Fundamentals of CS I (CS151 2001S)

Conditional Evaluation

When Scheme encounters a procedure call, it looks at all of the subexpressions within the parentheses and evaluates each one. Sometimes, however, the programmer wants Scheme to exercise more discretion -- specifically, to select just one subexpression for evaluation from two or more alternatives. In such cases, one uses a *conditional expression* -- that is, an expression that tests whether some condition is met and selects the subexpression to evaluate on the basis of the outcome of that test.

For instance, let's write a procedure to compute the *disparity* between two given real numbers, the amount by which one of them exceeds the other. We can compute this by subtraction, but before we can do the subtraction we need to know which of the two given numbers (let's call them `fore` and `aft`) is greater, so that we can make it the minuend and the other the subtrahend.

That is, if `fore` is greater, we compute the disparity by evaluating the expression `(- fore aft)`; otherwise, the expression we need is `(- aft fore)`.

If Expressions

A conditional expression, specifically, an *`if` expression*, selects one or the other of these expressions, depending on the outcome of a test, thus:

```
(if (> fore aft)   ; If fore is greater than aft ...
    (- fore aft)   ; ... subtract aft from fore ...
    (- aft fore))  ; ... otherwise subtract fore from aft.
```

Here is the complete definition of the `disparity` procedure:

```
;;; Procedure:
;;;   disparity
;;; Parameters:
;;;   fore, an exact number
;;;   aft, an exact number
;;; Purpose:
;;;   Compute the amount by which one number
;;;   exceeds another.
;;; Produces:
;;;   excess, an exact number.
;;; Preconditions:
;;;   Both fore and aft are exact numbers (unverified).
;;; Postconditions:
;;;   The greater of fore and aft is equal to the sum of excess
;;;   and the lesser of fore and aft.  If fore and aft are equal,
;;;   excess is equal to 0.
;;; Citation:
;;;   Based on a similar procedure created by John David Stone of
;;;   Grinnell College which is dated January 27, 2000.
(define disparity
```

```
(lambda (fore aft)
  (if (> fore aft)
      (- fore aft)
      (- aft fore))))
```

In an `if` expression, the *test* (the expression following the keyword `if`) is always evaluated first. If its value is `#t`, then the *consequent* (the expression following the test) is evaluated, and the *alternate* (the expression following the consequent) is ignored. On the other hand, if the value of the test is `#f`, then the consequent is ignored and the alternate is evaluated.

Scheme accepts `if` expressions in which the value of the test is non-Boolean. However, all non-Boolean values are classified as ‘‘truish’’ and cause the evaluation of the consequent.

Multiple Alternatives

When there are more than two alternatives, it is often more convenient to set them out using a *`cond` expression*. Like `if`, `cond` is a keyword. It is followed by zero or more lists of expressions called *`cond` clauses*. The first expression within a `cond` clause is a test, similar to the test in an `if` expression. When the value of such a test is found to be `#f`, the subexpressions that follow the test are ignored and Scheme proceeds to the test at the beginning of the next `cond` clause. But when a test is evaluated and the value turns out to be true, or even ‘‘truish’’ (that is, anything other than `#f`) each of the remaining expressions in the same `cond` clause is evaluated in turn, and the value of the last one becomes the value of the entire `cond` expression. Subsequent `cond` clauses are completely ignored.

In other words, when Scheme encounters a `cond` expression, it works its way through the `cond` clauses, evaluating the test at the beginning of each one, until it reaches a test that *succeeds* (one that does not have `#f` as its value). It then makes a ninety-degree turn and evaluates the other expressions in the selected `cond` clause, retaining the value of the last expression.

If all of the tests in a `cond` expression are found to be false, the value of the `cond` expression is unspecified (that is, it might be anything!). To prevent the surprising results that can ensue when one computes with unspecified values, good programmers customarily end every `cond` expression with a `cond` clause in which the keyword `else` appears in place of a test. Scheme treats such a `cond` clause as if it had a test that always succeeded. If it is reached, the subexpressions following `else` are evaluated, and the value of the last one is the value of the whole `cond` expression.

For example, here is a `cond` expression that inspects a list called `ls`:

```
(cond ((null? ls) 'none)
      ((symbol? (car ls)) 'first)
      (else 'other))
```

The expression has three `cond` clauses. In the first, the test is `(null? ls)`. If `ls` happens to be the empty list, the value of this first test is `#t`, so we evaluate whatever comes after the test to find the value of the entire expression -- in this case, the symbol `none`

If `ls` is not the empty list, then we proceed instead to the second `cond` clause. Its test is `(symbol? (car ls))` -- in other words, ‘‘look at the first element of `ls` and determine whether it is a symbol’’. If it is, then again we evaluate whatever comes after the test and obtain the symbol `first`.

However, if the first element of `ls` is not a symbol, then we proceed instead to the third `cond` clause. Since the keyword `else` appears in this `cond` clause in place of a test, we take that as an automatic success and evaluate `'other`, so that that value of the whole `cond` expression in this case is the symbol `other`.