CS416

Introduction to Computer Science II

Spring 2018

6b Recursion Mechanics

- Activation Records
- Control Flow
- Tail Recursion
- Recap: Iteration vs. Recursion

Activation Records

- As a program executes, methods call other methods.
- As each method is called, the system creates an **activation record**
- Activation record contains the <u>parameter</u> variables and <u>local</u> variables of the called method.

```
public int fact( int n )
{
  int ans = 1;
  if ( n > 1 )
    ans = n * fact( n-1 );
  return ans;
}
```

fact(3)

fact(3)

fact(3) AR

n: 3, ans: 1;

The values of the variables change during execution; these are the values when the *if* statement is executed

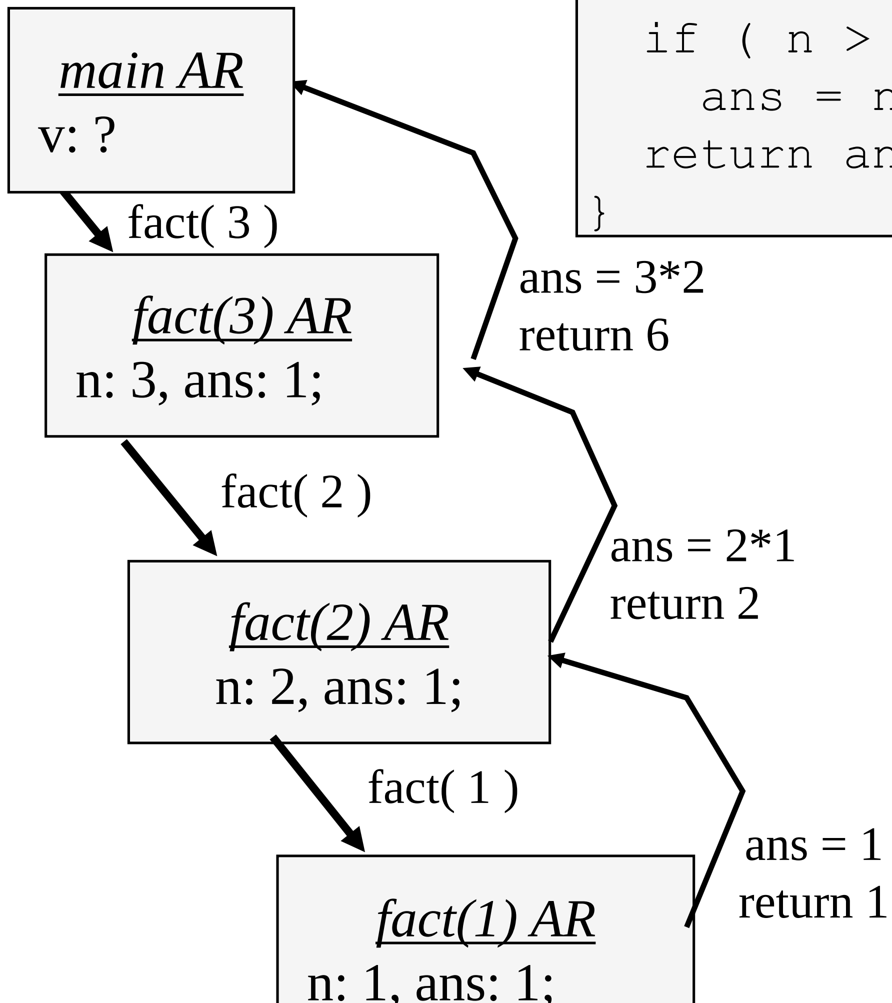
Recursion Mechanics

```
public ... main(...)
...
  int v = fact(3);}
```

```
public int fact(int n)
{
  int ans = 1;
  if ( n > 1 )
    ans = n*fact(n-1);
  return ans;
}
```

How does it work?

