Mutual Recursion | Coursera

Mutual Recursion

Mutual Recursion

We may also find it useful to write some functions using *mutual recursion*—two or more functions which call each other. Recall that recursive functions come up when our generalized steps call for us to solve the same problem on parameter values closer to the base case(s). Mutually recursive functions occur when we write one function, find a complex step that we want to abstract out into a second function, then go to write that second function only find a complex step which exactly matches the first function. Here, we again need to be careful to make sure the mutually recursive pair makes progress towards a base case, which does not require recursion—otherwise, we will recurse infinitely and our program will not terminate.

As a (somewhat contrived) example, suppose that we did not have the modulus (%) operator available, and wanted to write a function to figure out if a positive number is even. We might start from the fact that 0 (is even) and 1 (is not even) are easy cases (strictly speaking, we only need one base case: 0), and use the fact that *n* is even if (and only if) *n - 1* is odd. Such reasoning would lead us to the following code:

```
    return 0;
  }
  return isOdd (n - 1); //complicated step: abstract into a function
}
```

We would now need to proceed by writing the isOdd function which we relied on in implementing isEven. In writing that, we might start from the fact that 0 (is not odd) and 1 (is odd) are easy cases, and use the fact that *n* is odd if (and only if) *n - 1* is even. We would then write:

```
int isOdd (unsigned int n) {
  if (n == 0) {
    return 0;
  }
  if (n == 1) {
    return 1;
  }
  return isEven (n - 1); //already have a function to do this step
}
```

These two function are mutually recursive—they call each other. Note that we will need to write the prototype (recall that the prototype for a function tells the name, return type, and argument types without providing the body) for the second function before the first, to let the compiler know about the existence of the second function. The resulting code would look like this:

```c
int isOdd (unsigned int n); //prototype for isOdd
int isEven (unsigned int n) {
  if (n == 0) {
    return 1;
  }
  if (n == 1) {
    return 0;
  }
  return isOdd (n - 1); //complicated step: abstract into a function
}

int isOdd (unsigned int n) {
  if (n == 0) {
    return 0;
  }
  if (n == 1) {
    return 1;
  }
  return isEven (n - 1); //already have a function to do this step
}
```

While this even/odd example is rather contrived (if you want to know if a number is odd or even, it is much more efficient to just test if *n % 2 ==* or not), there are many important uses of mutually recursive functions. One common use is *recursive descent parsing*. Parsing is the process of analyzing input text to determine its meaning. A recursive descent parser is typically written by writing many functions (each of which parses a specific part of the input) which then mutually recurse to accomplish their jobs. We will not go into the details here, but you can imagine the mutually recursive nature by thinking about C functions and their arguments—for example, if you were writing the parser for a C compiler.

At a high level, to parse a function call (*f*(*arg1, arg2,...*)) you would write a function *parseCall*. The *parseCall* would read the function name, and the open parenthesis, then repeatedly call another function, *parseExpression* to parse each argument (which must be an expression)—as well as checking if the argument is followed by a comma or a close parenthesis. The *parseExpression* function itself may encounter a function call, in which case, it would need to call *parseCall*. Such a situation would occur if the text being parsed looked like this *f(3,g(42), h(x,y,z(1)))*—some of the arguments to f are themselves function calls, and in fact, one of the arguments to h is also a function call.

Such a parse often results in a mutually recursively defined data structure—meaning you have two (or more) types which recursively reference each other. Algorithms to operate on mutually recursive data structures typically lend themselves quite naturally to mutually recursive implementations. Of course, mutually recursive data structures may come up in a variety of other contexts as well.

✓

Completed