Understanding the Stack

Introduction

Assembly languages provide you the ability to implement conditional statements. If you add and jump/branches (the assembly equivalent of "goto" statements), you can implement loops.

The next step is some kind of support for functions. Functions are perhaps the most fundamental language feature for abstraction and code reuse. It allows us to refer to some piece of code by a name. All we need to know is how many arguments are needed, what type of arguments, and what the function returns, and what the function computes to use the function.

In particular, it's not necessary to know how the function does what it does.

How does assembly language give us this kind of support.

To think about what's required, let's think about what happens in a function call.

- When a function call is executed, the arguments need to be evaluated to values (at least, for C-like programming languages).
- Then, control flow jumps to the body of the function, and code begins executing there.
- Once a return statement has been encountered, we're done with the function, and return back to the function call.

Programming languages make functions easy to maintain and write by giving each function its own section of memory to operate in.

For example, suppose you have the following function.

```
int pickMin( int x, int y, int z ) {
   int min = x ;
   if ( y < min )
      min = y ;
   if ( z < min )
      min = z ;
   return min ;
 }
```

You declare parameters **x**, **y**, and **z**. You also declare local variables, **min**. You know that these variables won't interfere with other variables in other functions, even if those functions use the same variable names.

In fact, you also know that these variables won't interfere with separate invocations of itself.

For example, consider this recursive function,

```
int fact( int n ) {
   if ( n == 0 )
      return 1 ;
   else
      return fact( n - 1 ) * n ;
 }
```

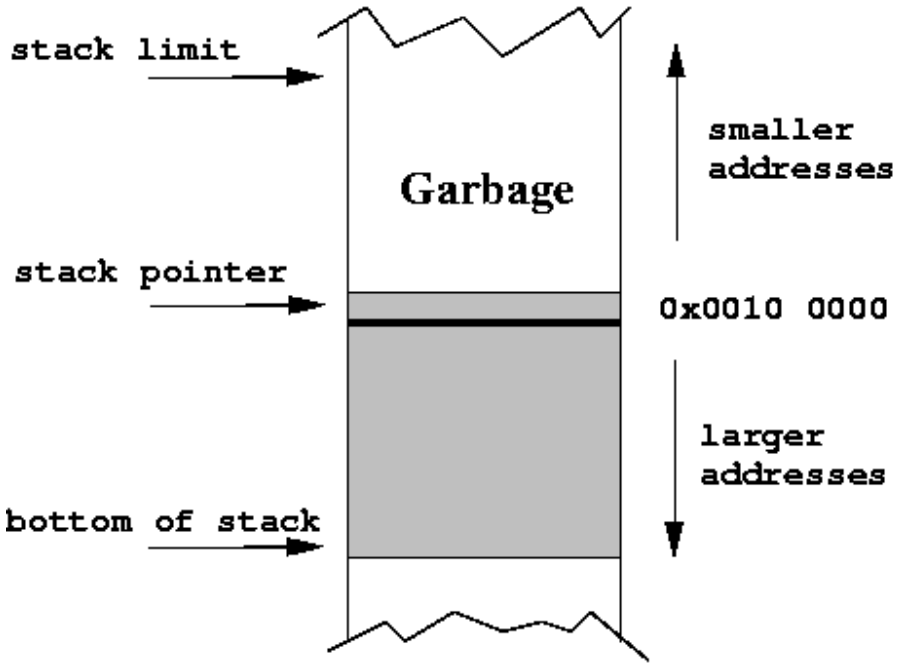
Each call to **fact** produces a new memory location for **n**. Thus, each separate call (or invocation) to **fact** has its own copy of **n**.

How does this get implemented? In order to understand function calls, you need to understand the stack, and you need to understand how assembly languages like MIPS deal with the stack.

Stack

When a program starts executing, a certain contiguous section of memory is set aside for the program called the **stack**.

Let's look at a stack.



The stack pointer is usually a register that contains the top of the stack. The stack pointer contains the smallest address **x** such that any address smaller than **x** is considered garbage, and any address greater than or equal to **x** is considered valid.

In the example above, the stack pointer contains the value **0x0000 1000**, which was somewhat arbitrarily chosen.

The shaded region of the diagram represents valid parts of the stack.

It's useful to think of the following aspects of a stack.

