Parse Trees

by

John Kennedy
Mathematics Department
Santa Monica College
1900 Pico Blvd.
Santa Monica, CA 90405

rkennedy@ix.netcom.com

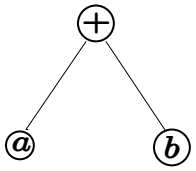
Except for this comment explaining that it is blank for some deliberate reason, this page is intentionally blank!

Parse Trees

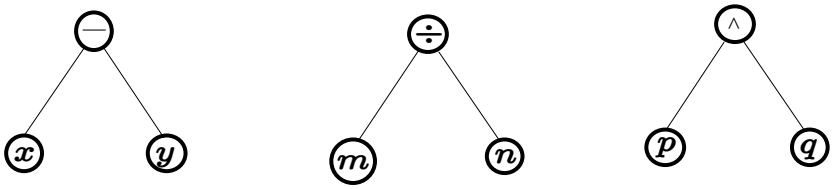
The process whereby a computer compiler takes parts of human written source code and produces an internal structure in memory that represents the correct parts of a program that can be executed is what we call ***Parsing***. This paper will demonstrate how to create parse trees on paper that represent well-formed mathematical formulas or expressions. We then show some examples of recursive functions that operate on those parse trees. It is these recursive functions that justify the importance of parse trees in the first place. Since most readers probably don't have sufficient programming expertise, this paper does not explain how to create parse trees programmatically. This in itself is another fascinating topic worthy of a more advanced paper. However, the current paper will have served its purpose if it motivates the reader to further investigate the use of parse trees in any application domain, let alone the use of trees in general.

Mathematical expressions represent only a very small part of any computer language, but they also involve the more intricate parts of the grammars that define a language. While we won't discuss the formal grammar of any particular language, we mention that most modern computer languages such as FORTRAN, BASIC, C++, PASCAL, and JAVA all handle and represent mathematical expressions by internally using the equivalent of binary trees. For this reason it is worthwhile to learn the relationship between mathematical expressions and binary trees.

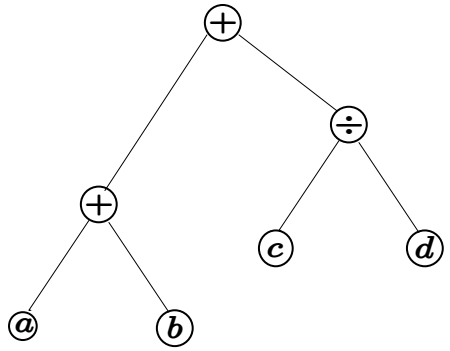
First you should try to understand that the trees we are going to create are recursive data structures. They are recursive in a sense that will become more clear as we gain some experience and show some program code. But first we should show several examples of parse trees before we show what to do with them. One of the simplest mathematical expressions we could write is something like $a + b$. The figure below is a binary tree that represents this simple expression.



What are called nodes in our tree will be drawn as circles whose content describes the node. In the above simple tree there are three nodes. The top node in any binary tree is called the root node. In the above tree the root node represents the operation of addition. The bottom two nodes are called leaf nodes and they represent the variables a and b. Since addition is commutative, the left-right order of the nodes a and b is not important. However, since subtraction and division and exponentiation are not commutative, we will show one standard way for drawing trees for expressions like $x - y$ and $m \div n$ and p^q. Note that the exponential expression p^q is often written as $p \wedge q$ where the up-arrow $\wedge$ represents the exponentiation operation.

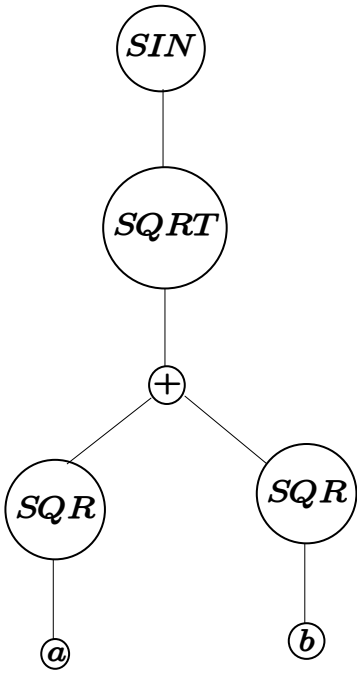


The next expression for which we draw a parse tree is the expression: $a + b + c \div d$. The entire tree for this expression is made by first drawing the simple trees for the two subexpressions $a + b$ and $c \div d$ and then combining these two subtrees with the middle addition operation. The final resulting tree looks like:



The top or root node in this case is the middle addition operation. The root node has two child nodes that can be considered as subtrees. The left child node of the root represents another addition operation for the $a + b$ sum while the right child of the root node represents the division operation.

