NOTE 223 – Lattice Expression Language

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

2003 November 10

1 Introduction

The Lattice Expression Language (LEL) makes it possible to do arithmetic on lattices (in particular on images [which are just lattices plus coordinates]) in AIPS++. An expression can be seen as a lattice (or image) in itself. It can be used in any operation where a normal image is used.

To summarize, the following functionality is supported:

- The common mathematical, comparison, and relational operators.
- An extensive list of mathematical and logical functions.
- Mixed data type arithmetic and automatic data type promotion.
- Support of image masks.
- Masking using boolean expressions.
- Handling of masks in an expression.
- Support of image regions.
- Interface from both Glish and C++.

The first section explains the syntax. The last sections show the interface to LEL using Glish or C++. The Glish interface makes it possible to embed Glish variables and expressions in a LEL command. At the end some examples are given. If you like, you can go straight to the examples and hopefully immediately be able to do some basic things.

LEL operates on lattices, which are a generalization of arrays. As said above a particular type of lattice is an image; they will be used most often. Because lattices can be very large and usually reside on disk, an expression is only evaluated when a chunk of its data is requested. This is similar to reading only the requested chunk of data from a disk file.

LEL is quite efficient and can therefore be used well in C++ and glish code. Note however, that it can never be as efficient as carefully constructed C++ code.

Note 216 gives a detailed description how LEL is implemented using various C++ classes.

2 Expressions

A LEL expression can be as simple or as complex as one likes using the standard arithmetic, comparison, and logical operators. Parentheses can be used to group subexpressions.
The operands in an expression can be lattices, constants, functions, and condition masks. lattice regions and masks. E.g.

```
lat1 + 10
lat1 + 2 * max(lat2,1)
amp(lat1, lat2)
lat1 + mean(img[region1])
lat1 + mean(lat2[lat2>5 && lat2<10])
```

The last example shows how a boolean expression can be used to form a mask on a lattice. Only the pixels fulfilling the boolean condition will be used when calculating the mean.

In general the result of a LEL expression is a lattice, but it can be a scalar too. If is is a scalar, it will be handled correctly by C++ and glish functions using it as the source in, say, an assignment to another lattice.

LEL fully supports masks. In most cases the mask of a subexpression is formed by and-ing the masks of its operands. It is fully explained in a later section.

LEL supports the following data types:

Bool

Float single precision real (which includes integers)

Double double precision real

Complex single precision complex

DComplex double precision complex

All these data types can be used for scalars and lattices.
LEL will do automatic data type promotion when needed. E.g. when a Double and a Complex are used in an operation, they will be promoted to DComplex. It is also possible to promote explicitly using the conversion functions (FLOAT, DOUBLE, COMPLEX and DCOMPLEX). These functions can also be used to demote a data type (e.g. convert from Double to Float), which can sometimes be useful for better performance.

Region is a specific data type. It indicates a region of any type (in pixel or world coordinates, relative, fractional). A region can only be applied to a lattice subexpression using operator `[]`.

2.1 Constants

Scalar constants of the various data types can be formed as follows (which is similar to Glish):

- A Bool constant can be given as T or F (meaning true and false respectively).
- A Float constant can be any integer or floating-point number. E.g.

  ```
  3
  3.14
  3.14e-2
  ```

- A Double constant is a floating-point number using a D for the exponent. One can also use the `DOUBLE` function. E.g.

  ```
  1d2
  3.14d-2
  double(2)
  ```

- The imaginary part of a Complex or DComplex constant is formed by a Float or Double constant immediately followed by a lowercase **i**. A full complex constant is formed by adding another constant as the real part. E.g.

```
1.5 + 2i
2i+1.5             is identical
```

Note that a full complex constant has to be enclosed in parentheses when, say, a multiplication is performed on it. E.g.

```
2 * (1.5+2i)
```

The functions `pi()` and `e()` should be used to specify the constants pi and e. Note that they form a Double constant, so when using e.g. pi with a Float lattice, it could make a lot of sense to convert pi to a Float. Otherwise the lattice is converted to a Double, which is time-consuming. However, one may have very valid reasons to convert to Double, e.g. to ensure that the calculations are accurate enough.

2.2 Operators

The following operators can be used (with their normal meaning and precedence):

Unary + and `-`
: Can not be used with Bool operands.

Unary `!`
: Logical NOT operator. Can only be used with Bool operands.
For a region it forms the complement.

Binary ∧, `*`, `/`, `%`, +, and `-`
: % is the modulo operator. E.g. `3%1.4` results in `0.2` and `-10%3` results in `-1`.
`^` is the power operator.
All operators are left-associative, except `^` which is right-associative; thus `2^1^2` results in `2`.
Operator % can only be used for real operands, while the others can be used for real and complex operands.
Operator `-` can also be used for regions. It forms the difference of the left and right operand.

