Truth and Falsehood

Why is there no mark engraved upon men' bodies,
By which we could know the true ones from the false ones?

— Euripides, *The Medea 518-519*

This week, we'll talk about how to get Scheme to figure out if something is true or false.

What is truth? In Scheme, everything except falsehood is truth. It sounds like a silly way of defining truth, but it works out pretty well.

Let's write a function called `truthsayer` that will say "truth" if we give it a true value, and "untruth" otherwise.

```
(define (truthsayer x)
  (if x
     'truth
     'untruth))
```

Conventionally, whenever we have a function that has a question mark at the end, like `truth?`, we're making a function that only returns booleans. These are *predicates*—functions that either say `#t` or `#f`. Since truthsayer will only return words, let's not go against tradition. Here's a few tests we can try on `truthsayer`:

```
> (truthsayer #t)
truth
> (truthsayer #f)
untruth
> (truthsayer 'apple)
truth
> (truthsayer 0)
truth
> (truthsayer 'false)
truth
```

So this confirms the notion of truth from falsehood in Scheme.

`if`, `cond`, `and`

As another example, let's write a function that plays around with truth and falsehood. It would be nice to make a function that will give us the `third` thing in a letter or sentence. What should we do if the sequence is too short? As an arbitrary choice, let's have it return `#f`. There are several different ways of writing this sort of function. (The book points out this sort of thing in its discussion on "function vs process.")

First, let's try it with the `if` special form.

```
(define (third x)
  (if (>= (count x) 3)
        (first (bf (bf x)))
        #f))
```

Let's try writing this another way, using `cond`:

```
(define (third x)
  (cond ((>= (count x) 3) (first (bf (bf x))))
        (else #f)))
```

Finally, we can even use `and` to make our `third` function:

```
(define (third x)
  (and (>= (count x) 3) (first (bf (bf x)))))
```

Often, which construction you prefer will do with personal taste. It takes a little bit of thought to see that the `and` version will do the same thing as the other two. Although its shorter, it's slightly tricker because it depends on the fact that `and` will either return `#f` or the very last true value that it looked at.

`if` vs `cond`

Which brings us to the question: why do we need both `if` and `cond`? Why provide two ways of doing things, and what makes these two different? To answer this, we actually need to practice with them, and get a feel for the situations that favor one over the other.

Let's try another problem: Imagine that we wanted to convert letters to numbers. We'll call this function `l->n` "letter to number". First, let's try to define this using `if` expressions alone.

```
(define (l->n letter)
  (if (equal? letter 'a) 1
      (if (equal? letter 'b) 2
          (if (equal? letter 'c) 3
              (if (equal? letter 'd) 4
                  (if (equal? letter 'e) 5
                      ...
```

In a certain sense, it's both pretty neat and hideous. We'll need to write 26 `if` expressions, and even worse, it will slant to the right like a staircase.

Let's see what `l->n` will look like if we use `cond` instead.

```
(define (l->n letter)
  (cond ((equal? letter 'a) 1)
((equal? letter 'b) 2)
((equal? letter 'c) 3)
((equal? letter 'd) 4)
((equal? letter 'e) 5)
...
```

This is still quite bad, but it gives us a hint of `cond`'s advantages: `cond` is good when we have several (three or more) conditions to search for. `if`, on the other hand, is good when we have an "either this or that" situation.

An example: `type-of`

We also talked about the `type-of` on pg. 85, how to write it, and what things to be careful about. To do this problem, we need to use the functions `sentence? word? number? boolean?` which are listed on the back cover. I'll try to show one tricky thing about this problem.

```
(define (type-of x)
  (cond ((word? x) 'word)
((sentence? x) 'sentence)
((number? x) 'number)
((boolean? x) 'boolean)))
```

What's buggy with this definition of `type-of`?

As a general strategy, when we write our case statements, we should take care of the most specific stuff first. That way, we won't prematurely handle certain situations that need special attention.

Digression with `position`

As a final note, both of these ways of `l->n` are equally horrendous. They will work, but they aren't aesthetically pleasing. There is a better way to go about this particular problem — but don't read this unless you have time; I'm going way off tangent on this one.

You've been introduced to the `item` function: if you give `item` a number as its first argument, and a word or sentence as its second, `item` will pick out the Nth element of that word or sentence.

What would be really nice is a function called `position` that can give us the position of an element in a sentence or word. Let's pretend that it exists, and see what makes such a function useful:

```
> (position 'pick '(this is an expression that will pick out a word))
7
> (position 'fififofum '(fi fi fo fum))
#f
> (position 'q 'abcdefghijklmnopqrstuvwxyz))
17
```

`position` is especially nice because it makes `l->n` like child's play:

```
(define alphabet 'abcdefghijklmnopqrstuvwxyz)
(define (l->n letter)
  (position letter alphabet))
```

But to my great horror, `position` isn't defined within Simply Scheme's standard functions!

To remedy this, I've put up a version of `position` on my web site here: http://hkn.eecs.berkeley.edu/~dyoo/cs3/
To use it, load it like any other file in Scheme:

```
(load "position.scm")
```

and you'll be able to use `position` afterwards.

WARNING WARNING: Do not study the code! That's because it uses recursive ideas, which won't be discussed until the second half of the semester. Treat it as a black box until we get to that point.

You can always email me your questions at `dyoo@hkn.eecs.berkeley.edu`. I'll see you next week!