# Lattakia

## Language Reference Manual (v1.0/2011)

**Heba Elfardy** <hme2110@columbia.edu>
**Dara Hazeghi** <dmh2186@columbia.edu>
**Wael Salloum** <wss2113@columbia.edu>
**Li Yifan** <yl2774@columbia.edu>

## Contents

1. Introduction. .... 4
2. Lattices .... 5
3. Lexical conventions .... 7
   a. Comments .... 7
   b. Identifiers .... 7
   c. Keywords .... 7
   d. Constants .... 7
4. Types .... 9
   4.1 Lattices .... 9
   4.1.1 Sequence Lattices .... 9
   4.1.2 Alternative Lattices .... 10
   4.1.3 Named variables: .... 10
   a. Independent V.S. Dependent Variables: .... 10
   b. Constrained V.S. Unconstrained Variables: .... 11
   4.1.4 Labels .... 11
   4.1.4 Evaluation .... 11
   4.2 Numbers .... 13
   4.2.1 Integers .... 13
   4.2.2 Real Numbers .... 13
   4.3 Strings .... 13
   4.4 Booleans .... 13
   4.5 Regular Expression .... 13
5. Expressions and Operators .... 15
   5.1 Arithmetic expressions .... 15
   Unary Operators .... 16
   5.2 String expressions .... 16
   5.3 Boolean expressions .... 16
   5.4 Comparison expressions .... 17
   5.5 Assignment expressions .... 18
   5.6 Evaluation Expressions .... 19
   5.6.1 Evaluation: *? expression* .... 19

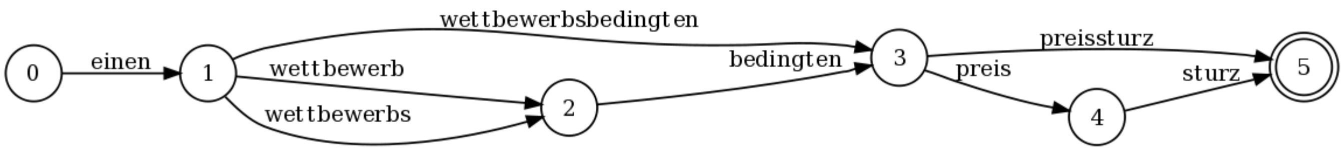
5.6.2 Preventing Optimization: ^ *expression* ..... 19
5.6.3 Application: *name (lattice)* ..... 20
5.7 Alternative expression: *expression | expression* ..... 20
5.8 Sequence expression: *expression ; expression* ..... 20
5.9 Concatenation expression: *expression ~ expression* ..... 21
5.10 Labeling expressions: *label: value* ..... 21
5.11 Conditional expressions: *[condition]* ..... 21
5.12 Each expressions: *each x* ..... 22
5.13 Parenthetic expressions: *(a; b; c)* ..... 22
5.14 Braces expressions: *x{index}* ..... 22
5.16 Accessor expressions: *'.'* ..... 23
5.17 Hash expressions: # ..... 23
6. Declarations ..... 24
7. Attributes and Keywords ..... 24
8. Built-in Library Functions ..... 27
9. Rule Application ..... 28
9.1 Rules ..... 28
9.2 Example: Part-of-Speech (POS) Tagger ..... 28
10. Examples ..... 30
10.1 Object-Oriented Programming ..... 30
10.2 Collections (Advanced Data Structures) ..... 32
10.2.1 Arrays: ..... 32
10.2.2 Stacks and Queues: ..... 32
10.2. 3 Hash Tables: ..... 33
10.2.4 Sets: ..... 33
10.2.5 Tuples: ..... 33
10.2.6 Trees: ..... 33
10.2.7 Graphs: ..... 34
Appendix A – Syntax Summary ..... 36
Appendix B – The Scanner ..... 39
Appendix C – The Parser ..... 40

## 1. Introduction.

This manual describes the Lattakia programming language. Lattakia is a compact functional language built around the *word lattice* structure. Word lattices are a special case of lattices (partially-ordered sets). In Lattakia, both program code and data are stored in these lattices. Standard programming constructs such as conditionals, loops and functions all lend themselves to this representation.

Word lattices are powerful representation models. For example in Natural Language Processing (NLP), they are commonly used in applications where there is ambiguity (uncertainty) in the meaning (or interpretation) of a word, such as in automatic speech recognition, machine translation, language and/or dialect identification, language models, paraphrasing (e.g. in information retrieval and question answering) as well as many other applications.

Despite their importance, researchers tend to avoid using word-lattices because of the inherent difficulty in implementing them. Only recently have some NLP tools such as Moses and SRILM started to support word lattices natively by accepting them as input. However, their application is not limited solely to this domain. Lattakia is designed to make the processing of such complicated data structures more convenient as well as providing the required functionality for developing general-purpose programs.

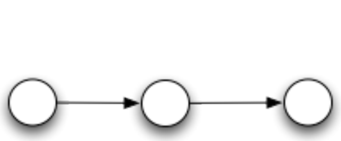


An example of a word lattice from Moses, http://www.statmt.org/moses/?n=Moses.WordLattices

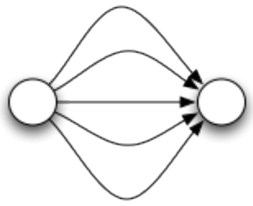
## 2. Lattices

Conceptually, lattices may be thought of as a specialized type of graph. Each lattice has a start and an end node. The arcs in between are the expression in the lattice. The nodes are the position in the execution of the lattice. Each arc that is traversed corresponds to the evaluation of the expression associated with that arc.

