Lattakia[1] (Lattice Kiaugh)

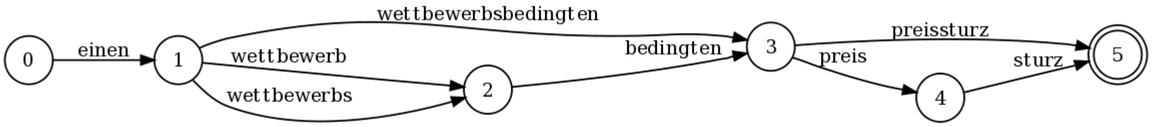
Language Reference Manual

Heba Elfardy <hme2110 >, Li Yifan < yl2774 >,
Dara Hazeghi<dmh2186>, Wael Salbum<wss2113>

Why Lattakia?

Lattakia is a programming language that deals with all the *latt*ice *kia*ugh (kiaugh means trouble or anxiety). It's designed to focus on using lattices as a main data structure and code structure. In fact, Lattakia work on *word lattices*, which are special kinds of lattices (partially-ordered sets).

An example of a word lattice is:



http://www.statmt.org/moses/?n=Moses.WordLattices.

Word lattices are powerful representation models. They are used in many applications; e.g., in NLP, they are useful wherever there is ambiguity (uncertainty) in the meaning (or interpretation) of a word; such as in speech recognition, machine translation, language (or dialect) identification, language models, paraphrasing (e.g., in information retrieval and question answering), and other applications.

However, many researchers avoid using them because they are hard to implement; instead they use top 'n' paths. Recently, many free NLP tools that accept lattices as input have been implemented (e.g. Moses, SRILM, etc.); thus, we will build a programming language that allows programmers to build lattices, process them and provide them to other tools.

[1] Lattakia (or Latakia) is a city on the Mediterranean Sea that is over 4000 years old. It was called Ramitha and was a part of the kingdom of Ugarit. Later, under the Roman rule, it was called Laodicea.

Table Of Contents

Introduction to Word Lattices

Defining the language:

1. Atom definition: an atom is any expression (as in other programming languages) such as assignment, mathematical expressions, conditions, function calls, etc.
2. Word lattice definition:
    1. An "atom" is a lattice;
    2. "lattice | lattice" is a lattice (we call it alternative lattice or altlat);
    3. "lattice ; lattice" is a lattice we call it sequence lattice or seqlat).

In this language, both data and code are written in this lattice format.

To represent the lattice in a way similar to other programming languages, we think of a lattice as an array of variables where variables can have multiple values. For example, if we want to build a Part-Of-Speech (POS) tagger for in following example, we need to give variables multiple values:

```
 Time   flies  like   an  arrow.
N|V|Adj  N|V    V|Adv  Det   N
   1.  N V Adv Det N
   2.  Adj N V Det N
   3.  V N Adv Det N
```

Variables (or alternative lattices; or altlats):

A variable can have multiple values at the same time:

```
a = 5;
x = 3 | 7 | 2;
isImportant = [x >= 75] false | [x < 75] true;
isEmptyStack = [stack==epsilon] true | false;
```

Notice that isImportant and isEmptyStack are variables here while they are functions in other programming languages. 'x' is called an "*independent variable*" while 'isImportant' is called an "*dependant variable*" because it depends on 'x'.

Lattakia is a language that handles this kind of variables.

INDEXING:

To access the second value of x (which is 7):

```
x{1}
```

As in C++, index start from 0.

Lattices (or sequence lattices; or seqlats):

An array of this kind of variables is actually a lattice.

```
yLat = (v1; v2; v3);
v1 = 4; v2 = x; v3 = a|2;
```

➔

```
yLat = (4; x; a|2);
```

INDEXING:

To access the second value of yLat (which is x):

```
yLat[1]
```

As in C++, index start from 0.

Note that {} are used for alternative lattices while [] are used for sequence lattices

We call a variable an "alterntives lattice" (or altlat) because a variable with multiple values is a lattice of multiple alternatives. In contrast, we call a lattice a "sequence lattice" (or seqlat)

Examples:

The following lattices are shown in the figures below.