- **stack bottom** The largest valid address of a stack. When a stack is initialized, the stack pointer points to the stack bottom.
- **stack limit** The smallest valid address of a stack. If the stack pointer gets smaller than this, then there's a *stack overflow* (this should not be confused with overflow from math operations).

Other sections of memory are used for the program and for the heap (the section of memory used for dynamic memory allocation).

Push and Pop

Like the data structure by the same name, there are two operations on the stack: push and pop.

Usually, push and pop are defined as follows:

- **push** You can push one or more registers, by setting the stack pointer to a smaller value (usually by subtracting 4 times the number of registers to be pushed on the stack) and copying the registers to the stack.

- **pop** You can pop one or more registers, by copying the data from the stack to the registers, then to add a value to the stack pointer (usually adding 4 times the number of registers to be popped on the stack)

Thus, pushing is a way of saving the contents of the register to memory, and popping is a way of restoring the contents of the register from memory.

MIPS support for push and pop

Some ISAs have an explicit **push** and **pop** instruction. However, MIPS does not. However, you can get the same behavior as **push** and **pop** by manipulating the stack pointer directly.

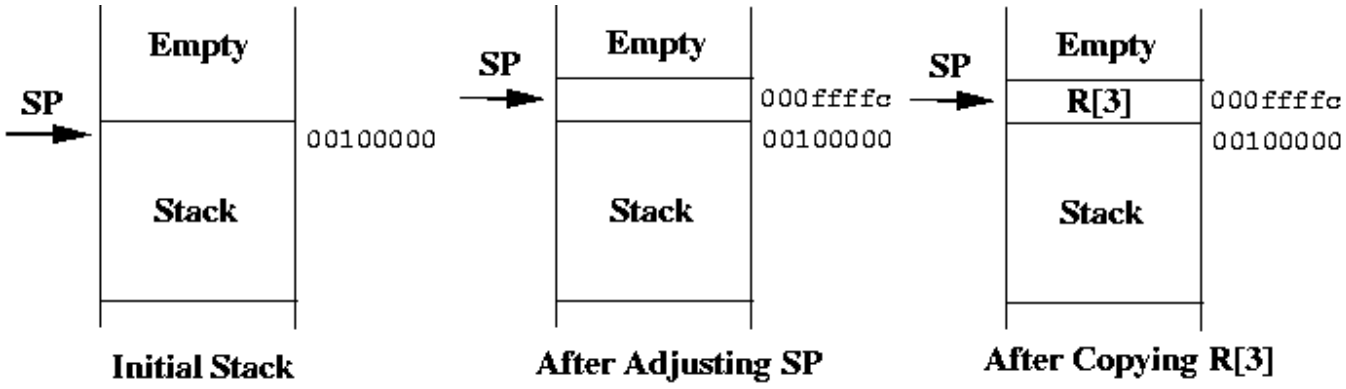
The stack pointer, by convention, is $r29. That is, it's register 29. We say, "by convention", because MIPS ISA does not force you to use register 29. However, because it's a convention, you should follow it, if you expect your code to work with other pieces of code.

Here's how to implement the equivalent of **push $r2** in MIPS, which is to push register **$r2** onto the stack.

```
push:  addi $sp, $sp, -4   # Decrement stack pointer by 4
       sw   $r3, 0($sp)    # Save $r3 to stack
```

The label "push" is unnecessary. It's there just to make it easier to tell it's a push. The code would work just fine without this label.

Here's a diagram of a push operation.



The diagram on the left shows the stack before the push operation.

The diagram in the center shows the stack pointer being decremented by 4.

The diagram on the right shows the stack after the 4 bytes from register 3 has been copied to address **0x000f fffc**

You might wonder why it's necessary to update the stack pointer. Couldn't you just do the following?

```
push:  sw $r3, -4($sp)    # Copy $r3 to stack
```

Certainly, this is equivalent in behavior as far as register 3 being saved to the stack.

However, we'd like to maintain the invariant that all addresses greater than or equal to the stack pointer hold valid data, and all the addresses less than the stack pointer hold invalid data. If we ran the above code, then there would be valid data at an address smaller than the stack pointer.

