

Notes on Compilation

CS536, Fall 2002

Kathi Fisler*

October 20, 2002

We've seen that interpreters implement languages: if someone gives you the specification of a language, you can write an interpreter to provide programming support for that language. The advantages of writing interpreters are that they are generally easy to prototype (once you understand the language!) and can take advantage of all of the features in the language you are writing the interpreter in. For example, you could write a Java compiler in Scheme using *let/cc* to implement exceptions, even though Java doesn't have *let/cc*. The main downside to interpreters is that they can be too slow on large programs. If execution speed becomes an issue for your language, it's time to write a compiler.

A compiler is a program that transforms a program in one language to a program in another. This is a broad definition; it admits compilers from a language to (maybe a subset of) itself, for example. In this lecture, we want our compiled programs to be written in a language with the constructs and expressive power of an assembly language. More specifically, our only control operators should be jumps, data should consist of only simple types (numbers and symbols, but not lists), and we should expect to use stacks, memory addresses (pointers), and registers in our code.

Your first instinct might be to toss away your interpreter and start writing your compiler from scratch. That wastes all the effort that you put into developing and debugging your interpreter though! Finding a way to *derive* your compiler code from your (working) interpreter code would be much less error prone. Fortunately, using CPS supports precisely this activity!

This class uses a series of small programs to demonstrate how we can compile programs starting from CPS. We'll compile tonight's programs by hand to build your intuition on how this process works. You could implement the techniques that we will perform manually in this lecture to compile programs. You could produce a compiler by implementing the techniques we discuss here. While we won't get all the way to actual assembly code tonight, we'll get to a form of programs that highly resembles assembly code; the final transformation to actual assembly would be reasonably straightforward.

Bogus Terminology Warning: You'll often hear people say "Language X is an *interpreted* language". That terminology is meaningless because nothing in the *language* (the syntax or semantics) requires it to be interpreted or compiled! The *context* in which you use the language may favor interpretation or compilation, but that is a separate issue. Keep these issues separate (or disavow that you took a languages course!).¹

Now, on with the show!

*Material adapted from Brown CS173 notes on compilation, fall 2001.

¹It is technically possible to define a language that cannot be compiled, but this very rarely happens.

Example 1: Compiling Factorial

Consider the old favorite factorial program:

```
(define (fact n)
  (if (zero? n)
      1
      (* n (fact (- n 1)))))
```

To create a compiled version of this program, we first convert it to CPS:

```
(define (fact/k n k)
  (if (zero? n)
      (k 1)
      (fact/k (- n 1)
               (lambda (val) (k (* n val))))))
```

This doesn't look much like assembly code: it passes closures as arguments to maintain contexts (assembly languages don't have closures). The contexts resemble stacks though: we add pending computation to them (when we build new contexts) and discharge pending computation when we get concrete values. To see this, consider the following program trace of *fact/k* – notice how the pending multiplications build up “stack-like” in the context.

```
(fact/k 3 (lambda (x) x))
= (fact/k 2 (lambda (val1)
              ((lambda (x) x)
               (* 3 val1))))
= (fact/k 1 (lambda (val2)
              ((lambda (val1)
                 ((lambda (x) x)
                  (* 3 val1)))
                (* 2 val2))))
= (fact/k 0 (lambda (val3)
              ((lambda (val2)
                 ((lambda (val1)
                    ((lambda (x) x)
                     (* 3 val1)))
                   (* 2 val2)))
                (* 1 val3))))
= ((lambda (val3)
     ((lambda (val2)
        ((lambda (val1)
           ((lambda (x) x)
            (* 3 val1)))
          (* 2 val2)))
      (* 1 val3)))
  1)
```