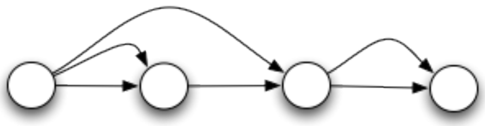
Lattices are divided into two categories: sequence lattices and alternative lattices. Sequence lattices have many nodes, but only a single arc from one to the next. Alternative lattices meanwhile have only two nodes but potentially many arcs in between them. One corresponds to a sequence of expressions or statements, the other to a collection of alternatives.



Sequence Lattice (seqlat)



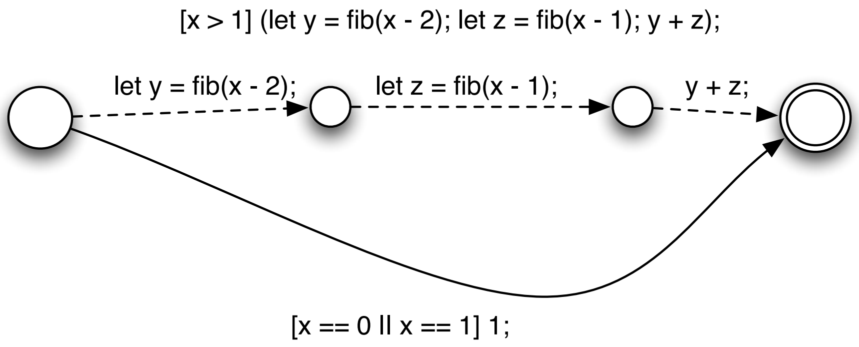
Alternative Lattice (altlat)



Lattice (combination of seqlat and altlat)

In Lattakia, there is no significant distinction between code and data. Both are represented as lattices. Executing code is simply a matter of evaluating each expressions in a lattice sequentially. These expressions correspond to the standard programming constructs - arithmetic, simple variable assignment and the like. Lattices may be composed of other lattices indefinitely, which allows for creating and operating upon complex data structures.

As a functional language, every valid expression in Lattakia can be evaluated. However, Lattakia also features delayed evaluation to allow for dependent variables. Dependent variables are variables bound to expressions that reference other variables. Thus when those other variables change, the result of evaluating the dependent variable will change as well.



Lattice for a simple Fibonacci number calculator

Because all data and code are contained within lattices, being able to access components of lattices is important. Lattakia has several operators to allow such access. The ability to label nodes within a lattice provides easy access to those expressions outside.

Scoping is done at the lattice level. Labels have global scope. Variables have lattice scope. As a result, variables are not visible outside the lattice where they are first bound, while labeled expressions can be accessed both from inside the same lattice, and from anywhere else.

## 3. Lexical conventions

There are four basic kinds of tokens: *identifiers, keywords, constants, and expression operators*.

In general blanks, tabs, newlines, and comments as described below are ignored except as they serve to separate tokens. At least one of these characters is required to separate adjacent identifiers, constants, and certain operator pairs.

### a. Comments

The language has both single-line and multi-line comments. Commented lines start with double-periods ".." and end when a newline is encountered while multi-line comments start and end with double-asterisks "**".

### b. Identifiers

An identifier is a sequence of letters, digits and underscores; the first character must be a character. Identifiers are case-sensitive.

### c. Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

| | | | | |
|---|---|---|---|---|
| General keywords: | **let** | **this** | **else**[1] | **return** |
| Constant keywords: | **true** | **false** | **nil** | **epsilon** |
| Attribute keywords: | **length** | **count** | **clone** | **labels** |

### d. Constants

There are several kinds of constants, as follows:

- **Integer Constants:**
  An integer consists of a sequence of digits and an optional minus sign preceding the first digit to indicate that the integer is negative.

- **Float Constants:**
  A floating constant consists of an integer part, a decimal point, a fraction part and an optional minus sign preceding the first digit of the integer part.

- **String Constants:**
  A string is a sequence of characters surrounded by double quotes " " ". String constants start and end with double-quotations and cannot contain double or single quotes.

- **Regular Expression Constants:**
  A regular expression constant is a sequence of characters surrounded by two forward slashes " / ". Regular expressions use the same wild cards used in Java.

- **"Java-Code" Constants:**

[1] The 'else' keyword is not implemented currently but reserved for future implementations.

A Java-code constant is a sequence of characters surrounded by two grave accents “ ` ”. Java code is not processed; instead, it’s passed as is to the generated code.

## 4. Types

Lattakia is a functional language. As with all functional languages, every expression in Lattakia must be possible to evaluate. Depending on which operators and operands appear in the expression, the type of this expression may be a lattice (sequence or alternative), a number (integer or real), a string, a boolean value or a regular expression. The types of these expressions are implicit (there are no explicit type 'casts'). At evaluation time, for an expression to be valid, the types of the operands must be compatible with each other and with the operator being used. Likewise, if the type of an expression cannot be inferred, the expression cannot be valid.

In Lattakia, there are atomic and composite types. Atomic types include numbers, string, booleans or regular expressions, whereas sequence lattices and alternative lattices are composite types. Fundamentally, composite types may be composed of other atomic or composite type expressions, whereas atomic types are composed only of their own expression types.

### 4.1 Lattices

Lattices are used to describe structured data and code. From the "code" perspective, a sequence lattice is a sequence of statements while an alternative lattice is a branching statement. From the "data" perspective, a sequence lattice is analogous to an array/list structure in common programming languages while an alternative lattice is analogous to a value of a variable in those languages and this allows variables in Lattakia to hold multiple values at the same time. As such, those variables have special treatment in Lattakia.