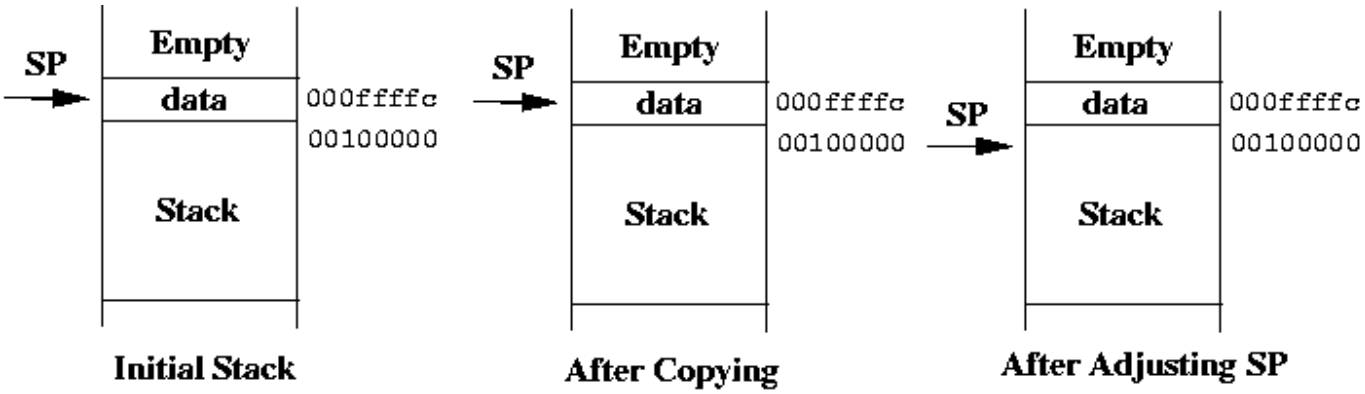
It just makes bookkeeping that much easier if we follow these conventions.

Popping off the stack

Popping off the stack is the opposite of pushing on the stack. First, you copy the data from the stack to the register, then you adjust the stack pointer.

```
pop:  lw   $r3, 0($sp)    # Copy from stack to $r3
      addi $sp, $sp, 4    # Increment stack pointer by 4
```

Here's a diagram of the pop operation.



The diagram on the left is the initial state of the stack.

The data is copied from the stack to the register 3. Thus, the diagram in the center is the same as the one on the left. If I had drawn a picture of the contents of register 3, it would have been updated with the data from the stack.

Then, the stack pointer is moved down (shown in the diagram on the right).

As you can see, the data still is on the stack, but once the pop operation is completed, we consider that part of the data invalid. Thus, the next push operation overwrites this data. But that's OK, because we assume that after a pop operation, the data that's popped off is considered garbage.

If you've ever made the error of returning a pointer to a local variable or to a parameter that was passed by value and wondered why the value stayed valid initially, but later on got corrupted, you should now know the reason.

The data still stays on the garbage part of the stack until the next push operation overwrites it (that's when the data gets corrupted).

Pushing and Popping, Part 2

It turns out you don't have to adjust the stack pointer each time you want to save a register to the stack. If you know you are going to push several registers to the stack, you can adjust the stack pointer at once.

For example, suppose you want to push registers $r2, $r3, and $r2 on the stack. This is code to do that:

```
push:  addi $sp, $sp, -12  # Decrement stack pointer by 12
       sw   $r2, 0($sp)    # Save $r2 to stack
       sw   $r3, 4($sp)    # Save $r3 to stack
       sw   $r4, 8($sp)    # Save $r4 to stack
```

Since each register takes up 4 bytes and since each memory address stores 1 byte, we need to decrement the value of the stack pointer by 12 to give us the space to store 3 registers.

At this point, we can copy the contents of registers 2, 3, and 4 to the stack. It's somewhat arbitrary which order we put the registers on the stack. In the code above, register 2 is on the very top of the stack. We could just have easily put register 3 or 4 on the top of the stack. The way shown above just seems more "orderly".