Let's make this stack explicit by rewriting our creation and use of contexts using stack operator names.

```
(define (fact/stack n stack)
  (if (zero? n)
      (Pop stack 1)
      (fact/stack (- n 1)
```

```
(Push stack (lambda (val) (Pop stack (* n val))))))
```

```
(define (Pop stack value)  
  (stack value))
```

```
(define (Push stack receiver)  
  (lambda (v) (receiver v))) ;; equivalent to just receiver
```

```
(define (EmptyStack value) value)
```

```
(define (fact n) (fact/stack n EmptyStack))
```

How did we transform the previous version into this one? We performed three steps:

- renamed the *k* parameter to *stack*
- replaced all calls to *k* with *Pop stack*
- wrapped a *(Push stack ...)* around all the **lambda** expressions passed as continuations

In the transformed code, our “stacks” are functions that pop when we send them values; in other words, a stack is a **lambda** (*v* ...) Note that we haven’t changed how our program manages contexts yet. We’ve only introduced abstract names for the operations we perform on contexts. Having introduced those names, though, we are now free to change how we implement Push and Pop.

Throughout this lecture, our goal is to move lower and lower down the abstraction hierarchy. In other words, we want to replace all high-level constructs with lower-level ones (this is what most compilers do). The current *fact* code uses lambdas in two ways: we represent the contexts (the stack contents) as lambdas, and we implement the stacks themselves as lambdas. Lists are a lower-level, and more intuitive, data structure for implementing stacks. Let’s first change our implementation of stacks. The stack will still hold lambdas, but Push and Pop will use list operations:

```
(define (fact/stack/list n stack)  
  (if (zero? n)  
      (Pop stack 1)  
      (fact/stack/list (- n 1)  
                        (Push stack (lambda (val) (Pop stack (* n val))))))
```

```
(define (Pop stack value)  
  ((first stack) value))
```

```
(define (Push stack receiver)  
  (cons receiver stack))
```

```
(define EmptyStack (cons (lambda (value) value) empty))
```

```
(define (fact n) (fact/stack/list n EmptyStack))
```

With this version, we’d expect the stack for *fact/stack/list* with input 3 to look like:

```
(list  
  (lambda (val3) (* val3 1))  
  (lambda (val2) (* val2 2))  
  (lambda (val1) (* val1 3))  
  (lambda (value) value))
```

Now the real fun begins! We need to develop a lower-level representation of the stack *contents* than functions. What do those functions do? Looking at the stack contents in the list above, once we get the final value (1), we run each function and send the value to the previous lambda on the stack. So all that really matters are the numbers (given those, *fact* just multiplies them up the stack to get the final answer). So we should be able to replace the functions with records that hold the numbers. Actually, we'll need two kinds of records: one for a stack content that holds another multiplicand, and one for the bottom of the stack:

```
(define-datatype stack-record StackRec?
  [stack-rec-mult (n number?)] ;; says "combine data with rest of stack using mult"
  [stack-rec-empty]) ;; this just signals bottom of stack
```

```
(define (fact/stack/rec n stack)
  (if (zero? n)
      (Pop stack 1)
      (fact/stack/rec (- n 1) (Push stack (stack-rec-mult n)))))
```

```
(define (Pop stack value)
  (let ([top-rec (first stack)])
    (cases stack-record top-rec
      [stack-rec-mult (n) (Pop (rest stack) (* n value))]
      [stack-rec-empty () value])))
```

```
(define (Push stack new-record)
  (cons new-record stack))
```

```
(define EmptyStack (cons (stack-rec-empty) empty))
```

```
(define (fact n) (fact/stack/rec n EmptyStack))
```

This *fact* program looks much lower-level than our original version: we're no longer using lambdas to capture any data or control, and we have an explicit notion of stacks in our programs. This program doesn't look syntactically like assembly code, but we're clearly making progress.

To recap, what steps did we perform to compile *fact*?

1. Convert the source program to CPS
2. Replace operations on contexts (k's) with stack operations
3. Replace functions on stack with records. We need a new record type for each new continuation created in the source program. The fields of the record store the environment of the original continuation (**lambda**).
4. Expand *Pop* to process the new records. The code executed in each case is just the body of the original **lambda** expression corresponding to that type of record. In that body, we replace uses of the parameter to the **lambda** with the *value* argument to *Pop*.