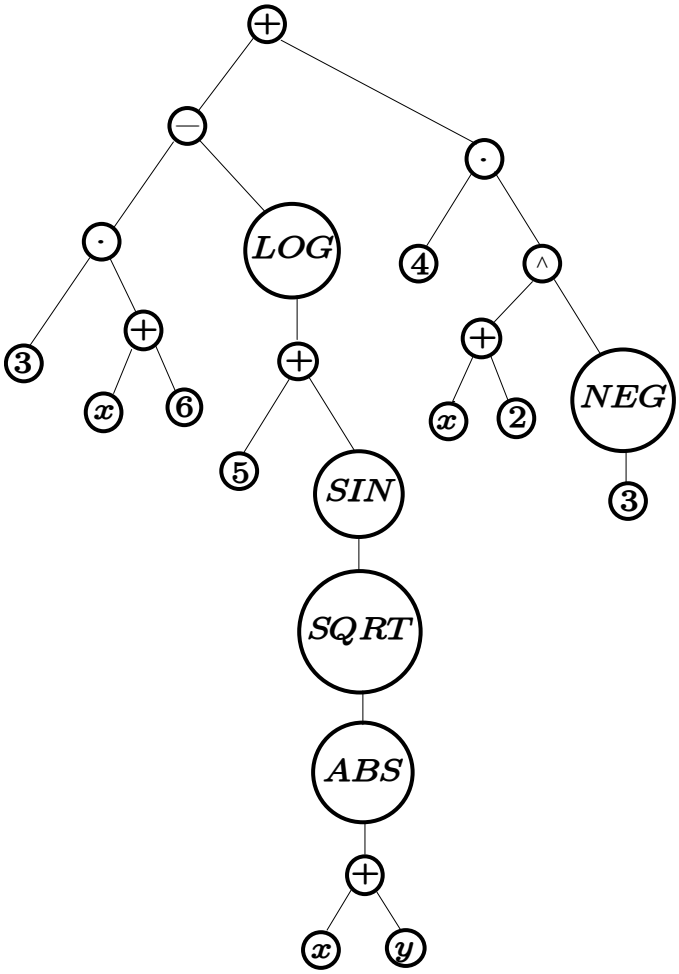
By combining subtrees we can build more complicated trees. So far all of our non-leaf nodes have been binary operations. If we wanted to draw the tree that represents $sin\left(\sqrt{a^2 + b^2}\right)$ we would need three unary operations. The first unary operation is the trigonometric function SIN and the second is $SQRT$ which represents the square root, and the third would be SQR for the squaring operation. In this example we consider squaring as its own unary operation distinct from exponentiation. The parse tree for $sin\left(\sqrt{a^2 + b^2}\right)$ is shown below. Note how each unary operation has only one child, the subtree that represents what that operation operates on. Be sure to make a distinction between $SQRT$ and SQR. Also note that parentheses are not part of any node in the tree.



As yet another example of a parse tree, consider the mathematical expression:

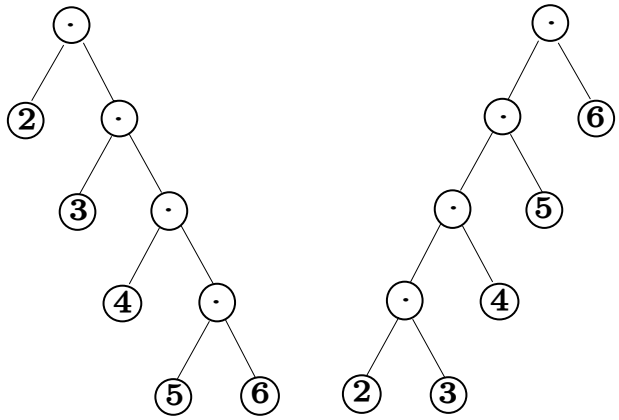
$$3(x+6) - log\big(5 + sin\big(\sqrt{|x+y|}\,\big)\big) + 4(x+2)^{-3}$$

To represent this expression we introduce three new unary operators. The logarithm function is denoted by LOG and the absolute value function will be denoted by ABS and the unary negation operator that appears in the exponent for x will be denoted by NEG. Note that multiplication is a binary operation that is represented by the dot $\cdot$. The parse tree for this expression appears below.



