COMP 161 - Lecture Notes - 18 - Recursion ... again

Spring 2014

In these notes we revisit our old friend recursion and make an attempt at generalizing the recursive design process for problems involving vectors of data.

Generalizing Functional Recursion

The general functional problem still has the same declaration it did when we used iteration. Problems don't change just based on the method of solution.

```
T mystery(vector<S> v, ...)
```

With a recursive strategy we look at the whole vector and assume that a recursive procedure call can solve our problem for all but the first of our vector[1]. As we noted in earlier lecture notes, we lack a proper *rest* selector for the recursion so we need a special helper that let's us specify the lower bound of the index range of the rest of the vector. By doing this we can recursively work with the index range and while doing so, work with vector. This is the same strategy we used with loops where we counted over the index range then processed the vector while we counted. The declaration for our new recursive procedure/problem includes a *fst* parameter for recursing over $[fst, v.size())$.

[1] or some other recursive decomposition we discussed

```
T mystery(vector<S> v, int fst,...)
```

When $fst = 0$ we cover the entire vector with the interval $[fst, v.size())$. We recursively deconstruct this as fst and $[fst + 1, v.size())$. Now we make the inductive assumption that *mystery(v,fst+1,...)* is the partial solution to our problem given all the data in v found in the index range $[fst, v.size())$. Our inductive assumption for iteration was that we could accumulate properly, our assumption for recursion is that *delegating the rest of the work to a recursive call will produce the correct partial result*. In both cases we end up with the need for an *accumulateNext* operation/procedure/function that maps an S value and a T value to another T value. In fact, the purpose of *accumulateNext* is exactly the same in both strategies, to accumulate one more value. The difference in the case of recursion is that the partial solution includes everything but $v[fst]$ and we have only one piece of data left to deal with. This is reflected in how we use *accumulate*, not necessarily what *accumlateNext* does.

We now arrive at the following characterization of *accumulateNext* in the context of recursion. The signature stays the same.

```
accumulateNext : S T -> T
```

The expected behavior is quite different.

$$mystery(v, fst, ...) \equiv accumulateNext(v[fst], mystery(v, fst + 1, ...))$$

Note that using *accumulateNext* in this manner leads to the complete solution.

All that's left is to imagine what happens when there's nothing in the vector. More concretely, we need to decide what T value should be returned if there's no values in the vector. But wait, we're recursing over the $[fst, v.size())$ and not v,so we need to realize that empty means an empty interval of index values. This occurs when $fst = v.size()$ and our interval is $[v.size(), v.size())$. The value we're looking to return for the empty interval is typically the same as the value we used to initialize the accumulator variable in our iterative solution because the underlying questions are the same, "What should I return if there's no data in the vector?" and "What should I accumulate a single item with to get the correct solution?"

Let's take a crack at the template. This time we'll start with the basic version that recurses through $[0, v.size())$ in increments of 1. Note we use the two procedure setup to maintain the original signature for our procedure/problem. The original procedure now just calls the recursive procedure with the appropriate initial value for *fst*.

```
T mystery(vector<S> v, ...){
  return mystery(v,0,...);
}

T mystery(vector<S> v, int fst,...){
  if( fst == v.size() ){
     return mtResult;
  }
  else{
     return accumulateNext(v[fst],mystery(v,fst+1));
  }
}
```

The recursive version, with the index parameter *fst*, utilizes *accumulateNext* as well as the value *mtResult* for the T value to be returned with the interval/vector is empty.

We can generalize this template for different traversal patterns. This time *init* is the first location in the vector we need or want to look at and *next* is a function/operation that produces the appropriate next *fst* value for the recursive function call. We've also adjusted the empty interval test in case *next* skips over *v.size()* rather than landing exactly on it.

```
T mystery(vector<S> v, ...){
  return mystery(v,init,...);
}

T mystery(vector<S> v, int fst,...){
  if( fst >= v.size() ){
     return mtResult;
  }
  else{
     return accumulateNext(v[fst],mystery(v,next(fst)));
  }
}
```

Predicates and Short-Circuiting

We can short circuit recursion for predicates just like we do with iteration. Let's look at the case where *accumulateNext* combines the partial solution with the result of evaluating $v[fst]$ with the predicate f using or (||). Recall that we stop on *true* and return false otherwise.

```
bool mystery(vector<S> v, ...){
  return mystery(v,init,...);
}

bool mystery(vector<S> v, int fst,...){
  if( fst >= v.size() ){
     return false;
  }
  else{
     if( f(v[fst]) ){
        return true;
     }
     else{
       return mystery(v,next(fst));
     }
  }
}
```

It turns out that && and || will short circuit themselves. So, if you do $a||b$ and a is false, then the computer ignores b and produces *true*. Likewise, if you do $a\&\&b$ and a is false, then the computer ignores b and produces *false*. So, let's let || do our short-circuiting for us.

```
bool mystery(vector<S> v, ...){
```

```
  return mystery(v,init,...);
}

bool mystery(vector<S> v, int fst,...){
  if( fst >= v.size() ){
     return false;
  }
  else{
    return f(v[fst]) || mystery(v,next(fst));
  }
}
```

If $f(v[fst])$ is *true,* then the computer won't evaluate the recursion. This is a slight optimization as we avoid explicit branching with another *if..else* statement, but the fact that this avoids work is obfuscated from a reader that is unaware of short-circuiting boolean operators. So, maybe we've traded efficiency for simplicity. You decided.

Recursion for Effect

What about recursion for side-effects? Not a huge change here, just remember to pass the vector by reference if your effect is mutation.

```
void mystery(vector<S> v, ...){
  return mystery(v,init,...);
}

void mystery(vector<S> v, int fst,...){
  if( fst >= v.size() ){
    return;
  }
  else{
    affectNext(v[fst]);
    mystery(v,next(fst),...);
    return;
  }
}
```

The *if* condition that does nothing isn't really a problem, but its definitely unnecessary code. So while it doesn't optimize anything to remove it, we can still re-write the recursive procedure to reduce the amount of code without complicating the logic. The *if* now checks for the non-empty case.

```
void mystery(vector<S> &v, ...){
  return mystery(v,init,...);
```

```
}

void mystery(vector<S> &v, int fst,...){
  if( fst < v.size() ){
     affectNext(v[fst]);
    mystery(v,next(fst),...);
  }
  return;
}
```