You could write programs to perform these transformations automatically, so we're still in the realm of what a real compiler could do. Before we refine our compiled program to resemble assembly more closely, let's try another example.

An Aside

Given the CPS version and the insight that only the numbers matter in the contexts for *fact*, you could also produce the following program, which matches the standard accumulator-style program taught in intro programming classes:

```
(define (fact/accum n stack)
  (if (zero? n)
      (Pop stack 1)
      (fact/accum (- n 1) (Push stack n))))

(define Pop *)
(define Push *)
(define EmptyStack 1)
(define (fact n) (fact/accum n EmptyStack))
```

Why didn't we use this version, instead of the stack records version? To develop this version, you need insight into the stack contents and you need to know that * is associative. Although we motivated the transformation to stack records by knowing what the program does, a compiler could perform that transformation without that knowledge. Our goal in this lecture is to stick to compiler-implementable transformations.

Example 2: Compiling Tree-Sum

Consider the following program for summing the numbers in a binary tree:

```
(define-datatype tree Tree?
  [empty-tree]
  [node (n number?)
        (left Tree?)
        (right Tree?)])

(define (tree-sum atree)
  (cases tree atree
    [empty-tree () 0]
    [node (n left right) (+ n
                            (tree-sum left)
                            (tree-sum right))]))
```

Let's produce a compiled version. We first convert this to CPS:

```
(define (tree-sum/k atree k)
  (cases tree atree
    [empty-tree () (k 0)]
    [node (n left right)
      (tree-sum/k left (lambda (lv)
                        (tree-sum/k right (lambda (rv)
                                          (k (+ n lv rv))))))]))

(define (tree-sum atree)
  (tree-sum/k atree (lambda (x) x)))
```

Next, eliminate the uses of lambdas by implementing the stack with lists and the contexts with records.

```
(define-datatype stack-record StackRecord?
  [rec-bottom]
  [rec-add-left (node-val number?)
                (right-tree Tree?)]
  [rec-add-right (node-val number?)
                 (left-value number?)])

(define (tree-sum/rec atree stack)
  (cases tree atree
    [empty-tree () (Pop stack 0)]
    [node (n left right)
         (tree-sum/rec left (Push stack (rec-add-left n right)))]))

(define (Pop stack value)
  (let ([top-rec (first stack)])
    (cases stack-record top-rec
      [rec-bottom () value]
      [rec-add-left (node-val right)
                    (tree-sum/rec right (Push (rest stack)
                                              (rec-add-right node-val value)))]
      [rec-add-right (node-val lv) (Pop (rest stack) (+ node-val lv value))])))

(define (Push stack record)
  (cons record stack))

(define EmptyStack (cons (rec-bottom) empty))

(define (tree-sum atree)
  (tree-sum/rec atree EmptyStack))
```

Compared to the *Pop* we wrote for *fact*, this *Pop* seems a lot more complicated. In fact, the recursive call to process the right tree happens inside of *Pop*, not inside the function. This is consistent with the steps we defined after transforming *fact* though: the recursive call to process the right tree is in the body of continuation for processing the left tree, and the body of the left continuation should become the computation to do in the *Pop* case.

Having done two examples, it's worth convincing ourselves that the stack records achieve the same role in the compiled program as the **lambdas** did in the CPSed version. The **lambda** expressions did two tasks for us: they delayed computation of the continuation body, and they captured the environment in the **lambda** to use when performing that later computation. We capture the environment in the stack records, and the computation is delayed until we pop each record off the stack. Once we push a record for the left subtree, that record isn't popped until the program has computed the sum of that whole subtree: thus, we continue the computation (invoke the continuation) only after we've finished processing the entire left subtree (if you don't see this, hand trace the stack contents on a small tree example). So our stack records do implement all of the features of the original **lambda** continuations.