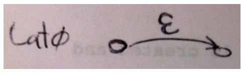
```
        lat0 = ();
```

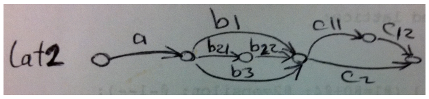
A lattice with an epsilon transition; i.e., an empty lattice.

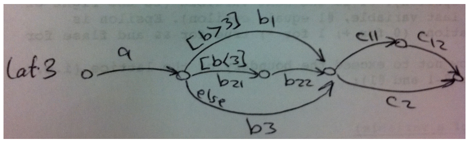
```
        lat1 = load ("file1.lat");
```

'load' is a optimized built-in function to read a lattice from a file in our format.

```
        lat2 = (a; b1|(b21; b22)|b3; (c11; c12)|c2);
        lat3 = (      a;
                      [b>3] b1 | [b<0](b21; b22) | b3;
                      (c11; c12) | c2);
        lat2[1]{1} = b2;
        lat3[1]{1}[0] = b21 * 3;
```







Labels:

```
        latLab = (l1: a; l2: b1|b2);
```

Labels are actually similar to labels in other programming languages; they are names assigned to be used later.

Think of labels here as variable handles or names of the nodes. When accessing a label you actually get the variable that comes after this label's node. Labels can be used later to access variables inside the lattice. Labels represent keys in hash tables and fields in objects. To access a label from inside it's lattice you just need to use it's name. To access a label from outside it's lattice you need to use the "." operator:

```
        x = latLab.l2;
```

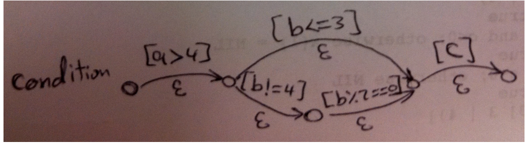
Conditions are also lattices:

```
        [a>4 && b<=3 || (b!=4 && b%2==0) && c]
```

or:

```
        [a>4; b<=3 | (b!=4; b%2==0); c]
```

Think of it as: this is a list of conditions that must be satisfied. Or think of it as a lattice of epsilons on the arcs guarded by these conditions as shown in the figure below:



Types, Operators, and Expressions

Assignment and Evaluation:

```
        y = 3;
```

The r-value is a constant ➔ y's value is 3

```
        yIsEven = (y % 2 == 0);
```

'yIsEven' is assigned the equation not the current value. '(y % 2 == 0)' is not evaluated here; i.e. when 'y' changes, 'yIsEven' changes too. Here, there is no problem if y is not initialized. This is as defining functions in programing languages, the only difference is that they look at it form an Assembly perspective (a function is a routine or procedure that returns a value) while we look at it from a mathematical perspective (variable yIsEven is a function of variable y);

```
        x = ?(y + 3); // or: x = ?y + 3;
```

The '?' operator takes one operand (to the left) and evaluates it. Here the value of x is 3+3 = 6.

```
        z = ?y - 2*x;
```

Here z is a function of x but not of y ➔ z = 3 – 2*x;

```
        w ?= z++ - x*y;
```

The operator '?=' evaluates the r-value and assigns it to the l-value.

The assigment statement evaluates to its l-value:

```
        x = y = 3;          // Here y=3 and x=y and the result is x
        x ?= y = 3;         // Here y=3 and x=3 and the result is x
        x = ?y = 3;         // Here y=3 and x=3 and the result is x
        x = ?y = z;         // Here y=z and x=z and the result is x
        ?x = y = 3;         // Here y=3 and x=y and the result is 3
```

If you don't want to return x or its value, use the keyword 'let' that returns epsilon:

```
        let x = y = 3;      // Here y=3 and x=y and the result is epsilon
```

Left-preference assignment V.S. Right-preference assignment:

Both '=' and '?=' are right-preference assignment operators while '^=' is the left-preference assignment operator:

```
?x = ?y;  // y is evaluated; x = y's value; x is evaluated; x returned.
?x ^= ?y; // x is evaluated; x returned; y is evaluated; x = y's value.
```

```
        x = 1; y = 2; z = 3
        ?x = ?y = ?z = 0;    ⇔    let z = 0; let y = ?z; let x = ?y; ?x;
```

Now all x, y and z are equal to 0. Exactly as in other programing languages.
While in the following statement:

```
        ?x ^= ?y ^= ?z ^= 0;    ⇔    ?x; let x = ?y; let y = ?z; let z = 0;
```

x equals old y, y = old z and z equals 0; then, x's value is returned.

Operations on Lattices:

Concatenation:

To concatenate two lattices:

```
lat41 = lat1 ~ lat2;
lat42 = (lat1; lat2);
lat42 ~= lat3;
lat42[2] ~= lat2;
lat3[1]{0} ~= b12;
```

Add alternative:

To add an alternative:

```
lat5 = lat1 | lat2; // lat5 is an altlat that has the two lattices
lat5 |= lat3;       // lat5 now is an altlat.
lat5[2] |= lat2;    // giving alternative to the third element in lat5
lat3[1]{1}[1] |= b222;
```

Remove a variable for a lattice:

```
        lat3[1] = epsilon; // 'epsilon' is a keyword.
```

Or:

```
        remove(x) = x = epsilon;
        remove(lat3[1]);
```

Example: Stack:

```
        stack = lat5;
```

Now we've two handles pointing to the same lattice temporarily. However, when we try to change the lattice using one of them, the system will creat a copy of the lattice and change the copy. This is for optimization purposes. However, if you want to only one lattice with two pointers to it, you can use the reference ('&') operator:

```
        Stack = & lat5;

        element = pop(stack); // pop is a built-in function.
        pop(lat) = (lat[0]; let lat[0] = epsilon);
```

or:

```
        pop(lat) = (lat[0]; let lat = & lat[1]);
```

Both evaluate to (lat[0]; epsilon) ➔ lat[0] that is then returned to 'element'.

We notice that the code itself is a lattice and 'pop' here is a conditioned lattice that's evaluated when called. By 'conditioned' we mean that the condition is the parameter 'lat' that may change to change to resulting lattice.

An important thing to notice is that calling the funtion evaluates it's lattice code and returns a value; however, this value is not evaluated. In the previous example, when calling 'pop(stack)', the code lattice of 'pop' is conditioned to 'stack' ➔ '(stack[0]; let stack = & stack[1])' which is evaluated to 'stack[0]'; however, 'stack[0]' is not evaluated; instead, it is returned as a variable with what it has inside it (which could be code). That means that calling a function does not evaluate it's result. To evaluate the result in the previous example:

```
        element = pop(stack)?;
```

Constrained Variables (Functions)

Types Of Variables:

Independent V.S. Dependent Variables:

1. Independent Variables: which are variables that hold values; i.e., they do not depend in their values on any other variable.
```
        x = 5 * 4;
        y = rand() % 10; // rand() returns a value
        z = ?x + ?y;
```

2. Dependent Variables: which are variables that hold code that contains another variable; i.e. they depend in their values on at least one variables.
```
        w = 5 * x;    // 'w' is a function of 'x'
```
The value of 'w' is '5 * x' and not the value of 5 times the value of 'x'; i.e., 'x' is not evaluated and '5 * x' is not evaluated. Later, when 'w' is evaluated then 'x' is evaluated and '5 * x' is evaluated, and the value is returned. Note that 'w' does not hold the value after evaluation; i.e. this process repeats every time 'w' is evaluated.

