This morning, I was reading [Steve Yegge's: When Polymorphism Fails](http://steve.yegge.googlepages.com/when-polymorphism-fails), when I came across a question that a co-worker of his used to ask potential employees when they came for their interview at Amazon.

As an example of polymorphism in action, let's look at the classic "eval" interview question, which (as far as I know) was brought to Amazon by Ron Braunstein. The question is quite a rich one, as it manages to probe a wide variety of important skills: OOP design, recursion, binary trees, polymorphism and runtime typing, general coding skills, and (if you want to make it extra hard) parsing theory.

At some point, the candidate hopefully realizes that you can represent an arithmetic expression as a binary tree, assuming you're only using binary operators such as "+", "-", "\*", "/". The leaf nodes are all numbers, and the internal nodes are all operators. Evaluating the expression means walking the tree. If the candidate doesn't realize this, you can gently lead them to it, or if necessary, just tell them.

Even if you tell them, it's still an interesting problem.

The first half of the question, which some people (whose names I will protect to my dying breath, but their initials are Willie Lewis) feel is a Job Requirement If You Want To Call Yourself A Developer And Work At Amazon, is actually kinda hard. The question is: how do you go from an arithmetic expression (e.g. in a string) such as "2 + (2)" to an expression tree. We may have an ADJ challenge on this question at some point.

The second half is: let's say this is a 2-person project, and your partner, who we'll call "Willie", is responsible for transforming the string expression into a tree. You get the easy part: you need to decide what classes Willie is to construct the tree with. You can do it in any language, but make sure you pick one, or Willie will hand you assembly language. If he's feeling ornery, it will be for a processor that is no longer manufactured in production.

You'd be amazed at how many candidates boff this one.

I won't give away the answer, but a Standard Bad Solution involves the use of a switch or case statement (or just good old-fashioned cascaded-ifs). A Slightly Better Solution involves using a table of function pointers, and the Probably Best Solution involves using polymorphism. I encourage you to work through it sometime. Fun stuff!

So, let's try to tackle the problem all three ways. How do you go from an arithmetic expression (e.g. in a string) such as "2 + (2)" to an expression tree using cascaded-if's, a table of function pointers, and/or polymorphism?

Feel free to tackle one, two, or all three.

***Floor 1:***

The tests are just building up the binary trees by using constructors.

program structure:

abstract base class: Node

* all Nodes inherit from this class

abstract base class: BinaryNode

* all binary operators inherit from this class
* process method does the work of evaluting the expression and returning the result

binary operator classes: Plus,Minus,Mul,Div

* two child nodes, one each for left side and right side subexpressions

number class: Num

* holds a leaf-node numeric value, e.g. 17 or 42

***Floor 2:***

Representing the Nodes

If we want to include parentheses, we need 5 kinds of nodes:

* the binary nodes: Add Minus Mul Div  
  these have two children, a left and right side

+

/ \

node node

* a node to hold a value: Val  
  no children nodes, just a numeric value
* a node to keep track of the parens: Paren  
  a single child node for the subexpression

( )

|

node

For a polymorphic solution, we need to have this kind of class relationship:

* Node
* BinaryNode : inherit from Node
* Plus : inherit from Binary Node
* Minus : inherit from Binary Node
* Mul : inherit from Binary Node
* Div : inherit from Binary Node
* Value : inherit from Node
* Paren : inherit from node

There is a virtual function for all nodes called eval(). If you call that function, it will return the value of that subexpression.

***Floor 3:***

Hm... I don't think you can write a top-down parser for this without backtracking, so it has to be some sort of a shift-reduce parser. LR(1) or even LALR will of course work just fine with the following (ad-hoc) language definition:

Start -> E1   
E1 -> E1+E1 | E1-E1   
E1 -> E2\*E2 | E2/E2 | E2   
E2 -> *number* | (E1)

Separating it out into E1 and E2 is necessary to maintain the precedence of \* and / over + and -.

But this is how I would do it if I had to write the parser by hand:

* Two stacks, one storing nodes of the tree as operands and one storing operators
* Read the input left to right, make leaf nodes of the numbers and push them into the operand stack.
* If you have >= 2 operands on the stack, pop 2, combine them with the topmost operator in the operator stack and push this structure back to the operand tree, *unless*
* The next operator has higher precedence that the one currently on top of the stack.

This leaves us the problem of handling brackets. One elegant (I thought) solution is to store the precedence of each operator as a number in a variable. So initially,

* int plus, minus = 1;
* int mul, div = 2;

Now every time you see a a left bracket increment all these variables by 2, and every time you see a right bracket, decrement all the variables by 2.

This will ensure that the + in 3\*(4+5) has higher precedence than the \*, and 3\*4 will not be pushed onto the stack. Instead it will wait for 5, push 4+5, then push 3\*(4+5).