

Preconditions and Postconditions

- Procedures as Contracts
- Generating Explicit Errors
- Husks and Kernels
- Documentation

Several of the Scheme procedures that we have written or studied in preceding labs presuppose that their arguments will meet specific *preconditions* -- constraints on the types or values of its arguments. For instance, we saw in the lab on recursion with lists that the `greatest-of-list` procedure requires its argument to be a non-empty list of real numbers. If some careless programmer invokes `greatest-of-list` and gives it, as an argument, the empty list, or a list in which one of the elements is not a real number, or perhaps even some Scheme value that is not a list at all, the computation that the definition of `longest-in-list` describes cannot be completed.

Procedures as Contracts

A procedure definition is like a contract between the author of the definition and someone who invokes the procedure. The *postconditions* of the procedure are what the author guarantees: When the computation directed by the procedure is finished, the postconditions shall be met. Usually the postconditions are constraints on the value of the result returned by the procedure. For instance, the postcondition of the `square` procedure,

```
(define square
  (lambda (val)
    (* val val)))
```

is that the result is the square of the argument `val`.

The preconditions are the guarantees that the invoker of a procedure makes to the author, the constraints that the arguments shall meet. For instance, it is a precondition of the `square` procedure that `val` is a number.

If the invoker of a procedure violates its preconditions, then the contract is broken and the author's guarantee of the postconditions is void. (If `val` is, say, a list of symbols, then the author can't very well guarantee to return its square.) To make it less likely that an invoker violates a precondition by mistake, it is usual to document preconditions carefully and to include occasional checks in one's programs, ensuring that the preconditions are met before starting a complicated computation.

Many of DrScheme's primitive procedures have such preconditions, which they enforce by aborting the computation and displaying a diagnostic message when the preconditions are not met:

```

> (/ 1 0)
/: division by zero
> (log 0)
log: undefined for 0
> (length 116)
length: expects argument of type <proper list> given 116

```

Generating Explicit Errors

To enable us to enforce preconditions in the same way, DrScheme provides a procedure named `error`, which takes a string as its argument. Calling the `error` procedure aborts the entire computation of which the call is a part and causes the string to be displayed as a diagnostic message.

For instance, we could enforce `greatest-of-list`'s precondition that its parameter be a non-empty list of reals as by rewriting its definition thus:

```

(define greatest-of-list
  (lambda (ls)
    (if (or (not (list? ls))
          (null? ls)
          (not (all-real? ls)))
        (error "greatest-of-list: requires a non-empty list of reals")
        (if (null? (cdr ls))
            (car ls)
            (max (car ls) (greatest-of-list (cdr ls)))))))

```

where `all-real?` is a predicate that we have to write that takes any list as its argument and determines whether or not all of the elements of that list are real numbers:

```

(define all-real? (lambda (ls) (or (null? ls) (and (real? (car ls)) (all-real? (cdr ls))))))

```

Now the `greatest-of-list` procedure enforces its precondition:

```

> (greatest-of-list 139)
greatest-of-list: requires a non-empty list of reals
> (greatest-of-list null)
: requires a non-empty list of reals
> (greatest-of-list (list 71/3 -17 23 'oops 16/15))
greatest-of-list: requires a non-empty list of reals

```

Husks and Kernels

Including precondition testing in your procedures often makes them markedly easier to analyze and check, so I recommend the practice, especially during program development. There is a trade-off, however: It takes time to test the preconditions, and that time will be consumed on every invocation of the procedure. Since time is often a scarce resource, it makes sense to save it by skipping the test when you can prove that the precondition will be met. This often happens when you, as programmer, control the context in which the procedure is called as well as the body of the procedure itself.

For example, in the preceding definition of `greatest-of-list`, although it is useful to test the precondition when the procedure is invoked “from outside” by a potentially irresponsible caller, it is a waste of time to repeat the test of the precondition for any of the recursive calls to the procedure. At the point of the recursive call, you already know that `ls` is a list of strings (because you tested that precondition on the way in) and that its `cdr` is not empty (because the body of the procedure explicitly tests for that condition and does something other than a recursive call if it is met), so the `cdr` must also be a non-empty list of strings. So it’s unnecessary to confirm this again at the beginning of the recursive call.

One solution to this problem is to replace the definition of `greatest-of-list` with two separate procedures, a “husk” and a “kernel.” The husk interacts with the outside world, performs the precondition test, and launches the recursion. The kernel is supposed to be invoked only when the precondition can be proven true; its job is to perform the main work of the original procedure, as efficiently as possible:

```
(define greatest-of-list
  (lambda (ls)
    ;; Make sure that ls is a non-empty list of real numbers.
    (if (or (not (list? ls))
          (null? ls)
          (not (all-real? ls)))
        (error "greatest-of-list: requires a non-empty list of reals")
        ;; Find the greatest number in the list.
        (greatest-of-list-kernel ls))))

(define greatest-of-list-kernel
  (lambda (ls)
    (if (null? (cdr ls))
        (car ls)
        (max (car ls) (greatest-of-list-kernel (cdr ls))))))
```

The kernel has the same preconditions as the husk procedure, but does not need to enforce them, because we invoke it only in situations where we already know that the preconditions are satisfied.

The one weakness in this idea is that some potentially irresponsible caller might still call the kernel procedure directly, bypassing the husk procedure that he’s supposed to invoke. In later labs, we’ll see that there are a couple of ways to put the kernel back inside the husk without losing the efficiency gained by dividing the labor in this way.

When documenting your procedures, you may wish to note whether a precondition is verified (in which case you should make sure to print an error message) or unverified (in which case you may still crash and burn, but the error will come from one of the procedures you call).

Documentation

The previous sections have covered the programming issues related to preconditions and their verification. However, for most procedures, I care at least as much that you have carefully *documented* your procedures, including their preconditions and postconditions. Such documentation is evidence that you’ve thought carefully about what your procedure should do (and should not do). In particular, you need to be careful to document,

- The *procedure* (that is, its name);
- The *parameters* (their names, their expected types, their semantics);
- The *purpose* (what the procedure does);
- The *produced value* (what the result is);
- The *preconditions* (and whether or not you've bothered to verify them); and
- The *postconditions* (some of which you should express as formal as you can).