Exercise 1.6.

Alyssa P. Hacker doesn't see why `if` needs to be provided as a special form. "Why can't I just define it as an ordinary procedure in terms of `cond`?" she asks. Alyssa's friend Eva Lu Ator claims this can indeed be done, and she defines a new version of `if`:

```
(define (new-if predicate then-clause else-clause)
  (cond (predicate then-clause)
        (else else-clause)))
```

Eva demonstrates the program for Alyssa:

```
(new-if (= 2 3) 0 5)
5

(new-if (= 1 1) 0 5)
0
```

Delighted, Alyssa uses `new-if` to rewrite the square-root program:

```
(define (sqrt-iter guess x)
  (new-if (good-enough? guess x)
          guess
          (sqrt-iter (improve guess x)
                     x)))
```

What happens when Alyssa attempts to use this to compute square roots? Explain.

Answer.

When Alyssa attempts to use this to compute square roots, say `(sqrt 2)` for example, the interpreter drops into an infinite recursion:

```
(sqrt 2)
;Aborting!: maximum recursion depth exceeded
```

We can exploit the rule of substitution to trace the behavior of the evaluator:

```
(sqrt 2)

(sqrt-iter 1.0 2)

(new-if (good-enough? 1.0 2)
        1.0
        (sqrt-iter (improve 1.0 2) 2))

(new-if (good-enough? 1.0 2)
        1.0
        (new-if (good-enough? 1.5 2)
                1.5
                (sqrt-iter (improve 1.5 2) 2)))

...
```

In section 1.1.5 we learnt that Scheme uses applicative-order evaluation, namely, that all the arguments to Scheme procedures are evaluated when the procedure is applied. Hence, the evaluator must obtain the value of `(sqrt-iter (improve 1.0 2) 2)` before get the compound procedure `new-if` applied. Expanding this expression would produce another application of `new-if` that contains `(sqrt-iter ...)`. This arouse an infinite recursion and therefore exhausted the interpreter.

Email address: informlarry@gmail.com