Example 3: Compiling Filter-Positive

The *fact* example showed how to make stacks explicit. With *tree-sum* we saw how to extend the techniques of the first example to process tree-shaped data. The compiled programs in both cases feel more assembly-like, but they still rely on many features of Scheme, such as Scheme's records (datatypes) and lists, not to mention Scheme's stack for holding arguments. In the rest of this lecture, we'll eliminate these dependencies on Scheme as well.

Let's write a function that traverses a list of numbers and gathers the positive numbers into a new list.

```
(define (filter-pos L)
  (cond [(empty? L) empty]
        [(cons? L)
         (cond [(> (first L) 0)
                  (cons (first L) (filter-pos (rest L)))]
               [else (filter-pos (rest L))])]))
```

Convert this to CPS:

```
(define (filter-pos/k L k)
  (cond [(empty? L) (k empty)]
        [(cons? L)
         (cond [(> (first L) 0)
                  (filter-pos/k (rest L) (lambda (lst)
                                           (k (cons (first L) lst)))]
               [else (filter-pos/k (rest L) k)])]))
```

```
(define (filter-pos L)
  (filter-pos/k L (lambda (x) x)))
```

Look at the two recursive calls in the CPSed version: one augments the continuation (k), while the other simply passes k along because there is no new context to add. This second kind of call is known as a *tail call*. Intuitively, a tail call is a function call that doesn't have to return to a pending expression in its enclosing function body or expression. The CPS transformation gives us a more technically precise definition: *a tail call is a function call in which the continuation is not augmented in the CPSed version of the program*. In the original *filter-pos* program, the first recursive call was not a tail call, while the second one was.

Why should we care about tail calls? Since tail calls accumulate no context, there's no need to remember where to return to after the function call; in other words, a compiler can implement a tail call as a *jump* rather than as a subroutine call. As jumps are a lot less expensive than subroutine calls, special handling of tail calls can provide significant improvements in run-time efficiency of programs.

Side Note: Not all languages require optimized handling of tail calls. Scheme does (its required in the language standard). Other languages like Pascal don't. Thus, even though your program may contain function calls that look like tail calls, language implementations may not provide the corresponding optimization.

As usual, the next step is to make the stack explicit.

```
(define-datatype stack-rec StackRec?
  [bottom-rec]
  [cons-rec (first-num number?)])
```

```
(define (filter-pos/stack L stack)
  (cond [(empty? L) (Pop stack empty)]
        [(cons? L)
         (cond [(> (first L) 0)
                  (filter-pos/stack (rest L) (Push stack (cons-rec (first L))))]
               [else (filter-pos/stack (rest L) stack)])]))
```

```
(define (Pop stack value)
  (let ([top-rec (first stack)])
```

```
(cases stack-rec top-rec
  [bottom-rec () value]
  [cons-rec (first-val) (Pop (rest stack) (cons first-val
value))]))))
```

```
(define (Push stack rec) (cons rec stack))
(define EmptyStack (cons (bottom-rec) empty))
```

Note: the *first-num* in the *stack-rec* here is a bit of an optimization: we really should put the list *L* into the record, since *L*, not (*first L*) is in the environment of the continuation. We'll continue with the optimization in this code; just be aware that an implemented compiler would store *L* instead.

Something subtle has happened in our code: the non-tail calls are gone! If we were to CPS *filter-pos/stack*, we'd find that all of the calls to *filter-pos/stack* are now tail-calls, unlike in the original version. This means we've optimized our program so that we can implement all of the function calls as jumps.

You may be asking why we would CPS *filter-pos/stack* again – didn't we do that step already? Yes, we know that we CPSed *filter-pos*, but *filter-pos/stack* is just an ordinary Scheme program. You could run a CPS transformation algorithm on it as with any Scheme program. In other words, the suggestion to CPS this program only sounds odd because we know the origin of *filter-pos/stack*.