- Each "activation" of a method creates an activation record
- An Activation Record contains
  - reference to caller
  - space for local variables
  - space for copies of parameters
- *return* from a method
  - deletes AR
  - restores caller's AR



More Activation Records

- Each method call gets it own activation record containing its parameters and local variables.
- There is another important role of the activation record:

It helps maintain correct execution <u>control flow</u>

Execution Control Flow

- Statement execution flow of control within a method is determined by *control* statements in the code.
- However, as a program executes, methods call other methods.
- When a method is called, flow of control is suspended in the calling method, and control is passed to the called method.

Activation Records and Control Flow

- How does the system keep track of where execution was suspended so that it can be resumed in the proper place?
- The **caller's** current position of execution, the *program counter (PC)*, is saved in the activation record of the **called** method.
- When the called method returns, this position is used to resume execution in the calling method.

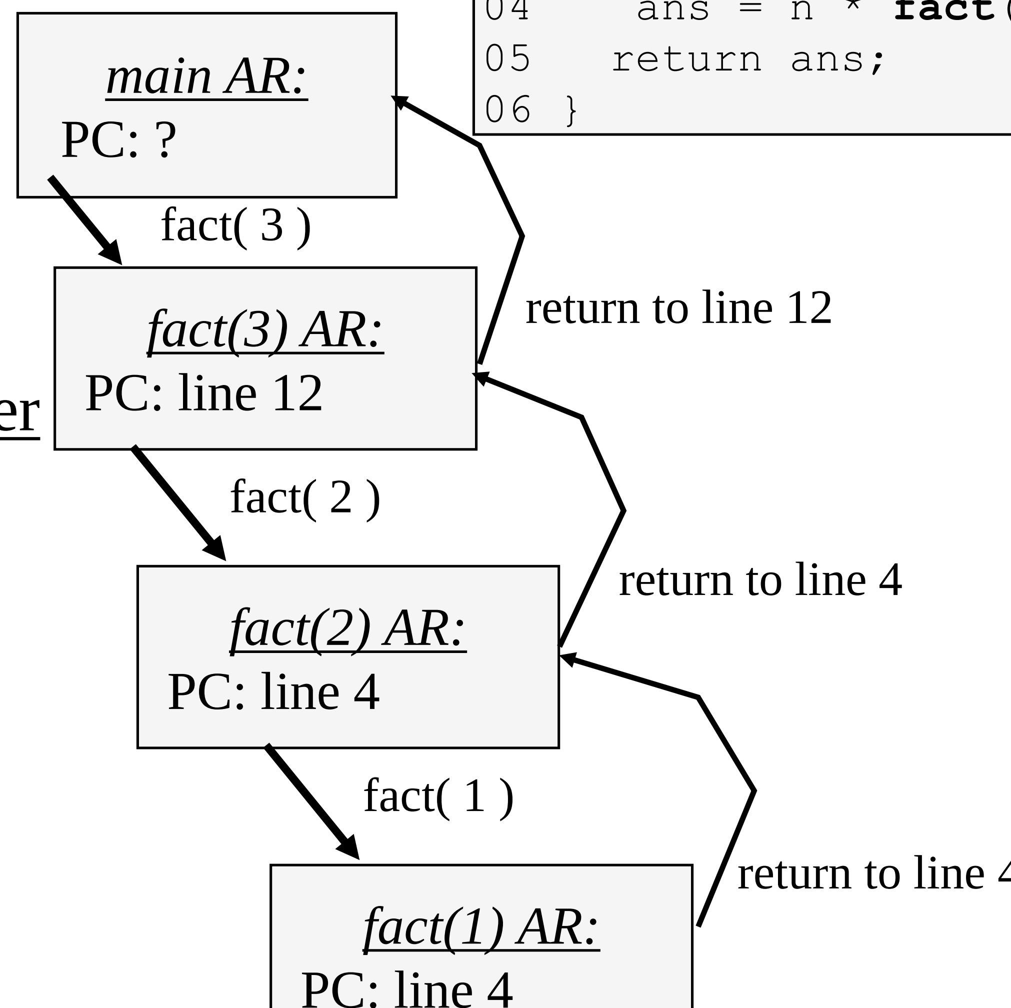
Control Flow Mechanics

```
00 public static void main(...)
...
12  int v = fact(3);
...
```

```
00 public int fact( int n )
01 {
02  int ans = 1;
03  if ( n > 1 )
04    ans = n * fact( n-1 );
05   return ans;
06 }
```

How does it work?

