Introduction to Object-Oriented Programming

Recursion

Christopher Simpkins
chris.simpkins@gatech.edu

Recursion

- A recursive processes or data structure is defined in terms of itself
- A properly written recursive function must
  - handle the base case, and
  - convergence to the base case.
- Failure to properly handle the base case or converge to the base case (divergence) may result in infinite recursion.

The Factorial Function

A mathematical definition: For a non-negative integer n:

$$fac(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ n * fac(n-1) & \text{otherwise} \end{cases}$$

- This definition tells us what a factorial is.
- Defined in cases: a base case and a recursive case

Factorial is defined in terms of itself

A Recursive Factorial Function

> *Mathematics provides a rigorous framework for dealing with notions of* what is*, computation provides a rigorous framework for dealing with notions of* how to*. – SICP*

To translate the mathematical definition of factorial (what a factorial *is* into a computational defiinition (*how to* compute a particular factorial), we need to

- identify the base case(s), and
- figure out how to get our computation to converge to a base case.

For factorial, the solution is straightforward:

```
public static int fac(int n) {
    if (n <= 1) {
        return 1;
    } else {
        return n * fac(n - 1);
    }
}
```

See Fac.java

The Substitution Model of Function Evaluation

- Functions are evaluated in an eval-apply cycle: function arguments are evaluated (which may in turn require function evaluation), then the function is applied to the arguments.
- The substitution model of evaluation is a tool for understanding function evaluation in general, and recursive processes in particular.

Here's `fac(5)`:

```
fac(5)
5 * fac(4)
5 * 4 * fac(3)
5 * 4 * 3 * fac(2)
5 * 4 * 3 * 2 * fac(1)
5 * 4 * 3 * 2 * 1
5 * 4 * 3 * 2
5 * 4 * 6
5 * 24
120
```

Activation Records

- Recursive subprograms cannot use statically allocated local variables, because each instance of the subprogram needs its own copies of local variables
- Most modern languages allocate local variables for functions on the run-time stack.
- The system provides a stack pointer pointing to the next available storage space on the stack.
- Subprogram instances use a frame pointer that points to their activation record, or stack frame, which contains its copies of local variables

Activation Record Example

Consider this simplified example code (type annotations elided for brevity):

```
void main(args) {
  foo();
}
int foo() {
  int r = 3;
  return fac(r);
}
int fac(n) {
  if (n <=1) {
    return 1
  } else {
    return n * fac(n-1)
  }
}
```

The stack just before `fac` returns with 6:

main frame	args = ... in main
foo frame	r = 3 in foo return value (TBD)
fac(3) frame	parameter n = 3 in fac return value (TBD)
fac(2) frame	parameter n = 2 in fac return value (TBD)
fac(1) frame	parameter n = 1 in fac return value (1 by definition)

Stack Overflow

- The run-time stack is finite in size.
- If you put too many activation records on the stack (for example by calling a recursive function with a “large” argument), you will overflow the stack.

```
$ java Fac 10000
facLoop(10000)=0
Exception in thread "main" java.lang.StackOverflowError
  at Fac.facIter(Fac.java:35)
  at Fac.facIter(Fac.java:38)
  at Fac.facIter(Fac.java:38)
...
```

Three ways to deal with this:

- limit input size (brittle – how do you know limit on a particular machine?),
- increase stack size (brittle – how do you know how big), or
- replace recursion with iteration.

Looping is Imperative Recursion

```
public static int facLoop(int n) {
    int factorialAccumulator = 1;
    for (int x = n; x > 0; x--) {
        factorialAccumulator *= x;
    }
    return factorialAccumulator;
}
```

- The base case is the termination condition for the loop.
- The loop variable converges to the termination condition.
- We “accumulate” the answer in the loop.

Recursive definitions are often more natural, but imperative/iterative definitions often perform better.

Tail Recursion - Recursive Iteration

```java
private static int facTail(int n) {
    return facIter(n, 1);
}
private static int facIter(int n, int accum) {
    if (n <= 1) {
        return accum;
    } else {
        return facIter(n - 1, n * accum);
    }
}
```

Tail call optimization creates an iterative, rather than a recursive process:

```
facTail(5);
facIter(5, 1);
facIter(4, 5);
facIter(3, 20);
facIter(2, 60);
facIter(1, 120);
120
```

Note: Java does not optimize tail calls, but many languages do.
ncluding all functional languages) do.

Closing Thoughts

- Remember: A properly written recursive function must
  - handle the base case, and
  - convergence to the base case.
- Today we learned recursive processes. We'll also learn recursive data structures.