**Expression API Design Doc**

**Design Overview**

The Expression API contains multiple interfaces and concrete classes. It makes use of the suggested **Emulated Extensible Enum** pattern and the **Visitor** pattern.

The whole API could be broken down into three parts: **expressions**, **arithmetic operations** and **functional operations**. Expressions include constants, variables and computation nodes. Arithmetic operations include mathematical operations and functional operations do things like evaluation, Lisp compilation and etc.. On top of these components, an **arithmetic environment** class is defined to enclose arithmetic components in a closed environment.

Expressions are modeled using an interface *ExpressionNode*, including an *accept()* function as described in the Visitor pattern. Four concrete classes implement this interface: *ConstantNode*, *VariableNode*, *UnaryNode* and *BinaryNode*.

Arithmetic operations are modeled with two interfaces: *BinaryOperation* and *UnaryOperation*. Each has one concrete enum class implementing some basic binary/unary operations. In this way, more unary and binary operations can be added to the library without modifying existing code.

Functional operations are implemented using the Visitor pattern. A *NodeVisitor* interface is defined and different *visit()* functions are declared in it. As of now, three visitors are implemented: *EvalVisitor* for evaluating an expression, *SetOfVariableVisitor* for obtaining set of referenced variables, and *LispVisitor* for compiling an expression to Lisp form.

The *ArithmeticEnv* class is used to organize related expressions/visitors in a closed session so that we can maintain some necessary states. For example, in this way we can forbid the user from creating variables/constants with the same name, which often leads to confusion. Static factories exist in the *ArithmeticEnv* to create constants/variables/expressions. Specifically, arithmetic operations are wrapped within factories to make better syntactical sense.

Variables can be created with or without initialization, but has to be initialized for evaluation. Otherwise, an *UninitializedVariableException* will be thrown. *VariableNode.setValue()* method is used to associate a numeric value with a variable.

In this framework, it is easy to add both new arithmetic operations (binary or unary) and to add new functional visitors. To add new arithmetic operation, the user only needs to implement the binary/unary operation interface and add corresponding factory in ArithmeticEnv class. To add new visitor, the user simply implements the NodeVisitor interface.

**Usecases**

1. Evaluation

ArithmeticEnv myenv = new ArithmeticEnv();

VariableNode var = myenv.createVariable(“x”);

UnaryNode sqr = myenv.SQUARED(var);

EvalVisitor visitor = new EvalVisitor();

var.setValue(1.0);

sqr.accept(visitor);

double eval = visitor.getResult();

System.out.println(“Result of ” + sqr + “(1.0) is ” + eval);

1. Lisp Compilation

ArithmeticEnv myenv = new ArithmeticEnv();

VariableNode var = myenv.createVariable(“x”);

UnaryNode sqr = myenv.SQUARED(var);

LispVisitor visitor = new LispVisitor();

sqr.accept(visitor);

String lispr = visitor.getResult();

System.out.println(lispr);

**Issues List**

1. Name of main expression interface: Expression or Evaluable?

Evaluable was a tempting name at first glance. When we think of expressions the next thing that we think of is evaluation because we evaluate expressions simply too often. It could have been a good name most of the time but probably not in this task. The reason is in the Visitor pattern, evaluation is simply one kind of visitor and in this task, evaluation is also only one functionality. It is bizarre to have an interface named Evaluable without declaring an *eval()* method. However, such method would be inconsistent with the Visitor pattern that we decide to use.

1. Put *BinaryNode* and *UnaryNode* together?

The difference between *BinaryNode* and *UnaryNode* only depends on the nature of the underlying operation so another alternative is to provide one unified class with its operands field as an array of *ExpressionNodes* and explicit bookkeep its underlying number of operands.

I chose to write them differently for several reasons:

1. Most mathematical operations are binary or unary, and the rare operations with arity 3 can often be represented by a composition of binary/unary operations.

2. Unary and Binary Nodes have quite different toString() methods.

3. It is not clean or beautiful to represent operands with an array. Basically the same information (number of operands) exists in multiple locations and we are responsible that they are always consistent

1. Naming variables and constants

Obviously there should be some way to identify variables/constants. One way is to let users give them String form names and another is to use the name of program variable. However, the latter is hard to achieve in java and assignment can mess up the correspondence. Thus, the current API requires the user to specify a name for variables/constants.

1. Introduction of an arithmetic environment

The introduction of an arithmetic environment seems to make the API less easy to use, and it does. The tradeoff here is that we want the user to give us enough information on how to manage the variables, constants and expressions he/she defines. For example, we usually don’t define two different variables with the same name. With an environment, we can bookkeep the used name and prevent users from reusing variable names.

1. Syntax for arithmetic operations

In my design, arithmetic operations are passed into the constructor of Binary and Unary Nodes. However, in order to bind expressions to an arithmetic environment, we need a cleaner way of making computation nodes. Therefore, I warp the construction of computation nodes in static factories, named after the corresponding operation name. In this way, the API also sounds more natural.

1. PrintVisitor or toString?

The printing method could also be implemented with the Visitor pattern. However, we already need to override *toString()* for good. It seems a better idea to just incorporate formatted printing in *toString()* so that it aligns with normal user expectiations.

1. Lifetime of expressions within an Arithmetic Environment

One problem with the existence of an Arithmetic Environment is that, because all expression associated with the environment are referenced by some data structure form inside the environment, they don’t go out of scope when the user usually feel like they do. Destruction methods could be added.