NOTE 216 – Lattice Expression Language Implementation

Neil Killeen (ATNF) and Ger van Diepen (NFRA)

1998 January 20

1 Introduction

The Lattice Expression Language (just a fancy name for some C++ classes !) allows manipulation of mathematical expressions involving `Lattices` directly from C++. The `Lattices` involved in the expressions are iterated through tile by tile, and the expression evaluated for each pixel in the tiles. Thus there are no large temporary `Lattices`, and the iteration is efficient.

LEL offers many of the standard numerical and logical operators and functions (that can be applied to scalars and `Arrays`), as well as some additional astronomically oriented ones. It can handle `Float`, `Double`, `Complex` and `DComplex` `Lattices`, including expressions involving mixtures of these types of `Lattices`. Conversion of data types is automatic, although the user can also embed explicit conversions.

The user can build expressions from subexpressions, finally evaluating the final expression.

Throughout this document we will refer to objects of class `Lattice`. In reality, this is an abstract class and the real objects would be derived from it.

2 Class Structures

The expression is parsed, by the compiler, into a tree, and the nodes of the tree are built with the LEL classes. The tree is also evaluated with the LEL classes.

LEL is implemented with a Letter/Envelope scheme. The relational structure between the classes is straightforward. The Envelope class is `LatticeExpr`. `LatticeExpr` invokes `LatticeExprNode`, which provides a bridge from `LatticeExpr` to the letter classes `LELBinary`, `LELBinaryCmp`, `LELBinaryBool`, `LELConvert`, `LELFunction1D`, `LELFunctionND`, `LELFunctionReal1D`, `LELFunctionFloat`, `LELFunctionDouble`, `LELFunctionComplex`, `LELFunctionDComplex`, `LELFunctionBool`, `LELLattice`, `LELUnaryConst`, `LELUnary`, and `LELUnaryBool`. The letter classes all inherit from `LELInterface` which defines their common interface. There is one more class, `LELAttribute`, which is a helper class containing some attribute information about the expression.

The purpose of the bridge class, `LatticeExprNode` is to handle type conversions. If all the data were of the same type (e.g. `Float`) we would not need the bridge class and `LatticeExpr` would directly invoke the letter classes.

The user is exposed to the classes `LatticeExpr` and `LatticeExprNode`. Exposure to `LatticeExpr` is largely implicit (see later). Use of `LatticeExprNode` may be explicit if subexpression manipulation is desired.

The classes are

LEL Classes				
Class	Source files	templation	inheritance	use
LatticeExpr	LatticeExpr.{h,cc}	$< T >$	Lattice$< T >$	Envelope
LatticeExprNode	LatticeExprNode.{h,cc}	none	none	bridge
LELAttribute	LELAttribute.{h,cc}	none	none	helper
LELInterface	LELInterface.{h,cc}	$< T >$	none	Base class
LELBinaryEnums	LELBinaryEnums.h	none	none	Enum
LELBinary	LELBinary.{h,cc}	$< T >$	LELInterface$< T >$	letter class
LELBinaryCmp	LELBinary.{h,cc}	$< T >$	LELInterface<Bool>	letter class
LELBinaryBool	LELBinary{.h,2.cc}	none	LELInterface<Bool>	letter class
LELConvert	LELConvert.{h,cc}	$< T,F >$	LELInterface$< T >$	letter class
LELFunctionEnums	LELFunctionEnums.h	none	none	enum
LELFunction1D	LELFunction.{h,cc}	$< T >$	LELInterface$< T >$	letter class
LELFunctionND	LELFunction.{h,cc}	$< T >$	LELInterface$< T >$	letter class
LELFunctionReal1D	LELFunction.{h,cc}	$< T >$	LELInterface$< T >$	letter class
LELFunctionFloat	LELFunction{.h,2.cc}	none	LELInterface<Float>	letter class
LELFunctionDouble	LELFunction{.h,2.cc}	none	LELInterface<Double>	letter class
LELFunctionComplex	LELFunction{.h,2.cc}	none	LELInterface<Complex>	letter class
LELFunctionDComplex	LELFunction{.h,2.cc}	none	LELInterface<DComplex>	letter class
LELFunctionBool	LELFunction{.h,2.cc}	none	LELInterface<Bool>	letter class
LELLattice	LELLattice.{h,cc}	$< T >$	LELInterface$< T >$	letter class
LELUnaryEnums	LELUnaryEnums.h	none	none	enum
LELUnaryConst	LELUnary.{h,cc}	$< T >$	LELInterface$< T >$	letter class
LELUnary	LELUnary.{h,cc}	$< T >$	LELInterface$< T >$	letter class
LELUnaryBool	LELUnary{.h,2.cc}	none	LELInterface<Bool>	letter class

3 How It Works

Let us look at an example expression and examined how the system works.

For example,

```
Lattice<Float> a;
Lattice<Float> b;
Lattice<Double> c;
a.copyData(b+c);
```

Thus, we evaluate the sum of the `Lattices` `b` and `c` and fill the `Lattice` `a` with the result. Note that the result of `b+c` is a `Double Lattice` which will be assigned to a `Float Lattice`.

There are two distinct steps in this; the first is the creation of the expression tree. The second is the evaluation of the tree after its creation. Both occur at run time; the creation first, and then, via the `copyData` call, the evaluation.

The tree is a structure which can be thought of as representing the hierarchy of operations. For our example above, it looks like

```
   +
b    c
```

Really, `b` and `c` are not operations, the actual software operation is something that gets the values of the `Lattice` into core from the `Lattice`. The tree is more accurately written as

```
                    +
  getLatticeValues       getLatticeValues
     b                            c
```

but we will use the short-hand tree expression style.

A more complex expression like `(a+sin(b)+2) / 10` would be

```
               /
         +          10.0
    +         a
sin    2.0
b
```

The tree is evaluated bottom up. Conceptually, the `sin` of the `Lattice b` is evaluated and `2.0` added to the resultant `Lattice`. Then that is added to the `Lattice a`, and that resultant `Lattice` is divided by `10.0`.

3.1 Tree Creation

Let us consider the creation of the expression tree first. The `Lattice::copyData` member function expects as its argument a `Lattice`. Thus, the expression in the argument has to find a way to be converted to a `Lattice`. It is the LatticeExpr class that knows how to evaluate expressions involving `Lattices`, and LatticeExpr inherits from `Lattice`. So any `LatticeExpr` object is a valid argument for the copyData call. We need to show that `b+c` is a `Lattice`; in this case a `LatticeExpr` object, derived from `Lattice`.