Misconception Warning: Most of us have learned that languages use the stack to process function calls. This view is not accurate for all languages! If your language requires tail call optimization and all of the function calls in your program are tail calls, then all of your function calls are jumps. What then do those languages use the stack for? In such cases, *the stack is used only to evaluate arguments* (recall how we evaluate expressions in postscript: push the operands; when you reach an operator, pop the operands and apply the operator; push the result back onto the stack). If you've taken a compilers class, recall that you often push return pointers onto the stack – this is for handling function calls. If you don't have to save a return pointer, you're using the stack for other purposes than function calls.

Handling Jumps

I've claimed that by converting all function calls to tail calls, we can simply implement our function calls as jumps. If you examine *filter-pos/stack* carefully though, you'll see that are function calls aren't quite jumps because they still have arguments. To honestly claim that our function calls are jumps, we have to eliminate the arguments.

Luckily, arguments are easy to eliminate. We'll define a set of registers, and just put the arguments into the registers. For this program, we'll need one register for the stack and one for the list of numbers.²

```
(define =reg1= 'dummy)
(define =stack= 'dummy)
```

Now, rewrite the rest of the code to use the registers. When we make a tail call, we'll mutate the registers instead of passing arguments explicitly.

```
(define-datatype stack-rec StackRec?
  [bottom-rec]
  [cons-rec (first-num number?)])
```

²Clearly, we're relying on our knowledge of the program here to know how many registers to define. In reality, a machine has a fixed number of registers and compilers perform a step called *register allocation* to map program data to registers.

(define =reg l= 'dummy) ;; will hold the L parameter
(define =stack= 'dummy) ;; will hold the stack parameter

```
(define (filter-pos/reg)
  (cond [(empty? =reg l=) (Pop =stack= empty)]
        [(cons? =reg l=)
         (cond [(> (first =reg l=) 0)
                  (begin
                     (set! =stack= (Push =stack= (cons-rec (first =reg l=))))
                     (set! =reg l= (rest =reg l=))
                     (filter-pos/reg))]
                [else
                 (begin
                    (set! =reg l= (rest =reg l=))
                    (filter-pos/reg))]]]))
```

```
(define (Pop stack value)
  (let ([top-rec (first stack)])
    (cases stack-rec top-rec
            [bottom-rec () value]
            [cons-rec (first-val) (Pop (rest stack) (cons first-val value))])))
```

```
(define (Push stack rec) (cons rec stack))
```

```
(define (filter-pos L)
  (begin
    (set! =reg l= L)
    (set! =stack= (cons (bottom-rec) empty))
    (filter-pos/reg)))
```

A few things to note here:

- Notice how the calls to *filter-pos/reg* are now truly jumps – the arguments are gone (they were already tail calls).
- Having a helper function for the original interface of *filter-pos* is very helpful now, as we have to remember to set the register contents before starting the program.
- We didn't rewrite *Pop* to use registers – why not? *Pop* is part of our *implementation* of stacks (i.e., the run-time system), it is not part of the program that we are trying to compile. Up to now, we are still using Scheme lists to implement the machine stack. But read on ...

Question: Is this approach safe (as in, will it preserve the correctness of our original code)? We've explicitly avoided mutation in this course because we can't undo assignments (unlike when we pass arguments recursively). Why isn't the mutation a problem here? (Remember, these techniques should allow you to compile any program – they are not specific to *filter-pos*).

Representing the Stack as a Vector

Actual machines don't implement stacks with lists; they use memory, which is just an array. In Scheme, arrays are called *vectors*. To reduce our reliance on Scheme lists, we will now implement the stack using vectors.