==, ! =, >, >=, <, and <=
: For Bool operands only == and ! = can be used.

A Bool operand cannot be compared with a numeric operand.
The comparison operators use the norm for complex values.

&& and ||
Logical AND and OR operator.
These operators can only be used with Bool operands.
When used on a region **&&** forms the intersection, while || forms the union.

The precedence order is:

`^`
unary `+, -, !`
`*, /, %`
`+, -`
==, ! =, >, >=, <, <=
`&&`
||

Note that `^` has a higher precedence than the unary operators.
E.g. `-3^2` results in `-9`.

The operands of these operators can be 2 scalars, 2 lattices, or a lattice and a scalar. When 2 lattices are used, they should in principle conform; i.e. they should have the same shape and coordinates. However, LEL will try if it can extend one lattice to make it conformant with the other. It can do that if both lattices have coordinates and if one lattice is a true subset of the other (thus if one lattice has all the coordinate axes of the other lattice and if those axes have the same length or have length 1). If so, LEL will add missing axes and/or stretch axes with length 1

2.3 Functions

In the following tables the function names are shown in uppercase, while the result and argument types are shown in lowercase. Note, however, that function names are case-insensitive. All functions can have scalar and/or lattice arguments.
When a function can have multiple arguments (e.g. atan2), the operands are automatically promoted where needed.

2.3.1 Mathematical functions

Several functions can operate on real or complex arguments. The data types of such arguments are given as 'numeric'.

`Double PI()` Returns the value of **pi**.

`Double E()` Returns the value of **e**.

`numeric SIN(numeric)`

`numeric SINH(numeric)`

`real ASIN(real)`

`numeric COS(numeric)`

`numeric COSH(numeric)`

`real ACOS(real)`

`real TAN(real)`

`real TANH(real)`

`real ATAN(real)`

`real ATAN2(real y, real x)` Returns `ATAN(y/x)` in correct quadrant.

`numeric EXP(numeric)`

`numeric LOG(numeric)` Natural logarithm.

`numeric LOG10(numeric)`

`numeric POW(numeric, numeric)` The same as operator `^`.

`numeric SQRT(numeric)`

`complex COMPLEX(real, real)` Create a complex number from two reals.

`complex CONJ(complex)`

`real REAL(numeric)` Real value itself or real part of a complex number.

`real IMAG(complex)` Imaginary part of a complex number.

`real NORM(numeric)`

`real ABS(numeric), real AMPLITUDE(numeric)` both find the amplitude of a complex number. If the numeric argument is real, imaginary part zero is assumed.

`real ARG(complex), real PHASE(complex)` both find the phase of a complex number.

`numeric MIN(numeric, numeric)`

`numeric MAX(numeric, numeric)`

`Float SIGN(real)` Returns -1 for a negative value, 0 for zero, 1 for a positive value.

`real ROUND(real)` Rounds the absolute value of the number.
E.g. `ROUND(-1.6) = -2`.

`real FLOOR(real)` Works towards negative infinity.
E.g. `FLOOR(-1.2) = -2`

`real CEIL(real)` Works towards positive infinity.

`real FMOD(real, real)` The same as operator %.

Note that the trigonometric functions need their arguments in radians.

2.3.2 Scalar result functions

The result of these functions is a scalar.

`double NELEMENTS(anytype)` Return number of elements in a lattice (1 for a scalar).

`double NDIM(anytype)` Return dimensionality of a lattice (0 for a scalar).

`double LENGTH(anytype, real axis)` Return length of a lattice axis (returns 1 for a scalar or if axis exceeds number of axes). Axis number is 1-relative.

`Bool ANY(Bool)` Is any element true?

`Bool ALL(Bool)` Are all elements true?

`Double NTRUE(Bool)` Number of true elements.

`Double NFALSE(Bool)` Number of false elements.

`numeric SUM(numeric)` Return sum of all elements.

`numeric MIN(numeric)` Return minimum of all elements.

`numeric MAX(numeric)` Return maximum of all elements.

`real MEDIAN(real)` Return median of a lattice. For smallish lattices (max. 512*512 elements) the median can be found in 1 pass. Other lattices usually require 2 passes.

`real FRACTILE(real,float)` Return the fractile of a lattice at the fraction given by the second argument. A fraction of 0.5 is the same as the median. The fraction has to be between 0 and 1. For smallish lattices (max. 512*512 elements) the fractile can be found in 1 pass. Other lattices usually require 2 passes.