#### 4.1.1 Sequence Lattices

Sequence lattices (seqlats) represent ordered sequences of data and code. Syntactically, a seqlat is the concatenation of one or more alternative lattices (altlats). Alternative lattices may be labeled. Sequence lattices are processed in order - one element at a time.

The primary operations supported by the seqlat are construction, access and application. Seqlats are constructed by stringing together some number of altlats separated from each other by semicolons (;). Each element in a seqlat may be given a label – a name to access it by. An altlat is assigned to a label using the colon (:) operator. Labels correspond to names on the nodes while altlats correspond to code on the arcs.

Example: (the code below is a seqlat itself)

```
lat0 = (); .. an empty lattice (of a single epsilon transition)
lat1 = (1; 3; 2); .. a lattice of three elements
print(lat1[0]); .. prints the first element in lat1
```

### 4.1.2 Alternative Lattices

Alternative lattices (altlats) are lists of alternatives. Each altlat is a sequence of one or more atoms each is preceded by an optional condition. Alternative lattices are processed sequentially one order at a time. Unlike sequence lattices though, elements are only processed if their condition is satisfied.

Like seqlats, altlats support construction operations. They are constructed by stringing together some number of atoms, separated by the bar (|) symbol. Each element may have a condition which is a preceding expression of boolean type, placed between square brackets ("[" and "]"). In evaluation, the condition of each atom is evaluated and if true, that atom will be evaluated.

### 4.1.3 Named variables:

Named variables are defined when assigned a value by either the colon (":") or the equal ("=") operators and their type is determined based on the right-value expression. Variables defined using ":" are labels and are accessible outside the lattice as opposed to local variables that are defined using "=". The value assigned to a label is an altlat while the local variable is assigned and expression except for constrained local variables (explained below) that are assigned altlats. A lattice surrounded by parentheses ("(" and ")") is an expression in Lattakia; thus, to assign a seqlat to a variable or an altlat to an unconstrained local variable, they should be enclosed by parentheses.

Example:

```
a = 5;              .. an integer variable with the value 5
x = (3 | 7 | 2); .. an integer variable that can be 3, 7 or 2
isImportant = ([x >= 75] false | [x < 75] true);
isEmptyStack = ([stack==epsilon] true | false);
```

In this example, isImportant and isEmptyStack are variables while they are functions in other programming languages. 'x' is called an "independent variable" while 'isImportant' is called a "dependant variable" for it depends on 'x'.

#### a. Independent V.S. Dependent Variables:

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

1. Independent Variables (analogous to variables in common programming languages): 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 (analogous to functions in common programming languages): 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. The type of ‘w’ is dependent on the type of ‘x’.

**b. Constrained V.S. Unconstrained Variables:**

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

1. Constrained Variables (analogous to functions with parameters in common programming languages): variables that are subject to constraints (parameters) and the lattice they hold may change when provided different constraints. Constrained variables are defined by specifying an identifier followed by a caret (“^”) and a constraints list. A constraint list is a semicolon-separated list of identifiers, each is optionally proceeded with a “?” sign (analogous to passing parameters by value) and optionally followed by a “=” and an altlat (analogous to default value parameters).
```
        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 (analogous to variables or functions with no parameters in common programming languages): variables that are not subject to constraints:
```
        u = 1;         .. An independent unconstrained variable.
        w = x + 1;     .. A dependent unconstrained variable.
```

To access 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 constraints:
```
        z^(x) = x * y;
        y = 5;
        w = z(3);      .. now ‘w’ is 3+5 = 8
```

### 4.1.4 Labels

A seqlat may have only one type of labels: either variable labels or hash-key labels:

- A variable label can be a constrained and an unconstrained variable.
- A hash-key label starts with the hash (“#”) operator followed by a string, a number, an ID, or an altlat surrounded by parentheses. The ID and the altlat must evaluate to a string or a number and their value is used as the hash-key label.

Labeled elements can be accessed through the use of the dot (.) operator.

Example:

```
a = (x: 1; increaseXBy^(a): x += a; 3);
a.x = 3;
a.increaseXBy(2);
```

### 4.1.4 Evaluation

The following pseudo code explains the evaluation of a seqlat (see Appendix B for the CFG of the language).

```
// Evaluate seqlat
For each element in the seqlat
    // Evaluate label
    Label (if exists) is evaluated if needed (when proceeded by #)
        and put it in a the Symbol Table
    // Evaluate altlat
    For each alternative in the altlat
        If (condition satisfied)
            // Evaluate atom
            if (atom is “expression”)
                evaluate expression
            else if (atom is “RETURN expression”)
                Stop execution
                Evaluate expression
                Return evaluated expression
            else if (atom is “LET expression”)
                Evaluate expression // for its side-effect
                Evaluate to epsilon
```

When encountering a label or a local variable definition, this variable is put in the Symbol Table along with it’s type.

The following code explains the evaluation of an expression:

```
Constants are Evaluated to themselves

For binary and unary operations:
    Evaluate expression operands recursively
    Evaluate the operation on the result

For assignment operators:
    Value = Evaluate the r-value expression
    Optimize Value unless otherwise specified
    Access the l-value
    Put Value in l-value

