The Role of the Study of Programming Languages in the Education of a Programmer

Daniel P. Friedman
Indiana University

I want to thank the Universidad Nacional Autônoma de México and especially Dr. Luis Alberto Pineda Cortés, Jefe del Departamento de Ciencias de la Computación for inviting me to speak at your seminar.

The role of the study of programming languages is two-fold. First, the study of programming languages should teach you how to avoid bad ideas. I demonstrate these with a little history of dynamic scope, some small discussion of types, and a disaster in the making concerning proper implementation of tail calls. Second, the study of programming languages should teach you how to embrace good ideas in a representationally-independent fashion. I show you how that is done by resolving the issue of tail calls until something better comes along. These issues require some small amount of formalism, but for the most part are intuitive and only require that the person who studies programming languages do so with an open mind. Of course, there are other roles, but they become clearer from the three long quotes from my former graduate students.

Jon Rossie describes the life of a programming languages expert in a world that does not know what one is and does not understand what one does. His knowledge is first-hand.

> When a significant software project requires a database, database specialists are called in; when networking solutions are needed, networking specialists are called in; when special demands are placed on the operating system, operating systems specialists are called in.

But the one thing on which every significant project makes special demands is programming languages. Every *successful* significant project I have seen has called in a languages specialist to address these special needs.

Programming languages are the box outside of which most programmers, architects, and managers cannot think. Languages, for them, are things you choose, not things you build. Where language specialists see simple solutions, the untrained see costly problems that will likely remain unaddressed. Where the untrained see "policies" and "standards," the languages specialist sees detection and enforcement. Problems that cut menacingly across the project are our special domain. Done well, the language expert's work becomes the unnoticed context in which other work is done. We put power back in the hands of architects by letting them build needed assumptions into the language and the development tools. We reduce entropy and error in software systems.

I work as a software systems analyst with two other languages people. Our group's technical expertise in languages has repeatedly demonstrated its value. Our system analyses have often revealed problems that are best addressed by the formal manipulation of programs. This kind of solution is so foreign to project managers that we are often forced to defend the practice. Several experienced and bright software project managers responded to our offerings by accusing us of claiming to solve the Halting Problem. They were not unreasonable people, just under-educated. Their long experience with programs was that they cannot usefully be treated as data. Our long experience is just the opposite. When we explained that our approach was essentially a variation on the front end of a compiler, they slowly warmed to it, but it was clearly with skepticism. Then we proved our ideas with prototype implementations, which we produced quickly and accurately because they were only minor variations on the well-learned lessons of a good training in programming languages.

It's a simple insight: "Programs are data." This is never more evident than with software porting projects, but we have seen major porting projects undertaken by managers who lack this insight. By luck, we found them and were able to offer our expertise, but they would never have thought to look for us.

I wish, but I don't expect, that every programmer and architect could see the solutions we see because of our training in languages. It's as if all the world were color-blind, and I wish they could see the colors with us. But I do think I can expect every programmer to at least know that colors exist and that some people can see them. A good undergraduate course in programming languages is all they need. It's enough to let them see the possibilities and think outside the language box. Later, when they see a problem that cuts across the system in a way that languages are uniquely suited to solve, they might recognize it and go find a languages expert. Thinking like this has saved projects.

Jon says that all programs are data. I agree, but I think that all data are programs. Every program that we write in some sense is an interpreter. The argument is simple, the hardware is an interpreter. But, what does that really mean? Every program processes data; the processor of the data is an interpreter. Even a compiler is an interpreter. The data it processes is a program and the output that this interpreter processes is another program. A type checker (or inferencer) is also an interpreter. It interprets programs and produces types. A program that produces code in monadic style or continuation-passing style is an interpreter, too. If I write a program whose input is the numeral 5, that numeral must be interpreted as a number if I am thinking about it as a number. But, what if the input is "V?" That could be interpreted as a character, or perhaps it could be interpreted as the Roman Numeral for the number 5. We have no way of knowing. If I write a function whose input is "XIV" and its output is "XV," would you know that this is the "successor" function? If you did not know anything about Roman Numerals, wouldn't you guess this was the "predecessor" function, since its output is shorter than its input? The data must be interpreted.

Most programmers are comfortable with several programming languages and some are upset that some things are very difficult to do in their favorite language. The study of programming languages to some extent alleviates this problem. In today's world, learning a new language is one of the expected activities a programmer must do. There are certain language design issues such that identifying how a language approaches each of them can make learning that language relatively easy. One of our goals is to learn enough about these issues so that when the time comes to program in the latest language, it is usually no more trouble than identifying various characteristics, which we call “essentials,” and piecing that language’s essentials together.

