Programming languages: Erlang higher order functions

Introduction

In these notes, the notion of higher order functions - functions that can take functions as inputs - is introduced. The primary focus will be on three list processing higher order functions - *map*, *reduce*, and *filter* - that encapsulate common processing strategies.

Idea of higher order functions

In most languages the parameters passed to a function are *data components* – items such as integers, strings, or objects. In Erlang, however, it is also possible to pass *functions* as arguments into a function. In a sense, this allows handling functions in ways in which we would normally handle data. Often times, we are interested in applying a function over an entire list of data, and it is in this case that higher order functions are particularly helpful.

Higher order functions catalog

map function

Consider the *map* function, *map(F, L)* which applies a function passed in as a parameter to map to each element of a list. This function takes a function F and a list L of [data1,data2,data3,. . . ,dataN], the function F is applied to each individual element of the list. and *map* returns the list [F(data1), F(data2), F(data3), . . . , F(dataN)].

The *map* function is defined in Erlang as follows:

```
map(F,[]) ->
        [];
map(F,[X|XS]) ->
        [F(X)|map(F,XS)].
```

An example application of map is as follows:

```
square(X) ->
        X*X;

squareList(L) ->
    map(fun square/1, L).

squareList([1,2,3]).
[1,4,9]
```

To pass a function as a parameter, as was done in the *squareList* function above, we have to reference it in the following way.

```
fun moduleName:functionName/arity
```

The *moduleName* isn't required if using the function within the module, but is if using the function from within the *erl* interpreter shell. Note that *moduleName:functionName/arity* uniquely refers to a function.

***filter* function**

The *filter* function, *filter(F, L)*, takes a predicate function F (a function which returns true/false) and a list L of [data1,data2,data3,...dataN] and returns the list M which contains all elements of L that satisfy the predicate F.

The *filter* function is defined in Erlang as follows:

```
filter(_,[]) ->
        [];
filter(P,[X|XS]) ->
        case P(X) of
            true ->
                [X|filter(P,XS)];
            _ ->
                filter(P,XS)
        end.
```

Note that I introduced a different style of *if, else* testing that we haven't seen before. You may remember that the general *if, else* structure in Erlang will not allow user defined functions to be part of the conditional to prevent side-effects from the conditional. The *case* statement in Erlang does not have these restrictions.

The case statement employs pattern matching instead of true Boolean conditional analysis in deciding which case to execute. The expression occurring after the word case is evaluated, and its result is then pattern matched against each of the values listed in the body of the case. When one matches, the resulting code is executed. In the code above, the predicate function *P()* is applied to *X*. If its result (which is either *true* or *false*) matches *true*, then the item is kept and pre-pended onto the results of filtering with the predicate against the tail of the list. If the predicate returns anything else than true, then the item is dropped and the results of filtering are set to be the results of filtering just the tail of the list.

An example of the use of the filter function is:

```
greaterThan10(X) ->
        X > 10.

dropSmall(L) ->
        filter(fun greaterThan10/1, L).

dropSmall([1,2,12,10,18,8]).
[12,18]
```

Note how just the values in the list greater than 10 are in the result list returned from the function. As another example, a predicate function can be easily written to determine if an integer is odd. If that predicate is applied over a list with the filter function, the returned list will only contain the odd integers.

reduce function

The *reduce* function, *reduce(F, L)*, takes an associative (binary operand) function F and a list L of [data1,data,data3,...,dataN] and returns the result of applying that function in the following manner: $F(data1, F(data2, F(data3, \ldots F(dataN))))$. As an example, assume that the function F was the *addition* operation. Then, *reduce* called with the addition function and an integer list as parameters would return the sum of the integers in the list.

The reduce function can be defined in Erlang as follows:

```
reduce(_, [A]) ->
                    A;
reduce(F, [A|AS]) ->
                    F(A, reduce(F,AS)).
```

Note that the reduce function is undefined for an empty list as there is no data to reduce over.

An example application of reduce is as follows:

```
add(X,Y) ->
        X + Y.

sumList(L) ->
        reduce(fun add/2, L).

sumList([1,2,3,4]).
10
```

Note that the + operator wasn't used as the function parameter to reduce. The + operator, and other infix operators, need to be mapped to a true function.

Also take note of how reduce is defined. The ordering of addition that is done in the above example

```
sumList([1,2,3,4])
```

is

```
1 + (2 + (3 + (4)))
```

since reduce is defined as: *F(X, reduce(F,XS))*.