For lvalue expression
    Access lvalue
    Evaluate lvalue’s lattice

For “? ( lattice )”
    Evaluate seqlat
```

```
For "^ ( lattice )"
    Value = Evaluate seqlat
    Don't optimize Value
```

Accessing lvalue expressions is done by the following operators: () [] {} . @ #
All evaluations needed to get to lvalue's lattice are performed.

### 4.2 Numbers

Numeric types follow the same model as numeric types in other languages. Integer and real number types follow the rules for integer and real number arithmetic and support the same set of standard arithmetic operators. If a numeric operator is given two integer operands, the type of the result will be integer. If one or more operands are a real number, the result will be of type real.

#### 4.2.1 Integers

Integers in Lattakia are used for whole number operations. In the few cases where integer operands and an integer operator result in a real (non-integer) result, the result of the expression will be automatically truncated to just the whole number portion.

#### 4.2.2 Real Numbers

Real numbers in Lattakia follow the rules of the float type in other languages.

### 4.3 Strings

Strings in Lattakia are an atomic type. Their function is to represent textual data. They are immutable. As such, they cannot be modified. Instead, in cases where a modified version of a string is needed, a new one must be constructed. In addition to construction, operations applying to strings include searching, replacing and concatenation.

### 4.4 Booleans

Booleans are different from the other basic types in that boolean operators accept most types. Booleans are used to represent conditions which may be either true or false. While equality and non-equality operators apply to any type of operand, comparison operators (less than, greater than, etc.) apply to all atomic types.

### 4.5 Regular Expression

Regular expressions are data types. Their representation in Lattakia is analogous to their representation in Java. Their operations are only “=” and “==”;

## 5. Expressions and Operators

The expression is the basic building block of the language.  To compose complex expressions, Lattakia includes a large range of operators, both for atomic types and for lattices.

For operators dealing with atomic type expressions, operands are simply the results of evaluating the respective expressions.  Operands in lattice expressions are not evaluated save where explicitly noted.

Lattakia relies on the notion of delayed evaluation, so that actual evaluation of expressions (according to the rules below) does not occur until explicitly required.  However, while the expressions themselves are not evaluated, they are checked for valid operand types and any errors reported.

### 5.1 Arithmetic expressions

Arithmetic operators supported by Lattakia include addition, subtraction, multiplication, division, remainder, negation and positive.  These operators require numeric operands.  Behavior is not defined in case of an overflow or underflow.  Operands must be of integer or real number type.  If both operands are integers, the result is of integer type.  Otherwise, the result is of real number type.

Standard arithmetic operators in Lattakia are right associative.  Precedence is determined using standard arithmetic rules.

**5.1.1 Addition**: *expression + expression*

The result is the sum of the two operands.

**5.1.2 Subtraction**: *expression - expression*

The result is the difference between the first and second operands.

**5.1.3 Multiplication**: *expression * expression*

The result is the product of the two operands.

**5.1.4 Division**: *expression / expression*

The result is the quotient of the two operands.  The second operand must be non-zero.

**5.1.5 Remainder**: *expression % expression*

The result is the remainder of the two operands.  The second operand must be positive.

### Unary Operators

### 5.1.6 Negation: - *expression*

The result is the negation of the operand.

```
3 * 4.5;
2 - 3;
4 / 3;
3 % 2;
- (-5);

.. The lines evaluate to 13.5; -1; 1; 1 and 5 respectively
```

## 5.2 String expressions

String operators in Lattakia include concatenating and matching.  All operands must be string expressions.  The result of these operations is of type string.

### 5.2.1 Concatenation: *expression + expression*

The result is a new string consisting of the first operand concatenated with the second operand. Note that this usage of the '+' operator has nothing semantically to do with addition.

### 5.2.2 Matching: */expression/*

The result is a new string consisting of the portion of the first operand that matches the second operand.  Regular expression matching is performed.

```
"hello" + " there";
/gre*n/;

.. The lines evaluate to "hello there" and "green" respectively.
```

## 5.3 Boolean expressions

Boolean expressions in Lattakia include equality, non-equality, logical negation, logical or and logical and.  Equality and non-equality require two operands of the same type of expression. Logical negation requires one operand of boolean type.  Logical or and logical and require two operands of boolean type. The result is of boolean type.

### 5.3.1 Equality: *expression == expression*

If the operands are atomic, the result is true if the two expressions evaluate to the same value and false otherwise. If the operands are lattices, the result is true if the corresponding elements in each lattice are equal, and false otherwise.

### 5.3.2 Non-equality: *expression != expression*

The result is evaluated to be the result of *!(expression == expression)*. (See 5.3.3 for *!(expression)*)

### 5.3.3 Logical Negation: *!expression*

The result is true if the operand is false, and false otherwise.

### 5.3.4 Logical Or: *expression || expression*

The result is true if either operand is true, and false otherwise.

### 5.3.5 Logical And: *expression && expression*

The result is true if both operands are true, and false otherwise.

```
2 == 3;
(2; 3; (4; 5)) != (1+1; (4; 6-1));
true && false;
!false;

.. The lines evaluate to false, false, false and true respectively.
```

## 5.4 Comparison expressions

Comparison operators in Lattakia include less than, less than or equal, greater than and greater than or equal. Comparison expressions require their operands to be of the same atomic type, save for numeric types where mixing and matching of integer expressions and real number expressions are allowed. The result is of boolean type.

### 5.4.1 Less than: *expression < expression*

If the operands are string expressions, a lexicographical comparison is made, and the result is true if the first operand is lexicographically before the second operand. Otherwise the result is false.

If the operands are boolean expressions, the result is true if the first operand is false and the second one is true, and false otherwise.

If the operands are numeric expressions, the result is true if the numeric comparison is true, and false otherwise.

### 5.4.2 Less than or equal: *expression <= expression*

The result is equal to the result of *(expression < expression) || (expression == expression)*.

### 5.4.3 Greater than: *expression > expression*

The result is equal to the result of *!(expression <= expression)*.

### 5.4.4 Greater than or equal: *expression >= expression*

The result is equal to the result of *!(expression < expression)*.