We will need the following Scheme vector primitives:

- *(make-vector n)* creates a vector of length *n*.
- *(vector-set! v val pos)* sets position *pos* of vector *v* to value *val*. Vectors use 0-based indexing and Scheme yields an error if *pos* is out of bounds.
- *(vector-ref v pos)* returns the value stored in position *pos* of vector *v*. Scheme yields an error if *pos* is out of bounds.

First, we need to define the stack, a stack pointer (to remember where the top of the stack is – actually, our stack pointer will point to the first open slot in the stack, not the last used slot), and the registers:

```
(define the-stack (make-vector 100))
```

```
(define =sp= 0) ;; the stack pointer
```

```
(define =reg l= 'dummy)
```

```
(define =tmp0= 'dummy)
```

The main change in the code is that the stack no longer needs to be a parameter because it is a global variable. In other words, we mainly need to edit *Pop* and *Push*.

```
(define-datatype stack-rec StackRec?
```

```
  [bottom-rec]
```

```
  [cons-rec (first-num number?)])
```

```
(define (filter-pos/stk)
```

```
  (cond [(empty? =reg l=) (Pop empty)]
```

```
        [(cons? =reg l=)
```

```
         (cond [(> (first =reg l=) 0)
```

```
                (begin
```

```
                  (set! =tmp0= (first =reg l=))
```

```
                  (set! =reg l= (rest =reg l=))
```

```
                  (Push (cons-rec =tmp0=))
```

```
                  (filter-pos/stk))]
```

```
        [else
```

```
         (begin
```

```
           (set! =reg l= (rest =reg l=))
```

```
           (filter-pos/stk))]]))
```

```
(define (Pop value)
```

```
  (let ([top-rec (vector-ref the-stack (- =sp= 1))])
```

```
    (begin
```

```
      (set! =sp= (- =sp= 1))
```

```
      (cases stack-rec top-rec
```

```
        [bottom-rec () value]
```

```
        [cons-rec (first-val) (Pop (cons first-val value))])))
```

```
(define (Push rec)
```

```
  (begin
```

```
    (vector-set! the-stack =sp= rec)
```

```
    (set! =sp= (+ 1 =sp=))))
```

```
(define (filter-pos L)
```

```
(begin
  (set! =sp= 0)
  (Push (bottom-rec))
  (set! =reg1= L)
  (filter-pos/stk))
```

Why the sudden introduction of register `=tmp0=` (the previous version just used `(first =reg1=)` in the code. In the spirit of making our code as close to machine code as possible, we're making the instructions (pushed onto the stack) operate on data in registers.

Time to take stock.³ How close is our code to language-independent machine code? Certainly a lot closer – we no longer use Scheme lists for the stack, the function calls are just jumps, and the arguments are passed in registers. What's left? Well, we are still using Scheme lists for the input data (the list of numbers). If we've really compiled the program, we need to reduce our dependencies on Scheme's lists for data as well.

How does Scheme handle data such as lists? When we call `cons`, Scheme allocates memory on the *heap*. The heap is another segment of memory for longer-term storage than the stack. We'll need to implement the heap explicitly to finish our compiler.

Making the Heap Explicit

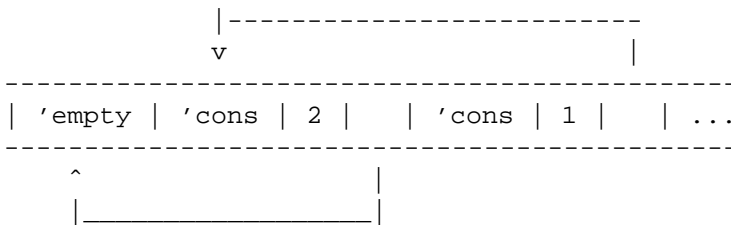
We will implement the heap as another vector (since it is just another portion of memory). We will allocate space on the heap for our list data structures.

First, the definition of the heap itself:

```
(define the-heap (make-vector 100))
(define heap-ptr 0)
```

Next, in order to use the heap to store lists, we'll need functions to allocate space for lists. We have two kinds of lists (`empty` and `cons`), so we'll need one allocator function for each. These functions advance the heap pointer by enough cells to store the corresponding data, then put the needed data into the appropriate cells.