Internally, `LatticeExpr` contains a `LatticeExprNode` object, so let us consider class `LatticeExprNode` first. `LatticeExprNode` exists to handle type conversions for mixed type expressions. It is a non-templated class and is not derived from any other class. It contains, as private data members, a variety of pointers to the class `LELInterface`. `LELInterface` is an abstract base class, from which are derived concrete classes. These derived classes are constructed in the tree, and when the expression is evaluated, they enable one to evaluate expressions such as binary expressions, or functions, or get chunks of a `Lattice` etc. These derived classes are (mostly) templated, and the `LatticeExprNode` class contains one `LELInterface` pointer object for each conceivable type (`Float`, `Double`, `Complex`, and `DComplex`). The appropriate type for the `LELInterface` pointer and the templated derived `LELInterface` object it is pointing to is the type of the data that it is manipulating. For example, if a `LELLattice` object is constructed from a `Lattice<Float>` then the appropriate type is `Float`.

Now, expressions like `b` and `c` can be converted to `LatticeExprNode` objects via constructors such as

```
LatticeExprNode(const Lattice<Float>& lattice);
LatticeExprNode(const Lattice<Double>& lattice);
```

Recall that the `LatticeExprNode` object contains private data members (`LELInterface` pointers) of many different types. Only the data member of the relevant type will be assigned. For example, the Float constructor looks like

```
LatticeExprNode::LatticeExprNode (const Lattice<Float>& lattice)
: donePrepare_p (False),
  dtype_p       (TpFloat),
  pExprFloat_p  (new LELLattice<Float> (lattice))
{
   pAttr_p = &pExprFloat_p->getAttribute();
}
```

The constructor notes that no optimizations (see later) have been performed, and also notes what type of data it is being asked to handle. Now look at the `new` statement. A pointer (`pExprFloat_p` of type `LELInterface<Float>`) to an object of class `LELLattice` is created (`LELLattice` is a class derived from `LELInterface`, the abstract base class).

Thus, from the expressions `b` or `c`, we can create `LatticeExprNode` objects from the `Lattice` objects associated with `b` and `c`. We must now look at the full expression, `b+c`. Remember that `Lattice b` is of type `Float` and `Lattice c` is of type `Double`, and the output `Lattice a` is of type `Float` (and therefore copyData is expecting a `Lattice<Float>` for its argument).

`LatticeExprNode` has an operator + function declared as

```
friend LatticeExprNode operator+ (const LatticeExprNode& left,
                                  const LatticeExprNode& right);
```

The friend keyword makes it a globally accessible operator. Now, you can see that it takes two other `LatticeExprNode` objects, in our case, those that we made from `b` and `c`.

The + operator returns another `LatticeExprNode` object; it is defined as

```
LatticeExprNode operator+ (const LatticeExprNode& left,
                           const LatticeExprNode& right)
{
   return LatticeExprNode::newNumBinary (LELBinaryEnums::ADD, left, right);
}
```

where the static function `LatticeExprNode::newNumBinary` has returned the desired `LatticeExprNode` object (which embodies the two subexpression, `b` and `c`).

Now, recall that what we really want in the copyData call is an object of type `LatticeExpr` (which is a `Lattice`). Currently we have a `LatticeExprNode` object. So there has to be an automatic conversion from the non-templated `LatticeExprNode` object to the templated `LatticeExpr` object. This is done with one of the operators in `LatticeExprNode` from the list

```
operator LatticeExpr<Float>();
operator LatticeExpr<Double>();
operator LatticeExpr<Complex>();
operator LatticeExpr<DComplex>();
operator LatticeExpr<Bool>();
```

For reasons that we don't understand, this could not be made to work yet with a constructor of the type

```
LatticeExpr (const LatticeExprNode& expr)
```

These operators are are in reality casting operators. For example, if you had

```
Double x;
Int i;
x = (Double)i;
```

the `i` would be cast to a `Double`. Similarly,

```
LatticeExprNode node;
LatticeExpr<Float> expr;
expr = (LatticeExpr<Float>)node;
```

converts the node to the `expr`.

Since they are in class `LatticeExprNode`, they expect to operate on a `LatticeExprNode` object. The name of the operator is the same as the return type: `LatticeExpr<T>` This is in general a dangerous practice, as one gets automatic conversions that weren't wanted sometimes. But we seem to have no choice for now.

Now, for our example, the type that `Lattice::copyData` is expecting is a `Float`, because that is the type of the `Lattice a`. Therefore, the casting operator that will be invoked is

```
LatticeExprNode::operator LatticeExpr<Float>()
{
   return LatticeExpr<Float> (LatticeExprNode(makeFloat()), 0);
}
```

So a `LatticeExpr` constructor of the form

```
LatticeExpr (const LatticeExprNode& expr, uInt iDummy)
```

is explicitly invoked by this casting operator. First however, the `makeFloat` function is invoked explicitly to convert the data in the `LatticeExprNode` object to the correct internal type, which is `Float` for our example. Actually, the return type from `makeFloat` is a `CountedPtr<LELInterface<Float>>`. Therefore, to convert that to a `LatticeExprNode`, the constructor

```
LatticeExprNode(LELInterface<Float>* expr);
```

is automatically invoked. This is given to the `LatticeExpr` constructor and finally returned.

Now returning to the `newNumBinary` static function above, there is another subtlety being handled. Here is where we handle some additional type conversion. We know that `Lattice a` wants a `Float Lattice` in the `Lattice::copyData` function. We saw above that the `newNumBinary` static function produced a `LatticeExprNode` which was automatically converted to a `LatticeExpr` object of type Float. The thing we didn't see yet was how the handling of the mixed type expression `b+c` was dealt with by `newNumBinary`. That is, we don't know yet what the type of `b+c` was, although we know that `makeFloat` was able to handle it, whatever it was. So let us look inside `newNumBinary`.