- Each "activation" of a method creates an activation record
- Activation Record contains
  - Program counter <u>of the caller</u>
- *return* from a method
  - deletes current AR
  - restores caller's AR
  - resumes execution at the program counters value



When an AR is removed, what AR should become active?

- When a method returns and its AR is deleted, the AR for the *calling* method should become active.
- Remember the Stack data structure:
  - Last In, First Out: LIFO
- The system stores AR's in a stack
  - a new method's AR is pushed onto the stack
  - when a method returns, its AR is popped
  - the calling method's AR is on top of the stack

Tail Recursion

- If there is no code to be executed in a function after the recursive call we call it *tail recursive*

```
// tail recursive
public static void print1( int n )
{
  if( n != 0 )
  {
     System.out.print( n % 10 + " " );
     print1( n / 10 );
  }
}
```

When recursive call returns, this recursion returns and so does the previous and so forth up to the initial call to *print1*

```
// non-tail recursive
public static void print2( int n )
{
  if( n != 0)
  {
     print2( n / 10 );
     System.out.print( n % 10 );
  }
}
```

print statement is done after recursion

What's the Difference?

- A tail recursive function does <u>nothing</u> after the recursive call -- we say that it does nothing “on the way back <u>up</u>” the recursion
- A non-tail recursive function does <u>something</u> “on the way back up”

What's the Difference?

- Remember, a stack reverses order:
  - push: 1 2 3   pop: 3 2 1
- A tail recursive function does not rely on the stack to do any work for it, a non-tail recursive function usually does.
- One result of this is that *it is very easy to convert a tail recursive function to a non-recursive, iterative version.*
- Many compilers convert tail-recursive code to iterative versions automatically.
  - Benfits of recursive code, without limits of stack size!

Tail-recursive or not?

```
public static int mod( int v, int d )
{
  if ( v < d )
    return v;
  else
    return mod( v - d, d );
}
```

Tail-recursive or not?

```
public static int mod( int v, int d )
{
  if ( v < d )
    return v;
  else
    return mod( v - d, d );
}
```

Yes, this is tail-recursive

Convert *mod* recursive method to non-recursive

- One or more method parameters may become local variables.
- The recursive call becomes a loop

```
public static int mod( int v, int d )
{
  if ( v < d )
     return v;
  else
     return mod( v - d, d );
}
```

```
public static int mod( int v, int d )
{
  while ( v >= d )
  {
    v = v - d;
  }
  return v;
}
```

The value of v at each new recursion level is v - d. This is the key component of the loop body in non-recursive version

The recursion *termination* condition (v < d) transforms to its complement as the *continuation* condition in non-recursive version

Tail-recursive or not?

```
public static String reverse( String word )
{
  if ( word.length() == 0 )
    return word;
  else
  {
     first = word.substring( 0, 1 ); // first char
     rest  = word.substring( 1 );    // all but 1st
     return reverse( rest ) + first;
  }
}
```

Not tail-recursive:
appends first *after* recursion

Convert non-Tail Recursive to Tail Recursive

- Non-tail recursive can often be converted to tail recursive
  - Rather than building the answer on the way back up, we want to build it on the way down.
  - This can often be done by adding a new parameter to pass the interim calculation *down* the recursion; the final calculation is then returned from the bottom unchanged.
  - Sometimes this requires a "helper" method.