I want to tell you a little bit about my background. When I was just starting out in computer science in the Spring of 1964, one of my goals as an undergraduate was to learn at least one new language per semester. This may seem tame compared to today’s standards, but you must understand that there were very few languages and access to these languages was through the equivalant of “man” pages. There were no texts, so if you did not know why the language was designed in the first place, it was nearly impossible to determine what the designer’s man-page writer meant. Often it was necessary to find the published paper and try to understand the “abstract” notions put forth in the paper and then try to correlate the “man” pages. I can tell you this was not easy, particularly because languages were not as well designed then. (As we observe later, today’s languages can still use some improvement.) When I went to graduate school, I chose to ratchet up my personal expectations a bit. Now, instead of understanding a language per semester, I wanted to be able to *implement* a language per semester. Later, I wanted to be able to implement a language per week.

This meant that I had to have much better tools than I had been using. About this time, Denotational Semantics was coming into popularity and we could see that from Denotational Semantics it was possible to obtain a high-level understanding of the essentials of a programming language, simply by abstracting away some of the tedious, inessential details.

Denotational Semantics studies languages, but does not implement them. We were not aware how to close the gap between Denotational Semantics and LISP, which was the best alternative at that time. They both had λ's, didn't they? But the ones in the LISP I was using were ill-conceived, since they did not close over free variables. It was in December of 1975, when Mitchell Wand brought home a satchel of papers from his alma mater, MIT, and plucked one out and said, "Dan, I think that you are going to like this."

Rarely have I heard such understatement. That paper (the very first Scheme paper) changed my research life overnight and it got the λ's right. The craziness of dreaming up strange variables (a direct consequence of dynamic scope) was about to be over. The ability to take the equations that existed in Denotational Semantics and use them in this new implementation of LISP solved all my problems at once. Just as I need a particular tool, it emerges. What luck!

But, I am digressing from the main topic, which is why you should care as a programmer about the study of programming languages. I find programming languages fascinating and thus have spent from 1964 to the present studying them; but that may not be enough of a reason for you to study them.

Anurag Mendhekar characterizes what he used from 311, my programming languages course, that led to success in his start-up company. (Warning this is a long letter.)

> I, like all good computer scientists, believe in the power of abstraction in software development. The irony to me, always, was that the tools that I used for programming (i.e., the programming languages), with a few exceptions, provided such poor tools for building abstractions, that the full power of abstraction could hardly be realized. You ended up breaking your abstractions in order to optimize your programs and hence needlessly complicating programs.

311-style programming opened my eyes to a whole new way of programming (which I still use)—build the abstractions you need. This, as opposed to forcing concepts from software architecture onto a small set of inadequate abstractions, ends up being a much better style of programming for complex applications, as I hope to demonstrate from my professional experience.

Based on this key idea about software development, I embarked on the task of building a software platform for Online Anywhere (now part of Yahoo!) that would enable the publication of internet content on devices other than desktop PCs; devices such as cell-phones, PDAs, and televisions.

I very quickly settled on a set of abstractions that I thought would be necessary. We then used Java and meta-programming tools around Java to implement these abstractions. The result of this effort was a system that very quickly helped us build transducers for transforming HTML content for non-PC devices. The abstractions technically formed an extension to Java, which we called Blue.

As we gained more experience building these transducers, we began to notice other abstractions that were even better to use in building certain kinds of transducers that were very frequently used. This led to the development of a language called Teal, which is used today to support a very large portion of Yahoo!'s non-pc internet traffic.

The other key challenge in building this software platform was to glue numerous transducers together in order to handle multiple different devices from the same platform. Guess what our solution was! We came up with another language, called Green, to specify how to do it.

The result of a combination of these languages is a very easy to maintain, highly efficient and extensible system that supports all of Yahoo!'s non-pc internet traffic. I attribute the elegant architecture of what's in there to key learnings in the construction and implementation of programming languages that I have gained from 311.

Obviously, not all programming languages are suitable for this style of programming. 311 also exposed me to Scheme, and from there to Common Lisp, both of which are excellent in supporting this style of programming. A fair number of programming languages (such as Java and C) have certain fundamental characteristics that will enable this style of programming, but require more work in order to build the abstractions that are needed.