This function is implemented as

```
LatticeExprNode LatticeExprNode::newNumBinary (LELBinaryEnums::Operation oper,
                                               const LatticeExprNode& left,
                                               const LatticeExprNode& right)
{
   DataType dtype = resultDataType (left.dataType(), right.dataType());
   switch (dtype) {
   case TpFloat:
      return new LELBinary<Float> (oper, left.makeFloat(),
                                   right.makeFloat());
   case TpDouble:
      return new LELBinary<Double> (oper, left.makeDouble(),
```

```
                                    right.makeDouble());
      case TpComplex:
         return new LELBinary<Complex> (oper, left.makeComplex(),
                                        right.makeComplex());
      case TpDComplex:
         return new LELBinary<DComplex> (oper, left.makeDComplex(),
                                         right.makeDComplex());
      default:
         throw (AipsError
               ("LatticeExpr: Bool argument used in numerical binary operation"));
      }
      return LatticeExprNode();
   }
```

This function returns an expression of one type, as the two expressions that go into it may have different types. Indeed, in our case, the left expression is a `Float` and the right a `Double`. The function `LatticeExprNode::resultDataType` says that mixing these two types should result in a `Double` so as not to lose precision. Therefore, the left and right expressions are converted to a `Double` expression and the `LELBinary` object that is created is a `Double` (see also the section on type conversions).

In addition, it can be seen that the return statements are returning pointers to objects of type `LELBinary`, which is derived from `LELInterface`. Yet, the function `newNumBinary` actually returns an object of type `LatticeExprNode`. So what is happening is an implicit conversion via a constructor. It's one of the private constructors

```
   LatticeExprNode(LELInterface<Float>* expr);
   LatticeExprNode(LELInterface<Double>* expr);
   LatticeExprNode(LELInterface<Complex>* expr);
   LatticeExprNode(LELInterface<DComplex>* expr);
   LatticeExprNode(LELInterface<Bool>* expr);
```

that is doing the work.

3.2 Evaluation

How does `copyData` manage to extract the result of the expression evaluation ? The `copyData` function ultimately calls the `Lattice` function `getSlice` (via an iterator) to fish out the data from its `Lattice` argument. `getSlice` is therefore implemented in `LatticeExpr` (as it inherits from `Lattice` where `getSlice` is declared). We have seen that `LatticeExpr` has one private data member, and it is of type `LatticeExprNode`. The implementation of `LatticeExpr::getSlice` is to call the `eval` function of its `LatticeExprNode` private data member (recall that `LatticeExprNode` has a variety of pointers like `CountedPtr<LELInterface<Float>>` for each data type). `LatticeExprNode` has many self-similar `eval` functions, one for each type (`Float, Double` etc). Although the `LatticeExprNode` object does know for what type it was constructed, it actually chooses the correct version of the `eval` function by the argument signature. This works because a buffer is included in the `eval` interface (this is where the result of the expression is put), and the buffer is of the appropriate type.

So invoking `eval` of `LatticeExprNode` invokes `eval` of the object (which has been derived from `LELInterface`) and is pointed to by the appropriately typed `CountedPtr<LELInterface<T>>`. In our example involving adding two `Lattices` together, those derived classes would be `LELLattice` (to read the data from the `Lattice`) and `LELBinary` (to add the data). For `LELLattice`, its `eval` function actually then uses the getSlice function on the actual `Lattice` from which it was constructed (`b` or `c`) to fish out the data. The `LELBinary eval` function will add the numbers together.

Finally, since copyData is actually iterating through the LatticeExpr (Lattice) object in optimally sized chunks. The Lattice expression is evaluated chunk by chunk (usually tile by tile). This means that there are no large temporary Lattices stored.

```
virtual void eval (Array<T>& result,
                   const PixelRegion& region) const = 0;
```

The derived classes make the actual implementation. The result of the evaluation of the expression is put in the result array. If the result of the expression evaluation is known to be a scalar (figured out at tree construction time) then the getScalar function is used to get the value instead.

```
virtual T getScalar() const = 0;
```

Let's look at eval implementations for LELBinary and LELLattice. First, the piece for LELBinary relevant to the + operator.

```
template <class T>
void LELBinary<T>::eval(Array<T>& result,
                        const PixelRegion& region) const
{
   switch(op_p) {
   case LELBinaryEnums::ADD :
       if (pLeftExpr_p->isScalar()) {
          pRightExpr_p->eval(result, region);
          result += pLeftExpr_p->getScalar();
       } else if (pRightExpr_p->isScalar()) {
          pLeftExpr_p->eval(result, region);
          result += pRightExpr_p->getScalar();
       } else {
          Array<T> temp(result.shape());
          pLeftExpr_p->eval(result, region);
          pRightExpr_p->eval(temp, region);
          result += temp;
       }
       break;
```

Three cases are handled here: (array,array), (scalar,array) and (array,scalar). The case of (scalar,scalar) is handled similarly in LELBinary::getScalar.

The important thing to see here is that the process is recursive. Each of the left and right expressions are evaluated first, before the + operation is done. So for example, since our example is the (array,array) case, we have

```
Array<T> temp(result.shape());
pLeftExpr_p->eval(result, region);
pRightExpr_p->eval(temp, region);
result += temp;
```

Both the left and right expressions are LELLattice objects. Evaluating them results in filling the result array with the values from the Lattice in the region. Then the two arrays (result and temp) are added to make the binary operation result. The LELLattice eval function looks like

```
template <class T>
void LELLattice<T>::eval(Array<T>& result,
                         const PixelRegion& region) const
{
// The rwRef function will make a copy when needed (i.e. when ptr
// contains a reference to the original data).

   COWPtr<Array<T> > ptr;
   pLattice_p->getSlice(ptr, region.box(), False);
   result.reference(ptr.rwRef());
}
```

the `Lattice` function `getSlice` is used to recover the pixels into the array `result`. Note we use a `COWPtr` so that for say an `ArrayLattice`, the array references the data only saving a copy, unless it is actually written to. There is no `LELLattice::getScalar` function as it doesn't make any sense. If you try to call it, you will throw an exception.

4 Data Type Conversions

There are two types of conversion going on in these classes, and one can get rather confused between them if not careful.

- There are conversions imposed by the `C++` compiler. These convert from `Lattice<T>` to `LatticeExprNode` and from `LatticeExprNode` to `LatticeExpr<T>`
- There are conversions done by the expression classes to convert from one data type to another (e.g. `Float` to `Double`). They are run-time conversions generated by the `makeXXX` functions.

