Equality Checking in Scheme

In Scheme = is used to test for numeric equality (including comparison of different numeric types). Non-numeric arguments cause a run-time error. Thus

```
(= 1 1) ⇒ #t
(= 1 1.0)  ⇒ #t
(= 1 2/2) ⇒ #t
(= 1 1+0.0i) ⇒ #t
```

To compare non-numeric values, we can use either:

pointer equivalence (do the two operands point to the same address in memory)

structural equivalence (do the two operands point to structures with the same size, shape and components, even if they are in different memory locations)

In general pointer equivalence is faster but less accurate.

Scheme implements both kinds of equivalence tests.

`(eqv? obj1 obj2)`

This tests if **`obj1`** and **`obj2`** are the exact same object. This works for atoms and pointers to the same structure.

(equal? obj1 obj2)

This tests if **obj1** and **obj2** are the same, component by component. This works for atoms, lists, vectors and strings.

```
(eqv? 1 1) ⇒ #t

(eqv? 1 (+ 0 1)) ⇒ #t

(eqv? 1/2 (- 1 1/2)) ⇒ #t

(eqv? (cons 1 2)(cons 1 2)) ⇒
   #f

(eqv? "abc" "abc") ⇒ #f

(equal? 1 1) ⇒ #t

(equal? 1 (+ 0 1)) ⇒ #t

(equal? 1/2 (- 1 1/2)) ⇒ #t

(equal? (cons 1 2)(cons 1 2)) ⇒
  #t

(equal? "abc" "abc") ⇒ #t
```

In general it is wise to use **equal?** unless speed is a critical factor.

I/O in Scheme

Scheme has simple read and write functions, directed to the "standard in" and "standard out" files.

`(read)`

Reads a single Scheme object (an atom, string, vector or list) from the standard in file. No quoting is needed.

`(write obj)`

Writes a single object, **`obj`**, to the standard out file.

`(display obj)`

Writes **`obj`** to the standard out file in a more readable format. (Strings aren't quoted, and characters aren't escaped.)

`(newline)`

Forces a new line on standard out file.

Ports

Ports are Scheme objects that interface with systems files. I/O to files is normally done through a port object bound to a system file.

`(open-input-file "path to file")`

This returns an input port associated with the **`"path to file"`** string (which is system dependent). A run-time error is signalled if **`"path to file"`** specifies an illegal or inaccessible file.

`(read port)`

Reads a single Scheme object (an atom, string, vector or list) from **`port`**, which must be an input port object.

`(eof-object? obj)`

When the end of an input file is reached, a special eof-object is returned. **`eof-object?`** tests whether an object is this special end-of-file marker.

`(open-output-file "path to file")`

This returns an output port associated with the **`"path to file"`** string (which is system dependent). A run-time error is signalled if **`"path to file"`** specifies an illegal or inaccessible file.

`(write obj port)`

Writes a single object, **`obj`**, to the output port specified by **`port`**.

`(display obj port)`

Writes **`obj`** to the output port specified by **`port`**. display uses a more readable format than write does. (Strings aren't quoted, and characters aren't escaped.)

`(close-input-port port)`

This closes the input port specified by **`port`**.

`(close-output-port port)`

This closes the output port specified by **`port`**.

Example—Reading & Echoing a File

We will iterate through a file, reading and echoing its contents. We need a good way to do iteration; recursion is neither natural nor efficient here.

Scheme provides a nice generalization of the **`let`** expression that is similar to C's for loop.

```
(let X ( (id1 val1) (id2 val2) ... )
       ...
       (X v1 v2 ...)
)
```

A name for the let (**`X`** in the example) is provided. As usual, **`val1`** is evaluated and bound to **`id1`**, **`val2`** is evaluated and bound to **`id2`**, etc. In the body of the let, the let may be "called" (using its

name) with a fresh set of values for the let variables. Thus **`(X v1 v2 ...)`** starts the next iteration of the let with **`id1`** bound to **`v1`**, **`id2`**, bound to **`v2`**, etc.

The calls look like recursion, but they are implemented as loop iterations.

For example, in

```
(let loop ( (x 1) (sum 0) )
  (if (<= x 10)
      (loop (+ x 1) (+ sum x))
      sum
  )
)
```

we sum the values of **`x`** from 1 to 10.

Compare it to

```
for (x=1,sum=0; x <= 10;
     sum+=x,x+=1)
 {}
```

Now a function to read and echo a file is straightforward:

```
(define (echo filename)
  (let (
   (p (open-input-file filename)))
    (let loop ( (obj (read p)))
      (if (eof-object? obj)
          #t ;normal termination
          (begin
            (write obj)
            (newline)
            (loop (read p))
          )
      )
    )
  )
)
```

We can create an alternative to **`echo`** that uses

```
(call-with-input-file
   filename function)
```

This function opens **`filename`**, creates an input port from it, and then calls **`function`** with that port as an argument:

```
(define (echo2 filename)
  (call-with-input-file filename
    (lambda(port)
      (let loop ( (obj (read port)))
        (if (eof-object? obj)
         #t
         (begin
           (write obj)
           (newline)
           (loop (read port))
         )
        )
       )
    )
) )
```

Control Flow in Scheme

Normally, Scheme's control flow is simple and recursive:

- The first argument is evaluated to get a function.
- Remaining arguments are evaluated to get actual parameters.
- Actual parameters are bound to the function's formal parameters.
- The functions' body is evaluated to obtain the value of the function call.