Make *reverse* tail recursive

```
public static String reverse( String word ) // not tail recursive
{
  if ( word.length() == 0 )
    return word;
  else
    return reverse( word.substring(1) ) + word.charAt(0);
}
```

```
public static String reverse( String word ) // revised reverse
{
  return reverseTail( word, "" );  // invoke "helper" method
}
```

Result value starts as empty string

```
// helper method has the extra parameter and is tail recursive
private static String reverseTail( String word, String rWord )
{
  if ( word.length() == 0 )
    return rWord;
  else
    return reverseTail( word.substring(1), word.charAt(0) + rWord );
}
```

Build reverse string as recursion proceeds and pass it along in the recursive call

At the end, final result is just passed back

Now, convert tail-recursive *reverse* to non-recursive

- The tail-recursive method's parameter becomes a local variable. The recursive call becomes a loop

```
private static String reverseTail( String word, rWord )
{
  if ( word.length() == 0 )
    return rWord;
  else
    return reverseTail( word.substring(1), word.charAt(0) + rWord );
}
```

```
public static String iterativeReverse( String word )
{
  String rWord = "";
  while( word.length() > 0 )
  {
     rWord = word.charAt(0) + rWord;
     word  = word.substring(1);
  }
  return rWord;
}
```

Note the complementary conditions

Changes to recursive parameters become update of iterative variables

Iteration vs. Recursion

- Any recursive solution can be done iteratively
  - *tail recursion* -- recursion that occurs at the end of the execution of the recursive method
    - this is so easy to convert that many compilers automatically convert a tail recursive method to an iterative one
    - if there are few local variables, it’s also very easy for a user
  - Other forms of recursion can also be done iteratively
- Any iterative solution can be done recursively
  - Turn *for* loop body into the recursive method with the loop index as a parameter (along with other needed stuff)
  - A *while* loop body can also become a recursive method

Iteration vs. Recursion

- It's a *design* and *performance* decision
  - Ease of coding?
    - In general, this is the most important criterion
  - Ease of testing?
    - Need to be extra careful with recursive solutions
  - Computational and memory resources
    - Recursive solutions can be less efficient

Ease of Coding

- Factorial
  - recursive factorial is simple
  - but so is iterative factorial

```
public int fact(int n)
{
  int ans = 1;
  for (int i=2; i<=n; i++)
     ans *= i;
}
```

- Anagram
  - recursive version took a little thought, but is nice
  - can you think of a reasonable iterative version?
- Towers of Hanoi
  - straightforward recursive solution
  - what would an iterative one be like?

Ease of Testing

- Some people think it’s easier to debug iterative solutions
- Dangers of recursive solutions are often easy to moderate with just a little care:
  - Be sure to identify the *base case*
  - Be sure that the *recursion step* reduces the problem domain and guarantees convergence to the base case
- A clean elegant recursive solution will be easier to debug than a messy complicated iterative one!

Performance Issues

- Computational resources
  - Creating activation records is expensive compared to doing a loop iteration; might be a factor in huge problems
- Memory resources
  - Recursive solutions are limited by stack size (small), where iterative solutions are limited by heap size (bigger)
- Key: is recursion overhead significant in application
  - Can you make it tail-recursive for the compiler to convert to iteration?

SpiralApp and *TreeApp*

- Review the *SpiralApp* and *TreeApp* examples in text
  - Recursion in both is based on drawing lines, such that the angle between each successive pair changes by a fixed amount <u>and</u> the length of each line is shorter than the last.
  - The *base case* is a line that is 3 pixels long; when that happens the recursion stops.
  - *TreeApp* has 2 new recursive paths spawned at each step
    - The *base case*, therefore must be reached along all paths.

Review

- Recursion is a powerful programming tool
  - Can make some programs much simpler to write
  - Which makes them easier to test and maintain
- Some performance overhead, but isn't always critical

Next, in 416

- Data structures
  - Lists, stacks, queues
  - Trees & graphs
    - Especially good models for recursion!