Constrained V.S. Unconstrained Variables:

Variables in Lattakia must be in one of these two types:

1. Constrained Variables: which are variables that are subject to constrained and the lattice they hold may change when provided different constraints:
```
        f(n) = 1;      // An independent constrained (by n) variable
        y(x) = 5 * x; // A independent constrained variable.
        z(x) = x * y; // A dependent (on y) constrained (by x) variable.
```
2. Unconstrained Variables: which are variables that do not hold constrains:
```
        u = 1;        // An independent unconstrained variable.
        w = x + 1;    // A dependent unconstrained variable.
```

To accessing the data hold by an unconstrained variable you just use its name. To access the data hold by a constrained variable you need to specify the constraint:
```
        z(x) = x * y;
        y = 5;
        w = z(3); // now 'w' is 3+5 = 8
```

Recursion:
```
    factorial(n) = [n<=1] 1 | n * factorial(n-1);
```

Passing parameters:

By reference: `y(x)`

By value: `y(?x)`

A function that returns the maximum of two values and throws an exception if they're equal:
```
    max(a; b) =
    (
        [a>b] return a |
        [a<b] return b |
        [a==b] return NIL;
                    // i.e. the function didn't work properly.
    );
```
The keyword 'return' returns the value and ends the execution of the function; i.e., it adds an epsilon path to the final node.

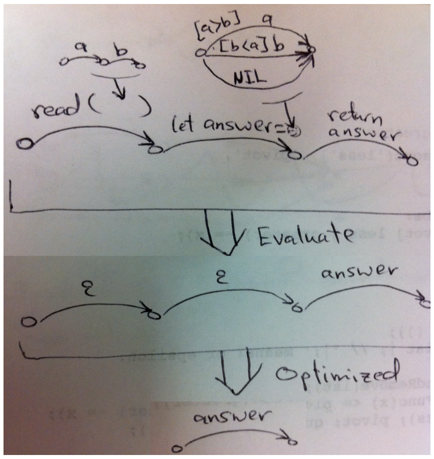
Or simply we can write:

```
        max(a; b) = [a>b] a | [a<b] b | NIL;
        max(a; b) = [a>b] a | [a<b] b otherwise NIL;
        max(a; b) = [a>b] a | [a<b] b else NIL;
```

Keywords ‘otherwise’ and ‘else’ are programming sugar that work as the final condition if no condition was satisfied. In comparison, ‘default’ is a keyword that works when an error happens such as an uninitialized variable evaluation or a type mismatch. ‘default’ works as an exception handler.

```
maxRead =
(
        read(a; b); // reads a and b; then
                    // evaluates to epsilon.

        let answer =
                [a>b] a | [a<b] b
                otherwise NIL;
        return answer;
);
```



NOTE: The previous function ‘maxRead’ evaluates to the seqlat:

```
        (epsilon; epsilon; answer) ➔ answer
```

However, there is no restriction that a function should evaluate to a single variable; it can evaluate to a seqlat which means it can return multiple variable. Let’s take an example:

```
        randomSelect(lat) = (lat[index = rand()%lat.length]; index);
        (x; i) = randomSelect(lat1);
```

This will return a seqlat of two elements that will be assigned to x and i respectively.
The rule here is that, in all assignemt types, if the l-value is an explicit seqlat, then the elements of the r-value lattice is assigned one by one. If the r-value was longer than the l-value, then the last l-value element will hold the rest of the lattice. If the l-value was longer than the r-value, then the rest of the elements in the l-value are assigned epsillons;

Epsilon V.S. NIL:

Epsilon is an empty transition which means that you can always pass through it while NIL is a broken arc that you can pass through it. In a seqlats, you can drop the epsilon and merge its endpoints while you can’t drop NIL and it means that this seqlat is broken and cannot be evaluated and it returns NIL to the lattice that contains it (similar to throwing exceptions). In altlats, you can drop the NIL path because there is other alternative paths (when evaluating an altlats, if NIL is the only accepted path, then it is part of the seqlat that hold this altlat). Epsilons cannot be dropped from altlats because they give an important information: when evaluating an altlat, if we cannot satisfy any condition on the paths before the epsilon path, this means that you still can path through this epsilon transition.

