Lambda: The Ultimate Declarative

Guy L. Steele, Jr.

1976

== 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.

- 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 easy 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.

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 `IF`s and `GOTO`s". 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 functional 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 again 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 inherent 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: ...

PUSHJ G

BAR1: ...

PUSHJ H

BAR2: ...

PUSHJ F

BAR3: POPJ

F: ...

POPJ

G: ...

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 turn 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 different 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 as 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.

We note that the textdual 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.

The 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 that:

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 necessary 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 turn 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:

L1:

PUSH [L1]

GOTO G

is the same as

L1:

PUSHJ G

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 turn adress on a stack, in a register, or in a memory location) works on 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 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 than 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 usually 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 know 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 those 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 of 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 locaiton.

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 leve, 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

```
(DEFINE FACT

(LAMBDA (N)

(LABELS ((FACT1

(LAMBDA (M A)

(IF (= M 0) A

(FACT1 (- M 1)

(* MA))))))

(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: 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:

FACT1:

GOTO FACT1

could be optimized by removing the GOTO.

These tricks, however, are know 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. === 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 bases 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.)

About this digital edition

This e-book comes from the online library Wikisource[1]. This multilingual digital library, built by volunteers, is committed to developing a free accessible collection of publications of every kind: novels, poems, magazines, letters...

We distribute our books for free, starting from works not copyrighted or published under a free license. You are free to use our e-books for any purpose (including commercial exploitation), under the terms of the Creative Commons Attribution-ShareAlike 3.0 Unported[2] license or, at your choice, those of the GNU FDL[3].

Wikisource is constantly looking for new members. During the transcription and proofreading of this book, it's possible that we made some errors. You can report them at this page[4].

The following users contributed to this book:

- Pi Delport
- Lithis
- Jesscmcmxc
- Mpaa
- Billinghurst
- ShakespeareFan00

- Hesperian
- CalendulaAsteraceae
- Kwj2772
- Santoposmoderno

---

1. ↑ https://en.wikisource.org
2. ↑ https://www.creativecommons.org/licenses/by-sa/3.0
3. ↑ https://www.gnu.org/copyleft/fdl.html
4. ↑ https://en.wikisource.org/wiki/Wikisource:Scriptorium