The beauty of building abstractions as extensions to languages is that you have the added benefit of not violating your abstractions for the sake of efficiency—all you do is make the implementation of your abstractions smarter. This could be harder to do in some cases (especially if you haven't taken 311), but will overall lead to a much more uniform way of optimizing performance, and code that is much easier to maintain (because the optimizations now occupy a module of their own!)

Anurag's perspective on separating what you want to do with how you are going to do it is critical. Why should we burden with details those who are thinking abstractly about solving a problem. The details can be the implementation of the language design or the implementation of the language, itself. Details are for later, after the ideas are developed and prototyped. We see this view re-iterated by Jonathan Sobel in another guise a bit later.

Let's get back to the topic at hand. First, we must ask ourselves, "What do we mean by *the study* of any topic?" By *the study* we mean "The application of one's mental faculties to the acquisition of knowledge in a particular field or to a specific subject." How does one acquire such knowledge? One should not acquire such knowledge by memorizing a bunch of facts; looking at a bunch of instances, etc. One acquires such knowledge by developing a firm foundation of concepts upon which one can absorb knowledge. And, in my opinion, that knowledge should be learned by modelling the actual artifact that one is studying! That is how I have been teaching programming languages for a long time.

For example, knowing that a language passes its parameters by value tells me a lot more than a description that explains call-by-value. Since I consider the call-by-value λ-calculus as my model of call-by-value, I do not need to see any further description, but I do need to know what items in the language are values. As the set of values changes, so does my perception of the language. Both ML and C are purported to be call-by-value languages, but their values differ considerably.

And, we must ask ourselves, "What do we mean by *the study of programming languages*?" We mean a semi-formal analysis of the programming language concepts that have lasted well beyond a decade of their discovery. I don't just mean the concepts in languages, but I mean the collection of the concepts plus their underlying principles

Most texts teach the features of many languages and ask students to write programs in these languages, demonstrating an understanding of the syntax and some features. That is not what I mean. I want the student to be able to implement (perhaps crudely) every language that they study; and I certainly would not burden them with concerns about the syntax of these languages. I am not saying that syntax is unimportant, it just does not matter in the study of programming languages, as I see it.

For example, the concept of lexical scope has been around in programming languages since at least Algol 60, but of course, the logicians have used lexical scope for much longer. Their quantifiers, $\forall$ and $\exists$, relied on lexical scope. In fact, logicians had a hard time getting substitution right because they were not used to the subtleties of lexical scope.

Here are some naive relational database system operators, where a relation is a set of tuples.

```
(∀ x E Tuple*) == (andmap (λ (x) E) Tuple*)
```

and

```
(∃ x E Tuple*) == (ormap (λ (x) E) Tuple*)
```

Our goal here is not to capture the cleverness of the implementation of database systems, but just to allow us to run simple database programs Our goal is to get the abstraction completely understood. Look how powerful λ is; it simply captures the notion of binding for the $\forall$ and $\exists$ quantifiers.

Let's consider another scoping mechanism: dynamic scope. When you first study scoping, you cannot fail to learn about dynamic scope. Why? Well, there are two reasons. The first one is that you are likely to discover dynamic scope first, as many language designers did, and regrettably still do. The second reason is that you need to know why dynamic scope is a mistake. Before you study that, however, you need to know that the choice of the name of your bound variables should be up to you. In λ-calculus terms α-substitution should be supported. For example:

```
    (λ (x) x) = (λ (y) y)
```

Of course, you cannot always just change the `x` to `y`, but you can use any name that is not used in the expression. For example,

```
    (λ (y) (λ (x) (y x)))
    =
    (λ (z) (λ (x) (z x)))
```

We could argue that the study of programming languages must include at some level this notion of renaming. Anyone who programs subconsciously knows the α-rule, but it is essential that the rule be explained. Watch what happens to the α-rule as we work with `map`, below.

```
(define map
  (λ (f ls)
    (if (null? ls)
      '()
      (cons (f (car ls))
        (map f (cdr ls))))))
```

Clearly this is a correct program. We assume that `f`:$\alpha \rightarrow \beta$ and `ls` = list of α, so we know that the result of this computation must be of type list of β. Here is a use of map:

```
(let ((ls '(1 2 3 4)))
  (map (λ (x) (cons x ls)) ls))
> ((1 1 2 3 4)
   (2 1 2 3 4)
   (3 1 2 3 4)
   (4 1 2 3 4))
```

This is what we would expect, since Scheme is lexically scoped. What does this return if `let` and λ are dynamically scoped? Remarkably, it starts out the same.

```
((1 1 2 3 4) ...
```

But, on the second recursion, `ls` gets smaller, so it affects the `ls` inside the definition of `map`! Isn't that a surprise! Should any language designer be allowed to inflict such horrifying thoughts on a language user?

```
((1 1 2 3 4)
 (2 2 3 4)
 (3 3 4)
 (4 4))
```

Look at the defining equations, below. The first one is for dynamic binding and the second one is for static (or lexical) binding. When you are not comparing them, the static binding equation can be simpler.