Loops (built-in recursive functions):

In the following, ‘codeLattice’ is a lattice of code (i.e. a piece of code):

```
        while(satisfiedCondition;   codeLattice);
        until(unsatisfiedCondition; codeLattice);
        foreach(element; lattice; codeLattice);               // for seqlats
        foreach(alternativesLattice; codeLattice);     // for altlats
        for(firstOnce; endCondition; lastEach; codeLattice); // as in C++
```

Example: Calculating $\sum_i |x_i|$

Latt

```
// 1. Considering x a seqlat:
sum = 0.0;
x = (5.4; 3.0; -9.2); // Seq
foreach(xi; x; sum+=[xi>0] xi|-xi);

// 2. Considering x an altlat(intuitive):
x = 5.4 | 3.0 | -9.2; // Alt
foreach(x; sum += [x>0] x|-x);

//or use the built-in ‘sum’ for altlats:
sum([x>0] x|-x);
```

C++

```
float x[3] = {5.4, 3, -9.2};
float sum = 0;
for (int i=0; i<len(x); i++)
{
   if (x[i] > 0)
      sum += x[i];
   else
      sum += - x[i];
}
```

Built-in Aggregation functions:

max(x) applyies to x where x is an alternatives lattice.
Aggregation functions are: max, min, avg, sum, and count (which is the same as aLat.count). Note: the size of a seqlat is lat.lenght while the size of an altlat is lat.count.

```
longestPath(lat) =
(
   let sum = 0;
   foreach(x; lat; sum += max(x));
   longestPath = sum;
);

path(lat; opr) =
(
   let sum = 0;
   foreach(x; lat; sum += opr(x));
   path = sum;
);
```

To calculate the shortest path:

```
path(lat; min);
```

Example: Quicksort:

Pseudo-code:

```
function quicksort('array')
       create empty lists 'less' and 'greater'
       if length('array') ≤ 1
           return 'array'  // an array of zero or one elements is
                           // already sorted
       select and remove a pivot value 'pivot' from 'array'
       for each 'x' in 'array'
           if 'x' ≤ 'pivot' then append 'x' to 'less'
           else append 'x' to 'greater'
       return concatenate(quicksort('less'), 'pivot', quicksort('greater'))
```

Quicksort: The for loop can be:

```
foreach(x; lat; ([x <= pivot] less | greater) ~= x);
```

or:

```
for(latt=lat; latt.length = 0; pop(latt);
       ([x <= pivot] less | greater) ~= x);
```

Quicksort in Lattakia (see the figure below):

```
quicksort(lat; AltEvalFunc) =
(
       let (less = (); greater = ());
       [lat.length <= 1] return lat |; // '|;' means: or epsilon.

       let pivot = randomSelectAndRemove(lat);
       foreach(x; lat; ([AltEvalFunc(x) <= pivot] less | greater) ~= x);
       quicksort = (quicksort(less); pivot; quicksort(greater));
);
randomSelectAndRemove(lat) = (
       let ind = rand()%lat.length;
       lat[ind];
       let lat[ind] = epsilon
);
```

// or using right-preference assignement '^='

```
randomSelectAndRemove(lat) = (lat[rand()%lat.length] ^= epsilon );
```

