Data Directed Programming

…let that data flow…

Intro to DDP

- Computers are not intelligent.
- By using different levels of Data Abstraction, we can make different levels of the computer hierarchy act intelligently.
- With Data Directed Programming (DDP) we can make our data "informative" to our "smart" generic procedures, and have it dictate the flow of our program. Instead of having generic data where we apply a generic function, we can have specific data that forces a generic program to act like a **type specific** function.

Example…

- Lets look at an example of our boring generic function perimeter, that cannot be expanded to a generic function.

```
(define (sq-perimeter square)
    (* 4 (get-side square)))
(define (ci-perimeter circle)
    (* pi (get-diameter cirle)))
```

- What's wrong with it?

Example…

- VERY INEFFICIENT!
- Need to know the type of data being passed to each procedure
- Let's create a general function that knows what kind of procedure to apply on a specific shape

Idea of DDP…

- The idea of DDP is that the data is still kind of dumb, but it knows what type it is.

```
(define (tag type data)  (cons type data))
(define (type-tag data)  (car data))
(define (extract-data data)  (cdr data))

(define (make-square side)  (tag side 'square))
(define (make-circle diameter)  (tag diameter 'circle))

(define (generic-perimeter obj)
   (let ((data (extract-data obj)))
       (cond ((equal? (type-tag data) 'square)
                 (* 4 (get-side data)))
                ((equal? (type-tag data) 'circle)
                 (* pi (get-diameter data)))
                (else (error "Bad type of object")))))
```

Idea of DDP…

- Now when we create squares and circles using tag on our data, generic-perimeter will apply the correct perimeter function to the data.
- In other words, our data is *directing* the flow of the function!

More shapes???

- What happens if we added more shapes?
- Well then inside the cond statement we would have to add hundreds of more conditionals and we might mess-up our earlier version of the code.
- Lets make our **procedure** smarter!
- We can store functions (or values) in a table and then simply retrieve the appropriate function depending on the type of the object. Lets look at our table commands:

The Global Table…

- **Table**
  A 2-dimensional table that stores your information.
- **Put**
  Places a value (or function) in a specific slot in your table. It's syntax is (put x-val y-val value). Where value is stored in the "box" (x-val, y-val)
- **Get**
  Gets your entry out of the table and returns the value you stored there (or #f if there is nothing at that point in the table). The syntax is (get x-val y-val). Where you return whatever is in the box at (x-val, y-val) or #f if nothing was stored there

Using the table…

- Now we can just store all the perimeter functions in a table and apply them to the correct arguments.
- Also we can write area functions that apply the correct area function for its shape! Remember, when we return a function we should use lambda!

```
(put 'square 'perimeter (lambda (side) (* side 4)))
(put 'square 'area (lambda (side) (* side side)))
(put 'circle 'perimeter (lambda (diameter) (* pi diameter)))
(put 'circle 'area (lambda (diameter) (* (* (/ diameter 2) (/ diameter 2)) pi)))

(define (perimeter shape) (really-generic shape 'perimeter))
(define (area shape) (really-generic shape 'area))

(define (really-generic shape op-type)
    (let ((procedure (get (type-tag shape) op-type)))
       (if proc
          (proc (extract-data shape))
          (error "Bad type"))))
```

Recap of what we know…

- We’ve seen functional programming.
  - Good for small programs that are easy to follow
- Now we have Tagged Data & DDP
  - Data can now dictate the flow of the program
  - Data is ‘informative’ & procedures are ‘smart’
- So…what’s message passing?

Brief Intro to Message Passing

- Similar idea to data directed programming, but this time data isn’t **just** informative.
- It’s **SMART**! More so a smart procedure!
- So to do something, all we need to do is ask the data to do it.
- So we pass the data a ***message***.

Brief Intro to Message Passing

- What’s a message?
  - Also known as a dispatch
  - An ***interface*** to the to the program, or data
- So you don’t need to know how it’s implemented, just how it works. Aha! Abstraction! ☺

Brief Intro to Message Passing

- Let's take a look at MP in action!

```
(define (number n)
   (lambda (message)
      (cond ((equal? message 'add) (lambda (x) (+ x n)))
               ((equal? message 'sub) (lambda (x) (- n x)))
               ((equal? message 'scale) (lambda (x) (* n x)))
               (else (se '(sorry, I don't know how to ) message)))))

(define (make-imaginary x y)
   (lambda (message)
      (cond ((equal? message 'real) x)
               ((equal? message 'imag) y)
               ((equal? message 'add)
                (lambda (z) (make-imaginary (+ (z 'real) x) (+ (z 'imag) y))))
               ((equal? message 'sub)
                (lambda (z) (make-imaginary (- (z 'real) x) (- (z 'imag) y))))
               ((equal? message 'scale)
                (lambda (n) (make-imaginary (* n x) (* n y)))))))
```

Brief Intro to Message Passing

- So numbers and imaginary numbers know how to add, subtract, and scale instances of themselves from/to other numbers.
- Try it out!
  - (define two (number 2))
  - (define 3+4i (make-imaginary 3 4))
  - two
  - (two 'add)
  - ((two 'add) 4)
  - ((two 'scale) 8)
  - ((3+4i 'add) (make-imaginary 4 5))
  - ((3+4i 'sub) (make-imaginary 4 5))

Brief Intro to Message Passing

- Even though there are completely different ways to add, subtract, etc. different numbers, we can just ask the 'instance' of the type of data to perform the task without knowing how to operate imaginary numbers

- We don't need to know the implementation, but just know what actions the data can perform (interface) and ask the data to perform the task for us.

DDP vs. MP

- **What's a dispatch????**

  With DDP, a dispatch call comes from the call to put, and get. The dispatch returns a procedure to the generic program, which uses the procedure to specifically implement a function. i.e. in DDP a dispatch is the way in which the data tells the generic op what to do. If we want to add another procedure when using DDP, we do not have to change our generic op. All we have to do is give our dispatch another procedure from which to choose. **Remember** DDP means generic op's, and smart data which pick specific procedures using an explicit dispatch.

  With Message Passing, the book often uses the convention that a "dispatch" is the message we pass to our argument. This dispatch is similar to the "dispatch" we use with DDP. However, it is an explicit dispatch. This dispatch (or message) does not depend on the type of data, it depends on the availability of the interface. With MP, you must explicitly call the dispatch procedure to work with data. It is difficult to add procedures for which the dispatch can access, when using MP. We actually have to go into the cond clause and add a new phrase to add functionality to our data.

DDP vs. MP

- Where are the functions stored for each data type?
  - With DDP you store the procedures in a table (using put/get).
  - With Message Passing you store the procedures in the data itself (in the lambda expression).

DDP vs. MP

- Which one's better?

  Well that question is quite type specific. Mostly, we want to know which style will provide us easier access to creating and mutating data/procedures, and which provides us the most encapsulation (well-defined abstraction barrier).
  - With DDP, it is much easier to add procedures. All we use is an explicit call to put, and Bam! our procedure is ready to be called via table lookup (get).
  - Message Passing is easier to use for encapsulation purposes. If someone implements an object (or class of objects) for us, we do not have to know how to call the procedure. All we have to know is the correct usage of such defined procedures.

Next Time…MP→OOP

…don't you love acronyms ☺