CS403: Programming Languages

Notes on Bindings

Object-orientation

Here is one way to achieve one of the hallmarks of object-orientation, *encapsulation*. The idea is to use the first class nature of functions to provide a wrapper for data and a means for penetrating that wrapper. For example, a constructor for rational numbers in this paradigm might look like:

```
(define (Rational numerator denominator)
    ; the formal parameters numerator and denominator are the members
    ; member functions / selectors go here
    (define type 'Rational)
        (define (convert object)
            (cond
                ((number? object)(Rational object 1))
                (else '())
                )
            )
    (define (show)
        (display numerator) (display "/") (display denominator)
        )
    (define (add b)
        (Rational
           (+ (* numerator (b 'denominator)) (* (b 'numerator) denominator))
              (* denominator (b 'denominator))
              )
        )
    ; function which handles components / selectors / member functions
    ; is returned here
    (define (rat msg . args)
        (cond
            ((eq? msg 'numerator) numerator)
            ((eq? msg 'denominator) denominator)
            ((eq? msg 'convert) (convert (car args)))
            ((eq? msg 'add) (add (car args)))
            ((eq? msg 'type) 'Rational)
            ((eq? msg 'show) show)
            )
        )
    rat
    )
```

Consider the following definition:

```
(define a (Rational 3 4))
```

If we wish to select the numerator, we simply type the expression:

```
(a 'numerator)
```

To select the denominator, we type

```
(a 'denominator)
```

To show a, we need to recall that the ?object" will return the function show; to execute it, we need to evaluate that function like so:

```
((a 'show))
```

Generic operators and object orientation

The *data-driven approach* is one method for writing generic procedures. But rather than installing functions into a table, let's take an object-oriented approach in which the objects carry along their functions with them. Let's look at the operator +. Can we write a generic form of + which not only adds numbers but adds objects (like Rationals and Complexes) as well? Try it! ***Hint: don't use get and put - don't use the table method at all!***

To begin, we save the original version of `+`:

```
(define basic-plus +)
```

Then we need to write an oveload of `+` so that it calls the add function for "objects".

```
(define (+ a b)
    (cond
        ((and (number? a) (number? b))
            (basic-plus a b))
        ((number? a) ; b must be an object
            (b 'add (b 'convert a)))
        ((number? b)
            (a 'add (a 'convert b)))
        ((eq? (a 'type) (b 'type))
            (a 'add b))
        ((a'convert b)
            (a 'add (a'convert b)))
        ((a'convert b)
            (a 'add (a'convert b)))
        )
    )
```

Note that the convert function is used to coerce unlike objects to the same type. As long as every "class" that wishes to overload `+` supplies appropriate *add* and *convert* functions, then our redefinition of `+` is completely general (at least for two arguments). The drawback is that every class has to know about every other class. When adding a new type of number, every other class needs modification, adding the appropriate clauses to the add and convert routines. A better way uses a *data driven approach* . In this approach adding a new type requires no modification of the existing types.

Note also that the "members" of a Rational object are complete walled off from the rest of the world. The only way to access them is through the "access functions". We have thus provided the first leg of the object oriented paradigm, encapsulation. We'll look at the other two legs, polymorphism and inheritance, later.