The first type of conversion (e.g. `LatticeExprNode` to `LatticeExpr<T>`) is handled by the casting operator discussed previously. In addition, inside that casting operation are calls to functions like `LatticeExprNode::makeFloat` which embed an object of class `LELConvert` into the tree and then at evaluation time `LELConvert::eval` actually converts the data (i.e. the values of the pixels in the `eval` interface buffer array) between types so that the `LatticeExpr<T> T` type is self-consistent with the type of the `LELInterface CountedPtr` inside `LatticeExprNode` (and hence the right-type eval functions get picked up in `LatticeExprNode`).

Let us look a little harder at the conversion functions like `LatticeExprNode::makeDouble` (and similar expressions) that does the type conversion for the actual data arrays in the `LELInterface::eval` interface. Here is the implementation

```
CountedPtr<LELInterface<Double> > LatticeExprNode::makeDouble() const
{
    switch (dataType()) {
    case TpFloat:
        return new LELConvert<Double,Float> (pExprFloat_p);
    case TpDouble:
        return pExprDouble_p;
    default:
        throw (AipsError ("LatticeExpr: conversion to Double not possible"));
    }
    return 0;
}
```

So what happens is that if a type conversion is required on the `LatticeExprNode` to which `makeDouble` is being applied, then the returned `CountedPtr<LELInterface<Double>>` is assigned to a `LELConvert` object (which inherits from `LELInterface`). Otherwise, it just returns the current `CountedPtr<LELInterface<Double>>` object already active inside the `LatticeExprNode` ($\texttt{pExprDouble}_{\texttt{p}}$). The `LELConvert` object is now embedded in the tree. Note that the actual conversion will happen at evaluation time, not at tree construction time, when the `eval` function of `LELConvert` gets called. `LELConvert::eval` will actually convert the data in the interface buffer between types (just by copying).

Let us look at the actual tree here. Imagine we have

```
   Lattice<Float> a;
   Lattice<Double> b;
   LatticeExprNode expr = a+b;
```

The tree, with all like types, would be

```
   +
a      b
```

Now however, because `a` and `b` are different types, we embed a conversion into the tree. In this case, the `Float` is converted to a `Double`.

```
     +
conv    b
  a
```

Now let us assign this result of `a+b` to an output `Lattice`

```
   Lattice<Float> c;
   c.copyData(expr);
```

The type of `a+b` is `Double`, and we need to convert it to `Float`, the type of `c`. Thus the tree looks like

```
    conv
     +
conv    b
  a
```

In summary, type conversions of the actual data are handled by embedding `LELConvert` objects in the tree where necessary. The embedding is done by the `LatticeExprNode::makeXXX` functions.

5 Scalar Results

So far, we have assumed that the result of all expressions is of the same shape. For example, adding two `Lattices` together where the `Lattices` have the same shape. However, we need to also handle expressions where the resultant of the operation on a `Lattice` is a scalar. For example, `min(a)` where the minimum value of the `Lattice a` is returned, must also be handled.

This is done in two places. Firstly, when any of the derived `LEL*` classes are constructed, it is known whether the result of the operation for which that class exists is a scalar or not. For example, the class

LELUnaryConst, which exists to handle an expression like 2.0 knows that its result, after evaluation, is a scalar. Similarly, class LELFunction1D, when it is handling functions min, max, mean and sum (which take one argument) knows that it returns a scalar. Otherwise, and for all other classes, it is seen whether the result of evaluating the tree below is a scalar or not. If the former, then the result is also a scalar. Storage of the knowledge about whether the result is a scalar or not is handled by the attribute class, LELAttribute.

Secondly, the knowledge that the result of the evaluation of the tree below the current location is scalar or not is used to optimize the computation (we could just replicate scalars into arrays of course).

For example, consider the expression b+min(c) where b and c are Lattices. The result of min(c) is a scalar (we will discuss the optimization of only evaluating this once later). Evaluating b+min(c) means that scalar is added to each element of b so the final result is not a scalar.

The tree looks like

```
     +
b    min
      c
```

The code handling the binary operator + needs to know whether the result of the left and right expressions are scalars or not. For example, if they were both scalars, it would just add those scalars together and pass them on up the tree (noting at tree construction time that its result was scalar). This operation is handled in the LELBinary class, and the relevant code for the + operation is

```
template <class T>
void LELBinary<T>::eval(Array<T>& result,
                        const PixelRegion& region) const
{
   switch(op_p) {
   case LELBinaryEnums::ADD :
       if (pLeftExpr_p->isScalar()) {
          pRightExpr_p->eval(result, region);
          result += pLeftExpr_p->getScalar();
       } else if (pRightExpr_p->isScalar()) {
          pLeftExpr_p->eval(result, region);
          result += pRightExpr_p->getScalar();
       } else {
          Array<T> temp(result.shape());
          pLeftExpr_p->eval(result, region);
          pRightExpr_p->eval(temp, region);
          result += temp;
       }
       break;
```

Here you can see that it checks the left and right arguments to see if they are scalar and acts optimally accordingly, using the getScalar function (rather than the eval function) to return the scalar result. But notice that the case of both right and left being scalar is missing. What happens is that the eval function is only called by the next object up the tree if the result of that current operation is NOT scalar. If the result is a scalar, then eval is not called, but getScalar is called. For LELBinary the piece relevant to operator + looks like

```
template <class T>
T LELBinary<T>::getScalar() const
```

```
{
   switch(op_p) {
   case LELBinaryEnums::ADD :
      return pLeftExpr_p->getScalar() + pRightExpr_p->getScalar();
```

So if we had asked for `2.0 + min(c)` then both the arguments would be scalars; the tree would be

```
      +
  2.0   min
         c
```

and `LELBinary::getScalar` would have been called to evaluate the sum rather than `LELBinary::eval`. Now if this expression was being used like

```
a.copyData(2+min(c));
```

then the `Lattice a` will have all of its pixels assigned the same scalar value that resulted from the expression evaluation. This final assignment decision is handled in the `LatticeExprNode eval` functions.

6 Optimizations

In the previous section, we considered an expression like

```
a.copyData(b+min(c));
```

where the result of `min(c)` is a scalar. The `Lattice` expressions classes, normally evaluate their expressions chunk by chunk (tile by tile). However, it is clear that an expression like `min(c)` should only be evaluated once, and thereafter, for every chunk of the output `Lattice`, `a`, that pre-evaluated scalar result used.

