The History, Controversy, and Evolution of the Goto statement

Andru Luvisi

Sonoma State University

ABSTRACT

This talk will touch on how standard usage patterns for the Goto statement became embodied in higher level control structures, ways in which higher level control structures can often express the intent of the programmer more clearly than lower level control structures, some of the controversy surrounding the Goto statement, common reasons for and ways of using Goto, and common ways of implementing arbitrary control structures in languages that do not contain the Goto statement. The latest version of this handout is available from http://www.sonoma.edu/users/l/luvisi/goto/

1. History

1.1. The Stored Program Computer

The main predecessors of the stored program computer were the Mark I and the Eniac. Both were used primarily for running a calculation multiple times, and both lacked anything which we would recognize today as general control capabilities.

The Mark I was controlled by instructions read from paper tape, but it had very limited control capabilities. There was an instruction for transferring control from one tape reader to another, which was used for implementing a sort of subroutine facility.

The Eniac was programmed by hooking its various components up to each other in different configurations. It was, in theory, possible to have decisions about the flow of control within the machine be based on the value of a piece of data, but none of our surviving program diagrams from the early days show this being done.

The earliest publications on the stored program computer, even those which predate the first implementations[1, 2] include jumps and conditional jumps in the proposed instruction sets.

The Edsac[3], inspired heavily by the *First Draft*, was probably the

earliest computer that a modern programmer would recognize. Shortly after becoming operational, it had been used in all sorts of unanticipated ways, including games such as Tic Tac Toe which used a debugging display as a crude bitmap.

Comparing the stored program computer to its most immediate predecessors, the big difference, the final feature which turned the computer from a rapid automatic calculator into a fully universal Turing machine, was the conditional jump instruction.

1.2. Fortran

Fortran[1] introduced the automatic conversion of mathematical formulas into assembly and machine code, but kept the unconditional and conditional jumps of machine code, naming them "goto" and "if." Fortran also added the "computed goto" which offered the programmer the ability to jump to one of many targets based on the value of a variable, offering the programmer something similar to the jump tables of machine code.

1.3. Algol 60

Algol 60 [4] kept the goto statement and added some truly psychotic variations. Jump targets could be either numbers or identifiers. It also added a powerful "for loop" capabiility and blocks. Blocks allowed multiple statements to be grouped together and treated as a single statement, and made it possible for any number of statements to be conditionally executed by an "if" statement without using goto. The computed goto of Fortran was replaced by the switch, which had a syntax resembling an array of labels, and bears no resemblance to the more familiar switch statement from C.

1.4. Pascal

Pascal kept the goto statement, but only allowed the programmer to use numbers as labels. The Algol switch was replaced by the case statement[5] . The case statement would execute one of many chunks of code based on the value of a variable.

1.5. C

The C switch statement (based on the BCPL[6] SWITCHON statement) had the now familiar fall-through.

2. Evolution

[1] See http://www.softwarepreservation.org/projects/FORTRAN

Over time, people noticed that conditional jumps were often being used over and over in the same ways. Abbreviations and higher level syntax were developed for some of these usage patterns.

2.1. Jump Tables

The jump table of assembly gave way to the computed goto of Fortran

```
ON I GOTO 110, 120, 130, 140
```

which gave way to Algol switch

```
switch s = label1, label2, label3;
...
goto s[i];
```

which gave way to Pascal case

```
case i of
    1: ...
    2: ...
    3: ...
end
```

which gave way to the C switch.

```
switch(i) {
    case 1: ...
    case 2: ...
    case 3: ...
}
```

2.2. Conditional Jump for skipping stuff

The conditional jump used for skipping stuff

```
IF ... THEN GOTO 50
...
50 ...
```

turned into an if statement with a compound body.

```
if (...) {
    ...
}
```

2.3. Conditional Jump for looping

The conditional jump for looping turned into the now ubiquitous loop syntaxes such as for, while, until, and do/while loops.

2.4. Jumping to an error handler

The jump to an error handler

```
ON ERROR GOTO 9999
```

evolved into modern exceptions.

```
try {
    ...
    throw(...);
    ...
} catch(...) {
    ...
}
```

2.5. Jumping out of loops

The conditional jump for escaping out of a loop

```
FOR I = 1 to 10
    ...
    IF ... THEN GOTO 10
    ...
NEXT I
10 ...
```

evolved into break statements and labeled break statements.

```
for(i = 0; i < 10; i++) {
    if(...) break;
}

LOOP:
for(i = 0; i < 10; i++) {
    if(...) break LOOP;
}
```

3. The Controversy

In 1963 Peter Naur wrote an article[7] where he pointed out that many of the gotos in people's Algol 60 code were actually for loops in disguise or ifs with a compound body in disguise. His basic argument was that if a piece of code is doing something which the language has a builtin syntax for, then it will be easier for a reader to understand the code if it uses that builtin syntax. For example, upon seeing a for loop with a

compound body, a reader will immediately know that the code is a loop, what code is in the loop, and what the start and end conditions for the loop are. If the same code is written using gotos, then the reader must figure this all out.

