upon Tyne (January 1974).

[Church 41]
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).

[Dijkstra 67]
Dijkstra, Edsger W. "Recursive Programming." In Rosen, Saul (ed.), Programming Systems and Languages. McGraw-Hill (New York, 1967).

[Dijkstra 76]
Dijkstra, Edsger W. A Discipline of Programming. Prentice-Hall (Englewood CLiffs, N.J., 1976).

[Fateman 73]
Fateman, Richard J. "Reply to an Editorial." SIGSAM Bulletin 25 (March 1973), 9-11.

[Forte 67]
Forte, Allen. SNOBOL3 Primer. The MIT Press (Cambridge, 1967).

[Galley 75]
Galley, S.W. and Pfister, Greg. The MDL Language. Programming Technology Division Document SYS.11.01. Project MAC, MIT (Cambridge, November 1975).

[Griss 76]
Griss, Martin L. "The Definition and Use of Data Structures in REDUCE." Proc. ACM Symposium on Symbolic and Algebraic Computation (August 1976).

[Hauck 68]
Hauck, E.A., and Dent, B.A. "Burroughs' B6500/B7500 Stack Mechanism." Proc. AFIPS Conference Vol. 32 (1968).

[Hewitt 73]
Hewitt, Carl. "Planner." In Project MAC Progress Report X (July 72-July 73). MIT Project MAC (Cambridge, 1973), 199-230.

[Hewitt 76]
Hewitt, Carl. Personal communications and talks (1975-76).

[Hoare 74]
Hoare, C.A.R. "Monitors: an Operating System Structuring Concept." Comm. ACM 17, 10 (October 1974).

[Holloway 70]
Holloway, J. PDP-10 Paging Device. Hardware Memo 2. MIT AI Lab (Cambridge, February 1970).

[Johnsson 75]
Johnsson, Richard Karl. An Approach to Global Register Allocation. Ph.D. Thesis. Carnegie-Mellon University (Pittsburgh, December 1975).

[Knight 74]
Knight, Tom. The CONS microprocessor. Artificial Intelligence Working Paper 80, MIT (Cambridge, November 1974).

[Lipton 76]
Lipton, R.J., Eisenstat, S.C., and DeMillo, R.A. "Space and Time Heirarchies for Classes of Control Structures and Data Structures." Journal ACM 23, 4 (October 1976), 720-732.

[Liskov 74]
Liskov, Barbara, and Zilles, Stephen. "Programming with Abstract Data Types." Proc. Symp. on Very High Level Languages. SIGPLAN Notices, April 1974.

[Liskov 76]
Liskov, Barbara, et al. CLU Design Notes. MIT Lab. for Computer Science (Cambridge, 1973-1976).

[McCarthy 60]
McCarthy, John. "Recursive functions of symbolic expressions and their computation by machine - I." Comm. ACM 3, 4 (April 1960), 184-195.

[McCarthy 62]
McCarthy, John, et al. LISP 1.5 Programmer's Manual. The MIT Press (Cambridge, 1962).

[McDermott 74]
McDermott, Drew V. and Sussman, Gerald Jay. The CONNIVER Reference Manual. AI Memo 295a. MIT AI Lab (Cambridge, January 1974).

[McLennan 76]
McLennan, Marilyn. Private communication, 1976.

[MITRLE 62]
COMIT Programmers Reference Manual. MIT Research Laboratory of Electronics. The MIT Press (Cambridge, 1962).

[Moon 74]
Moon, David A. MACLISP Reference Manual, Revision 0. Project MAC, MIT (Cambridge, April 1974).

[Moses 70]
Moses, Joel. The Function of FUNCTION in LISP. AI Memo 199, MIT AI Lab (Cambridge, June 1970).

[Pratt 76]
Pratt, Vaughan R. CGOL - An Alternative External Representation for LISP Users. AI Working Paper 121. MIT AI Lab (Cambridge, March 1976).

[Presser 75]
Presser, Leon. "Structured Languages." Proc. National Computer Conference 1975. Reprinted in SIGPLAN Notices 10, 7 (July 1975), 22-24.

[Reynolds 72]
Reynolds, John C. "Definitional Interpreters for Higher Order Programming Languages." ACM Conference Proceedings 1972.

[Smith 75]
Smith, Brian C. and Hewitt, Carl. A PLASMA Primer (draft). MIT AI Lab (Cambridge, October 1975).

[Snyder 75]
Snyder, Alan. A Portable Compiler for the Language C. MAC TR-149. Project MAC, MIT (Cambridge, May 1975).

[Steele 76]
Steele, Guy Lewis Jr., and Sussman, Gerald Jay. LAMBDA: The Ultimate Imperative. AI Lab Memo 353. MIT (Cambridge, March 1976).

[Stoy 74]
Stoy, Joseph. The Scott-Strachey Approach to the Mathematical Semantics of Programming Languages. Project MAC Report. MIT (Cambridge, December 1974).

[Sussman 71]
Sussman, Gerald Jay, Winograd, Terry, and Charniak, Eugene. Micro-PLANNER Reference Manual. AI Memo 203A. MIT AI Lab (Cambridge, December 1971).

[Sussman 75]
Sussman, Gerald Jay, and Steele, Guy L. Jr. SCHEME: An Interpreter for Extended Lambda Calculus. AI Lab Memo 349. MIT (Cambridge, December 1975).

[Teitelman 74]
Teitelman, Warren. InterLISP Reference Manual. Xerox Palo Alto Research Center (Palo Alto, 1974).

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

[Wulf 71]
Wulf, W.A., Russell, D.B., and Habermann, A.N. "BLISS: A Language for Systems Programming." Comm. ACM 14, 12 (December 1971), 780-790.

[Wulf 72]
Wulf, William A. "Systems for Systems Implementors -- Some Experiences from BLISS." Proc. AFIPS 1972 FJCC. AFIPS Press (Montvale, N.J., 1972).

[Wulf 75]
Wulf, William A., et al. The Design of an Optimizing Compiler. American Elsevier (New York, 1975).

[Yngve 72]
Yngve, Victor H. Computer Programming with COMIT II. The MIT Press (Cambridge, 1972).