Let us look at one of the `eval` function calls in `LatticeExprNode`. Recall that this is the function that the `Lattice::copyData` is eventually going to end up invoking and there is one for each data type depending upon the type of the output `Lattice`. Let us look at the `Float` version.

```
void LatticeExprNode::eval (Array<Float>& result,
                            const PixelRegion& region) const
{
   DebugAssert (dataType() == TpFloat, AipsError);
   if (!donePrepare_p) {

// If first time, try to do optimization.

      LatticeExprNode* This = (LatticeExprNode*)this;
      LELInterface<Float>::replaceScalarExpr (This->pExprFloat_p);
      This->donePrepare_p = True;
   }
   if (isScalar()) {
      result = pExprFloat_p->getScalar();
   } else {
      Array<Float> temp(result.shape());
      pExprFloat_p->eval(temp, region);
```

```
        result = temp;
    }
}
```

The only optimization we do at present is to replace expressions that result in a scalar by a scalar expression (`LELUnaryConst`).

So, the first time this function is entered, the optimization is attempted. Thereafter, the expression is just evaluated. Note that the caste is necessary to convert the const `LatticeExprNode` object to a non-const one so we can change it (yuck) !

The implementation of the static function `LELInterface::replaceScalarExpr` is

```
template<class T>
void LELInterface<T>::replaceScalarExpr (CountedPtr<LELInterface<T> >& expr)
{
    expr->prepare();
    if (expr->isScalar()) {
        expr = new LELUnaryConst<T> (expr->getScalar());
    }
}
```

So it takes a `CountedPtr<LELInterface>` and then does two things. First, the `prepare` function is called. Second, if the result of the input expression is a scalar, it evaluates the value of the scalar with the `getScalar` function and replaces the expression by a `LELUnaryConst` object of that scalar value and appropriate type.

Each of the `LEL*` classes derived from `LELInterface` has a prepare function. These either do nothing, or call replaceScalarExpr. Thus the process is recursive.

Let us consider a couple of simple examples. One that does no optimzation and one that does.

Consider the expression

```
LatticeExprNode myExpr = a+b;
```

where `a` and `b` are `Float Lattices`. The tree is

```
  +
a   b
```

After construction of the tree, the `LatticeExprNode` object `myExpr` has one active `LELInterface` pointer, `pExprFloat`$_{p}$, which points at the `LELBinary` object constructed to handle the + operation. The `LELBinaryObject` has two internal `LELInterface` pointers, one for each of the left and right expressions (call them `pLeft_p` and `pRight_p`). These pointers each point at a `LELLattice` object, one for each Lattice. The `LELLattice` objects maintain pointers to the actual `Lattice` objects. This is summarized in the following table.

Object	Contains a LELInterface<Float> *	Points at a	Expressions
myExpr	pExprFloat_p	LELBinary<Float>	a+b
pExprFloat_p	pLeft_p	LELLattice<Float>	a
	pRight_p	LELLattice<Float>	b

Here is the sequence of events. The numbers indicate the layer of the tree that we have penetrated to. 1 is the top of the tree.

1. In `LatticeExprNode::eval`, `replaceScalarExpr(myExpr.pExprFloat_p)` is called. `replaceScalarExpr` renames the pointer passed to it in the argument list expr. This points at the `LELBinary` object

2. In `LELInterface::replaceScalarExpr`, the `LELBinary` object calls prepare with `expr->prepare()`

3. In `LELBinary::prepare`, `replaceScalarExpr(pLeft_p)` is called.

4. In `LELInterface::replaceScalarExpr`, the `LELLattice` object calls prepare with `expr->prepare()`

5. In `LELLattice::prepare`; this does nothing and we return to `LELInterface::replaceScalarExpr`

4. In `LELInterface::replaceScalarExpr`, the result of evaluating the `LELLattice` expression is seen to not be a scalar and we return to `LELBinary::prepare`

3. In `LELBinary::prepare`, `replaceScalarExpr(pRight_p)` is called.

4. In `LELInterface::replaceScalarExpr`, the `LELLattice` object calls prepare with `expr->prepare()`.

5. In `LELLattice::prepare`; this does nothing and we return to `LELInterface::replaceScalarExpr`

4. In `LELInterface::replaceScalarExpr`, the result of evaluating the LELLattice expression is seen to not be a scalar and we return to `LELBinary::prepare`

3. In `LELBinary::prepare`, we return to `LELInterface::replaceScalarExpr`

2. In `LELInterface::replaceScalarExpr`, the result of evaluating the `LELBinary` expression is seen to not be a scalar and we return to `LatticeExprNode::eval`

1. In `LatticeExprNode::eval` we note that we have done the optimization and now evaluate the expression `a+b`.

The net result of all this was that nothing happened. This was because there were no scalar expressions to optimize.

Now let's consider an expression where the optimization will occur. Consider the expression

```
LatticeExprNode myExpr = a+min(b);
```

where `a` and `b` are `Float Lattices`. The tree is

```
  +
a   min
     b
```

The min function returns a scalar - the minimum of the `Lattice b` which should be added to the pixels of `Lattice a`. We should be able to optimize it out of the iteration loop and replace the tree by

```
  +
a   constant
```

After construction of the tree, the `LatticeExprNode` object `myExpr` has one active `LELInterface` pointer, `pExprFloat_p`, which points at the `LELBinary` object constructed to handle the + operation. The `LELBinaryObject` has two internal `LELInterface` pointers, one for each of the left and right expressions (call them `pLeft_p` and `pRight_p`). `pLeft_p` points at a `LELLattice` object, for `Lattice a`. `pRight_p` points at a `LELFunction1D` object to handle the `min` function. This `LELFunction1D` object has a `LELInterface` pointer called `pExpr_p` which points at a `LELLattice` object, for `Lattice b`. This is summarized in the following table.

Object	Contains a LELInterface<Float> *	Points at a	Expressions
myExpr	pExprFloat_p	LELBinary<Float>	a+min(b)
pExprFloat_p	pLeft_p	LELLattice<Float>	a
	pRight_p	LELFunction1D<Float>	min(b)
pRight_p	pExpr_p	LELLattice<Float>	b

Here is the sequence of events. The numbers indicate the layer of the tree that we have penetrated to. 1 is the top of the tree.