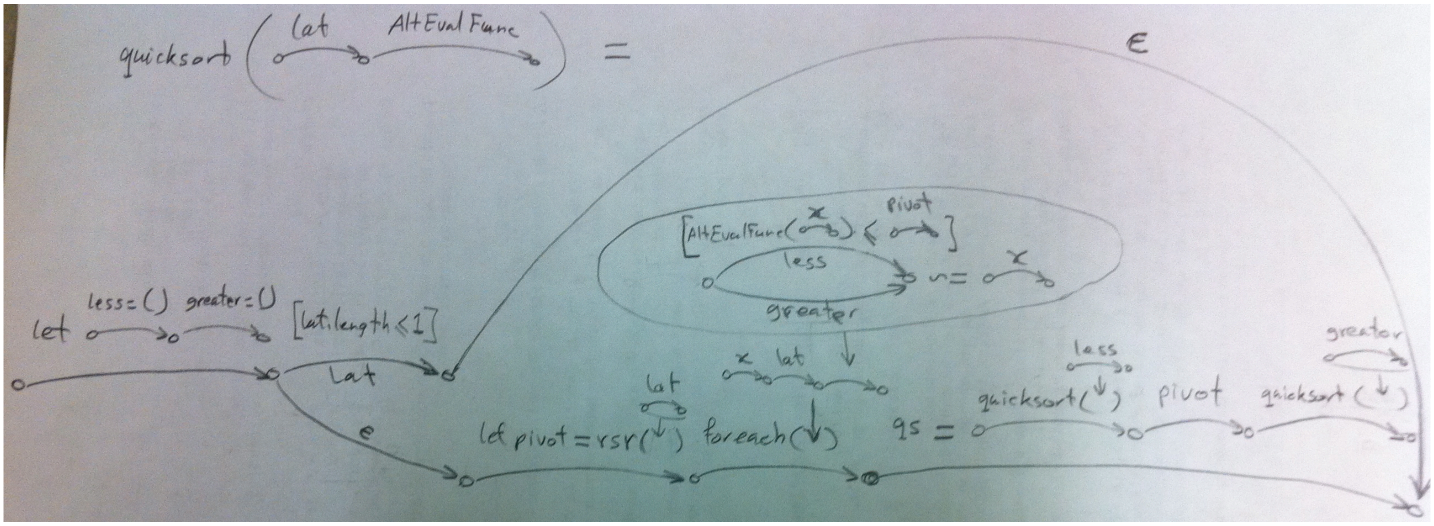
example:

```
       quicksort(lat1; max); // to sort, it selects the 'max' value of x
       quicksort(lat1; avg); // to sort, it selects the 'avg' value of x
       quicksort(lat1; eval); // to sort, it evaluates the conditions and selects
```

the right value of x (which is the first value of a satisfied condition). This is the default case when using x.

To sort a lattice by how many alternatives each variable has:

```
       quicksort(lat1; count);
```



Advanced Data Structures

Lists:

Defined in the language as lattices, and the concatenation operations, remove, and indexing, work as list operations.

Stack and Queue:

Defined in the language as a lattice:

```
        push(lat; x) = lat = lat ~ x;
        pop(lat) = (lat[0]; let lat[0] = epsilon);
        enqueue(lat; x) = lat ~= x;
        dequeue(lat) = pop(lat);
```

Hash Tables:

A hash table is a lattice with a label on each node (except the last node); thus, the labels are keys in the hash table and the variables are values.

Perl	Lattakia

`my %hash = {`
`        x => “a”,`
`        y => “b”,`
`};`
	`hash = (`
	`        x: “a”;`
	`        y: “b”;`
	`);`

Furthermore, to solve **collision** in a hash table, you have the “|” operator to add alternatives to a value in the same location (variable). This is a built-in feature that does not exist in other languages.

Sets:

Labels are set elements.

```
        lat = (red:; blue:; yellow:;);
        [defined(lat.red)] (doSomething);
```

Tuples:

Tuples are seqlats in Lattakia and their elements are the lattices variable.

```
        engine = (8; ‘V’; 300);
```

Search and Rule Triggering

Search (Regular Expressions):

```
        x =~ /^ab$/;
```

A Regular Expression is a lattice. We use Java's notation for regex but inside Perl's double slashes.

```
        latRegEx = (/^ab$/; /abc(.*)/ | /a*cd+/);
```

Regex's are atomic. Thus, We can have lattices of regex's. DEPRICATED. This is programming sugar.

A regex is a condition on strings. When it applies to a string variable, it applies to all its values and returns only what matches.
A regex applies to lattices too. It returns a lattice of each variable match. If a variable doesn't match, it returns NIL in it's place. NIL in a seqlat is an error in the program while NIL in a altlat is an ignored (deleted) path.