```
2 < 3
"abc" <= "abb"
true < false

.. The lines evaluate to true, true and false respectively.
```

## 5.5 Assignment expressions

Simple assignment for expressions in Lattakia means binding a name to an expression. In particular, no evaluation is carried out. Combined assignment works much the same way. Evaluation is delayed except for constant expressions whose values cannot change (i.e. expressions that don't contain variables). There is an additional operator to force evaluation prior to assignment. The name bound to the expression receives the same name as the expression. Assignment expressions are left associative.

### 5.5.1 Simple assignment: *lvalue = expression*

The result of this expression is the second operand. In addition, the first operand is bound to the second operand. Subsequent references to the first operand will retrieve the second operand. *lvalue* has several options, the most common of which is a name (ID), although it can also be an accessor for a lattice or several other more complicated expressions.

### 5.5.2 Combined assignment: *lvalue op= expression*

The result of this expression is the same as writing:
*lvalue = lvalue op expression*.

Valid options for *op* are: +, -, *, /, |, ~ and ?

### 5.5.3 Prefix inc/decrement: *++lvalue* and *--lvalue*

These expressions are equivalent to writing:
*lvalue += 1* and *lvalue -= 1* respectively.

### 5.5.4 Postfix inc/decrement: *lvalue++* and *lvalue--*

The result is the current expression in *lvalue*. The expression *lvalue* is then either incremented or decremented.

```
x = 3;
x += 2;
// x = 5
y = 2 | 1;
y |= 3;
// y = (2 |  1 | 3);

.. The combined assignment operators expand to x = x + 2 and y = y | 3.
```

## 5.6 Evaluation Expressions

There are two types of evaluation expressions. In both cases, the result is not simply the expression itself; it is the result of recursively evaluating the expression according to the rules of the operands.

### 5.6.1 Evaluation: *? expression*

The result is simply the result of evaluating the operand.

```
x = 3;
// x is set 3 because 3 is a constant expression
y = x;
// y is bound to x
x++;
// note that y will have the same value as x when evaluated
y = ?x;
x++;
// x is 5 here, but y = 4;

.. The ?  in the second case causes y to be bound to the result of evaluating the expression x
is
.. bound to.
```

### 5.6.2 Preventing Optimization: *^ expression*

Before assignment, the r-value lattice is optimized to the minimal lattice. Optimization means removing all epsilon transition from the seqlat, removing all nil transitions from the altlat, merging inner seqlats into the outer seqlat, and merging inner altlats into outer altlats. Using the caret operator prevents optimization from taking place.

```
x =  ((1; 2; 3|(4|5)); (1;3)); .. x is (1; 2; 3|4|5; 1; 3)
y = ^((1; 2; 3|(4|5)); (1;3)); .. y is ^((1; 2; 3|(4|5)); (1;3))
```

### 5.6.3 Application: *name (lattice)*

We have three overloaded meanings of application that is identified via type checking.

1. If "name" is a constrained variable (function) (this means "lattice" is a list of actuals), replace these actuals in the constraints (parameters) of the function in the code lattice associated with this function and execute the lattice.
2. If "name" is a data lattice and "lattice" is a rule lattice (it has an "@" operator in it), then loop through this data lattice and apply the rules on it. Rules application is described in section 9.
3. Else (If "name" is a data lattice and "lattice" is not a rule lattice), (this is analogous to constructors in OOP), create a copy of "name", and then, for every variable in "lattice", if "name" has the same variable, assign "lattice" variable value to "name" variable.

```
Ex. Function call:
sum((1; 2; 3; 4); 1);

.. This applies a function named sum to a lattice consisting of 2 elements.  The first element
is
.. the lattice (1; 2; 3; 4).  The second element is the integer value 1.
```

## 5.7 Alternative expression:  *expression | expression*

The result is a new alternative lattice with two alternatives, one for each operand.  Condition expressions can be used to select among the alternatives.

```
options = 1 | 2 | 3 | 4 | 5;
moreoptions = options | 6;

.. This creates two altlats, one with 5 possible values (alternatives), the other with 6.
```

## 5.8 Sequence expression:  *expression ; expression*

The result is a new sequence lattice where the second operand is tagged onto the end.

```
sequences = (1; 2; 3);
longer = sequence; 4;
```

```
.. This creates two seqlats, one with contents 1; 2; 3, the other with contents 1; 2; 3; 4.
```

### 5.9 Concatenation expression: *expression ~ expression*

The result is a new sequence lattice created by concatenating the two operands.

```
stmt1 = (1; 2; 3; 4);
stmt2 = (5; 6; 7; 8);
stmt = ^(stmt1 ~ stmt2);

.. This creates a new seqlat with contents (1; 2; 3; 4); (5; 6; 7; 8);
.. This is a sequence of 2 seqlats, each with 4 integers for arcs
```

### 5.10 Labeling expressions: *label: value*

Labels are quite similar to labels in other programming languages; names assigned to be used later. They represent handles or names of the nodes. Using a label you can access the variable that follows this label. Labels provide a neat way of accessing values inside a variable/lattice. Accessing a label can happen either:

a. from inside the lattice where it is defined; so you just use the label's name
b. from outside the lattice where the label is defined and in this case you need to use the dot operator (defined later in this section)
c.