In 1968, Dijkstra wrote his now famous letter to the editor[8] where he called for abolishing all use of the goto statement. He also famously argued[9] that all programs should be constructed from loops, conditionals, and sequential execution, with every piece of code (loop, conditional, function, etc.) having only one entry point and only one exit point.

Donald Knuth wrote a wonderful response[10], from which I have taken some of the following ideas. If you read only one of my references, Knuth's *Structured Programming with Goto Statements* is the one that you should read. It's also interesting to note that Knuth speaks of communications with Dijkstra where Dijkstra conceded that some of Knuth's gotos were appropriate.

Here are a few of my favorite quotes from Knuth's article.

> "The real goal is to formulate our programs in such a way that they are easily understood."
>
> "[W]e shouldn't merely remove go to statements because it's the fashionable thing to do; the presence or absence of go to statements is not really the issue. The underlying structure of the program is what counts, and we want only to avoid usages which somehow clutter up the program."
>
> "[U]ndisciplined go to statements make program structure harder to perceive, and they are often symptoms of a poor conceptual formulation. But there has been far too much emphasis on go to elimination instead of on the really important issues; people have a natural tendency to set up an easily understood quantitative goal like the abolition of jumps, instead of working directly for a qualitative goal like good program structure."

4. Common Uses

4.1. The read/process loop and the process/read loop

Some problems are most naturally solved with a loop that is run n and a half times.

Soloway, Bonar, and Ehrlich[11] ran an experiment where they gave two groups of students the same problem. One group was given the equivalent of a break statement, and one was not. 24% of the novices who had a break statement were able to write correct programs, while only 14% of those without a break statement succeeded. For intermediates, the

percentages were 61% and 36%. For advanced students, 96% and 69%.

The problem was this:

> Write a Pascal program which reads in a series of integers, and which computes the average of these numbers. The program should stop reading integers when it has read the number 99999. NOTE: the final 99999 should NOT be reflected in the average you calculate.

I'm going to write my examples in C since I don't happen to have a Pascal compiler with their special additions handy.

One way to solve this sort of problem is with a loop that reads and then processes each piece of data.

```
#include <stdio.h>

int main(int argc, char *argv[], char *envp[]) {
    int count = 0;
    int total = 0;
    int current;
    for(;;) {
        scanf("%d", &current);
        if(current == 99999) {
            break;
        } else {
            total = total + current;
            count = count + 1;
        }
    }
    if(count > 0)
        printf("%d\n", total/count);
    else
        printf("No numbers input.  average undefined\n");
    return 0;
}
```

One way to avoid the break statement is to change fron a read/process loop to a process/read loop.

```
    scanf("%d", &current);
    while(current != 99999) {
        total = total + current;
        count = count + 1;
        scanf("%d", &current);
    }
```

This has some disadvantages.

The read code is duplicated. This creates the danger that if the read code needs to be changed later, the programmer may forget to change it in both places.

The process and read statements that are executed during a single pass through the loop don't really have anything to do with each other. Item N is being processed, but item N+1 is being read for use during the next pass through the loop. This means that a single pass through the loop cannot be understood by itself, as "These are the statements to read and process item N" but must be understood as it relates to the previous and next executions of the loop.

It is possible to avoid using a break statement while still reading and writing the same item during a single pass through the loop, but at the expense of either a flag or a duplicated test.

```
int done = 0;
while(!done) {
    scanf("%d", &current);
    if(current == 99999)
        done = 1;
    else {
        total = total + current;
        count = count + 1;
    }
}
```

```
while(current != 99999) {
    scanf("%d", &current);
    if(current != 99999) {
        total = total + current;
        count = count + 1;
    }
}
```

Both of these approaches contain two tests each time through the loop. One has a flag, whose only purpose is to simulate a break statement and get control to the end of the loop, but in a more difficult to understand manner than with a break statement. The other duplicates the code which tests for the end of the input data, which poses the already mentioned dangers to code maintenance. Both approaches are also unnecessarily inefficient, performing two tests each time through the loop when one would suffice. This may not be a problem in most problem domains, but it is in some, and it is always ugly. Whether or not it more ugly than an exit from the middle of the loop is, of course, a matter of opinion.

