Eric Roberts
CS 106B

Handout #47
March 4, 2013

Expression Trees

Expression Trees

Eric Roberts
CS 106B
March 4, 2013

Where Are We Heading?

- In Assignment #6, your task is to implement an interpreter for the BASIC programming language. In doing so, you will learn a little bit about how interpreters and compilers work that will almost certainly prove useful as you write your own programs.
- The goal for the next two days is to give you a sense of how a compiler can make sense of an arithmetic expression like

```
y = 3 * (x + 1)
```

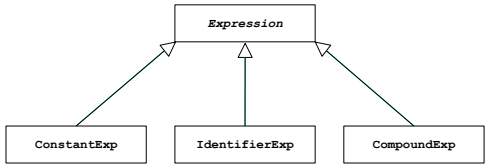
- Accomplishing that goal will unify three topics that we have already covered in CS106B:
  - Recursion
  - Tree structures
  - Class hierarchies and inheritance

Recursive Structure of Expressions

- In most programming languages, an ***expression*** is a recursive structure that can contain subexpressions.
- In the model I use in Chapter 19, every expression has one of the following forms:
  - An integer constant
  - A variable name that holds an integer value
  - Two expressions joined by an operator
  - An expression enclosed in parentheses
- Before the interpreter can work with expressions, it must convert the string representation to a recursive data structure that mirrors the recursive definition of expressions. This process is called ***parsing***.

The `Expression` Class Hierarchy

- Because expressions have more than one form, a C++ class that represents expressions can be represented most easily by a class hierarchy in which each of the expression types is a separate subclass, as shown in the following diagram:



The `exp.h` Interface

```
/*
 * File: exp.h
 * -----------
 * This interface defines a class hierarchy for arithmetic expressions.
 */

#ifndef _exp_h
#define _exp_h

#include <string>
#include "map.h"
#include "tokenscanner.h"

/* Forward reference */

class EvaluationContext;

/*
 * Type: ExpressionType
 * --------------------
 * This enumerated type is used to differentiate the three different
 * expression types: CONSTANT, IDENTIFIER, and COMPOUND.
 */

enum ExpressionType { CONSTANT, IDENTIFIER, COMPOUND };
```

The `exp.h` Interface

```
/*
 * Class: Expression
 * -----------------
 * This class is used to represent a node in an expression tree.
 * Expression itself is an abstract class.  Every Expression object
 * is therefore created using one of the three concrete subclasses:
 * ConstantExp, IdentifierExp, or CompoundExp.
 */

class Expression {

public:

   Expression();
   virtual ~Expression();
   virtual int eval(EvaluationContext & context) = 0;
   virtual std::string toString() = 0;
   virtual ExpressionType type() = 0;

/* Getter methods for convenience */

   virtual int getConstantValue();
   virtual std::string getIdentifierName();
   virtual std::string getOperator();
   virtual Expression *getLHS();
   virtual Expression *getRHS();

};
```

The `exp.h` Interface

```
/*
 * Class: ConstantExp
 * ------------------
 * This subclass represents a constant integer expression.
 */

class ConstantExp: public Expression {

public:

   ConstantExp(int val);

   virtual int eval(EvaluationContext & context);
   virtual std::string toString();
   virtual ExpressionType type();

   virtual int getConstantValue();

private:

   int value;

};
```

The `exp.h` Interface

```
/*
 * Class: IdentifierExp
 * --------------------
 * This subclass represents a expression corresponding to a variable.
 */

class IdentifierExp: public Expression {

public:

   IdentifierExp(string name);

   virtual int eval(EvaluationContext & context);
   virtual std::string toString();
   virtual ExpressionType type();

   virtual string getIdentifierName();

private:

   std::string name;

};
```

The `exp.h` Interface

```
/*
 * Class: CompoundExp
 * ------------------
 * This subclass represents a compound expression.
 */

class CompoundExp: public Expression {

public:

   CompoundExp(string op, Expression *lhs, Expression *rhs);
   virtual ~CompoundExp();

   virtual int eval(EvaluationContext & context);
   virtual std::string toString();
   virtual ExpressionType type();

   virtual std::string getOperator();
   virtual Expression *getLHS();
   virtual Expression *getRHS();

private:

   std::string op;
   Expression *lhs, *rhs;

};
```

The `exp.h` Interface

```
/*
 * Class: EvaluationContext
 * ------------------------
 * This class encapsulates the information that the evaluator needs to
 * know in order to evaluate an expression.
 */

class EvaluationContext {

public:

   void setValue(std::string var, int value);
   int getValue(std::string var);
   bool isDefined(std::string var);

private:

   Map<std::string,int> symbolTable;

};

#endif
```

Selecting Fields from Subtypes

- The **`Expression`** class exports several methods that allow clients to select fields from one of the concrete subtypes:

```
virtual int getConstantValue();
virtual std::string getIdentifierName();
virtual std::string getOperator();
virtual Expression *getLHS();
virtual Expression *getRHS();
```

- The implementation of these methods in the **`Expression`** class always generates an error. Each subclass overrides that implementation with code that returns the appropriate instance variable.
- These methods exist primarily for the convenience of the client. A more conventional strategy would be to have each subclass export only the getters that apply to that subtype. Adopting this strategy, however, forces clients to include explicit type casting, which quickly gets rather tedious.

Code for the `eval` Method

```
int ConstantExp::eval(EvaluationContext & context) {
   return value;
}

int IdentifierExp::eval(EvaluationContext & context) {
   if (!context.isDefined(name)) error(name + " is undefined");
   return context.getValue(name);
}

int CompoundExp::eval(EvaluationContext & context) {
   int right = rhs->eval(context);
   if (op == "=") {
      context.setValue(lhs->getIdentifierName(), right);
      return right;
   }
   int left = lhs->eval(context);
   if (op == "+") return left + right;
   if (op == "-") return left - right;
   if (op == "*") return left * right;
   if (op == "/") {
      if (right == 0) error("Division by 0");
      return left / right;
   }
   error("Illegal operator in expression");
   return 0;
}
```