```
ex.
    student = (name: "tiffany";
                         age: 20);
    x = student.age;
    .. returns 20
```

### 5.11 Conditional expressions: *[condition]*

These are conditions on certain paths in the lattice. Conditions are enclosed in brackets and can contain comparison and logical operators.

```
ex.1
max(a; b) = [a>b] action1| [a<b] action2| action3
.. This indicates that if the first condition is satisfied, action 1 is performed, and if the
.. second condition is satisfied action 2 is performed and in all cases action 3 is also ..
.. performed.

ex. 2
max(a; b) = [a>b] action1| [a<b] action2| else action3
```

In this example because of the use of the keyword else, action3 is only performed if both first and second conditions are false.

## 5.12 Each expressions: *each x*

The default behavior when applying any operation on an *altlat* is to apply the operation to the first path that has a satisfied condition. **'each'** overrides this behavior. Using each we instruct the program to apply the operation of all paths in the *altlat*

```
ex. 1.
x = (5 | 3); y = (8 | 2 | 1); condition [x < each y];
.. Result:   false since the first value of x (5) is compared to all value in y i.e. 8,2, and 1
.. so the operation yields false.
.. However the result of [x< y] is true because the first value of the altlat x which is 5 .. is less
than the first value of the altlat y which is 8.
```

```
ex 2.
z = each(x) + y
..result is = (x{0}+y{0} | x{1}+y{0});
```

## 5.13 Parenthetic expressions: *(a; b; c)*

Parentheses are used to surround elements of a lattice. These elements can be constants, expressions, conditions or code-statements. In the latter case a semicolon terminates each statement. A semicolon after the last constant or statement is optional.

```
ex 1. const = (1;2|3;4);
ex 2. code = ( a++;b=a;)
ex 3. expr = [b==4 | ( b!= 4; b%2==0)]
..This expression corresponds to "&&" in C++ (i.e. b != 4 && b%2==0)
```

## 5.14 Braces expressions: *x{index}*

Braces are used to access a particular alternative of an altlat using its index.

```
ex.
student = ( "alice"|"tiffany")
student{0} ..returns alice
```

## 5.15 Brackets expressions: *x[index]*

Brackets are used to access a particular element of a seqlat using its index.

```
ex.
courses = ( PLT; MT)
courses[0] ..returns PLT
```

## 5.16 Accessor expressions: '.'

The dot operator is used to access labels and properties inside a certain lattice.

```
ex.
student = (name: “alice”; age: 23; major: “CS”; “MATH”);

student. name  ..returns alice
student.major ..returns CS;MATH
student.major{0} ..returns CS
student.major.count ..returns the count of alternatives in the major field which is 2
..(Count is a built-in property of alt-lat)
```

## 5.17 Hash expressions: #

The hash is used in:

a. Function-declaration.

```
ex.     getMax^(a,b);  ..defines a function getMax that takes 2 arguments.
        getMax(a,b); ..calls the previously defined function getMax and passes
        the variables a and b to it.
```

b. Accessing elements of a previously created hashtable.

```
ex.     myHash = (key1 = value1;
        key2 = value2|value3;
        key3 = value4);
        ..Then using myHash#key2 returns the value of key2 which is
        “value2|value3”
```

## 6. Declarations

The only declarator that we use is the function delarator. A function declarator is exactly the same as the call to the function except for an added caret before the opening parenthesis that precedes the first argument.

```
ex.     func^(a; b; c ) .. an example of a function declaration
```

There are no scope specifiers and no type specifiers. Each variable/lattice infers its type from the value it gets bound with.

## 7. Attributes and Keywords

Reserved keywords and attributes are meant to facilitate the use of the language. Some of these keywords such as true, false, this, return, else are borrowed from other programming languages and are used in quite a similar fashion while others are meant to handle lattices and are not part of languages that do not deal with lattices. Some keywords such as count and length also function as attributes to sequence and alternative lattices (seqlat and altlat)

The list of reserved keywords (and attributes) and their use:

### this

Refers to the current lattice/variable

### let

Instead of returning the value of the current statement/expression (which is the default behavior), let returns epsilon.

```
let x = y = 3;
.. This assigns the value 3 to the variable y, assigns the value of y to the value of x and
instead of returning 3 returns epsilon
```

### return

Terminates the execution of statements in the current code-lattice and returns the value of the current statement.

```
ex.   maxRead =
          (
          read(a; b); // reads a and b; then
          // evaluates to epsilon.
          let answer =
          [a>b] a | [a<b] b
          otherwise NIL;
          return answer;
          answer= -1 .. unreachable code
          );
```

### true

Similar to other programming languages, true indicates that a particular condition or value of variable is true.

### false

Similar to other programming languages, true indicates that a particular condition or value of variable is true.

### NIL

'NIL' is a broken arc that you cannot pass through.
In a seqlat you can't drop NIL and it means that this seqlat is broken and cannot be evaluated. In this case NIL is returned (similar to throwing exceptions).
In an altlat, a path that contains NIL can be dropped because there are other alternative paths.
However when evaluating an altlat, if NIL is the only accepted path, then it is part of the seqlat that holds this altlat.

### epsilon

'epsilon' is an empty transition or a transition whose condition is always true.
In a seqlat, you can drop the epsilon and merge its endpoints.
In an altlat epsilon is not dropped because it corresponds to an alternative whose condition is always true.

### labels

Retrieves all the labels in a particular lattice.