It is also worth noting that the original read/process loop above still contains only one entry and only one exit. The exit just happens to be in the middle of the loop. To construct a formal proof, those predicates which are known to be true at the point of the break statement can

be assumed immediately after the end of the loop.

Roberts[12] points out some other very compelling evidence for the naturalness of the read/process loop over the process/read loop. Most authors of Pascal textbooks describe data processing problems first in terms of a read/process loop which they then convert into the process/read loop!

This issue is sometimes handled by performing the read operation within the condition of the loop. Most C programmers are familiar with the idiom

```
while((ch = getchar()) != EOF) {
    ...
}
```

and Perl programmers will recognize

```
while(<>) {
    ...
}
```

but this is just a tricky way of writing what is essentially the first read/process loop above. As the methods for reading a record and testing for termination get longer, this approach leads to horribly tortured code. Furthermore, it confuses expressions and statements. Performing side effects within expressions creates its own cognitive difficulties for the reader of the code.

Trying to solve the averaging problem this way, we get something like

```
while(scanf("%d", &current), current != 99999) {
    total = total + current;
    count = count + 1;
}
```

All in all, I think the read/process loop with an exit in the middle is the easiest to write, the easiest to understand, and the easiest to prove correct.

4.2. Removing recursion

Some algorithms are most naturally expressed recursively. Sometimes we want a non-recursive implementation of one of these algorithms. Maybe we are on a platform where function calls are slow, the recursive version would perform many function calls, and we need the speed. Maybe our function call stack isn't deep enough to handle the recursive version. Judicious use of the goto statement can allow us to remove recursion while still keeping the general structure of the recursive version.

For example, consider a postorder traversal of a tree.

```
void postprint(struct node *np) {
    if(np != NULL) {
        postprint(np->left);
        postprint(np->right);
        printf("%s\n", np->s);
    }
}
```

This is a fairly straightforward implementation. It is easy to prove correct with induction on the height of the tree. In fact, it is almost correct by definition. It traverses the left subtree, then the right subtree, and finally the current node.

Removing recursion by using gotos is a basically mechanical process. Each recursive call saves some information on a stack, reassigns the arguments (in a call by value language), and jumps to the beginning of the function. At the end of the function, we test to see whether the stack is empty. If not, we do a "fake return" by popping some information off of the stack, and jumping back to the appropriate place. If the stack is empty, we just return.

```
void postprint(struct node *np) {
    int flag;
    start:
    if(np != NULL) {
        /* fake recursive call postprint(np->left); */
        push(np, 0);
        np = np->left;
        goto start;
        after_first:

        /* fake recursive call postprint(np->right); */
        push(np, 1);
        np = np->right;
        goto start;
        after_second:

        printf("%s\n", np->s);
    }
    if(stack_empty()) {
        return;
    } else {
        /* fake return */
        pop(&np, &flag);
        if(flag == 0) goto after_first;
        if(flag == 1) goto after_second;
    }
}
```

This version is much longer than the recursive version, but the translation is mechanical, and by squinting a little bit we can still see the

basic structure of the recursive version hiding in this code, and convince ourselves that it still does what we want it to.

With some additional effort, it is possible to remove the gotos, but the result no longer bears any resemblance to the recursive version.

```
void postprint(struct node *np) {
    int flag;

    while(np != NULL) {
        push(np, 0);
        np = np->left;
    }

    while(!stack_empty()) {
        pop(&np, &flag);
        if(flag == 0) {
            push(np, 1);
            np = np->right;
            while(np != NULL) {
                push(np, 0);
                np = np->left;
            }
        } else if(flag == 1) {
            printf("%s\n", np->s);
        }
    }
}
```

This version uses no gotos, and is a little bit shorter than the version with goto, but it bears no structural resemblance to the recursive version. Indeed, the only reasons why I have any confidence that it will behave correctly are the fact that I derived this through meaning preserving transformations from the version using goto, and the fact that I have tested it!

Also note the duplication of the while loop that chases down left pointers.

When manually removing recursion, the programmer is basically pretending to be a compiler, by manually creating the machinery necessary to perform recursive function calls. When pretending to be a compiler, maybe it is appropriate to use the tools of a compiler, such as jump statements.