E[(λ (x) M)]env = (λ (arg) (λ (env) E[M]env[x←arg]))
versus
= (λ (arg) (λ (en**w**) E[M]env[x←arg]))

It is no wonder that we have had some confusion. These equations differ by exactly one character. That character is used to distinguish two names: env and en**w**. I have made my point about one scope at least being better than another.

Here is something else to ponder. What is wrong with this Scheme program?

```
(if (= n 0)
  (+ n 5)
  (not (= (length ls) 4)))
```

Nothing. Should we be content with this state of affairs? How about this one?

```
((if (= n 0)
   5
   (λ (x) (+ x 7)))
 6)
```

Nothing. Should we be content with this state of affairs?

We should not be happy with either. Why? In the first case, we see that the value of the conditional is either a number or a boolean. In the second case, we see that the value of the conditional is either a number or a function. We should be uncomfortable thinking about applying 5 to a number. In a language without types, such as Scheme, this is certainly possible. In a language, like ML, it is not possible. But, ML still gives you enough rope to hang yourself. You can design a data type that says it holds either a number or a function and then this code can be rewritten with some tagging and untagging, but at least ML makes it possible. So, it is clear that we must study types on some level in order to be proficient in understanding languages. Languages such as ML and Haskell allow you to specify or infer types. Java, on the other hand, requires that the user specify types. That's an essential distinction.

We don't want to be bothered with design flaws that have been dropped into languages by well-meaning designers or implementors. Some examples of this are $\TeX$'s dynamically-scoped macros, LISP's dynamic-scope and shallow binding, and

Java's lack of support for tail calls. There are likely very well-intentioned reasons for these mistakes, but mistakes they are, nonetheless. We have talked about dynamic scope, and I could write volumes about problems with macros, so we'll focus on the implementation flaw in Java. Those of you who do not think it is a flaw should take heed that the people at Microsoft have assured the programming languages research community that its Intermediate Language for .Net will properly handle tail calls.

There are lots of good features in Java, so I don't want you to think that I am here just to criticize it. Java is being used all over the world, so it has at least some popularity. In the second edition of "Essentials," which has eight chapters, two of them are devoted to object-oriented concepts, so we feel that understanding them is essential. But, there is something wrong with Java that does not relate directly to object-oriented programming. Guy Steele, one of the two authors of the 1975 paper I referred to earlier and a co-author of 'The Java Language Specification" now works for SUN and he has communicated with me that he was promised back in 1997 that this flaw would be fixed. Here it is 2001 and there is still no resolution of this problem on the horizon.

But, if you study programming languages, you discover that although there is a problem, through a small number of correctness-preserving transformations, this problem can be averted. And by being aware of the problem, you can better appreciate the merits of languages that manage to avoid it.

We consider this kind of reasoning about one language in terms of another language to be very much in keeping with the essense of learning the essentials of a programming language.

Most implementations of Java (and C) do not handle tail calls properly. The problem is not just one of recursive calls, but method calls, in general, since in object-oriented programming they are one of the primary operations. What you must do in Java to make recursive programs work does not encourage the programmer to use recursion, which in principle is rampant in every method table. When you consider programs such as operating systems that are designed not to terminate, this issue of efficiency becomes an issue of correctness. But, with the tools I present today, Java can be okay even if you need to write recursive programs.

Let's for the moment assume that the Scheme code that I write can easily be converted to similarly-structured Java code. It can; we demonstrate this in Chapter 5 of "Essentials" and in "A Little Java, A Few Patterns."

For much of the talk, you will see correctness-preserving transformations and I will make every attempt to make it clear how they work, but I have made this lecture public, so that you can read it at your leisure. The most important thing I want you to take away from this talk is that these transformations are simple and logical, that you could use them when the opportunity presents itself, and that they solve an existing problem in language implementations that do not handle tail calls properly as in most implementations of Java and C.