```
ex. student = (name: "tiffany";
                    age: 20);
x = student.labels;
.. returns name; age
```

## length

Returns the number of elements in a seqlat.

```
grades = 100;85;50;35
result = grades.length;   ..returns 4
```

## count

Returns the number of alternative in an altlat.

```
grade = A|B|C
result = grade.count;   ..returns 3
```

## clone

Returns a deep copy of a specific lattice.

```
major1 = “CS” | “MATH”;
lat = (name:”tiffany”; age: ”20”;major:major1)
clonedLat = lat.clone()

..clonedLat = (name:”tiffany”; age: ”20”; major: “CS” | “MATH”)
```

## 8. Built-in Library Functions

The standard library for Lattakia consists of 3 built-in functions, used for input, output and iteration.

### 8.1 Read: *read(arg1; arg2; ...)*

The read function is used for constructing lattices from user input. It receives an arbitrary number of arguments. Each argument is a name or lvalue, to which the corresponding input value will be bound. The reading is done from standard input.

```
read(x;y;z);
// user then types 1; (2 | 3; 4); "hi"

.. x = 1, y = (2 | 3; 4), and z = "hi"
```

### 8.2 Print: *print(expression)*

The print function is used to display output. It receives a single argument consisting of an expression, which it evaluates and displays to the standard output. Print returns epsilon.

```
x = true;
y = false;
print(((x || y); (x && y); !x));

.. displays 'true; false; false'
```

### 8.3 Loops:

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++
```

For example, the foreach function is used to iterate over the elements in a sequence lattice. Each item in the second argument is bound to the name used as the first argument, and the expression that is the third argument is applied to it. Foreach returns epsilon.

```
let x = (1; 2; 3; 4);
foreach(y; x; print(y));

.. displays '1 2 3 4'
```

## 9. Rule Application

Syntax:

```
      dataLattice(ruleLattice);
```

Rule application is a process that loops through a dataLattice (seqlat) and, in each loop, applies the ruleLattice. A ruleLattice is a lattice that contains the "@" operator in it. The "@" operator takes an integer the represent the offset from the current counter of the loop, and returns dataLattice[counter - offset]. For example, @0 is the current element of the dataLattice.
Example: accumulator:

```
      lat1 = (1; 3; 2);
      lat1(@0 += @-1); .. returns (1; 4; 6)
```

### 9.1 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);

      lat1 = lat1(r1); .. Code optimization: lat1 is changed and returned
      lat1(r1); .. lat1 is not changed. The result lattice is created and
      returned.
```

To keep epsilons in the resulted lattice:

```
      lat2 = ^lat1(r1)
```

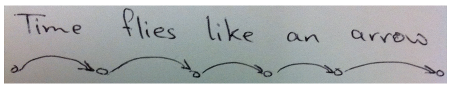
and that's because the optimization happens after the assignment. Now, this lattice is reserved.

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 ||).

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



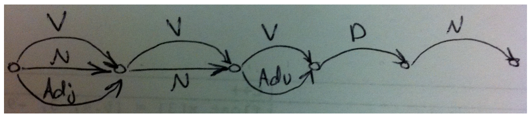
Look at lattice l1 in the figure above. To create a Part-of-Speech (POS) Tagger for English, we need the following:
1. Lexical Rules: E.g.,

```
            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."

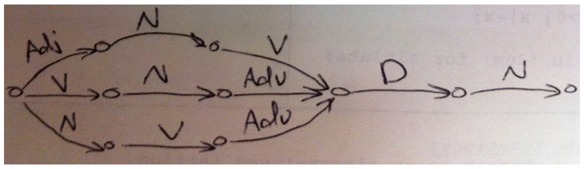
```
      sr101 = [@0==V; @1==N; @2==Adv; @3==D; @4==N] (@0; @1; @2; @3; @4);
```

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.

## 10. Examples

**Example 1** - determine and print out the greatest-common-divisor of two numbers using Euclid's algorithm.

```
gcd^(x; y) =
(
 [y == 0] x |
 [y != 0] gcd(y, x % y);
);

read(a; b);
print("gcd is ");
print(gcd(a; b));

.. displays the gcd of the two numbers entered by the user
```

**Example 2** - use Quicksort to sort a list of user-entered strings alphabetically.

```
quicksort^(input) =
(
   let (less = (); greater = ());
   [input.length <= 1] return input | epsilon;

   let pivot = lat[0];
   let lat[0] = epsilon;

   foreach(x; input; [x < pivot] less = (less; x) | greater = (greater; x));

   (quicksort(less); pivot; quicksort(greater));
);

read(values);
let sorted = quicksort(values);
foreach(x; sorted; print(x));

..  displays the user input, sorted from least to greatest
```

### 10.1 Object-Oriented Programming

**Example 1 –** Inheritance

```
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.
```

**Example 2** – Generalization

```
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
```

### 10.2 Collections (Advanced Data Structures)

Arrays, Lists, vectors, tuples, sets and matrices are supported in Lattakia. The structure of the word-lattice supports them without any added complexity. In this section we give examples on how to use these data structures.

#### 10.2.1 Arrays:

```
ex.
        array = (10; 3; 2; 8; -1);
        array[i]; .. the element of order (i+1). Index start from 0
        array.length; .. the number of elements in ‘array’.
        a1 = array; .. Creates a copy of ‘array’ and assign it to ‘a1
        a2= &array
        ..Both ‘a2’ and ‘array’ point to the same data
```

*Adding and removing elements from an array are discussed in the following section: Stacks and Queues.*

#### 10.2.2 Stacks and Queues:

Stacks and queues are defined in the language as lattices.