`real FRACTILERANGE(real,float,float)` Return the range between the fractiles at the fraction given by the second and third argument. The fractions have to be between 0 and 1 and the second fraction has to be greater than the first one. The second fraction is optional and defaults to `1-fraction1`. Thus

```
FRACTILERANGE(lat, 0.1)
FRACTILERANGE(lat, 0.1, 0.9)
FRACTILE(lat,0.9) - FRACTILE(lat,0.1)
```

are equal, be it that the last one is about twice as slow.
For smallish lattices (max. 512*512 elements) the fractile range can be found in 1 pass. Other lattices usually require 2 passes.

`numeric MEAN(numeric)` Return mean of all elements.

`numeric VARIANCE(numeric)` Return variance
(`sum((a(i) - mean(a))**2) / (nelements(a) - 1)`).
All calculations are done in double precision.

`numeric STDDEV(numeric)` Return standard deviation (the square root of the variance).

`real AVDEV(numeric)` Return average deviation.
(`sum(abs(a(i) - mean(a))) / nelements(a)`).
All calculations are done in double precision.

2.3.3 Miscellaneous functions

`numeric REBIN(numeric,[f1,f2,...])` rebins the image using the given (integer) factors. It averages the pixels in each bin with shape [f1,f2,...]. Masked-off pixels are not taken into account. If all pixels in a bin are masked off, the resulting pixel will be masked off. The length of the

factor list [f1,f2,...] has to match the dimensionality of the image. The factors do not need to divide the axes lengths evenly. Each factor can be a literal value, but it can also be any expression resulting in a real scalar value.
For instance, for a 3-dimensional image:

```
rebin(lat,[2,2,1])
```

will halve the size of axis 1 and 2.

`real AMP(real,real)` It returns the square root of the quadrature sum of the two arguments. Thus

```
amp(lat1,lat2)
```

gives $\sqrt{(lat1^2 + lat2^2)}$. This can be used to form, for example, (biased) polarized intensity images when lat1 and lat2 are Stokes Q and U images.

`real PA(real,real)` It returns a "position angle" (in degrees) from the two lattices. That is,

```
pa(lat1,lat2)
```

gives $180/\pi * atan2(lat1, lat2)/2$. This can be used to form, for example, linear polarization position angle images when lat1 and lat2 are Stokes Q and U images, *respectively.*

`real SPECTRALINDEX(real,real)` It returns a the spectral index made from the two lattices. That is,

```
log(s1/s2) / log(f1/f2)
```

where s1 and s2 are the source fluxes in the lattices and f1 and f2 are the frequencies of the spectral axes of both lattices. Similar to e.g. operator + the lattices do not need to have the same shape. One can be extended/stretched as needed.

`anytype VALUE(anytype)` It returns the argument without its possible mask, thus it removes the mask from the argument. The section about mask handling discusses it in more detail.

`Bool MASK(anytype)` It returns the mask of the argument. The section about mask handling discusses it in more detail.

`Bool ISNAN(anytype)` It tests lattice elements on a NaN value and sets the corresponding output element to T if so; otherwise to F.

`anytype REPLACE(anytype, anytype)` The first argument has to be a lattice (expression). The optional second argument can be a scalar or a lattice (expression). It defaults to 0.
The result of the function is a copy of the first argument, where each masked-off element in the first argument is replaced by the corresponding element in the second argument. The result's mask is a copy of the mask of the first argument.

```
replace (lat1, 0)
replace (lat1, lat2)
```

The first example replaces each masked-off element in `lat1` by 0.
The second example replaces it by the corresponding element in `lat2`.
A possible mask of `lat2` is not used.

`anytype IIF(Bool, anytype, anytype)` The first argument is a boolean expression. If an element in it is true, the corresponding element from the second argument is taken, otherwise from the third argument. It is similar to the ternary `?:` construct in C++. E.g.

```
iif (lat1>0, lat1, 0)       same as max(lat1,0)
iif (sum(lat1)>0, lat1, lat2)
```

The examples shows that scalars and lattices can be freely mixed. When all arguments are scalars, the result is a scalar. Otherwise the result is a lattice.
Note that the mask of the result is formed by combining the mask of the arguments in an appropriate way as explained in the section about mask handling.

`Bool INDEXIN(real axis, set indices)` The first argument is a 1-relative axis number. The second argument is a set of indices. It returns a Bool array telling for each lattice element if the index of the given axis is contained in the set of indices.
The 1-relative indices have to be given as elements with integer values enclosed in square brackets and separated by commas. Each element can be a single index, an index range as `start:end`, or a strided index range as `start:end:stride`. The elements do not need to be ordered, but in a range start must be <= end. For example:

```
image[indexin(2, [3,4:8,10:20:2])]
```

masks `image` such that only the pixels with an index 3, 4, 5, 6, 7, 8, 10, 12, 14, 16, 18, or 20 on the second axis are set to True.