The following diagram shows how we would store the list `(cons 1 (cons 2 empty))` in the heap. Before we allocated the `cons`, the heap-pointer was at the cell now containing the `'empty`. Note two things: first, we store pointers (really numbers giving a memory address/vector index) for the rest of the list because we don't know how large that list will be (we can't allocated a fixed number of cells to store it). Second, when we store data in the heap, we also tag it with its type – this is necessary to implement operations such as `empty?` and `cons?`.



³since we've just taken the stack ...

```

;; alloc-empty : → location
(define (alloc-empty)
  (begin
    (vector-set! the-heap heap-ptr 'empty)
    (set! heap-ptr (+ heap-ptr 1))
    (- heap-ptr 1)))

```

```

;; alloc-cons : num location → location
(define (alloc-cons fst rst)
  (begin
    (vector-set! the-heap heap-ptr 'cons)
    (vector-set! the-heap (+ 1 heap-ptr) fst)
    (vector-set! the-heap (+ 2 heap-ptr) rst)
    (set! heap-ptr (+ heap-ptr 3))
    (- heap-ptr 3)))

```

Now that we've changed our representation of lists, we need to provide our own implementations of the list primitives *first*, *rest*, *empty?*, and *cons?*. Instead of taking lists as inputs, our new primitives take addresses on the heap (vector indices). Note that the new list primitives rely on the implementation details of the allocation primitives.

```

(define (first/heap addr)
  (vector-ref the-heap (+ 1 addr)))

(define (rest/heap addr)
  (vector-ref the-heap (+ 2 addr)))

(define (cons?/heap addr)
  (eq? (vector-ref the-heap addr) 'cons))

(define (empty?/heap addr)
  (eq? (vector-ref the-heap addr) 'empty))

```

Finally, we rewrite the *filter-pos* code to use our new list primitives in place of Scheme's primitives:

```

(define (filter-pos/stk)
  (cond [(empty?/heap =reg1=) (Pop (alloc-empty))]
        [(cons?/heap =reg1=)
         (cond [(> (first/heap =reg1=) 0)
                (begin
                  (set! =tmp0= (first/heap =reg1=))
                  (set! =reg1= (rest/heap =reg1=))
                  (Push (cons-rec =tmp0=))
                  (filter-pos/stk))]
               [else
                (begin
                  (set! =reg1= (rest/heap =reg1=))
                  (filter-pos/stk))]]]))

(define (Pop value)
  (let ([top-rec (vector-ref the-stack (- =sp= 1))])
    (begin
      (set! =sp= (- =sp= 1))
      (cases stack-rec top-rec
        [bottom-rec () value]
        [cons-rec (first-val) (Pop (alloc-cons first-val value))])))

```

How do we write our new *filter-pos* helper function? Normally, that function would input a Scheme list. You could write a function that takes a Scheme list and allocates the heap memory accordingly, but that's not interesting as far as this lecture is concerned. Instead, we can just hardcode the list into the program.

```
(define (filter-pos)
  (let ([mem (alloc-cons
              -2 (alloc-cons
                  3 (alloc-cons
                      0 (alloc-cons
                          -1 (alloc-cons 9 (alloc-empty))))))]
        (begin
          (set! =sp= 0)
          (Push (bottom-rec))
          (set! =reg1= mem)
          (printf "answer: ~a~nheap: ~a ~n" (filter-pos/stk) the-heap))))
```

This final version of the program uses no Scheme-specific constructs: we've reduced everything to basic, low-level language operations. You could translate this program to C or assembly fairly easily at this point. Thus, we've achieved our goal of seeing how to compile programs starting from CPS. If you want a compiler for one of our languages, implement each of these steps as a separate program, then compose the steps to get a compiled version.