Consider the sum program below.

```
(define sum
  (λ (n)
    (if (= n 0)
      0
      (+ n (sum (- n 1))))))
> (sum 1000000)
```

It is simple but contains the necessary attributes. It takes the non-negative integer n and returns the sum of the integers from 0 to n. This program has an embedded call, referred to as a *nontail* call.

Of course, you know that we could implement this particular program, using an accumulator so that it would contain no nontail calls. Such programs are said to be in *tail* form.

```
(define sum
  (λ (n acc)
    (if (= n 0)
      acc
      (sum (- n 1) (+ n acc)))))
> (sum 1000000 0)
```

I have chosen rather simple programs that we would likely not even bother to write, since these are just sums and there are nice theorems $(n^2 + n)/2$ about sums. I only want to show you the idea and this simple program suffices for our purposes.

If you were to run either of these programs in Java, I can almost guarantee that they will not produce the right answer. Why? The problem is that we are relying on the control stack on each procedure call and the control stack is not very deep. The presumption by the designers and implementors is that you are heavily into writing programs with while loops and assignment statements. But, some programs don't naturally take on that shape! The two Java programs will produce the same result: an exception will be thrown.

I say "almost" guarantee that they will not produce the right answer, because some Java implementation might work. Some Java implementation might produce an answer when others might fail. So much for portability, but let us not digress.

Let's review. We have written a program in Java that will not work. Our next goal will be to transform it so that it will.

I want to show you that the seeds of the right answer to this problem are sitting in the code ready to be sown. The first procedure is not in tail form, whereas the second one is. This allows us to rewrite the tail form one in *register* form.

```
(define n)
(define acc)
(define sum
  (λ ()
    (if (= n 0)
      acc
      {(← acc (+ n acc)) ;; (begin (set!
       (← n (- n 1))
       (sum)})))
> {(← acc 0)
   (← n 1000000)
   (sum)}
```

When we run this program in Java, it still blows up. But, we are getting closer to our goal. Intending to replace procedure calls with gotos, as a first step we removed all arguments from calls and passed the arguments through global variables (or registers), setting them before going to the procedure.

Now, we make a little change that will break our program by putting it into *suspended-goto* form.

```
(define sum
  (λ ()
    (if (= n 0)
      acc
      {(← acc (+ n acc))
       (← n (- n 1))
       (λ () (sum))}
> {(← acc 0)
   (← n 1000000)
   (λ () (sum))}
```

Instead of invoking the goto label, we freeze and return it. We observe, however, that $(\lambda\ ()\ (f)) = f$, a variant of the λ-calculus η-rule restricted to variables. This results in the *η-suspended-goto* form.

```
(define sum
  (λ ()
    (if (= n 0)
      acc
      {(← acc (+ n acc))
       (← n (- n 1))
       sum})))
> {(← acc 0)
   (← n 1000000)
   sum}
```

But, now we must invoke the label after it is returned. We do this in a while loop, which is built into Java and does not grow the stack. We use `false` as the return value instead of *acc* to force the termination of the while loop. Instead, the accumulator is dereferenced at the end of the computation. Labels count as true.

```
(define sum
  (λ ()
    (if (= n 0)
      false
      {(← acc (+ n acc))
       (← n (- n 1))
       sum})))
(define run
  (λ ()
    {(while (sum) 'no-op) acc}))
> {(← acc 0)
   (← n 1000000)
   (run)}
```

We introduce another register, which we call *action*, which allows us to avoid any reliance on the "return" facility of the language. We simply set the action at each return point. We place the value returned by sum in the *action* register, thus neither relying on the arguments being passed nor the value being returned, leading to *trampoline* form.

```
(define action)
(define sum
  (λ ()
    (if (= n 0)
      (← action false)
      {(← acc (+ n acc))
       (← n (- n 1))
       (← action sum)})))
(define run
  (λ ()
    {(while action (action)) acc}))
```

```
> {(← acc 0)
   (← n 1000000)
   (← action sum)
   (run)}
```
We can model the assignment to *action* in Java by creating an instance *o* of a class that contains the `sum` method and using it like this: (← *action* *o*). Modelling the setting of *action* to `false` can be accomplished in several obvious ways.

We are relying on fewer and fewer facets of the underlying runtime architecture. By so doing, we are able to guarantee that we will not be surprised by it.

We have a correct program that implements tail calls, so we no longer have the problem alluded to above. But, we should not have to live in the world of the designer of the language. We would have preferred that our language already handled this problem. The designer's mistakes should not become our nightmares.