The following special syntax exists for this function.

```
INDEXi IN set
```

where `i` is the axis number. So the example above can also be written as:

```
image[index2 in [3,4:8,10:20:2]]
```

Negated versions of this function exist as:

```
INDEXNOTIN(axis, set)
INDEXi NOT IN set
```

2.3.4 Conversion functions

`Float FLOAT(real)` Convert to single precision.

`Double DOUBLE(real)` Convert to double precision.

`Complex COMPLEX(numeric)` Convert to single precision complex. If the argument is real, the imaginary part is set to 0.

`DComplex DCOMPLEX(numeric)` Convert to double precision complex. If the argument is real, the imaginary part is set to 0.

`Bool BOOLEAN(region)` Convert to boolean. This can be useful to convert a region to a boolean lattice. Only a region in pixel coordinates can be converted, so in practice only an image mask can be converted.

Note that, where necessary, up-conversions are done automatically. Usually it may only be needed to do a down-conversion (e.g. Double to Float).

2.4 Lattice names

When a lattice (e.g. an image) is used in an expression, its name has to be given. The name can be given directly if it consists of the characters `-.$~` and alphanumeric characters.

If the name contains other characters or if it is a reserved word (currently only T and F are reserved), it has to be escaped. Escaping can be done by preceeding the special characters with a backslash or by enclosing the string in single or double quotes. E.g.

```
~/myimage.data
~/myimage.data\-old
'~/myimage.data-old'
```

Note that when LEL is used from Glish, it is also possible to use a Glish image variable as a lattice operand (e.g. `$im`). This is explained in the section describing the Glish binding. It means that in Glish a name starting with a `$` should be escaped too.

3 Masks

3.1 Access to Image Masks

A boolean mask associated with an image indicates whether a pixel is good (mask value True) or bad (mask value False). If the mask value is bad, then the image pixel is not used for computation (e.g. when finding the mean of the image).

An image can have zero (all pixels are good) or more masks. One mask can be designated as the default mask. By default it will be applied to the image (from Glish, designation of the default mask is handled by the maskhandler function of the Image tool).

When using LEL, the basic behaviour is that the default mask is used. However, by qualifying the image name with a suffix string, it is possible to specify that no mask or another mask should be used. The suffix is a colon followed by the word `nomask` or the name of the alternative mask.

```
myimage.data
myimage.data:nomask
'myimage.data:othermask'
```

The first example uses the default mask (if the image has one). The second example uses no mask (thus all pixels are designated good) and the third example uses mask `othermask`.

Note that if the image name is enclosed in quotes, the mask name should be enclosed too. It means that a colon cannot be part of an image name.

It is also possible to use a mask from another image like

```
myimage.data:nomask[myotherimage::othermask]
```

This syntax is explained in the section describing regions

3.2 Lattice Condition Mask

We have seen in the previous section that lattices (in this case images) can have an associated mask. These masks are stored with the image – they are persistent.

It is also possible to create transient masks when a LEL expression is executed (dawn, usually). This is done with the operator `[]` and a boolean expression. For example,

```
sum( lat1[lat1<5 && lat1>10] )
```

creates a mask for `lat1` indicating that only its elements fulfilling the boolean condition should be taken into account in the `sum` function. Note that the mask is local to that part of the expression. So in the expression

```
sum( lat1[lat1<5 && lat1>10] ) + sum(lat1)
```

the second `sum` function takes all elements into account. Masking can also be applied to more complex expressions and it is recursive.

```
(lat1+lat2)[lat3<lat4]
sum( lat1[lat1<5][lat1>10] )
(lat1 + lat2[lat3<lat4]) [lat1<5]
```

The first example applies the mask generated by the `[]` operator to the expression `lat1+lat2`. The second example shows the recursion (which ANDs the masks). It is effectively a (slower) implementation of the first example in this subsection. In the last example, the expression inside the parentheses is only evaluated where the condition `[lat1<5]` is true and the resulting expression has a mask associated with it.

Please note that it is possible to select pixels on an axis by means of the function `INDEXIN` (or by the `INDEXi IN` expression) as shown in the previous section about miscellaneous functions.

