C for Schemers

Max Hailperin

September 10, 1996

The title of this document is something of a lie; you won't learn C by reading this. However, you will learn enough C to be able to use Hennessy and Patterson's *Computer Organization and Design: The Hardware/Software Interface*. How much C is that? Not much; they use a few scattered elements of C notation to explain what's going on inside the computer, and a few real short, simple C procedures as examples for translation into assembly language. Think of this handout as being like a crash course in Greek someone gives you while you are already on the plane flying to Greece—you learn to say "Excuse me, but do you speak English?" in Greek.

It might be best to dive right in to translating examples from C to Scheme, following along what Hennessy and Patterson translate into Pascal in sections D.3 through D.6.

Assignment statements

Examples in C	**Corresponding Scheme code**
`a = b + c;`	`(set! a (+ b c))`
`d = a - e;`	`(set! d (- a e))`
`f = (g + h) - (i + j);`	`(set! f (- (+ g h) (+ i j)))`
`g = h + P[i];`	`(set! g (+ h (vector-ref P i)))`
`P[i] = h + P[i];`	`(vector-set! P i (+ h (vector-ref P i)))`

For those assignment statements that involve "array elements," like `P[i]`, the above translation in terms of Scheme's vectors (using `vector-set!` and `vector-ref`) is completely natural, because C's arrays and Scheme's vectors correspond very closely.

For the other assignment statement's, however, such as the ones assigning values to `a`, `d`, `f`, and `g`, there is a bit of a problem, in that the Scheme version uses `set!`, which most of you aren't familiar with (since we don't use it in our introductory courses).

So, as an alternative, consider making a very simple data type called "cell" with just a single setable and getable component; we can represent cells naturally as one-element vectors:

```
(define make-cell
  (lambda ()
    (make-vector 1)))

(define cell-get
  (lambda (cell)
    (vector-ref cell 0)))

(define cell-set!
  (lambda (cell new-contents)
    (vector-set! cell 0 new-contents)))
```

Now, with cells in hand, assume that `a`, `b`, `c`, etc. have all been defined as cells:

```
(define a (make-cell))
(define b (make-cell))
(define c (make-cell))
```

With this, we can now rewrite the C assignment statement `a = b + c` as

```
(cell-set! a (+ (cell-get b) (cell-get c))).
```

Hennessy and Patterson go on in section D.3 to introduce C's logical operators, like `<<` and `&`. These have no direct analog in Scheme, and are explained where they get used, in section 4.4, so you might as well wait until we get to 4.4 to worry about what these mean.

Relational expressions and conditional statements

Recall from the previous section that C uses the equal sign, `=`, as its assignment operator, like Scheme's `set!` (or `cell-set!`) and `vector-set!`. This means when you want to test whether two numbers are equal or not, you can't use `=`, unlike in Scheme or Pascal; instead, C uses the double equal sign, `==`, for equality testing. Another difference from Scheme is that there is an not-equal test, written `!=`, which Scheme doesn't have at all. Other than that, the numerical comparison operators are the same, for example `<=` means less than or equal to in both languages. Of course, they get used differently: in C you'd say `a <= b` where in Scheme you'd say `(<= a b)`. Oh yes, another difference: C comparisons evaluate to 1 for true and 0 for false, instead of `#t` and `#f`. Just as in Scheme anything except `#f` counts as true, so too in C any non-zero value counts as true.

The C *if* statement

```
if (i == j) f = g + h;
    else f = g - h;
```

would be written by anyone except Hennessy and Patterson as

```
if (i == j)
  f = g + h;
else
  g = g - h;
```

Either way it would translate into Scheme as

```
(if (= i j)
    (set! f (+ g h))
    (set! f (- g h)))
```

or

```
(if (= (cell-get i) (cell-get j))
    (cell-set! f (+ (cell-get g) (cell-get h)))
    (cell-set! f (- (cell-get g) (cell-get h))))
```

From here onward, I'll stop giving the alternative translations using cell operations, and stick with the version using `set!`.

The `else` and the statement after it can be omitted, if there isn't anything you want to do in that case. Just as in Scheme you can group a bunch of statements together using `(begin ...)`, in C you can group a bunch of statements together using curly braces, { ... }.

The second example Hennessy and Patterson give of an if statement is a bit harder to translate into Scheme:

```
    if (i == j) goto L1;
    f = g + h;
L1: f = f - i;
```

Perhaps the most literal translation of this into Scheme would be

```
(define L1
  (lambda ()
    (set! f (- f i))))

(if (= i j)
    (L1)
    (begin (set! f (+ g h))
           (L1)))
```

Of course, no one would do it that way; truth be known, no one (except Hennessy and Patterson) would write the above C code either, it would make much more sense to write

```
if (i != j)
  f = g + h;
f = f - i;
```

There is a good reason why Hennessy and Patterson show the weird code with the "goto" around the first assignment, however. That is because it reflects the way the computation would be expressed in assembly language. So, they want you to be familiar with this kind of C so they can use it to explain the assembly language version.