Again note that grouping symbols do not appear anywhere in the tree. Also note that only variables and constants appear as the bottom-most leaf nodes in the tree. All the interior tree nodes are either unary operators or binary operations.

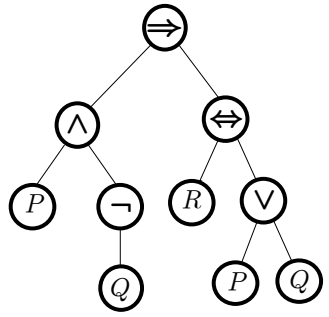
Although we have already noted that parentheses never appear as any part of any parse tree, the next two examples show how parentheses used in different ways result in different parse trees. The parse tree on the left is for the expression $2 \cdot (3 \cdot (4 \cdot (5 \cdot 6)))$ while the parse tree on the right is for the expression $(((2 \cdot 3) \cdot 4) \cdot 5) \cdot 6$.



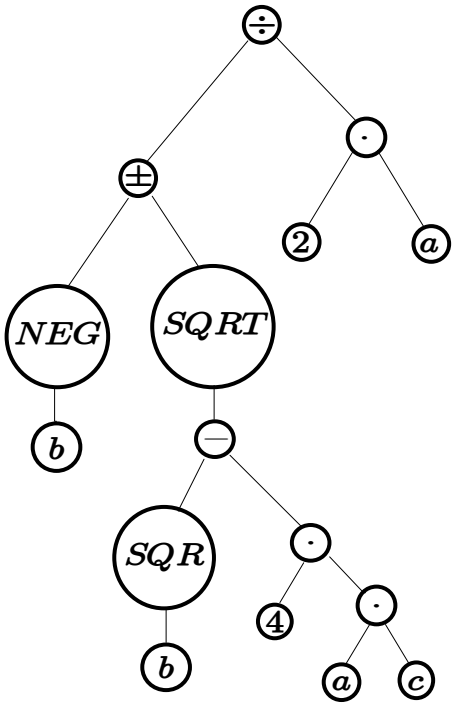
We can tell at a glance the difference between these two tree structures. These two trees are not isomorphic to each other even though both trees represent the same expression that could be written without any parentheses as $2 \cdot 3 \cdot 4 \cdot 5 \cdot 6$.

All of the previous examples of trees have represented mathematical expressions. The next example shows that parse trees can also be applied to logical expressions from sentential calculus, a branch of symbolic logic. For the next parse tree example, the symbol $\neg$ stands for NOT, $\vee$ stands for OR, and $\wedge$ stands for AND while $\Rightarrow$ and $\Leftrightarrow$ stand for the CONDITIONAL and BICONDITIONAL respectively. Then we can make the following parse tree that represents the symbolic logic expression:

$$(\mathrm{P} \wedge \neg \mathrm{Q}) \Rightarrow (\mathrm{R} \Leftrightarrow (\mathrm{P} \vee \mathrm{Q})).$$

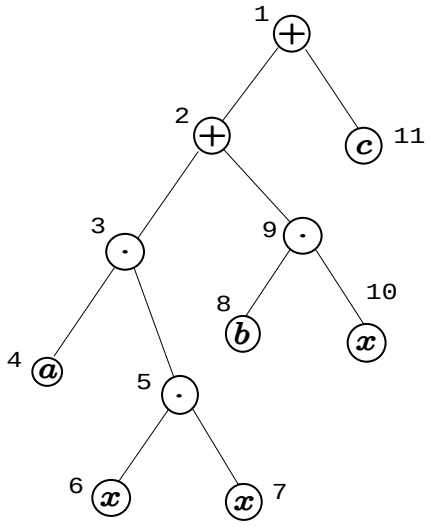


By now you should be gaining a better idea about how to construct a parse tree for any mathematical or logical expression. Just to test your full understanding, try to write down the mathematical expression that corresponds to the following parse tree.