3.3 Mask Handling

As explained in the previous subsections, lattices can have a mask. Examples are a mask of good pixels in an image, a mask created using a boolean condition and the operator `[]`, or a mask defining a region within its bounding box.
A pixel is bad when the image has a mask and when the mask value for that pixel is False. Functions like `max` ignore the bad pixels.
Note that in a MeasurementSet a False mask value indicates a good visibility. Alas this is a historically grown distinction in radio-astronomy.

Image masks are combined and propagated throughout an expression. E.g. when two lattices are added, the mask of the result is formed by and-ing the masks of the two lattices. That is, the resultant mask is True where the mask of lattice one is true AND the mask of lattice 2 is True. Otherwise, the resultant mask is False.

In general the mask of a subexpression is formed by and-ing the masks of the operands. This is true for e.g. +, *, `atan2`, etc.. However, there are a few special cases:

- The mask created by `operator[condition]` is formed by and-ing the condition result, the mask of the result, and the mask of the subexpression where the condition is applied to. For example, suppose `lat1` and `lat2` each have a mask. Then in

```
sum( lat1[lat2<5] )
```

the `sum` function will only sum those elements for which the mask of `lat1` and `lat2` is valid and for which the condition is true.

- The logical AND operator forms the resultant mask by looking at the result and the masks of the operands.

  ```
  lat1[lat1<0 && lat2>0]
  ```

  Let us say both `lat1` and `lat2` have masks. The operand `lat1<0` is true if the mask of `lat1` is true and the operand evaluates to true, otherwise it is false. Apply the same rule to the operand `lat2 > 0`. The AND operator gives true if the left and right operands are both true. If the left operand is false, the right operand is no longer relevant. It is, in fact, 3-valued logic with the values true, false, and undefined.

  Thus, the full expression generates a lattice with a mask. The mask is true when the condition in the `[]` operator is true, and false otherwise. The values of the output lattice are only defined where its mask is true.

- The logical OR operator works the same as the AND operator. If an operand has a true value the other operand can be ignored.

- The mask of the result of the `replace` function is a copy of the mask of its first operand. The mask of the second operand is not used at all.

- The `iif` function has three operands. Depending on the condition, an element from the second or third operand has to be taken. The resultant mask is formed by the mask of the condition and-ed with the appropriate elements from the masks of the second or third operand.

- The `value` function returns the value without a mask, thus it removes the mask from a value. It has the same effect as the `image:nomask` construction discussed above. However, the `value` function is more general, because it can also be applied to a subexpression.

- The `mask` function returns the mask of a value. The returned value is a boolean lattice and has no mask itself. When the value has no mask, it returns a mask consisting of all True values. When applied to an image, it returns its default mask.

Consider the following more or less equivalent examples:

```
value(image1)[mask(image2)]
image1:nomask[mask(image2)]
image1:nomask[image2::mask0]
```

The first two use the default mask of `image2` as the mask for `image1`. The latter uses `mask0` of `image2` as the mask for `image1`. It is equivalent to the first two examples if `mask0` is the default mask of `image2`.

It is possible that the entire mask of a subexpression is false. For example, if the mean of such a subexpression is taken, the result is undefined. This is fully supported by LEL, because a scalar value also has a mask associated with it. One can see a masked-off scalar as a lattice with an all false mask. Hence an operation involving an undefined scalar results in an undefined scalar. The following functions act as described below on fully masked-off lattices:

- MEDIAN, MEAN, VARIANCE, STDDEV, AVDEV, MIN, MAX
  result in an undefined scalar:
- NELEMENTS, NTRUE, NFALSE, SUM
  result in a scalar with value 0.
- ANY
  results in a scalar with value F.
- ALL
  results in a scalar with value T.
- LENGTH, NDIM
  ignore the mask because only the shape of the lattice matters.

You should also be aware that if you remove a mask from an image, the values of the image that were previously masked bad may have values that are meaningless.

4 Regions

A region-of-interest generally specifies a portion of a lattice which you are interested in for some astronomical purpose (e.g. what is the flux density of this source). Quite a rich variety of regions are supported in AIPS++.

There are simple regions like a box or a polygon, and compound regions like unions and intersections. Regions may contain their own "region masks". For example, with a 2-d polygon, the region is defined by the vertices, the bounding box and a mask which says whether a pixel inside the bounding box is inside of the polygon or outside of the polygon.

In addition, although masks and regions are used somewhat differently by the user, a mask is really a special kind of region; they are implemented with the same underlying code.

Like masks, regions can be persistently stored in image. From Glish, regions are generated, manipulated and stored with the Regionmanager tool.

We saw in the previous section how the condition operator `[]` could be used to generate masks with logical expressions. This operator has a further talent. A region of any type can be applied to a lattice with the `[]` operator. You can think of the region as also effectively being a logical expression. The only difference with what we have seen before is that it results in a lattice with the shape of the region's bounding box. If the lattice or the region (as in the polygon above) has a mask, they are and-ed to form the result's mask.

