CS403: Programming Languages

Conventional Interfaces

Enumeration, filtering, and accumulation

The nature of programming is such that there is pressure to provide languages in which problems can be specified with ever higher constructs. Assemblers replaced machine code and high-level programming languages replaced assembers for that very reason. Generally, the higher-level the language, the more productive are programmers. CASE tools attempt to provide even higher level programming constructs than languages such as C and Java. CASE stands for Computer-aided Software Engineering. Programs are developed within a CASE environment by selecting off the shelf software components and connecting them together to get the final product, much as an electrical engineer constructs a device on a breadboard wiring up discrete electronic components.

A good programming language supports the use of off-the shelf components and a way to glue them together. Scheme is especially suited to this kind of programming. Consider a routine which counts the number of numbers in a (possibly nested) list:

```
(define (count items)
    (cond
        ((null? items)
            0)
        ((list? (car items))
            (+ (count (car items)) (count (cdr items))))
        ((number? (car items))
            (+ 1 (count (cdr items))))
        (else
            (count (cdr items)))
        )
    )
```

Suppose we have *a* bound thusly:

```
(define a '(1 z (2 y 3) x))
```

The expression `(count a)` would yield a value of 3 since there are three numbers in the list. There is another way of solving this problem using a CASE style approach. In such an approach, we design some general components and string them together to solve this particular problem. The difficulty here, as with all CASE approaches, is the "stringing". How do ensure that the output from one module is compatible with another? The solutions is the "conventional interface". For the UNIX system, which uses CASE style in the way utilities are strung together with pipes, the conventional interface is the ASCII or Unicode file. The obvious conventional interface for Scheme is the *list*.

Now that we have settled on an interface, what are the general purpose modules. A great many number of problems can be solved using some combination of these four steps:

- *enumerate* all the possible candidates
- remove (or *filter*) the unacceptable ones
- transform (or *map*) the candidates into a usable form
- *accumulate* the result

In fact, Google's Map-Reduce programming paradigm is built on these modules, with *reduce* being another name for *accumulate*.

For our particular problem, we need to:

- *filter*: remove all the elements which are not numbers
- *map*: replace each remaining element with unity
- *accumulate*: sum the transformed elements

Here is one possible implementation of an enumeration function:

```
(define (enum items)
    (cond
        ((null? items)
            nil)
        ((list? (car items)) (append (enum (car items))
            (enum (cdr items))))
        (else
            (cons (car items) (enum (cdr items))))
        )
    )
```

If we bind *b* to `(enum a)`, the value of *b* becomes `(1 z 2 y 3 x)`. Next, we need to filter out those elements which are not numbers. Here is a filtering routine which takes a predicate *p?*. If the predicate is true for an item, that item is retained in the resulting list.

```
(define (filter p? items)
    (cond
        ((null? items)
            '())
        ((p? (car items))
            (cons (car items) (filter p? (cdr items))))
        (else
            (filter p? (cdr items)))
        )
    )
```

Scheme has a useful predicate *number?* which has an obvious meaning. If we bind *c* thusly:

```
(define c (filter number? b))
```

the value of *c* should be `(1 2 3)`. Now we know that, in this case, we want our final result to be three. The approach mentioned above suggests that we replace each number in *c* with the number one and then sum all the numbers in the resulting list. Obviously, this will produce our desired answer. Here is a definition of map which can be used to accomplish the desired replacement.

```
(define (map f items)
    ((null? items)
        nil)
    (else
        (cons (f (car items)) (map f (cdr items))))
        )
    )
```

We also need to define the function to perform the replacement.

```
(define (cardinality x) 1)
```

We now can bind *d* thusly:

```
(define d (map cardinality c))
```

At this point, the value of *d* should be `(1 1 1)`. Finally we accumulate our result by adding all the numbers in the list together. Here is a routine for accumulating:

```
(define (accum f base items)
    (cond
        ((null? items)
            base)
        (else
            (f (car items) (accum f base (cdr items))))
        )
    )
```

We sum all the ones together by supplying `+` as the function and 0 as the base, as in:

```
(accum + 0 d)
```

Of course, there is no need for the temporary variables. We could have generated the result by making the result of the previous step the input of the current step:

```
(accum + 0 (map cardinality (filter number? (enum a))))
```

This seems like a lot of work but, as in most cases, writing general code takes longer than writing specific code. The benefit is that the general code need only be written once, whereas the specific code must be written again and again. For example, suppose the problem changes to "add all the numbers in list a". We get the result with this expression:

```
(accum + 0 (filter number? (enum a)))
```

If the problem changes to "sum the squares of all the numbers in list a", we can get that result with:

```
(accum + 0 (map square (filter number? (enum a))))
```

If the problem changes to "find all the non-numbers in list a" we can get that result with:

```
(define (nan? x) (not (number? x)))
```

followed by:

```
(filter nan? (enum a))
```

Alternatively, we use a lambda to reduce polluting the namespace:

```
(filter (lambda (x) (not (number? x))) (enum a))
```

The superior student will ponder why:

```
(filter (not number?) (enum a))
```

doesn't work until enlightenment dawns.

Finally, we could have solved this problem by skipping the filtering step and mapping the non-numeric items to 0 and the numeric items to 1 and then accumulating.