Objectives:

1. Gain experience in advanced techniques for supporting polymorphism, specifically the visitor pattern.

Description: In this lab we will explore an advanced technique for implementing polymorphism using the Visitor Pattern. The exercise includes a parser that reads arithmetic expressions to produce a tree structure called an abstract syntax tree (AST), which represents the input expression. The directory:

```
/user/cse370/Labs/Lab11
```

contains a class for each type of expression. For example, class Add implements a tree node whose left and right subtrees correspond to the left and right operands of an add operation. Add’s instance variables are of type Expr, which is an abstract class that generalizes all of the different types of expressions. Consider the expression \((x + 5) - (y/z)\). The AST produced would be:

In this lab, you will develop a pretty printer that emits the expression in a textual form. One way to implement the pretty printer is to introduce a polymorphic “print” operation into class Expr and then provide a different method for this operation in each subclass. A problem with this approach is that if you wanted to add more operations later (such as an evaluate operation that computes the value of the expression), then you would have to add yet another polymorphic operation to
class Expr and (again) modify every class in the Expr hierarchy. A nice solution
to this general problem is called the Visitor Pattern, which works as follows: Class
Expr is extended with a polymorphic operation called Accept, which takes as
a parameter a reference to an object of class Visitor. Each subclass of Expr
then provides an Accept method that invokes an operation on the (Visitor)
parameter, passing itself (i.e., the object of some class derived from Expr) as a
parameter. Each different Accept method will invoke a different operation on the
visitor object. For example, method Add::Accept might look like:

```cpp
void Add::Accept( Visitor& v )
{ v.visitAdd(this); }
```

whereas method Subtract::Accept might look like:

```cpp
void Subtract::Accept( Visitor& v )
{ v.visitSubtract(this); }
```

Operations visitAdd and visitSubtract are both polymorphic in class Visitor.
This process is known as double dispatching because the actual method that gets
invoked by an Accept operation depends upon both the run-time type of the AST
node on which Accept is invoked and the run-time type of the visitor object that
is passed as a parameter to Accept.

For this trick to work, class Visitor must contain a polymorphic “VisitX”
operation for each concrete class “X” in the Expr hierarchy. For example:

```cpp
class Visitor {
public:
    virtual void VisitAdd(Add*) {}
    virtual void VisitSubtract(Subtract*) {}
    // and so on for other classes in the Expr hierarchy

protected:
    Visitor();
};
```

We can then implement the pretty printer as a class that derives from Visitor
and provides new methods (which recursively visit sub-expression ASTs if any
and emit a textual depiction of the operator to standard output) for all of the visit
operations. For example, we could define a PPVisitor class that derives from
Visitor. Each method in PPVisitor must know how to pretty print ASTs rooted
by nodes of the given type. By a similar reasoning, we could implement an evaluator
as a class that also derives from Visitor and provides new methods (which
recursively visits sub-expression ASTs if any and applies an arithmetic operator, such as plus or minus, to the results computed by visiting the sub-expression ASTs. By implementing these functions (i.e., pretty printing and evaluation) as visitors, we need not modify the classes in the Expr hierarchy more than once.

**Tasks:**

1. Create the base class Visitor from which one or more concrete visitor classes (e.g., class PPVisitor) can be derived.

2. Modify the AST classes (Add, Subtract, Multiply, Divide, Negate, Variable, and Literal) to provide an Accept method that takes a Visitor object as a parameter and calls the proper visit method back on this object.

3. Write a pretty printer visitor that prints out the expression stored in the AST. Details such as the proper number of parenthesis, exact spacing, and such are not important for this lab. Proper tree traversal and calling the correct functions is important (in other words, you can somewhat sacrifice the “pretty” part).

All of the files that you need can be found in:

```
/user/cse370/Labs/Lab11
```

The system is assembled such that it will include classes and functions from a library, which is contained in the Parser directory. You will have to run “make” to recompile the system once your modifications are in place. The Makefile is written such that you will not need to recompile the parser, and indeed you should not modify any of the files in the Parser directory.