Calculations Using Parse Trees

Next we will learn how parse trees can be used to perform mathematical calculations of the expressions they represent. Consider the quadratic expression: $ax^2 + bx + c$. The tree structure below is how this expression could be "parsed" inside a computer. In this case we have replaced the unary operator for squaring with a simple multiplication operation. The numbering of the nodes in the tree shown below is completely arbitrary. We are only using the numbers as a means of uniquely identifying the various nodes because we have multiple nodes with $+$, and $\cdot$ and x contents.



A binary tree is a recursive data structure and one of the most powerful ideas in computer science is a recursive function that acts over a tree. There are only 3 different recursive functions that can be written to "traverse" any binary tree like the one shown above. By traversing we mean that we always start at the root node and visit all the nodes in the tree just once. The connecting lines between the nodes dictate the paths we take to move from one node to another. These recursive functions are also sometimes considered as algorithms and they have the names PRE-ORDER, POST-ORDER and IN-ORDER.

In the PRE-ORDER algorithm you visit a node before you visit its children. In the POST-ORDER algorithm you visit the children of a node before you visit the node itself. In the IN-ORDER algorithm you visit the left child first, then you visit the node, and finally you visit the right child. The three orders for the above quadratic tree are summarized in the next three tables below. The *'s represent the binary multiplication operation. In this case we prefer using the * over the dot · for multiplication.

PRE-ORDER	visitation time order-->	1	2	3	4	5	6	7	8	9	10	11
1. Visit the node	node content-->	$+$	$+$	$*$	a	$*$	x	x	$*$	b	x	c
2. Visit the children	node number-->	1	2	3	4	5	6	7	9	8	10	11

POST-ORDER	visitation time order-->	1	2	3	4	5	6	7	8	9	10	11
1. Visit the children	node content-->	a	x	x	$*$	$*$	b	x	$*$	$+$	c	$+$
2. Visit the node	node number-->	4	6	7	5	3	8	10	9	2	11	1

IN-ORDER	visitation time order-->	1	2	3	4	5	6	7	8	9	10	11
1. Visit left child	node content-->	a	$*$	x	$*$	x	$+$	b	$*$	x	$+$	c
2. Visit the node	node number-->	4	3	6	5	7	2	8	9	10	1	11
3. Visit right child												

When you read across the middle row of each table you should note that only the IN-ORDER table gives the output for the expression as a normal human being would read it. But you should also note that for the purposes of computation, the POST-ORDER row is even more valuable. Consider the following as an application of the post-order algorithm where the calculations are carried out in a stack whose elements are named $X, Y,$ and Z. A stack is an example of a Last-In-First-Out structure that can be compared to a Dixie Cup dispenser.

When you read a variable, it is like marking that variable name on a Dixie Cup and pushing the cup up onto the bottom of a stack of Dixie cups. When you read a binary operation, the effect is as if you pull down the last two Dixie cups off the stack as the operands for that operation. You compute the operation answer and replace the two Dixie cups with one Dixie cup that holds the answer to the operation. Note that in this manner only math subexpressions appear as Dixie cups. In the table below we can think that $X, Y,$ and Z are three Dixie Cup *positions,* not the Dixie cups themselves.

A stack with only three elements high is all that is required to evaluate the above quadratic tree. We read the elements going across the middle row of the POST-ORDER table. The first three rows of the table below show the contents of the stack as the calculation progresses through reading operands and operators. The read order is the same as the node content order in the middle row of the above POST-ORDER table. To make the read order perfectly clear we have labeled the time order as the last row in the table. The table below shows exactly what takes place inside a Reverse Polish Logic calculator when it calculates the expression $ax^2 + bx + c$.

Z				a				ax^2				
Y			a	x	a		ax^2	b	ax^2		$ax^2 + bx$	
X		a	x	x	x^2	ax^2	b	x	bx	$ax^2 + bx$	c	$ax^2 + bx + c$
read order →		***a***	***x***	***x***	*	*	***b***	***x***	*	+	***c***	+
time order →	0	1	2	3	4	5	6	7	8	9	10	11

You might note that only those trees all of whose nodes are binary operations with complete left and right child nodes can have the IN-ORDER algorithm applied to them. Any unary operator can only have either the PRE-ORDER or POST-ORDER algorithm applied to it.