1. In `LatticeExprNode::eval`, `replaceScalarExpr(myExpr.pExprFloat_p)` is called. `replaceScalarExpr` calls the pointer passed to it in the argument list "expr" This points at the `LELBinary` object

2. In `LELInterface::replaceScalarExpr`, the `LELBinary` object calls prepare with `expr->prepare()`

3. In `LELBinary::prepare`, `replaceScalarExpr(pLeft_p)` is called. `pLeft_p` points at a `LELLattice` object

4. In `LELInterface::replaceScalarExpr`, the `LELLattice` object calls prepare with `expr->prepare()`.

5. In `LELLattice::prepare`; this does nothing and we return to `LELInterface::replaceScalarExpr`

4. In `LELInterface::replaceScalarExpr`, the result of evaluating the `LELLattice` expression is seen to not be a scalar and we return to `LELBinary::prepare`

3. In `LELBinary::prepare`, `replaceScalarExpr(pRight_p)` is called; `pRight_p` points to a `LELFunction1D` object this time.

4. In `LELInterface::replaceScalarExpr`, the `LELFunction1D` object calls prepare with `expr->prepare()`.

5. In `LELFunction1D::prepare`, `replaceScalarExpr(pExpr_p)` is called

6. In `LELInterface::replaceScalarExpr`, the `LELLattice` object calls prepare with `expr->prepare()`.

7. In `LELLattice::prepare`; this does nothing and we return to `LELInterface::replaceScalarExpr`

6. In `LELInterface::replaceScalarExpr`, the result of evaluating the `LELLattice` expression is seen to not be a scalar and we return to `LELFunction1D::prepare`

5. In `LELFunction1D::prepare`, we return to `LELInterface::replaceScalarExpr`

4. In `LELInterface::replaceScalarExpr`, the result of evaluating the `LELFunction1D` expression IS seen to be a scalar. We evaluate that scalar value (another recursive chain) with the `getScalar` function via the call `expr->getScalar()`. We replace the object pointed at by `expr` (in this case, `expr` is pointing at a `LELFunction1D` object) by a `LELUnaryConst` object constructed with the result of the `getScalar` call

We return to `LELBinary::prepare`

3. From `LELBinary::prepare`, we return to `LELInterface::replaceScalarExpr`.

2. In `LELInterface::replaceScalarExpr`, the result of evaluating the `LELBinary` expression is seen to not be a scalar and we return to `LatticeExprNode::eval`

1. In `LatticeExprNode::eval` we note that we have done the optimization and now evaluate the expression `a+constant`

Note that the call to getScalar by the `LELFunction1D` object invokes a recursive chain as well, although it doesn't go far in this case. Let's look inside the `LELFunction1D::getScalar` function and see what happens there. The relevant piece is implemented according to

```
template <class T>
T LELFunction1D<T>::getScalar() const
```

```
{
   switch(function_p) {
   case LELFunctionEnums::MIN1D :
   {
      if (pExpr_p->isScalar()) {
         return pExpr_p->getScalar();
      }
      Bool firstTime = True;
      T minVal = T();
      LatticeExpr<T> latExpr(pExpr_p, 0);
      RO_LatticeIterator<T> iter(latExpr, latExpr.niceCursorShape());
      while (! iter.atEnd()) {
         T minv = min(iter.cursor());
         if (firstTime  ||  minv < minVal) {
            firstTime = False;
            minVal = minv;
         }
         iter++;
      }
      return minVal;
   }
```

The `LELInterface` pointer `pExpr_p`, is pointing at a `LELLattice` object, which was constructed from the actual `Lattice`, `b`. First it looks to see whether the expression in hand, the `LELLattice` expression, is a scalar or not. If it is, it finds the value and returns. For example, if we had asked for `min(2.0)` this would happen. If it isn't, then we continue on. Now again, `pExpr_p` is pointing at a `LELLattice` object (derived from `LELInterface`). But that is not a `Lattice`; we need to get at the `Lattice` from which it was constructed. Since we are inside class `LELFunction1D`, we can't get at the pointer inside the `LELLattice` class which does point at the `Lattice`. Thus, instead, we construct a `LatticeExpr<T>` object (which does inherit from `Lattice`) from the `LELLattice`.

This happens with the constructor

```
LatticeExprNode(LELInterface<Float>* expr);
```

which makes a `LatticeExprNode`. From there it is converted to a `LatticeExpr` via the casting operators described in section 2.1 Then it is a simple matter to create a `Lattice` iterator, iterate through the `Lattice` (via the `LatticeExpr`) and work out the minimum value.

7 Relational and Logical Expressions

7.1 Relational Expressions

So far in the discussion, it has been assumed that the result of all expressions was either a numeric array or scalar. However, we also want to be able to handle operations which result in `Booleans`. For example, consider `Lattices`

```
Lattice<Float> a;
Lattice<Float> b;
Lattice<Bool> c;
```

and the expression

```
c.copyData(a>b);
```

so the Bool Lattice c is True or False depending upon whether the data values of a were greater than those of b or not.

What has to be handled here is that the output of the > operation is Boolean, whereas the type of the data which went into the operation was Float.

This relational operator, and like ones (<, >=, <=, ==, !=) are handled in the class LELBinaryCmp. It is templated on class T but inherits from LELInterface<Bool> rather than LELInterface<T>.

The LELInterface class eval function is declared as

```
// Evaluate the expression and fill the result array
   virtual void eval (Array<T>& result,
                      const PixelRegion& region) const = 0;
```

This indicates that the array, result, which results from evaluating the expression is of type T. Since LELBinaryCmp inherits from LELInterface<Bool>, the type of its evaluation array, result, is Bool. This is just what we want. The result of b>c is a Bool array.

The LELBinaryCmp class itself is still templated in class T because that is the type of the Lattices that go into it.

7.2 Logical Expressions

Take as an example,

```
Lattice<Bool> a;
Lattice<Bool> b;
Lattice<Bool> c;
c.copyData(a&&b);
```

so the Bool Lattice c is True if Lattice a and b are True. This kind of operator can only be defined for Boolean Lattices. Therefore the class LELBinaryBool is not templated and inherits from LELInterface<Bool>. If the data types of the Lattices are not Bool it will throw an exception.