The example C switch statement:

```
switch (k) {
   case 0: f = i + j; break;
   case 1: f = g + h; break;
   case 2: f = g - h; break;
   case 3: f = i - j; break;
};
```

can be translated into Scheme you are familiar with as follows:

```
(cond ((= k 0) (set! f (+ i j)))
      ((= k 1) (set! f (+ g h)))
      ((= k 2) (set! f (- g h)))
      ((= k 3) (set! f (- i j))))
```

(There is also a `case` expression in Scheme, which would allow for an even closer translation than `cond` does, but you may not be familiar with it.)

Loops

The C while loop:

```
while (save[i] == k)
    i = i + j;
```

could be translated directly into Scheme as

```
(while (lambda () (= (vector-ref save i) k))
       (lambda () (set! i (+ i j))))
```

provided that you used the following definition of `while`:

```
(define while
  (lambda (test body)
    (if (test)
        (begin (body)
               (while test body))
        'done)))
```

Hennessy and Patterson say that "of course" while loops can be constructed from `gotos`, and then proceed to show a loop made using a goto that *isn't* the equivalent of a while loop. So, let's first look at a goto-based loop that *is* equivalent to a while loop, namely one equivalent to the above while loop:

```
Loop:  if (save[i] != k) goto Done;
       i = i + j;
       goto Loop;
Done:
```

This is like saying in Scheme:

```
(define Loop
  (lambda ()
    (if (= (vector-ref save i) k)
        (begin (set! i (+ i j))
               (Loop))
        'done)))

(Loop)
```

Now, we can move on to the goto-based loop that Hennessy and Patterson show, which isn't like a while loop because its test is at the bottom rather than the top; thus the loop body is done at least once, even if the test expression is never true:

```
Loop:   g = g + P[i];
        i = i + j;
        if (i != h) goto Loop
```

In Scheme, we might write this one as

```
(define Loop
  (lambda ()
    (set! g (+ g (vector-ref P i)))
    (set! i (+ i j))
    (if (not (= i h))
        (Loop)
        'done)))

(Loop)
```

Hennessy and Patterson next introduce *for* statements, which are best understood with the help of the following Scheme procedure:

```
(define for
  (lambda (initialization test stepper body)
    (define loop
      (lambda ()
        (if (test)
            (begin (body)
                   (stepper)
                   (loop))
            'done)))
    (initialization)
    (loop)))
```

With this procedure in hand, we can translate the following C loop (where the $\{\ldots\}$ is a placeholder for the loop body):

```
for (i = 0; i < n; i = i + 1)
  {...}
```

In Scheme, this would be

```
(for (lambda () (set! i 0))
     (lambda () (< i n))
     (lambda () (set! i (+ i 1)))
     (lambda () ...))
```

The C operators `&&`, `||`, and `!` correspond to the Scheme `and`, `or`, and `not`. For example, here is a for loop using `&&` and its translation into Scheme, using `and`:

```
for (j = i - 1; j >= 0 && v[j] > j[j + 1]; j = j - 1)
   { ... }

(for (lambda () (set! j (- i 1)))
     (lambda () (and (>= j 0)
                     (> (vector-ref v j)
                        (vector-ref v (+ j 1)))))
     (lambda () (set! j (- j 1)))
     (lambda () ...))
```

Examples to put it all together

The most literal translation of the C procedure

```
swap(int v[], int k)
{
   int temp;

   temp = v[k];
   v[k] = v[k + 1];
   v[k + 1] = temp;
}
```

into Scheme would be as follows:

```
(define swap
  (lambda (v k)
    (let ((temp 'undefined-value-but-it-should-be-an-integer))
      (set! temp (vector-ref v k))
      (vector-set! v k (vector-ref v (+ k 1)))
      (vector-set! v (+ k 1) temp))))
```

In the C version of the program, the *declaration* "`int temp`" sets aside a memory location for the variable `temp` that can only hold integers, but until a value is assigned to that location, the contents of the location are undefined; this is what the let is intended to indicate. Just as the `temp` variable is declared to be of type `int` (i.e., an integer small enough to fit in a standard sized memory location), so too the parameters are declared to be an array of `int`s and another `int`. This declaration of types has no direct analog in Scheme.

Moving on to the longer example:

```
sort(int v[], int n)
{
   int i, j;

   for (i = 0; i < n; i = i + 1)
    for ( j = i - 1; j >= 0 && v[j] > v[j + 1]; j = j - 1)
     swap(v,j);
}
```

The direct translation into Scheme, using the previously presented `for` procedure, would be be as follows:

```
(define sort
  (lambda (v n)
    (let ((i 'undefined-value-but-it-should-be-an-integer)
          (j 'undefined-value-but-it-should-be-an-integer))
      (for (lambda () (set! i 0))
           (lambda () (< i n))
           (lambda () (set! i (+ i 1)))
           (lambda ()
             (for (lambda () (set! j (- i 1)))
                  (lambda () (and (>= j 0)
                                  (> (vector-ref v j)
                                     (vector-ref v (+ j 1)))))
                  (lambda () (set! j (- j 1)))
                  (lambda () (swap v j))))))))
```