For the remaining parts of this paper we assume you have both an interest in and a familiarity with a computer language that can employ recursive functions and procedures, if not also some object oriented facilities. Rest assured that PASCAL, C++, and JAVA qualify as decent languages in these respects. Sorry, if you are only familiar with BASIC or FORTRAN, two languages that have outlived any real usefulness. We prefer to give code examples in PASCAL if only because this language is excellent at exposing algorithms and was designed by a mathematician and named after a mathematician. Our second choice of a language would be JAVA. As a language, C++ would be our distant third choice. Both PASCAL and JAVA have the advantage that they were designed by language designers. No further comments about C++ would endear us with anyone who prefers that language.

Although we are going to show pseudo PASCAL code, we cannot emphasize enough that if you know how to program in a modern language then by all means you should write and test programs to illustrate for yourself what is going on under the hood. Only by writing and testing code can you fully appreciate what we are describing.

Assuming a tree node has the following structure we can write the three functions to evaluate the quadratic expression tree using the three tree-traversal orders. The following code is pseudo PASCAL. Note that each function calls itself from within itself. That is why these three functions are called recursive functions and that is why the tree structures they operate on are called recursive data structures.

```
ATreeNode
  NodeName   : string
  LeftChild  : NodePointer
  RightChild : NodePointer
```

```
procedure PreOrder(TreeNode : NodePointer);
begin
  Output(TreeNode.NodeName);
  if TreeNode.LeftChild<>nil then PreOrder(TreeNode.LeftChild);
  if TreeNode.RightChild<>nil then PreOrder(TreeNode.RightChild);
end;


procedure PostOrder(TreeNode : TreeNode);
begin
  if TreeNode.LeftChild<>nil then PostOrder(TreeNode.LeftChild);
  if TreeNode.RightChild<>nil then PostOrder(TreeNode.RightChild);
  Output(TreeNode.NodeName);
end;


procedure InOrder(TreeNode : TreeNode);
begin
  if TreeNode.LeftChild<>nil then InOrder(TreeNode.LeftChild);
  Output(TreeNode.NodeName);
  if TreeNode.RightChild<>nil then InOrder(TreeNode.RightChild);
end;
```

If you were to execute these functions where the initial input was the root node of the above quadratic parse tree and if each time a node was **`Output`**, we just printed the node contents then you would generate the middle row in each of the above three tables with the names PRE-ORDER, POST-ORDER, and IN-ORDER.

The last two code examples that we give illustrate how you could build a function to evaluate any mathematical or logical expression tree. Such a function is one of two key elements required to actually compute a result based on the expression the tree represents. The other key element is a program that actually builds the parse tree, but such a program is beyond the scope of this paper.

First, the mathematical expression evaluator function. As you will note, this function is a recursive function since it calls itself within itself. We only show a limited set of both binary and unary operators. But once you understand how the code works, you could extend this function to make it work with all the common math operators that are found on any scientific calculator. There are about 40 such common operators.

We now expand the definition of the **`ATreeNode`** structure to include the following fields. A **`NodePointer`** is a variable that points to **`ATreeNode`** and **`NodePointerType`** is a type of variable that can point to a tree node.

```
ATreeNode
  NodeType     : (Add,Subtract,Multiply,Divide,Power,AbsValue,Log,Negation,Sine,
                  Square,SquareRoot,XVariable,Constant);
  LeftChild    : NodePointer;
  RightChild   : NodePointer;
  UnaryChild   : NodePointer;
  ConstantData : float;
```

The **ConstantData** field holds the floating point value of those nodes that are leaf node constants. For an expression as simple as X+3 it would be the node representing the constant number 3 that would hold the value 3 in the **ConstantData** field. In this case the **OperatorName** for that node would be **Constant**. Note that the node that represents the variable X need not hold the floating point value for X. Only **Constant** nodes make use of the **ConstantData** field.