For most programs, it is not always evident that it is possible to take the program as it is and rewrite it in tail form, but that is okay, because I am going to show how to do it, in general.

If you look carefully at the two programs for "sum," you will notice that the tail form version has the property that it should produce the same result as the nontail form version. However, the first two numbers "added" in the conventional sum program are 1 and 0, whereas the first two numbers "added" in the tail form version are *n* and 0. How is it that we were able to use the tail-form program? Clearly they are not the same. We took advantage of the fact that the order of "additions" in a sequence of numbers does not matter. That is, we used both the associative and commutative properties of addition. Not every problem is open to such analysis. In fact, most are not!

So, we need a general way of doing this. Furthermore, we need a way of doing this that does not rely on anything very powerful that sits in the host language, in our case, Java or C.

In Chapter 7 of "Essentials," we warm you up by showing that the interpreter of Chapter 3 can be put into tail form by just thinking hard about what is going on and following some guidelines. We also show you how to put the interpreter into register form. This enables any tail-form program to run through the interpreter to exhibit iterative control behavior. But, to be a useful tool in the programmer's toolbox, we would hope to be able to transform any program to one that can be written (and run) in some host language. Such an algorithm exists, and has existed since 1975, with the publication of the seminal paper by Gordon Plotkin, "Call-by-value, call-by-name, and the lambda calculus."

There is a problem with this paper: the result of applying the algorithm produces unbearably ugly code. It is so bad that papers have been published on how to remove so-called "administrative redexes." In the earlier edition of "Essentials," we took a new approach, which was to think harder about the nature of the process and less about the mathematics and we produced a transformation whose output was more readable. But we found that our algorithm and its explanation were complicated enough that most people ignored that chapter. This had to change in the later edition. Knowing how to put programs in tail form is such a fundamental tool.

But, fate lent a hand. My student, Matthias Felleisen, challenged his student, now my colleague Amr Sabry, to find a better way to characterize our algorithm. That effort succeeded admirably and now Chapter 8 of the later edition uses Amr's and Matthias's CPS Algorithm.

Now, let's transform the first sum into *preregister-tail* form without taking advantage of the aforementioned properties of addition. Once in that form, we can transform it to trampoline form, as above.

We say that the tail-form code resulting from this process has been written in continuation-passing style. This is an easy style to learn, but you may also use the algorithm that is given in "Essentials."

Let's take another look at the code:

```
(define sum
  (λ (n)
    (if (= n 0)
      0
      (+ n (sum (- n 1))))))
> (sum 1000000)
```

The first thing we do is take every λ-expression and add an additional argument representing what to do next. In our case, there is only one such expression, so we add a *cont* parameter to `sum` and apply it to the λ-expression's body. We pass in the identity function, *id*, in the outer and inner calls of `sum`.

Let's see where we are.

```
(define id (λ (acc) acc))
(define sum
  (λ (n cont)
    (cont
      (if (= n 0)
        0
        (+ n (sum (- n 1) id))))))
> (sum 1000000 id)
```

All the λ-expressions have been handled, but we have introduced a new λ-expression. Do we have to deal with it, too? No, since it is just our representation of continuations.

Let's try to push the continuation through the branches of the `if` expression. If the computation in the test part is *simple* (loosely "obviously terminates"), then we can push the *continuation* into both the consequent and the alternative of the conditional expression, like this:

```
(define sum
  (λ (n cont)
    (if (= n 0)
      (cont 0)
      (cont (+ n (sum (- n 1) id))))))
```

Next, we must deal with the last resulting expression, since it contains an embedded call `(sum (- n 1) id)`.

```
(define sum
  (λ (n cont)
    (if (= n 0)
      (cont 0)
      (sum (- n 1)
        (λ (acc)
          (cont (+ n acc)))))))
```

What we did is mostly routine. We made the nontail call into a tail call by creating a continuation that was responsible for doing what was left over. In our case, what was left over was to add *n* to the accumulator and invoke the continuation. Now the code is in tail form.