Similarly, the class LELUnaryBool exists to handle unary logical operations such as

```
c.copyData(!a)
```

8 Other Specializations

There are a few other specialized classes because not all possible data types can be handled by some functions. For example, LELFunctionReal1D is a specialized version of LELFunction1D. The former exists to handle functions such as asin, acos etc which only function with real data.

Similarly, the classes LELFunction{Float,Double,Complex,DComplex} exist to handle functions with an arbitrary number of arguments for each data type. Probably some of these could be combined into a templated class in the same way as the 1-argument LELFunction*1D classes, but there is enough difference between them to make this worthwhile.

Deserving of special mention for their cunning implementation are the functions, nelements, ntrue, and nfalse. These are implemented in LELFunctionDouble, which is not templated and it inherits from LELInterface<Double>.

8.1 Function nelements

Consider the expression

```
Lattice<Bool> b;
Lattice<Double> a;
a.copyData(nelements(b));
```

Function nelements operates on a Lattice of any data type, and returns the number of elements in the Lattice in a Double. LELFunctionDouble is not templated, and yet this function handles Lattices of any type. It is implemented directly in LatticeExprNode

```
LatticeExprNode nelements(const LatticeExprNode& expr)
{
   Block<LatticeExprNode> arg(1, expr);
   return new LELFunctionDouble (LELFunctionEnums::NELEM, arg);
}
```

The new statement creates a pointer to a LELFunctionDouble object, which inherits from LELInterface<Double>. This is then automatically converted to a LatticeExprNode by the constructor

```
LatticeExprNode(LELInterface<Double>* expr);
```

Now, LELFunctionDouble knows that the result of function nelements is a scalar, so it is only implemented in getScalar. The implementation in LELFunctionDouble::getScalar is

```
case LELFunctionEnums::NELEM :
    if (arg_p[0].isScalar()) {
       return 1;
    }
    return arg_p[0].shape().product();
```

arg_p[0] is the first element in a Block<LatticeExprNode>. In our example, it is a LatticeExprNode housing the LELLattice object that is needed to access the Lattice¡Bool¿ (b). Now recall that LELLattice is fully templated, so it can of course handle any type of Lattice. But LELFunctionDouble doesn't know anything at all about the type of this Lattice in the path that is followed for this function; all type checking is bypassed. The statement arg_p[0].shape().product() invokes the appropriate LatticeExprNode function to return the shape attribute.

8.2 Functions ntrue and nfalse

These functions only work on Bool Lattices and count up the number of True or False values. Like nelements they are implemented directly from LatticeExprNode. E.g.

```
LatticeExprNode ntrue (const LatticeExprNode& expr)
{
   AlwaysAssert (expr.dataType() == TpBool, AipsError);
   Block<LatticeExprNode> arg(1, expr);
   return new LELFunctionDouble(LELFunctionEnums::NTRUE, arg);
}
```

Immediately though a test is made for the type of the expression that is having the function applied to it. If it's not a `Bool`, an exception is thrown. Otherwise we proceed into `LELFunctionDouble` again. Since the result is a scalar they are only implemented in `LELFunctionDouble::getScalar` For example, for `ntrue`

```
switch (function_p) {
case LELFunctionEnums::NTRUE :
{
   uInt ntrue = 0;
   Bool deleteIt;
   LatticeExpr<Bool> latExpr(arg_p[0], 0);
   RO_LatticeIterator<Bool> iter(latExpr, latExpr.niceCursorShape());
   while (! iter.atEnd()) {
      const Array<Bool>& array = iter.cursor();
      const Bool* data = array.getStorage (deleteIt);
      uInt n = array.nelements();
      for (uInt i=0; i<n; i++) {
         if (data[i]) {
            ntrue++;
         }
      }
      array.freeStorage (data, deleteIt);
      iter++;
   }
   return ntrue;
}
```

A `LatticeExpr<Bool>` (which is a `Lattice`) is explicitly created from the `LatticeExprNode` via the constructor. This is then iterated through to get implement the function.

9 Static LatticeExprNode functions

The following table lists helper functions in `LatticeExprNode` and their uses for creating appropriate nodes in the tree.

LatticeExprNode function	Reason
newNumUnary	Create a new node for a numerical unary operation. The result has the same data type as the input
newNumBinary	Create a new node for a numerical binary operator. The result has the same data type as the combined input type.
newBinaryCmp	Create a new node for a comparison binary operator. The result has the same data type as the combined input type.
newNumFunc1D	Create a new node for a numerical function with 1 argument. The result has the same data type as the input.
newRealFunc1D	Create a new node for a real numerical function with 1 argument. The result has the same data type as the input.
newComplexFunc1D	Create a new node for a complex numerical function with 1 argument. The result has the same data type as the input.
newNumReal1D	Create a new node for a real numerical function with 1 argument. The resultant type is non-complex
newNumFunc2D	Create a new node for a numerical function with 2 arguments. The result has the same data type as the combined input type.

10 Memory Management

Although there are many `new` statements in these classes, there are no matching `delete` statements. This is because all the pointers are `CountedPtr` objects and that class handles the cleanup of released memory.

11 Functionality

In this section we list the full functionality available in the `LEL` classes.

The next small table lists the data type codes used subsequently.

Type	Code
Float	1
Double	2
Complex	3
DComplex	4
Bool	5

First we give a descriptive table of the available classes and the generic input Lattice expression types and output types that they handle.

Class	Description	Type that Operates on	Return Type
LELLattice	Reads Lattice pixels	1,2,3,4,5	1,2,3,4,5
LELUnary	Handles numerical unary operators	1,2,3,4	1,2,3,4
LELUnaryConst	Handles scalar constants	1,2,3,4,5	1,2,3,4,5
LELUnaryBool	Handles logical unary operators	5	5
LELBinary	Handles numerical binary operators	1,2,3,4	1,2,3,4
LELBinaryCmp	Handles relational binary numerical operators	1,2,3,4	5
LELBinaryBool	Handles logical binary operators	5	5
LELFunction1D	Handles numerical 1-argument functions	1,2,3,4	1,2,3,4
LELFunctionND	Handles numerical N-argument functions	1,2,3,4	1,2,3,4
LELFunctionReal1D	Handles real numerical 1-argument functions	1,2	1,2
LELFunctionFloat	Handles numerical N-argument functions returning Float	1,3	1
LELFunctionDouble	Handles numerical N-argument functions returning Double	2,4	2
LELFunctionComplex	Handles complex numerical N-argument functions	3	3
LELFunctionDComplex	Handles double complex numerical N-argument functions	4	4
LELFunctionBool	Handles logical N-argument functions	5	5

Note that some classes are essentially non-templated specializations of others. For example, `LELFunctionReal1D` handles functions that couldn't be in `LELFunction1D` because there was no complex version of that function (e.g. `asin`) so templating would fail.