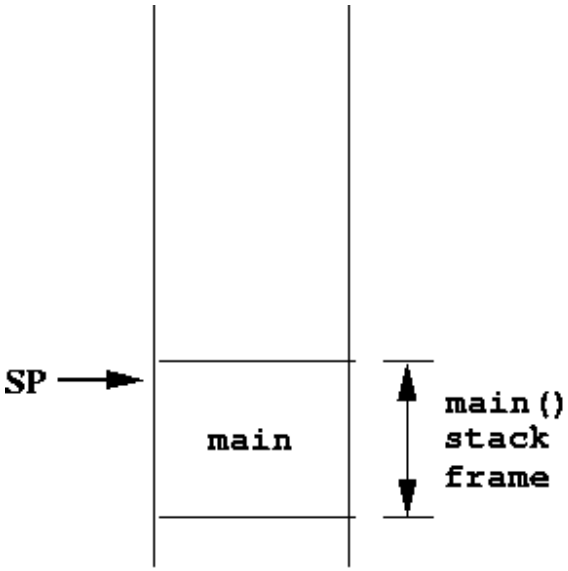
Popping is very similar.

```
pop:  sw   $r2, 0($sp)   # Copy from stack to $r2
      sw   $r3, 4($sp)   # Copy from stack to $r3
      sw   $r4, 8($sp)   # Copy from stack to $r4
      addi $sp, $sp, 12  # Increment stack pointer by 12
```

Stacks and Functions

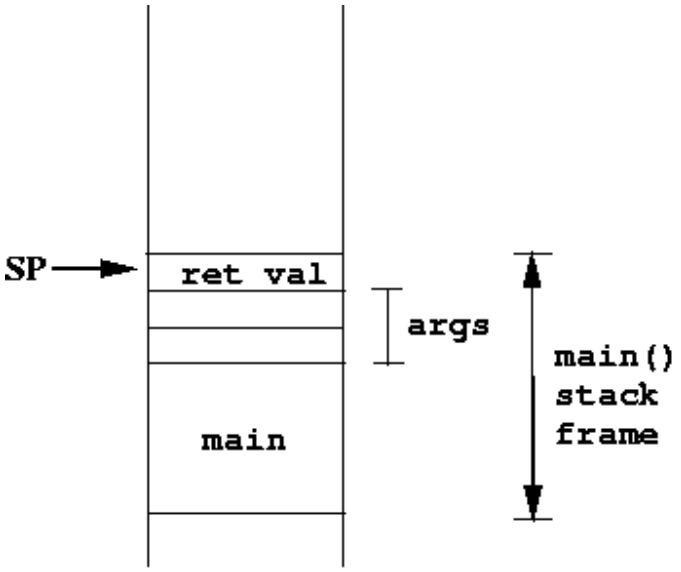
Let's now see how the stack is used to implement functions. For each function call, there's a section of the stack reserved for the function. This is usually called a *stack frame*.

Let's imagine we're starting in **main()** in a C program. The stack looks something like this:



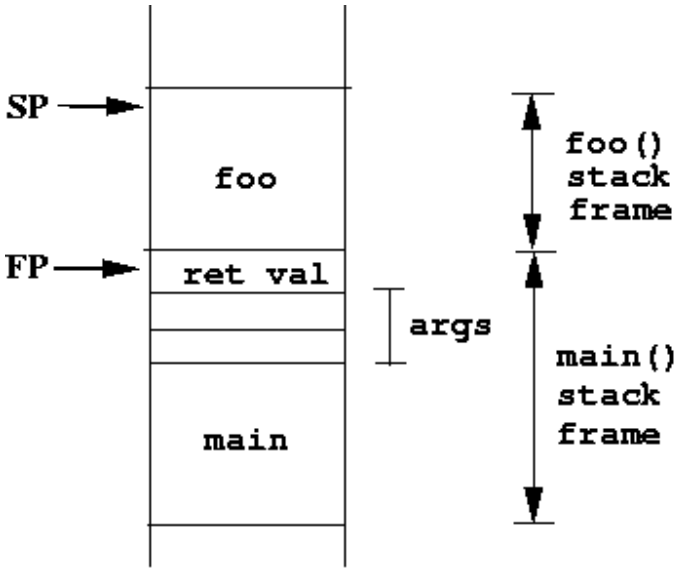
We'll call this the *stack frame* for **main()**. It is also called the *activation record*. A stack frame exists whenever a function has started, but yet to complete.

Suppose, inside of body of **main()** there's a call to **foo()**. Suppose **foo()** takes two arguments. One way to pass the arguments to **foo()** is through the stack. Thus, there needs to be assembly language code in **main()** to "push" arguments for **foo()** onto the the stack. The result looks like:



As you can see, by placing the arguments on the stack, the stack frame for **main()** has increased in size. We also reserved some space for the return value. The return value is computed by **foo()**, so it will be filled out once **foo()** is done.