In order to prepare this code to be in register form, we must dereference every free variable used in the continuation. This is just a bit subtle. In our case, the one continuation with free variables is

```
(λ (acc)
  (cont (+ n acc)))
```
Its two free variables are `n` and *cont*, so we replace the entire expression by
```
(let ((n n) (cont cont))
  (λ (acc)
    (cont (+ n acc))))
```
leading to the preregister-tail form, below. From here we are free to follow the path to trampoline form.
```
(define sum
  (λ (n cont)
    (if (= n 0)
      (cont 0)
      (sum (- n 1)
        (let ((n n) (cont cont))
          (λ (acc)
            (cont (+ n acc))))))))
```
We might be in a language that does not directly support higher-order functions, such as

$(\lambda\ (acc)\ acc)$

or
```
(λ (acc)
  (cont (+ n acc)))
```
When this happens, we can still use continuation-passing style, but change the representation of continuations. When *you* have put something into continuation-passing style, *you know* where the continuations get invoked. We find all the calls of the form `(`*cont s*`)` and replace them with `(apply-cont` *cont s*`)`.

```
(define sum
  (λ (n cont)
    (if (= n 0)
      (apply-cont cont 0)
      (sum (- n 1)
        (let ((n n) (cont cont))
          (λ (acc)
            (apply-cont cont (+ n acc))))))))
(define apply-cont
  (λ (cont acc)
    (cont acc)))
```

We then replace all the λ's that represent continuations by datatype constructions. The datatypes contain the free variables of the λ-expressions. In "Essentials," we use an ML-style datatype facility that we highly recommend, but for the purpose of this talk, we use an empty list and "cons," since we only have two variants and one of them has zero free variables and the other one has two free variables. The first is the empty continuation and the second is the continuation that "adds n to the argument coming in and applies the prior continuation, *cont*" leading to the *representation-independent-preregister-tail* form.

```
(define id '())
(define sum
  (λ (n cont)
    (if (= n 0)
      (apply-cont cont 0)
      (sum (- n 1) (cons n cont)))))
```

```
(define apply-cont
  (λ (cont acc)
    (if (null? cont)
      acc
      (apply-cont (cdr cont) (+ (car cont) acc)))))
> (sum 1000000 id)
```

Now all calls are tails calls, the derefencing of the free variables happens automatically when `cons` is invoked, and we have no higher-order procedures. So, we can proceed following the same steps we used in the tail form definition of `sum` leading to trampoline form.

```
(define sum
  (λ ()
    (if (= n 0)
      {(← cont cont)
       (← acc 0)
       (← action apply-cont)}
      {(← cont (cons n cont))
       (← n (- n 1))
       (← action sum)})))
(define apply-cont
  (λ ()
    (if (null? cont)
      (← action false)
      {(← acc (+ (car cont) acc))
       (← cont (cdr cont))
       (← action apply-cont)})))
> {(← cont id)
   (← n 1000000)
   (← action sum)
   (run)}
```

We can model the `apply-cont` dispatch using an abstract method `apply-cont` by inheriting from the associated abstract class of continuation types in the style of Chapter 1 of "A Little Java, a Few Patterns." In this example, there will be exactly two subclasses, one for the continuation modeled by the empty list and one for the continuation modeled by `cons`.

Alternatively, we can go back to the version with continuations represented as procedures. If we do that, we can also treat those procedures as actions.

```
(define id
  (λ ()
    (← action false)))
(define sum
  (λ ()
    (if (= n 0)
      {(← acc 0)
       (← action cont)}
      {(← cont (let ((n n) (cont cont))
                  (λ ()
                    {(← acc (+ n acc))
                     (← action cont)})))
       (← n (- n 1))
       (← action sum)})))
```

So, we see that when we have a problem because of poor implementation technology, it is possible to get around this problem with good correctness-preserving transformations. Our approach to programming languages relies heavily on such transformations. You may not learn the exact attributes of every language with our approach, but you will learn to develop your own perspectives on how to implement things elegantly. When the implementation falls short of expectations, you will be in a position to live with its shortcomings.

Jonathan Sobel's comments, below, are as relevant today as they were when they were written in 1994. Interestingly, the derivation to trampoline form (work done with Steve Ganz, a current student) had its beginnings in 1999. (Last warning: this is even longer than the previous two quotes.)

> During my first term of graduate school at Indiana University, I had the good fortune to be taking the analysis of algorithms and programming languages courses at the same time. One of the assignments in the algorithms course was a major term project in which we were to implement and optimize the Fast Multiplication algorithm. Fast Multiplication is a recursive, divide-and-conquer algorithm for multiplying two numbers, especially large numbers: hundreds or thousands of digits. (It is also used at the hardware level, where the units are bits, instead of digits.)
>
> Part of our grade for this project was based on how fast the program ran. Everyone else started writing in C, immediately, and even some in assembly language. Of course, they all spent hours upon hours debugging as they tried to get their highly optimized programs running. Modify, re-compile, test; modify, re-compile, test; on and on....
>
> To everyone's amazement and surprise, I started writing in Scheme. I had only seen Scheme for the first time two months before, in my programming languages course, but its simplicity made it attractive. Even more importantly, I had discovered the joy of incre-

mental compilation: make a change or an addition without recompiling everything. I wonder how many hours I saved.... On the other hand, the speed of the program was the most important factor, and even though Chez Scheme (the Scheme implementation we use here at Indiana University) is amazingly fast, it can't quite keep up with C in most cases. So why choose Scheme?