All types of regions supported in AIPS++ can be used, thus:

- regions in pixel or world coordinates
- in absolute, relative and/or fractional units
- basic regions box, ellipsoid, and polygon
- compound regions union, intersection, difference, and complement.
- extension of a region or group of regions to higher dimensions
- masks

The documentation in the classes LCRegion, LCSlicer, and WCRegion) gives you more information about the various regions.

At this moment a region can not be defined in LEL itself. It is only possible to use regions predefined in an image or another table.

When using Glish (as will normally be done), it is also possible to use a region defined in Glish using the $-notation. This is explained in more detail in the section discussing the interface to LEL.

A predefined region can be used by specifying its name. There are three ways to specify a region name:

1. `tablename::regionname`
   The region is looked up in the given table (which will usually be an image) in which it is stored.

2. `::regionname`
   The region is looked up in the last table used in the expression.

3. `regionname`
   Is usually equivalent to above. However, there is no syntactical difference between the name of a region and a lattice/image. Therefore LEL will first try if the name represents a lattice or image. If not, the name is supposed to be a region name. The prefix `::` in the previous way tells that the name should only be looked up as a region.

Examples are

```
myimage.data[reg1]
(myimage.data - otherimage)[::reg1]
(myimage.data - otherimage)[myimage.data::reg1]
myimage.data:nomask[myotherimage::othermask]
```

In the first example region `reg1` is looked up in image `myimage.data`. It is assumed that `reg1` is not the name of an image or lattice. It results in a lattice whose shape is the shape of the bounding box of the region. The mask of the result is the and of the region mask and the lattice mask.

In the second example it is stated explicitly that `reg1` is a region by using the :: syntax. The region is looked up in `otherimage`, because that is the last table used in the expression. The result is a lattice with the shape of the bounding box of the region.

In the third example the region is looked up in `myimage.data`. Note that the this and the previous example also show that a region can be applied to a subexpression.

In the fourth example we have been very cunning. We have taken advantage of the fact that masks are special sorts of regions. We have told the image `myimage.data` not to apply any of its own masks. We have then used the `[]` operator to generate a mask from the mask stored in a different image, `myotherimage`. This effectively applies the mask from one image to another. Apart from copying the mask, this is the only way to do this.

Unions, intersections, differences and complements of regions can be generated and stored (in C++ and Glish). However, it is also possible to form a union, etc. in LEL itself. However, that can only be done if the regions have the same type (i.e. both in world or in pixel coordinates).
The following operators can be used:

- `reg1 || reg2` to form the union.
- `reg1 && reg2` to form the intersection.
- `reg1 - reg2` to form the difference.
- `!reg1` to form the complement.

The normal AIPS++ rules are used when a region is applied:

- A region in world or relative coordinates can only be applied to an image (or a subexpression resulting in an image). Otherwise there is no way to convert it to absolute pixel coordinates.
- The axes of a region in world coordinates have to be axes in the image (subexpression). However, the region can have fewer axes.
- If a region has fewer axes than the image or lattice the region is automatically extended to the full image by taking the full length of the missing axes.

5 Some further remarks

5.1 Optimization

When giving a LEL expression, it is important to keep an eye on performance issues.

LEL itself will do some optimization:

- As said in the introduction a LEL expression is evaluated in chunks. However, a scalar subexpression is executed only once when getting the first chunk. E.g. in

  ```
  lat1 + mean(lat2)
  ```

the subexpression `mean(lat2)` is executed only once and not over and over again when the user gets chunks.

- Often the exponent 2 is used in the `pow` function (or operator `^`). This is optimized by using multiplication instead of using the system pow function.
- When LEL finds a masked-off scalar in a subexpression, it does not evaluate the other operand. Instead it sets the result immediately to a masked-off scalar. Exceptions are the operators AND and OR and function `iif`, because their masks depend on the operand values.

The user can optimize by specifying the expression carefully.

- It is strongly recommended to combine scalars into a subexpression to avoid unnecessary scalar-lattice operations. E.g.

  ```
  2 * lat1 * pi()
  ```

  should be written as

  ```
    lat1 * (2 * pi())
  or
    2 * pi() * lat1
  ```

  because in that way the scalars form a scalar subexpression which is calculated only once. Note that the subexpression parentheses are needed in the first case, because multiplications are done from left to right.
  In the future LEL will be optimized to shuffle the operands when possible and needed.