4.3. The Twelve Datums of Christmas

Sometimes it is necessary to allocate multiple resources in a single function call. If, while these resources are being allocated, one of them cannot be allocated, then any resources that have been successfully allocated up to that point must be deallocated.

This often looks something like this:

```
if((partridge = make_partridge()) == NULL)
    return ERROR;
if((doves = make_doves()) == NULL) {
    free_partridge(partridge);
    return ERROR;
}
if((hens = make_hens()) == NULL) {
    free_doves(doves);
    free_partridge(partridge);
    return ERROR;
}
...
```

To allocate 12 items, you end up with 66 deallocation statements, 55 of which are duplicates. The duplicates can be avoided with a bunch of nested if statements, but then we're likely to use a return that's not at the end of the function (a hidden goto) and we'll end up with heavily nested blocks, which tend to be confusing in their own rite.

If we are willing to use goto, it is possible to make both the allocation and the error handling/deallocation into fairly straight line code.

```
if((partridge = make_partridge()) == NULL)         goto partridge_error;
if((doves = make_doves()) == NULL)                 goto doves_error;
if((hens = make_hens()) == NULL)                   goto hens_error;
if((calling_birds = make_calling_birds()) == NULL) goto calling_birds_error;
if((rings = make_rings()) == NULL)                 goto rings_error;
...
return ...; /* successful return */

/* error handling code */
...
rings_error:         free_calling_birds(calling_birds);
calling_birds_error: free_hens(hens);
hens_error:          free_doves(doves);
doves_error:         free_partridge(partridge);
partridge_error:     return ERROR;
```

4.4. Retry

Sometimes you will find a goto used to retry a piece of code in an exceptional situation[13]. In such cases, the label is usually called something like "again" or "retry." Although most such cases can be written as a loop, using a goto indicates to the reader that the code is normally expected to execute only once, and repeating it is an unusual occurrence. Loop syntax suggests to the reader that the enclosed code is intended to run multiple times.

4.5. Efficiency

Martin Hopkins has pointed out[14] that sometimes the goto statement can offer a middle ground between staying at a high level and dropping into assembly, allowing you to hand optimize some code in a higher level language without having to work in assembly and give up portability. Linus has done this in Linux kernel code[2], and W. Richard Stevens has done this in networking kernel code[3] from time to time.

Knuth has proved[4] that without multi-level break statements, some gotos cannot be eliminated without losing efficiency.

4.6. Other control structures which your language does not supply

Sometimes a problem is most naturally solved using a control structure which is not present in the language being used. In these situations, a programmer must find a way to make do with those control structures which are available, perhaps simulating the desired control structure using an existing control structure.

When faced with this situation, sometimes the goto statement is the easiest base upon which to build.

4.7. When compiling from one language into another

Sometimes, when translating from one programming language into another, it is easier to generate code that uses goto in the target language rather than trying to map source language control structures into higher level control structures in the target language.

4.8. Other

There are some other uses of goto which I don’t have time to go into right now.

- Tail Recursion Optimization
- Coroutines
- Implementing algorithms that are already expressed as an arbitrary flowchart, such as those in *The Art of Computer Programming*[15]
- Breaking out of nested loops in languages without a multi-level break (like C).

---

[2] See http://kerneltrap.org/node/553/2131

[3] See http://www.kohala.com/start/rstevensfaq.html

[4] See "Structured Programming with Goto Statements"

5. Goto replacements

Sometimes, a programmer needs to implement arbitrary flow control, but the programmer is using a language which does not contain a goto statement. Here are some possible approaches to solving this problem.

For these explanations, I will use a simple Basic program from a book on programming in Basic[16].

```
 10 PRINT "INTEGERS"
 20 LET N = 1
 30 PRINT N
 40 LET N = N + 1
 50 IF N <= 5 THEN 30
999 END
```

I will show a few ways to translate this program into one that does not use goto while retaining the "flat" nature of its control flow.

5.1. A Goto by any other name

Some languages have a jump statement but it goes by another name. Common Lisp and Logo have "go." Other names to look for include "jump," "branch," and "transfer."