This approach routinely leads to deeply nested expression evaluation.

As an example, consider a simple function that multiplies a list of integers:

```
(define (*list L)
  (if (null? L)
      1
      (* (car L)(*list (cdr L)))
  )
)
```

The call `(*list '(1 2 3 4 5))`

expands to

```
(* 1 (* 2 (* 3 (* 4 (* 5 1)))))
```

But,

what if we get clever and decide to improve this function by noting that if 0 appears *anywhere* in list `L`, the product *must be* 0?

Let's try

```
(define (*list0 L)
  (cond
    ((null? L) 1)
    ((= 0 (car L)) 0)
    (else (* (car L)
             (*list0 (cdr L))))
  )
)
```

This helps a bit—we never go past a zero in **L**, but we still unnecessarily do a sequence of pending multiplies, all of which must yield zero!

Can we escape from a sequence of nested calls once we know they're unnecessary?

Exceptions

In languages like Java, a statement may *throw* an exception that's *caught* by an enclosing exception handler. Code between the statement that throws the exception and the handler that catches it is *abandoned*.

Let's solve the problem of avoiding multiplication of zero in Java, using its exception mechanism:

```
class Node {
  int val;
  Node next;
}
class Zero extends Throwable
  {};
```

```
int mult (Node L) {
  try {
    return multNode(L);
  } catch (Zero z) {
    return 0;
  }
}
int multNode(Node L)
   throws Zero {
  if (L == null)
    return 1;
  else if (L.val == 0)
    throw new Zero();
  else return
   L.val * multNode(L.next);
}
```

In this implementation, *no* multiplies by zero are ever done.

Continuations

In our Scheme implementation of **`*list`**, we'd like a way to delay doing any multiplies until we know no zeros appear in the list. One approach is to build a *continuation*—a function that represents the context in which a function's return value will be used:

```
(define (*listC L con)
  (cond
    ((null? L) (con 1))
    ((= 0 (car L)) 0)
    (else
     (*listC (cdr L)
        (lambda (n)
         (* n (con (car L)))))
     )
  )
)
```

The top-level call is

```
(*listC  L (lambda (x) x))
```

For ordinary lists *`listC` expands to a series of multiplies, just like `*list` did.

```
(define (id x) x)
(*listC '(1 2 3) id) ⟹
(*listC '(2 3)
  (lambda (n) (* n (id 1)))) ≡
(*listC '(2 3)
  (lambda (n) (* n 1))) ⟹
(*listC '(3)
  (lambda (n) (* n (* 2 1)))) ≡
(*listC '(3)
  (lambda (n) (* n 2))) ⟹
(*listC ()
  (lambda (n) (* n (* 3 2)))) ≡
(*listC () (lambda (n) (* n 6)))
  ⟹ (* 1 6) ⟹ 6
```

But for a list with a zero in it, we get a different execution path:

```
(*listC '(1 0 3) id) ⇒
(*listC '(0 3)
  (lambda (n) (* n (id 1)))) ⇒ 0
```

No multiplies are done!

Another Example of Continuations

Let's redo our list multiply example so that if a zero is seen in the list we return a function that computes the product of all the non-zero values and a parameter that is the "replacement value" for the unwanted zero value. The function gives the caller a chance to correct a probable error in the input data.

We create

```
(*list2 L) ≡
 Product of all integers in L if
no zero appears
else
(lambda (n) (* n product-of-all-
nonzeros-in-L)
```

```
(define (*list2 L) (*listE L id))

(define (*listE L con)
  (cond
    ((null? L) (con 1))
    ((= 0 (car L))
      (lambda(n)
       (* (con n)
          (*listE (cdr L) id))))
    (else
     (*listE (cdr L)
      (lambda(m)
       (* m (con (car L))))))
  )
)
```

In the following, we check to see if **`*list2`** returns a number or a function. If a function is returned, we call it with 1, effectively removing 0 from the list

```
(let  ( (V (*list2 L)) )
  (if (number? V)
      V
      (V 1)
  )
)
```

For ordinary lists **`*list2`** expands to a series of multiplies, just like **`*list`** did.

```
(*listE '(1 2 3) id) ⇒
(*listE '(2 3)
  (lambda (m) (* m (id 1)))) ≡
(*listE '(2 3)
  (lambda (m) (* m 1))) ⇒
(*listE '(3)
  (lambda (m) (* m (* 2 1)))) ≡
(*listE '(3)
  (lambda (m) (* m 2))) ⇒
(*listE ()
  (lambda (m) (* m (* 3 2)))) ≡
(*listE () (lambda (n) (* n 6)))
   ⇒ (* 1 6) ⇒ 6
```

But for a list with a zero in it, we get a different execution path:

```
(*listE '(1 0 3) id) ⇒
(*listE '(0 3)
  (lambda (m) (* m (id 1)))) ⇒
(lambda (n) (* (con n)
   (* listE '(3) id))) ≡
(lambda (n) (* (* n 1)
   (* listE '(3) id))) ≡
(lambda (n) (* (* n 1) 3))
```

This function multiplies **n**, the replacement value for 0, by 1 and 3, the non-zero values in the input list.

But note that only one zero value in the list is handled correctly!

Why?

```
(define (*listE L con)
  (cond
    ((null? L) (con 1))
    ((= 0 (car L))
      (lambda(n)
       (* (con n)
          (*listE (cdr L) id))))
    (else
     (*listE (cdr L)
      (lambda(m)
       (* m (con (car L)))))))
  )
)
```