- It is important to be careful with the automatic data type promotion of single precision lattices. Several scalar functions (e.g. pi) produce a double precision value, so using `pi` with a single precision lattice causes the lattice to be promoted to double precision. If accuracy allows it, it is much better to convert `pi` to single precision. E.g. assume `lat1` and `lat2` are single precision lattices.

  ```
  atan2(lat1,lat2) + pi()/2
  ```

The result of `atan2` is single precision, because both operands are single precision. However, `pi` is double precision, so the result of `atan2` is promoted to double precision to make the addition possible. Specifying the expression as:

```
atan2(lat1,lat2) + float(pi())/2
```

avoids that (expensive) data type promotion.

- `POW(LAT,2)` or `LAT^2` is faster than `LAT*LAT`, because it accesses lattice `LAT` only once.
- `SQRT(LAT)` is faster than `LAT^0.5` or `POW(LAT,0.5)`
- `POW(U,2) + POW(V,2) < 1000^2` is considerably faster than `SQRT(SQUARE(U) + SQUARE(V)) < 1000`, because it avoids the `SQRT` function.
- LEL can be used with disk-based lattices and/or memory-based lattices. When used with memory-based lattices it is better to make subexpressions the first operand in another subexpression or a function. E.g.
  `lat1*lat2 + lat3`
  is better than
  `lat3 + lat1*lat2`
  The reason is that in the first case no copy needs to be made of the lattice data which already reside in memory. All LEL operators and functions try to reference the data of their latter operands instead of making a copy.
  In general this optimization does not apply to LEL expression. However, when using the true C++ interface to classes like `LatticeExprNode`, one can easily use memory-based lattices. In that case it can be advantageous to pay attention to this optimization.

5.2 Mask Storage

In many of the expressions we have looked at in the examples, a mask has been generated. What happens to this mask and indeed the values of the expression depends upon the implementation. If for example, the function you are invoking with LEL writes out the result, then both the mask and result will be stored. On the other hand, it is possible to just use LEL expressions but never write out the results to disk. In this case, no data or mask is written to disk. You can read more about this in the interface section.

6 Interface to LEL

There are two interfaces to LEL. One is from Glish and the other from C++. It depends upon your needs which one you access. Most high level users of AIPS++ will access LEL only via the Glish interface.

6.1 Glish Interface

The LEL interface in Glish is provided by the Image tool (loaded with the script `image.g`) imagecalc constructor, and through the calc tool function. These can be used to compose and/or execute a LEL expression.

6.1.1 Simple String Expressions

The `imagecalc` constructor evaluates the expression and stores the result and mask in an output image. If you specify the output image name, it is written to a disk file of that name. If you don't give it, the output image is never written out; it is evaluated every time an action (like statistics) is requested.

```
im := imagecalc(outfile='outimage', pixels='inimage1+inimage2');
im.statistics();
```

The first command creates an image file `outimage` filling it with the sum of the input images. The second command does statistics on that new image.

Writing it as

```
im := imagecalc(pixels='inimage1+inimage2');
im.statistics();
```

would do the same with the exception of creating the output image. Instead the created image is transient; it only lives as an expression and each time it is used the expression is evaluated.

We can use the function `calc` on an already existing image. Thus

```
im := image('ngc1213');
im.calc('ngc1213^2');
```

would replace the pixels by the square of their value.

Sometimes you need to double quote the file names in your expression. For example, if the images reside in a different directory as in this example.

```
im := imagecalc ('"dir1/im1" + "/nfs/data/im2"');
```

6.1.2 Symbol (or tool) substitution

Images created/opened in Glish can be used with their Glish name in a LEL expression. Similarly a region created in Glish with the Regionmanager can be used with its Glish name.

It is also possible to embed other Glish variables and expressions in a LEL command using the syntax `$variable` and `$(expression)`. A variable can be a standard numeric scalar. An expression has to result in a numeric scalar.

E.g. one can make a Glish function like

```
img1 := image('"/data/inimage'");
img2 := imagecalc(pixels='min($img[$someregion], $clipvalue)');
```

to return a transient image formed by a region in an image clipped at the given clipvalue.

The substitution mechanism is described in more detail in the substitute functions of module misc of the User Reference Manual.

The substitution mechanism uses the eval function in Glish. As of 15-Jan-1999 eval only looks at global variables. This means that in a function one needs to create a global variable (with a unique name) if the variable is to be used in a LEL command. The global variable should be deleted at the end of the function (with `symbol_delete`). The name can be made unique by using the function name as a suffix.