Once we get into code for **foo()**, the function **foo()** may need local variables, so **foo()** needs to push some space on the stack, which looks like:



foo() can access the arguments passed to it from **main()** because the code in **main()** places the arguments just as **foo()** expects it.

We've added a new pointer called **FP** which stands for *frame pointer*. The frame pointer points to the location where the stack pointer *was*, just before **foo()** moved the stack pointer for **foo()**'s own local variables.

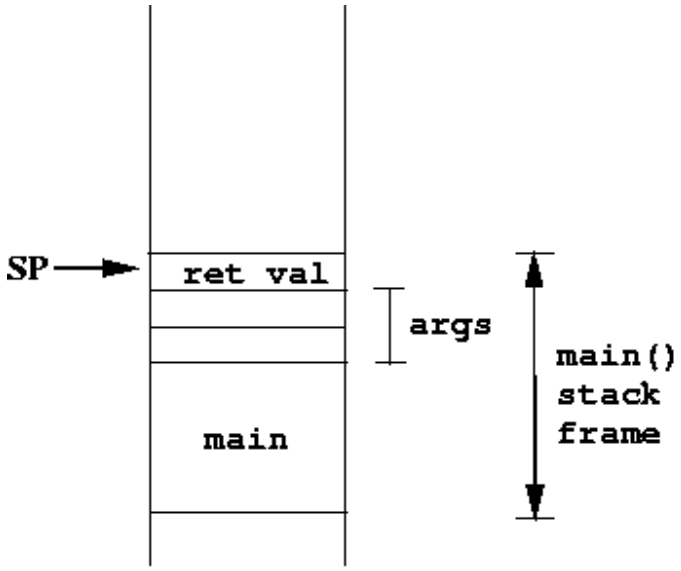
Having a frame pointer is convenient when a function is likely to move the stack pointer several times throughout the course of running the function. The idea is to keep the frame pointer fixed for the duration of **foo()**'s stack frame. The stack pointer, in the meanwhile, can change values.

Thus, we can use the frame pointer to compute the locations in memory for both arguments as well as local variables. Since it doesn't move, the computations for those locations should be some fixed offset from the frame pointer.

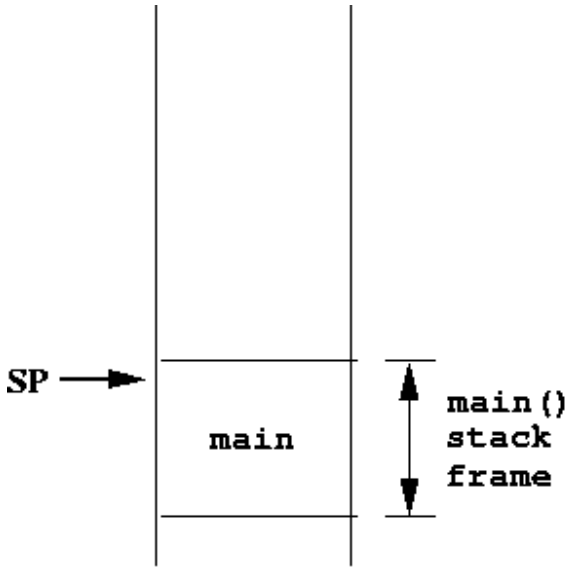
And, once it's time to exit **foo()**, you just have to set the stack pointer to where the frame pointer is, which effectively pops off **foo()**'s stack frame. It's quite handy to have a frame pointer.

We can imagine the stack growing if **foo()** calls another function, say, **bar()**. **foo()** would push arguments on the stack just as **main()** pushed arguments on the stack for **foo()**.

So when we exit **foo()** the stack looks just as it did before we pushed on **foo()**'s stack frame, except this time the return value has been filled in.



Once **main()** has the return value, it can pop that and the arguments to **foo()** off the stack.



Recursive Functions

Surprisingly enough, there's very little to say about recursive functions, because it behaves just as non-recursive functions do. To call a recursive function, push arguments and a return value on the stack, and call it like any other function.

It returns back the same way as well.

Stack Overflow

While stacks are generally large, they don't occupy all of memory. It is possible to run out of stack space.