5.2. Tail call optimization

Tail calls are essentially gotos with the ability to send parameters. Here is a possible, and fairly straightforward translation of the above program into Scheme[5], such as might be created by a very naive compiler.

```
(define n #f)
(define (line-10)  (display "Integers") (newline) (line-20))
(define (line-20)  (set! n 1) (line-30))
(define (line-30)  (display n) (newline) (line-40))
(define (line-40)  (set! n (+ n 1)) (line-50))
(define (line-50)  (if (<= n 5) (line-30) (line-999)))
(define (line-999) #f)
(line-10)
```

[5] See http://www.schemers.org/

Of course, if a programmer were writing this by hand, the programmer would probably combine some of these procedures.

```
(define n #f)
(define (line-10)
    (display "Integers") (newline)
    (set! n 1)
    (line-30))
(define (line-30)
    (display n) (newline)
    (set! n (+ n 1))
    (if (<= n 5) (line-30) (line-999)))
(define (line-999) #f)
(line-10)
```

As an apology for doing such a horrible thing to such a beautiful language, I feel that I must include the following option, even though it doesn't really have anything to do with this talk.

```
(define (print-ints low high)
    (if (> low high)
        #f
        (begin
            (display low) (newline)
            (print-ints (+ low 1) high))))
(display "Integers") (newline)
(print-ints 1 5)
```

And this one.

```
(display "Integers") (newline)
(do ((i 1 (+ i 1)))
    ((> i 5) #f)
    (display i) (newline))
```

5.3. Trampoline

If a language has some way to represent a piece of code (usually a procedure or function) as a piece of data which can be stored in a variable or returned from a procedure, for the price of a little bit of speed, a trampoline[17] can be used.

In C it might look like this.

```
#include <stdio.h>
int n;

void (*curr_line)();
void line_10();
void line_20();
void line_30();
void line_40();
void line_50();
void line_999();

void line_10()  { printf("Integers\n"); curr_line = line_20; }
void line_20()  { n = 1; curr_line = line_30; }
void line_30()  { printf("%d\n", n); curr_line = line_40; }
void line_40()  { n = n + 1; curr_line = line_50; }
void line_50()  { if(n <= 5) curr_line = line_30; else curr_line = line_999; }
void line_999() { curr_line = NULL; }

int main(int argc, char *argv[], char *envp[]) {
    curr_line = line_10;
    while(curr_line != NULL) {
        (*curr_line)();
    }
    return 0;
}
```

This same basic idea will work in any language which has something resembling function pointers. JavaScript's closures or Forth's execution tokens or anything similar will work.

Tarditi, Lee and Acharya report[18] that using this approach (with a few other optimizations) their C backend for the SML/NJ compiler generated code that was 70% to 100% slower than the native assembly language backend.

Guy Steele used a similar idea in his Scheme compiler RABBIT[19].

5.4. Run and return successor

This object oriented design pattern is a variation of the State pattern from the Gang of Four[6][20]. It is the object oriented version of a trampoline.

[6] See also http://www.c2.com/cgi/wiki?RunAndReturnSuccessor

The main driver might look something like this.

```
while (r != ...) {
    r = r.run();
}
```

Create a class for each line and you're off.

5.5. Loop and switch

This approach has the advantage that it will work in any language with loops and conditionals. It is an existence proof that any program which can be written with gotos can be written without them. The primary disadvantage to this approach is that code can only jump to other code within the same compilation unit. In a language where the multiway branch does not fall through, every line would have to end with an assignment to the line variable.

```
#include <stdio.h>

int main(int argc, char *argv[], char *envp[]) {
    int line = 10;
    int running = 1;
    int n;

    while(running) {
        switch(line) {
            case  10: printf("Integers\n");
            case  20: n = 1;
            case  30: printf("%d\n", n);
            case  40: n = n + 1;
            case  50: if(n <= 5) line = 30 else line = 999; break;
            case 999: running = 0; break;
        }
    }
}
```

5.6. Continuations

Continuations[21, 22, 23] are a very powerful generalization of goto. With continuations, not only do you get to transfer control to any code you like, not only do you get to transmit data to the place you are sending control to, but you also get to replace the entire call stack. Continuations are so powerful that it is possible to implement goto in terms of them. I did this for Scheme using some of Peter Landin's ideas[7].

[7] See http://www.sonoma.edu/users/l/luvisi/scheme/prog.scm

Using my implementation of prog for scheme, a more or less direct translation might look like this.

```
(prog (n)
     line-10  (display "Integers") (newline)
     line-20  (set! n 1)
     line-30  (display n) (newline)
     line-40  (set! n (+ n 1))
     line-50  (if (<= n 5) (go line-30))
     line-999 (return #f))
```

6. Come From

No discussion of goto would be complete without a mention of Come From[24]. Consider it mentioned.