Rules: When triggered (condition satisfied), apply action:

Syntax:

```
        ruleName = condition  action;
```

A condition can be of any type including regex. An action can be any statement.

```
r1 = [@0 > 3] @0++;
r2 = /he has a (.*) / => increase(@1);
```

```
apply r1 to lat1; // depricated.
lat1(r1);     // lat1 is changed and returned
lat1->(r1); // lat1 is not changed. The result lattice is created and returned.
lat2 = lat1->(r1); // lat2 is pointing to the result lattice.
```

To keep epsilons in the resulted lattice, define a reserved lattice:

```
        reserved lat2 = lat1->(r1)
```

and that's because the optimization happens after the assignment. Now, this lattice is reserved.

A lattice rule:

```
        rl1 = [@0>0; @1==2; @3+1<@-1] (@1=@0+@4; @2=epsilon; @-1--);
```

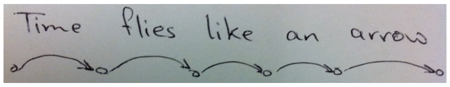
Variable @0 is the current variable in the 'foreach' function. @-1 is the previous variable (to the left of @0); (when @0 is the first variable, @-1 equals epsilon). @1 is the next varaible (to the right of @0); (when @0 is the last variable, @1 equals epsilon). Epsilon is neutral for all operations (0 for +; 1 for *; true for && and flase for ||).
If you want the window not to exceed the boundary of the lattice (i.e. no epsilon values for @-1 and @1):

```
        lat1->(:r1:);
```

Accessing the value of a variable: (Not part of our plan for this semester)

```
x = 5 | 3 | 4;
[x{1} == 3] // true
x = [b<3] 5 | [b>3; c<0] 3 | 4;
[x{1} == [b>3; c<0] 3] // true
[x{1} == 3] // true if b>3 and c<0; otherwise x{1} = NIL
[x{1:b=4} == [c<0] 3] // true
[x{1:b=4} == 3] // true if c<0; otherwise NIL
[x{1:b=4; c=-2} == 3] // true
[x{c=-2} == ([b<3] 5 | [b>3] 3 | 4)]
[x{b=4} == ( [c<0] 3 | 4)]
```

Example: Part-of-Speech (POS) Tagger:



Look at lattice l1 in the figure above. To create a Part-of-Speech (POS) Tagger for English, we need the following:

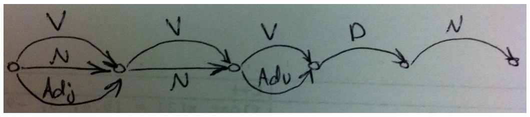
1. Lexical Rules: E.g.,

```
        lr1 = [@0 == "time"] (@0 = V|N|Adj);
        lr2 = [@0 == "flies"] (@0 = V|N);
```

Then we apply these rules like this:

```
        l2 = l1->(lr1 | lr2);
```

See l2 in the following figure:



Well, some of these paths are not possible in English (e.g., V V V D N); thus we need the next set of rules (Syntactic rules) to eliminate them.

2. Syntactic Rules:

```
        sr1 = [@0==D; @1==N] (@0=NP; @1=epsilon);
        sr2 = [@0==Adj; @1==N] (@0=NP; @1=epsilon);
```

These two rules will change "Determiner + Noun" and "Adjective + Noun" to "Noun Phrase".

```
        sr3 = [@-1==epsilon; @0==NP; @1==V; @2==NP; @3==epsilon] (@0; @1; @2);
```

This rule will apply to a full sentence. So if we go and say:

```
        synRules = ((sr1 | sr2); sr3);
        l3 = l2->(synRules);
```

Here "phrase rules" sr1 and sr2 apply first, then "sentence rules" sr3 apply.

The more English grammar rules you add, the more possible paths will be returned. For example the following rules:

```
        sr100 = [@0==Adj; @1==N; @2==V; @3==D; @4==N] (@0; @1; @2; @3; @4);
```

this rule interprets the sentence as in: "Fruit flies like an apple."

```
        sr101 = [@0==V; @1==N; @2==Adv; @3==D; @4==N] (@0; @1; @2; @3; @4);
```

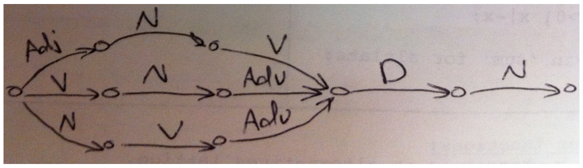
this rule interprets the imperative sentence as in: "Time rehearsal like an actor!"

```
        sr102 = [@0==N; @1==V; @2==Adv; @3==D; @4==N] (@0; @1; @2; @3; @4);
```

this rule gives the correct semantic interpretation.