In the usage column of the last table, the examples use the following objects:

```
Lattice<Float> a;
Lattice<Float> b;
Lattice<Double> aDouble;
Lattice<Double> bDouble;
Lattice<Complex> aComplex;
Lattice<Complex> bComplex;
Lattice<DComplex> aDComplex;
```

```
Lattice<DComplex> bDComplex;
Lattice<Bool> aBool;
Lattice<Bool> bBool;
Double const;
LatticeExprNode expr;
```

Class	Operation	Input Data Type	Result dim.	Arguments	Usage example
LELLattice	getSlice	1,2,3,4,5	Array	1	expr = a
LELUnary	-	1,2,3,4	Scalar,Array	1	expr = $-a$
	+	1,2,3,4	Scalar,Array	1	expr = +a (does nothing)
LELUnaryConst	constant	1,2,3,4,5	Scalar	1	expr = const
LELUnaryBool	!	5	Scalar,Array	1	expr = !aBool
LELBinary	+	1,2,3,4	Scalar,Array	2	expr = a+b
	-	1,2,3,4	Scalar,Array	2	expr = a−b
	*	1,2,3,4	Scalar,Array	2	expr = a*b
	/	1,2,3,4	Scalar,Array	2	expr = a/b
LELBinaryCmp	==	1,2,3,4	Scalar,Array	2	expr = a==b
	!=	1,2,3,4	Scalar,Array	2	expr = a!=b
	>	1,2,3,4	Scalar,Array	2	expr = a>b
	<	1,2,3,4	Scalar,Array	2	expr = a<b
	>=	1,2,3,4	Scalar,Array	2	expr = a>=b
	>=	1,2,3,4	Scalar,Array	2	expr = a<=b
LELBinaryBool	==	5	Scalar,Array	2	expr = aBool==bBool
	!=	5	Scalar,Array	2	expr = aBool!=bBool
	&&	5	Scalar,Array	2	expr = Bool&&bBool
	||	5	Scalar,Array	2	expr = Bool||bBool
LELFunction1D	sin	1,2,3,4	Scalar,Array	1	expr = sin(a)
	sinh	1,2,3,4	Scalar,Array	1	expr = sinh(a)
	cos	1,2,3,4	Scalar,Array	1	expr = cos(a)
	cosh	1,2,3,4	Scalar,Array	1	expr = cosh(a)
	exp	1,2,3,4	Scalar,Array	1	expr = exp(a)
	log	1,2,3,4	Scalar,Array	1	expr = log(a)
	log10	1,2,3,4	Scalar,Array	1	expr = log10(a)
	sqrt	1,2,3,4	Scalar,Array	1	expr = sqrt(a)
	min	1,2,3,4	Scalar	1	expr = min(a)
	max	1,2,3,4	Scalar	1	expr = max(a)
	mean	1,2,3,4	Scalar	1	expr = meann(a)
	sum	1,2,3,4	Scalar	1	expr = sum(a)
LELFunctionND	iif	1,2,3,4	Scalar,Array	1	expr = iif(aBool,a,b)
LELFunctionReal1D	asin	1,2	Scalar,Array	1	expr = asin(a)
	acos	1,2	Scalar,Array	1	expr = acos(a)
	tan	1,2	Scalar,Array	1	expr = tan(a)
	atan	1,2	Scalar,Array	1	expr = atan(a)
	tanh	1,2	Scalar,Array	1	expr = tanh(a)
	ceil	1,2	Scalar,Array	1	expr = ceil(a)
	floor	1,2	Scalar,Array	1	expr = floor(a)
LELFunctionFloat	min	1	Scalar,Array	2	expr = min(a,b)
	max	1	Scalar,Array	2	expr = max(a,b)
	pow	1	Scalar,Array	2	expr = pow(a,b)
	atan2	1	Scalar,Array	2	expr = atan2(a,b)
	fmod	1	Scalar,Array	2	expr = fmod(a,b)
	abs	1,3	Scalar,Array	1	expr = abs(a), abs(aComplex)
	arg	3	Scalar,Array	1	expr = arg(aComplex)
	real	1,3	Scalar,Array	1	expr = real(aComplex)
	imag	1,3	Scalar,Array	1	expr = imag(aComplex)

Class	Operation	Input Data Type	Result dim.	Arguments	Usage example
LELFunctionDouble	min	2	Scalar,Array	2	expr = min(aDouble,bDouble)
	max	2	Scalar,Array	2	expr = max(aDouble,bDouble)
	pow	2	Scalar,Array	2	expr = pow(aDouble,bDouble)
	atan2	2	Scalar,Array	2	expr = atan2(aDouble,bDouble)
	fmod	2	Scalar,Array	2	expr = fmod(aDouble,bDouble)
	abs	2,4	Scalar,Array	1	expr = abs(aDouble), abs(aDComplex)
	arg	4	Scalar,Array	1	expr = arg(aDComplex)
	real	2,4	Scalar,Array	1	expr = real(aDComplex)
	imag	2,4	Scalar,Array	1	expr = imag(aDComplex)
LELFunctionDouble	ntrue	5	Scalar	1	expr = ntrue(aBool)
	nfalse	5	Scalar	1	expr = nfalse(aBool)
LELFunctionDouble	nelements	Any	Scalar	1	expr = nelements(a)
LELFunctionComplex	pow	3	Scalar,Array	2	expr = pow(aComplex,bComplex)
	conj	3	Scalar,Array	1	expr = conj(aComplex)
LELFunctionDComplex	pow	4	Scalar,Array	2	expr = pow(aDComplex,bDComplex)
	conj	4	Scalar,Array	1	expr = conj(aComplex)
LELFunctionBool	all	5	Scalar	1	expr = all(aBool)
	any	5	Scalar	1	expr = any(aBool)
LatticeExprNode	amp	1,2,3,4	Scalar,Array	2	expr = amp(a,b)
LatticeExprNode	pa	1,2	Scalar,Array	2	expr = pa(a,b)