References

1. John Von Neumann, *First Draft of a Report on the EDVAC* (Jun 1945). http://en.wikipedia.org/wiki/First_Draft_of_a_Report_on_the_EDVAC.
2. Martin Campbell-Kelly (ed) and Michael R. Williams (ed), *The Moore School Lectures,* MIT Press, Cambridge, Massachusetts (1985).
3. Maurice V. Wilkes, David j. Wheeler, and Stanley Gill, *The Preparation of Programs for an Electronic Digital Computer (2nd ed),* Addison-Wesley (1957).
4. Peter Naur (ed), “Revised Report on the Algorithmic Language Algol 60,” *Communications of the ACM,* 3, 5, pp. 299-314 (May 1960). http://www.standardpascal.org/Algol60-RevisedReport.pdf.
5. O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming,* Academic Press Inc. (London) Ltd. (1972). http://portal.acm.org/citation.cfm?id=SERIES11430.1243380.
6. Martin Richards and Colin Whitby-Stevens, *BCPL - the language and its compiler,* p. 27, Cambridge University Press (1980).
7. Peter Naur, “Go To Statements and Good ALGOL Style” in *Computing: A Human Activity,* ACM Press (1992). http://portal.acm.org/citation.cfm?id=113602.
8. Edsger W. Dijkstra, “Go To Statement Considered Harmful,” *Communications of the ACM,* 11, 3, pp. 147-148 (Mar 1968). http://doi.acm.org/10.1145/362929.362947.
9. Edsger W. Dijkstra, *Notes on Structured Programming* (Apr 1970). http://www.cs.utexas.edu/~EWD/ewd02xx/EWD249.PDF.
10. Donald D. Knuth, “Structured Programming with Goto Statements,” *ACM Computing Surveys,* 6, 4, pp. 261-301 (Dec 1974). http://pplab.snu.ac.kr/courses/adv_pl05/papers/p261-knuth.pdf.
11. Elliot Soloway, Jeffrey Bonar, and Kate Ehrlich, “Cognitive Strategies and Looping Constructs: An Empirical Study,” *Communications of the ACM,* 26, 11, pp. 853-860 (Nov 1983). http://citeseer.ist.psu.edu/555332.html.
12. Eric S. Roberts, “Loop Exits and Structured Programming,” *ACM SIGCSE Bulletin,* 27, 1, pp. 268-272 (Mar 1995).

http://doi.acm.org/10.1145/199691.199815.

13. Diomidis Spinellis, *Code Reading,* p. 44, Addison-Wesley (2003).
14. Martin E. Hopkins, “A Case for the Goto,” *Proceedings of the ACM annual conference,* 2, pp. 787-790 (1972). http://doi.acm.org/10.1145/800194.805860.
15. Donald E. Knuth, *The Art of Computer Programming,* Volumes 1-3.
16. George Ledin, Jr. in *A Structured Approach to General Basic,* p. 127, Boyd & Fraser Publishing Company, San Francisco, California (1978).
17. Steven E. Ganz, Daniel P. Friedman, and Mitchell Wand, “Trampolined Style,” *International Conference on Functional Programming* (1999). http://citeseer.ist.psu.edu/ganz99trampolined.html.
18. David Tarditi, Peter Lee, and Anurag Acharya, “No Assembly Required: Compiling Standard ML to C,” *ACM Letters on Programming Languages and Systems* (1990). http://citeseer.ist.psu.edu/62903.html.
19. Guy Lewis Steele, Jr., “RABBIT: A Compiler for SCHEME,” AITR-474, MIT (May 1978). ftp://publications.ai.mit.edu/ai-publications/pdf/AITR-474.pdf.
20. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison-Wesley (1995).
21. John C. Reynolds, “Definitional interpreters for higher-order programming languages,” *Proceedings of the ACM annual conference,* 2, pp. 717-740 (1972). http://doi.acm.org/10.1145/800194.805852.
22. P. J. Landin, “Correspondence between ALGOL 60 and Church’s Lambda-notation: part I,” *Communications of the ACM,* 8, 2, pp. 89-101 (Feb 1965). http://doi.acm.org/10.1145/363744.363749.
23. P. J. Landin, “Correspondence between ALGOL 60 and Church’s Lambda-notation: part II,” *Communications of the ACM,* 8, 3 (Mar 1965). http://doi.acm.org/10.1145/363791.363804.
24. R. Lawrence Clark, “A Linguistic Contribution to GOTO-Less Programming,” *Communications of the ACM,* 27, 4, pp. 349-350 (Apr 1984). http://doi.acm.org/10.1145/358027.358043.