Each call to **fast-multiply** produces three recursive calls to **fast-multiply** (or to a simple **multiply** routine, once you reach small enough numbers). What I noticed was that each of those three recursive calls had nearly the same control context. In a simple-minded implementation in C, that context would be saved and restored three times, being destroyed after the return of each call. I thought to myself: If only I had explicit control over the flow of my program, I could speed it up significantly by creating that context only once and using it three times, destroying it only after the return of the third recursive call. Function calls are so costly! But I would never want to attempt such a thing from scratch in C.

What I had been learning in my programming languages course, however, was that I really could manage my own control flow if I wanted. Furthermore, I could start with a simpler, more naive program and basically derive the sophisticated one by a series of correctness-preserving program transformations. This is where Scheme really won. Because of its extremely algorithmic—almost mathematical—nature, Scheme can be easily manipulated in a sort of algebraic style. One can follow a series of rewrite rules (just about blindly) to transform a program into another form with some desirable property. This was exactly what I needed.

I started by implementing the algorithm very directly, following the steps given in our analysis of algorithms textbook line by line. (I definitely did *not* spend much time debugging in this phase.) Then I converted the program to continuation-passing style. Next I made the continuations into explicit record structures, rather than Scheme procedures. Finally, I transformed all the procedures to pass information via registers, instead of calling each other with arguments. At this stage, a procedure call is merely a simple **goto**, quite a cheap operation. I completed the entire transformation in a few hours.

With the Scheme program in its final form, it was a simple matter to translate it into C. The C program contained no function calls whatsoever; I really did use **goto**. I had never had the courage to use a **goto** in C before; it was just too dangerous a tool. It was kind of fun to use them now, knowing that I was completely safe in doing so. That was the amazing part: I had produced a program that I could not have written, and in any case would not have wanted to write.

Of course, you might be asking yourself, as I was, "But does it run fast?" Speed was, after all, my main goal. The answer is a very resounding "Yes!" Out of a class of about 15 students, only one person beat me (and only barely), and he wrote significant portions of his program directly in assembly language. The next fastest after me took nearly twice as long to do the same work.

Now I know of several more transformations that I could have applied to my Scheme program before I translated it into C, which would have put mine in first place. A runtime profile of my program revealed that the majority of time was spent in the C routines **malloc** and **free**. I could have eliminated that heap usage by transforming my program into a form in which all data allocation and control management would have been completely stack-based, with an explicitly managed stack. I could have pushed onto the stack exactly that control information that I deemed necessary. There were places where I could have modified the existing stack record, rather than popping it off and creating a (similar) new one in its place. These transformations were also presented in my programming languages course. Unfortunately, at the time that I did my algorithms project, I did not yet grasp them well enough to use them.

What I learned from this experience was the importance of a structured, systematic approach to programming. I have found that it is better to write a simple, direct program to solve a problem, having become convinced through experience that radical structural transformations can come later; and I can depend on those transformations to preserve the semantics of my program. I have also learned that I can

> be free to focus on the nature of whatever problem I am trying to solve, rather than on how efficient my solution is. Efficiency comes from elegant solutions, not optimized programs. Optimization is just a few correctness-preserving transformations away.

Jonathan's perception is accurate. The study of programming languages yields general-purpose tools that allow you to do things that are too hard to do without them. His comment that "I had produced a program that I could not have written, and in any case would not have wanted to write." tells the whole story. Learn these tools that are standard fare for researchers in programming languages, and you will be able to do things that, for you, may well have been impossible.

I want to close today's talk with one final quote, a quote from Christopher Strachey.

> I always worked with programming languages because it seemed to me that until you could understand those, you really couldn't understand computers. Understanding them doesn't really mean only being able to use them. A lot of people can use them without understanding them.

For those who wish to run this code in Scheme instead of Java or C, here is the definition of `while`.

```
(define-syntax while
  (syntax-rules ()
    ((while exp stmts ...)
     (let loop ()
       (if exp (begin stmts ... (loop)))))))
```