CMPSCI 250: Introduction to Computation

Lecture #23: Recursion on Trees
David Mix Barrington
16 March 2012

Recursion on Trees

- Trees to Represent Expressions
- A Recursive Definition of Expression Trees
- Types of Expressions
- Prefix, Infix, and Postfix Strings for Expressions
- Parsing and Evaluating Expressions
- The Definition of Truth
- Call Trees for Algorithms

Trees to Represent Expressions

- Trees are useful for representing collections of objects in a hierarchical structure, where every object except one has a unique “parent” object. We’ve mentioned people in an organization, classes in an inheritance hierarchy, and files in a directory/folder system.

- **Expressions** are collections of **atomic values** connected by **operators**. We’ve seen **boolean expressions** in the first part of the course, and we’ll see them again in today’s discussion. There are also **arithmetic expressions** as in Java. Even whole programs can be thought of as expressions.

- Operators can be **unary**, meaning that they take one argument (like ¬ or -) or **binary**, meaning that they take two (like ∧, ∨, +, or ×). In general we could also have ternary, 4-ary, or k-ary operators for any natural k.

- Our expressions are trees because each proper subexpression has exactly one parent. (The entire expression, the **root** of the tree, has no parent.)

A Recursive Definition of Expression Trees

- We can give a recursive definition of expression trees that is very similar to our other recursive definitions:
- (1) A single atomic value is an expression tree.
- (2) A k-ary operator, acting on a sequence k expression trees, gives an expression tree.
- (3) The only expression trees are those given by rules (1) and (2).
- Rule (3) gives us a **Law of Induction**: if we prove that P(a) is true for any atomic value a, and prove that P(E) is true whenever E is any k-ary operator acting on any k expression trees E_1,..., E_k such that P(E_i) is true for all i, then we have proved that P(E) is true for any expression tree E.

Types of Expressions

- In **boolean expressions** the atomic values are 0 and 1 (`false` and `true`), or variables ranging over those values, and the operators are ¬, ∧, ∨, ⊕, →, and ↔. In today's discussion we'll use just ∧, ∨, and ¬. The ¬ operator is unary and all the other operators are binary.

- In Java **arithmetic expressions** the atomic values come from one of the number types, and the operators are +, ×, -, /, % (for integer types), and so forth. The - operator can be either unary or binary -- the others are binary. We'll consider our own arithmetic expressions to use just +, ×, -, and /.

- Later in Chapter 5 and 14 we'll work with **regular expressions**, where the atomic values are letters and ∅, and there is one unary operator * and two binary operators + and ·. An induction over all regular languages will have *two* base cases and *three* inductive cases.

Prefix, Infix, and Postfix Strings for Expressions

- There are three ways to represent a boolean or arithmetic expression by a string. For an example, consider the arithmetic expression "b×b - 4×a×c" that occurs in the quadratic formula. The expression tree for this formula has nine nodes -- the root is a - operator, its children are × operators, and the leaves are atomic values a, b, c, and 4.

- "b×b - 4×a×c" is the **infix** string for this expression. The **prefix** string for it is "-×bb××4ac" and the **postfix** string is "bb×4a×c×-". Note that each string contains the same atomic and operator symbols, just in a different order. (Some infix strings also contain parentheses, making them longer than the other two.)

- We can recursively define each of the three strings from the expression. For example, "the postfix string of an atomic value is itself, and the postfix string of an operator applied to k subexpressions is the concatenation of the postfix strings for the subexpressions, followed by the symbol for the operator".

Parsing and Evaluating Expressions

- A major problem in computer science is to take a string and **parse** it, which means to determine the expression tree that it represents. A compiler must take a string in a computer language and determine (1) whether it is a valid program, (2) how the string is broken down into language parts, and (3) what the meaning of the resulting program is. You'll see more about parsing in courses like CMPSCI 401 and 410.

- The most common thing to do with an expression is to **evaluate** it, which means to determine its value by applying the operators to the atomic values. The basic evaluation algorithm for an expression is "if the expression is an atomic value, return the value, and if it is an operation applied to subexpressions, evaluate the subexpression, apply the operator to the results, and return the result of the operator".

- Parsing a program is just evaluating an expression over a complex set of values and operators, where the "value" is the meaning of each subprogram (as, for example, a machine-language program).

The Definition of Truth

- In this course we have been informal about what it means for a logical statement to be true or false in a given situation. But to do **metamathematics** (mathematics about mathematics), we would need a formal definition of what a **model** for a statement is and whether a given statement is true in a given model.

- The Polish logician Alfred Tarski gave a formal **definition of truth** in 1933, using induction on the definition of logical statements. A statement is built up from atomic statements using boolean operators and quantifiers. The truth of atomic statements is assumed to be given in a model. The truth of more complex statements can be defined by inductive rules, such as "∃x:P(x) is true if and only if there is an object z such that P(z) is true", in terms of the truth of simpler statements.

- Tarski's definition gives a justification for our four quantifier proof rules.

Call Trees for Algorithms

- If we have a method that makes recursive calls upon itself, but eventually terminates, we can make a diagram called a **call tree** that represents all the recursive calls. A node in the call tree represents a call to the method, and node x has node y as a child if the call y is made by the version of the method called by call x.

- The call tree is finite if every version eventually terminates, and the leaves of the call tree are the nodes for calls that cause no recursion.

- If we prove that every leaf call terminates with the right answer, and that every non-leaf call terminates with the right answer *if* all of its child calls do so, then the Law of Induction for trees means that we have proved that every recursive call (every call tree) corresponds to a version of the method that terminates with the right answer. This is how we prove that the method is correct.