In Glish applying a region to an image can also be done using the subim function of an Image tool. It forms an image from the given region in the input image. This new image can be used in a LEL expression. E.g.

```
img := image('inimage');
subimg := img.subimage(region=someregion);            # Virtual Image tool (a reference)
img2 := imagecalc(pixels='min($subimg, $clipvalue)');
```

has the same result as the earlier example. Both **subimg** and **im2** are virtual images (not written to disk).

6.2 C++ interface

This consists of 2 parts.

1. The function `command` in Images/ImageExprParse.h can be used to execute a LEL command. The result is a LatticeExprNode object. E.g.

   ```
   LatticeExprNode seltab1 = ImageExprParse::command
          ("imagein1 + imagein2");
   ```

   This example does the same as the Glish one shown above.

2. The other interface is a true C++ interface having the advantage that C++ variables can be used. Class LatticeExprNode contains functions to form an expression. The same operators and functions as in the command interface are available. E.g.

   ```
   Float clipValue = 10;
   PagedImage<Float> image("imagein");
   LatticeExpr<Float> expr(min(image,clipValue));
   ```

   forms an expression to clip the image. Note that the expression is written as a normal C++ expression. The overloaded operators and functions in class LatticeExprNode ensure that the expression is formed in the correct way.
   Note that a `LatticeExprNode` object is usually automatically converted to a templated `LatticeExpr` object, which makes it possible to use it as a normal `Lattice`.
   So far the expression is only formed, but not evaluated. Evaluation is only done when the expression is used in an operation, e.g. as the source of the copy operation shown below.

   ```
   PagedImage<Float> imout("imageout");
   imout.copyData (expr);
   ```

7 Examples

The following examples show some LEL expressions (equally valid in C++ or Glish).

Note that LEL is readonly; i.e. it does not change any value in the images given. A function in the `image` client has to be used to do something with the result (e.g. storing in another image).

`lat1+lat2`
: adds 2 lattices

`mean(myimage:nomask)`
: results in a scalar value giving the mean of the image. No mask is used for the image, thus all pixels are used. The scalar value can be used as a lattice. E.g. it can be used as the source in the `image` function `replacemaskedpixels` to set all masked-off elements of a lattice to the mean.

`complex(lat1,lat2)`
: results in a complex lattice formed by `lat1` as the real part and `lat2` as the imaginary part.

`min(lat1, 2*mean(lat1))`
: results in a lattice where `lat1` is clipped at twice its mean value.

`min(myimage, 2*mean(mymage[myregion]))`
: results in an image where `myimage` is clipped at twice the mean value of region `myregion` in the image..

`lat1[lat1>2*min(lat1)]`
: results in a lattice with a mask. Only the pixels greater than twice the minimum are valid.

`replace(lat1)`
: results in a lattice where each masked-off element in `lat1` is replaced by 0.

`iif(lat1<mean(lat1),lat1*2,lat1/2)`
: results in a lattice where the elements less than the mean are doubled and the elements greater or equal to the mean are divided by 2.

Here follows a sample Glish session showing some of the LEL capabilities and how Glish variables can be used in LEL.

```
duw01> glish -l image.g
- a := array(1:50,5,10)              # make some data
- global im1 := imagefromarray('im1', a);   # fill an image with it
```

```
- im1.shape()
[5 10]
- local pixels, mask
- im1.getregion(pixels, mask);       # get pixels and mask
- mask[1,1] := F                     # set some mask elements to False
- mask[3,4] := F
- im1.putregion(mask=mask);          # put new mask back
- global reg:=drm.box([1,1],[4,4]);  # a box region
- im2 := imagecalc(pixels='$im1[$reg]')     # read-only image applying region
- local pixels2, mask2
- im2.getregion(pixels2, mask2);     # get the pixels and mask
- print pixels2
[[1:4,]
    1 6 11 16
    2 7 12 17
    3 8 13 18
    4 9 14 19]
- print mask2
[[1:4,]
    F T T T
    T T T T
    T T T F
    T T T T]
- im1.replacemaskedpixels ('mean(im2)'); # replace masked-off values
- im1.getregion (pixels2, mask2);        # by mean of masked-on in im2
- print pixels2
[[1:5,]
    10.0714283 6  11 16         21 26 31 36 41 46
    2          7  12 17         22 27 32 37 42 47
    3          8  13 10.0714283 23 28 33 38 43 48
    4          9  14 19         24 29 34 39 44 49
    5          10 15 20         25 30 35 40 45 50]
```

8 Future developments

In the near or more distant future LEL will be enhanced by adding new features and by doing optimizations.

- Handle slices like `image[1:10:2, 5:]`
  which defines the section directly in pixel coordinates (with stride 2

for the first axis).

- Do optimization by reordering the expression.
- Do optimization by recognizing common subexpressions.