For example, consider the code we had for **factorial**.

```
int fact( int n ) {
   if ( n == 0 )
      return 1 ;
   else
      return fact( n - 1 ) * n ;
 }
```

Suppose **fact(-1)** is called. Then, the base case is never reached (well, it might be reached once we decrement so far that **n** wraps around to 0 again). This causes one stack frame after another to be pushed. Once the stack limit has been reached, we enter into invalid memory addresses, and the operating system takes over and kills your programming, telling you your program has a stack overflow.

Probably the most common cause of stack overflow is a recursive function that doesn't hit the base case soon enough. For fans of recursion, this can be a problem, so just keep that in mind.

Some languages (say, ML) can convert certain kinds of recursive functions (called "tail-recursive" functions) into loops, so that only a constant amount of space is used.

How MIPS Does It

In the previous discussion of function calls, we said that arguments are pushed on the stack and space for the return value is also pushed.

This is how CPUs used to do it. With the RISC revolution (admittedly, nearly 20 years old now) and large numbers of registers used in typical RISC machines, the goal is to (try and) avoid using the stack.

Why? The stack is in physical memory, which is RAM. Compared to accessing registers, accessing memory is much slower---probably on the order of 100 to 500 times as slow to access RAM than to access a register.

MIPS has many registers, so it does the following:

- There are four registers used to pass arguments: **$a0**, **$a1**, **$a2**, **$a3**.
- If a function has more than four arguments, or if any of the arguments is a large structure that's passed by value, then the stack is used.
- There must be a set procedure for passing arguments that's known to everyone based on the types of the functions. That way, the caller of the function knows how to pass the arguments, and the function being called knows how to access them. Clearly, if this protocol is not established and followed, the function being called would not get its arguments properly, and would likely compute bogus values or, worse, crash.
- The return value is placed in registers **$v0**, and if necessary, in **$v1**.

In general, this makes calling functions a little easier. In particular, the calling function usually does not need to place anything on the stack for the function being called.

However, this is clearly not a panacea. In particular, imagine **main()** calls **foo()**. Arguments are passed using **$a0** and **$a1**, say.

What happens when **foo()** calls **bar()**? If **foo()** has to pass arguments too, then by convention, it's supposed to pass them using **$a0** and **$a1**, etc. What if **foo()** needs the argument values from **main()** afterwards?

To prevent its own arguments from getting overwritten, **foo()** needs to save the arguments to the stack.

Thus, we don't entirely avoid using the stack.

Leaf Procedures

In general, using registers to pass arguments and for return values doesn't prevent the use of the stack. Thus, it almost seems like we postpone the inevitable use of the stack. Why bother using registers for return values and arguments?

Eventually, we have to run code from a leaf procedure. This is a function that does not make any calls to any other functions. Clearly, if there were no leaf procedures, we wouldn't exit the program, at least, not in the usual way. (If this isn't obvious to you, try to think about why there must be leaf procedures).

In a leaf procedure, there are no calls to other functions, so there's no need to save arguments on the stack. There's also no need to save return values on the stack. You just use the argument values from the registers, and place the return value in the return value registers.

Stackless Languages?

Is the stack absolutely necessary? FORTRAN 77 (77 refers to the year 1977) did not use a stack. Basically, each function had a section of memory for its own arguments and data. Because there were no stacks, you couldn't write recursive functions in FORTRAN 77. However, it goes to show you that a stack isn't always needed.

There's also a complicated language feature called *continuations* (which appear in languages like **Scheme**). *Continuations* can be coded in a way that doesn't require a stack, and in effect, acts like a queue.

Nevertheless, the vast majority of languages that implement functions use the stack.

Heaps?

What about heaps? The heap is a section of memory used for dynamic memory allocation. Heaps are more difficult to implement. Typically, you need some code that manages the heap. You request memory from the heap, and the heap code must have data structures to keep track of which memory has been allocated.

The code can be written in, say, C in a few hundred lines of code, but is rather unwieldy to write in assembly language.

Notice that the heap in memory is not the same as the heap data structure. Those two are different kinds of heaps, and aren't really related.

Summary

To understand how functions work, you need to understand the behavior of the stack.

We've discussed how arguments and return values are used in the stack, and how each function call results in a stack frame being pushed on the stack. We've also talked about how a stack can overflow, typically, from a recursive function that's either got a bad base case or unexpected arguments. Occasionally, stack overflow can just occur because the stack just grows too big.