

Conditional Evaluation in Scheme

Summary: Many programs need to make choices. In this reading, we consider Scheme's *conditional expressions*, expressions that allow programs to behave differently in different situations.

Contents:

- Introduction
- If Expressions
- Dropping the Alternative
- Supporting Multiple Alternatives with `cond`
- Multiple Consequents
- `and` and `or` as Conditionals

Introduction

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, the programmer wants to select just one subexpression for evaluation from two or more alternatives. In such cases, one uses a *conditional expression*, 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 result 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)`.

To write a disparity procedure, we need a mechanism to choose which expression to evaluate. Such mechanisms are the key subject of this reading.

If Expressions

A conditional expression, specifically, an *if expression*, selects one or the other of these expressions, depending on the outcome of a test. The general form is

```
(if test consequent alternative)
```

We'll return to the particular details in a moment. For now, let's consider the conditional we might write for the disparity procedure.

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

To turn this expression into a procedure, we need to add the `define` keyword, a name (`disparity`), a lambda expression, and such. We also want to give appropriate documentation.

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 of the form `(if test consequent alternate)`, the *test* is always evaluated first. If its value is `#t` (which you may recall, means “yes” or “true”), then the *consequent* 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.

Dropping the Alternative

It is possible to write an `if` expression without the alternative. Such an expression has the form `(if test consequent)`. In this case, the test is still evaluated first. If the test holds (that is, has a value of `#t` or anything other than `#f`), the consequent is evaluated and its value is returned. If the test fails to hold (that is, has value `#f`), the `if` expression has no value.

We mention this issue here for completeness. At this stage of your career, you are unlikely to need or want the alternative-free `if`, and you should avoid using it.

Supporting Multiple Alternatives with `cond`

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. (Recall that keywords differ from procedures in that the order of evaluation of the parameters may differ.) The `cond` keyword is followed by zero or more lists-like expressions called *cond clauses*.

```
(cond
  (test-0 consequent-0)
  ...
  (test-n consequent-1)
  (else alternate))
```

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 subexpression that follows the test is 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`), the consequent for that test is evaluated and its value is the value of the whole `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 consequent in the selected `cond` clause, retaining the value of the consequent.

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) 'empty)
      ((symbol? (car ls)) 'list-that-starts-with-a-symbol)
      (else 'something-else))
```

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 `empty`.

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 `list-that-starts-with-a-symbol`.

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 `'something-else`, so that that value of the whole `cond` expression in this case

is the symbol `something-else`.

Multiple Consequents

Although we have written our conditionals with one consequent per test (and we encourage you to do the same), it is, in fact, possible to have multiple consequents per test.

```
(cond
  (test-0 consequent-0-0 consequent-0-1 ... consequent-0-m)
  ...
  (else alternate-0 alternate-1 ... alternate-a))
```

In this case, when a test succeeds, each of the remaining subexpressions (that is, consequents) in the same `cond` clause is evaluated in turn, and the value of the last one becomes the value of the entire `cond` expression.

and and or as Conditionals

Note that both `and` and `or` provide a type of conditional behavior. As you may recall from the reading on Boolean values, `and` evaluates each argument in turn until it hits a value that is `#f` and then returns `#f` (or returns the last value if none return `#f`). Similarly, `or` evaluates each argument in turn until it finds one that is not `#f`, in which case it returns that value, or until it runs out of values, in which case it returns `#f`.

That is, `(or exp0 exp1 ... expn)` behaves much like the following `cond` expression, except that the `or` version evaluates each expression once, rather than twice.

```
(cond
  (exp0 exp0)
  (exp1 exp1)
  ...
  (expn expn)
  (else #f))
```

Similarly, `(and exp0 exp1 ... expn)` behaves much like the following `cond` expression.

```
(cond
  ((not exp0) #f)
  ((not exp1) #f)
  ...
  ((not expn) #f)
  (else expn))
```

Most beginning programmers find the `cond` versions much more understandable, but some advanced Scheme programmers use the `and` and `or` forms because they find them clearer. Certainly, the `cond` for `and` is quite repetitious.

Copyright © 2006 Samuel A. Rebelsky. This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc/2.5/> or send a letter to Creative

Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.