These three rules will give three syntactically-correct paths in the output lattice. Other English grammar rules will not apply to this sentence.

```
        l4 = l2->(sr100 | sr101 | sr102);
```



To select the semantically-correct path, we need another set of rules that can be obtained, for example, by training an English language model and then representing the resulted database as Semantic Rules.

Object-Oriented Lattakia (Under construction)

Classes:

To create a class in Lattakia, you simply need to create lattice that hold the code of that class. In other programming languages, you think of a class as a collection of data and functionality. In Lattakia functions are data.

```
        Car = (
                type: sedan | coupe;
                color: epsilon;
                engine: epsilon;
                numDoors: [type==coupe] 2|4;
                numWheels: 4;
                transmit: transSpeed++; // perform transmission.
                transSpeed: 0;       // transmission speed when started.
                changeColor(newColor): color = newColor;
        );
```

Labels are exactly the same as variables and they are treated in expressions exactly the same. Labels can also be constrained: changeColor.

Objects:

To build an object we simply assign a copy of a class lattice to the desired object:

```
        c1 = Car; // create an exact copy with the default initialization.
        c2 = Car->(color = red; engine = (8; V));
        c1.changeColor(blue);
        print (“number of cylinders is ” + c2.engine[0]);
        c2.transmit; // c2.transSpeed now equals 1
```

Note that you don’t need to write a constructor (except if you have special processing before initialization); the constructor is built in (same as applying rules with no conditions to a lattice).

Inheritance: Start from the general to the specific:

Inheritance is simply the same as composition.

A car can be a Bus or an SUV:

```
        Bus = (Car;
                type: school | public | company;
        );
        Truck = Car ~
        (
                maxCapacity: epsilon;
        );
```

Overriding members:

```
        TiptronicCar = (car: Car;
                transmit: ( // overriding ‘transmit’
                        car.transmit; // calling parent’s ‘transmit’.
                        increaseMaxSpeed(transSpeed)
                        );
                increaseMaxSpeed(amount): Engine.increaseMaxSpeed(amount);
                maxSpeed: epsilon;
        );
```

In this example ‘increaseMaxSpeed’ is actually brought from another class (Engine) to be a member of this class.

Omitting unwanted inherited members: (only in Lattakia)

```
	AutomaticCar = Car->(transmit: NIL) ~
	(
		capacity: epsilon;
	);
```

<u>Generalization: Start from the specific to the general:</u>

```
We want to build a general type of ‘Vehicle’ starting from specific vehicle
types:
	Cart = (
		color: epsilon;
		material: epsilon;
		numWheels: epsilon;
	);
	Car = (Cart->(material=metal; numWheels=4);
		engine: epsilon;
	);
	Motorcycle = (
		engine: epsilon;
		numWheels: 2;
	);
	Horse = (
		numLegs: 4;
		age: epsilon;
	);
	Vehicle = (
		[type==c] Car |
		[type==m] Motocycle;
		[type==hc] (Horse; Cart->(material=wood));
		type: c|m|hc;
	);

	v1 = Vehicle->(type=m);
	v1.numWheels; // equals 2
```

Inheritance means that you define the general and then specify while generalization means that you define the specific and then the general (in Lattakia this does not mean that you lose the details). You see that numWheels in Vehicle is not generalized to ‘x’ (an unknown value); instead, it is generalized to “2 when the vehicle is a car and 4 when the vehicle is a motorcycle”.

To check if a label is a member of an object, use the built-in “defined” function which has the following code:

```
	defined(x) = [x==NIL] false | true;
```

then you can write this code:

```
	[defined(v1.numDoors)] print (“Number of doors is ” + v1.numDoors);
```