You call the function **EvaluateNode** with two parameters. Assuming your expression has the variable **X** as its only variable, the first parameter is the floating point value of **X**. The second parameter is the root node of the entire tree. The function returns the floating point value that represents the entire expression. This single function is all that is required to compute any mathematical expression represented by any parse tree, no matter how big or complex the parse tree.

```
function EvaluateNode(X           : float;
                      NodePointer : NodePointerType) : float;
var  RightTemp : float;
     LeftTemp  : float;
     UnaryTemp : float;
begin
  with NodePointer do
    begin
      case NodeType of
        Add        : EvaluateNode := EvaluateNode(X,LeftChild) + EvaluateNode(X,RightChild);
        Subtract   : EvaluateNode := EvaluateNode(X,LeftChild) - EvaluateNode(X,RightChild);
        Multiply   : EvaluateNode := EvaluateNode(X,LeftChild)*EvaluateNode(X, RightChild);
        Divide     : begin
                       RightTemp := EvaluateNode(X,RightChild);
                       if (RightTemp = 0.0) then ErrorMessage := 'Divison by 0'
                         else EvaluateNode := EvaluateNode(X,LeftChild)/RightTemp
                     end;
        Power      : begin
                       LeftTemp := EvaluateNode(X,LeftChild);
                       RightTemp := EvaluateNode(X,RightChild);
                       if LeftTemp <= 0.0 then ErrorMessage := 'Improper base'
                         else EvaluateNode := exp(RightTemp*ln(LeftTemp))
                     end;
        AbsValue   : EvaluateNode := abs(EvaluateNode(X,UnaryChild));
        Log        : begin
                       UnaryTemp := EvaluateNode(X,UnaryChild);
                       if (UnaryTemp <= 0.0) then ErrorMessage := 'Log(X) logarithm';
                         else EvaluateNode := Log(UnaryTemp)
        Negation   : EvaluateNode := -EvaluateNode(X,UnaryChild);
        Sine       : EvaluateNode := sin(EvaluateNode(X,UnaryChild));
        Square     : EvaluateNode := sqr(EvaluateNode(X,UnaryChild));
        SquareRoot : begin
                       UnaryTemp := EvaluateNode(X,UnaryChild);
                       if (UnaryTemp < 0.0) then ErrorMessage := 'Square root';
                         else EvaluateNode := sqrt(UnaryTemp)
                     end;
        XVariable  : EvaluateNode := X;
        Constant   : EvaluateNode := ConstantData
      end
    end
end;  {function EvaluateNode}
```

The last function we write is one to evaluate logical expressions. Since there are fewer logical operators this function seems simpler than the one above for math expressions. However, both functions are very similar in nature.

In a logical expression we will normally have multiple variables so we assume there are global boolean variables named **PVariableTruthValue**, **QVariableTruthValue** and **RVariableTruthValue** that hold the boolean values of variables with the names P, Q and R. These global variables would be assigned their respective boolean truth values before this function would get called.

This function is called only once with the root node of the entire logical expression as its only input parameter. This function returns the boolean value that represents the truth value for the entire expression.

```
ALogicNode
   NodeType     : (LogicOR, LogicAND, Implies, IfOnlyIf, LogicNOT, VariableP,
                   VariableQ, VariableR, Constant);
   RightChild   : NodePointer;
   LeftChild    : NodePointer;
   UnaryChild   : NodePointer;
   ConstantData : boolean;
```

```
function EvaluateNode(NodePointer : NodePointerType) : boolean;
begin
  with NodePointer do
    begin
      case NodePointer.NodeType of
        LogicOR   : EvaluateNode := EvaluateNode(LeftChild) or EvaluateNode(RightChild);
        LogicAND  : EvaluateNode := EvaluateNode(LeftChild) and EvaluateNode(RightChild);
        Implies   : EvaluateNode := not (EvaluateNode(LeftChild) and
                                    (not EvaluateNode(RightChild)));
        IfOnlyIf  : EvaluateNode := EvaluateNode(LeftChild) = EvaluateNode(RightChild);
        LogicNOT  : EvaluateNode := not EvaluateNode(UnaryChild);
        VariableP : EvaluateNode := PVariableTruthValue;
        VariableQ : EvaluateNode := QVariableTruthValue;
        VariableR : EvaluateNode := RVariableTruthValue;
        Constant  : EvaluateNode := ConstantBooleanValue;
      end;
    end;
end;  {function EvaluateNode}
```

Except for the names, and number of logical variables, the above function is all that is needed to compute all the values to fill in any truth table for any logical expression, no matter how large or complicated the expression.