##### 10.2.2.1 Stack:

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

##### 10.2.2.2 Queue

```
        enqueue(lat; x) = lat ~= x;
        dequeue(lat) = pop(lat);
```

### 10.2. 3 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.
*The use of '|' operator, which adds an alternative to a given value, solves the collision problem.*

```
        hash = (
                x: "a";
                y: "b";
        );
```

### 10.2.4 Sets:

Lattakia uses labels as set elements because they are unique inside a lattice.

```
        color = (red:; blue:; yellow:;);
```

To check if an element exists in this set:

```
        [defined
                (color.red)
        ] (doSomething);
        ..defined(x) = [x==NIL] false | true;
```

### 10.2.5 Tuples:

Tuples are seqlats. The elements of the tuples are the variales inside the corresponding.

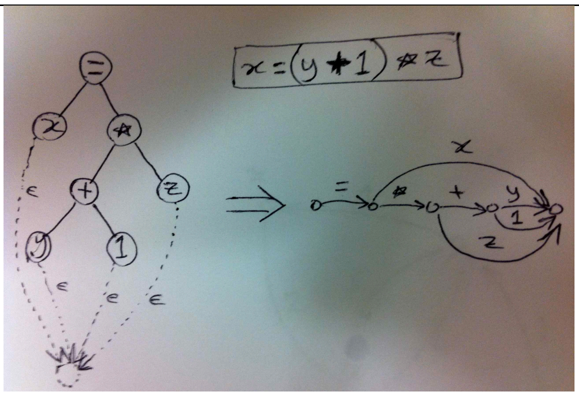
```
        engine = (8; 'V'; 300);
                car = (
                model: ('Coupe'; 'Bentley');
                color: red;
                engine: (8, 'V', 300)
        );
        .. Labels are not necessary
```

### 10.2.6 Trees:

Similar to previous data structures, a tree is just another lattice.
Adding an artificial "last node" and an epsilon transition from each leaf in the tree to this last node is all what you need to do to create a tree.

The following notations are proposed for trees:
An atom is a tree. (atom; lat1 | lat2 | … | latN) is a tree, where 'atom' is the parent and lat1 to latN are children of 'atom' (in the exact order).

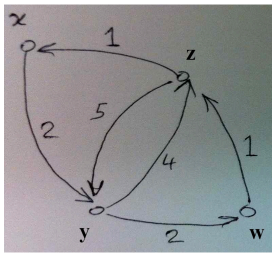
Note that the original tree's nodes become lattice arcs Each sub-tree can be given an identifier name using labels.

| | |
|---|---|
|  x=(y+1)*z<br>ε<br>ε<br>ε<br>ε<br>ε<br>=<br>*<br>+<br>y<br>1<br>x<br>z | Statement: x = (y + 1) * z;<br><br>tree =<br>('=';<br>'x' \|<br>('*';<br>('+';<br>'y' \|<br>'1'<br>) \|<br>'z'<br>)<br>); |

### 10.2.7 Graphs:

To define a graph in Lattakia, one can do the following:

1. Add a lattice node for each graph node (use graph node name as a label);
2. foreach node 'x' in the graph,
   foreach arc going from node 'x' to a node 'y' and labeled with label 'w',
   add lattice arc: (w; y) to 'x' alternatives.



```
Graph to the left
represented as:
(
  x: (2; y);
  y: (4; z) | (2; w);
  z: (1; x) | (5; y);
  w: (1; z)
);

path(a; b) = [a==b] 0 |
foreach (a; (
        let (w; next) = a;
                w + path(next; b)
));
```

```
minPath(a; b) ) = [a==b] 0 |
foreach (a; (
            let (w; next) = a;
                        w + min(path(next; b))
));
```

## Appendix A – Syntax Summary

1. Expressions.

*expr:*

```
lvalue
{ altlat }
constant
expr infix_op expr
lvalue ++
lvalue --
++ lvalue
-- lvalue
- expr
! expr
lvalue assign_op expr
? ( lattice )
^ ( lattice )
```

*constant:*

```
string_literal
number
“epsilon”
“nil”
“true”
“false”
regex_match
java_code
```

*number:*

```
INTEGER
FLOAT
```

*lvalue:*

```
name
( lattice )
? name
^ name
```

*name:*

```
rest_of_name
"this"
"this" . rest_of_name
at_operation
at_operation . rest_of_name
```

*at_operation:*

@ integer
@ ID
@ ( altlat )

*rest_of_name:*

attribute
label
rest_of_name . rest_of_name
rest_of_name ( altlat )
rest_of_name { lattice }
rest_of_name [ lattice ]

*label:*

ID
# String_literal
# number
# ID
# ( altlat )

*attribute:*

```
"length"
"count"
"labels"
"clone"
```

*infix_op:*

```
*   /   %
+  -
>  <  >=  <=
==   !=
&&  ||
~
```

Assignment operators all have the same priority, and all group right-to-left while all other operators group left-to-right.

*assign_op:*

```
=   +=   -=    *=   /=    ?=    ~=    |=
```

2. Statements

*lattice:*

/*epsilon*/
labeled_altlat
labeled_altlat ; lattice

*labeled_altlat:*

altlat

name : altlat
name ^ ( param_list ) : altlat
name ^ ( param_list ) = altlat

*param_list:*
by_value_opt ID default_value_opt
param_list ; ID default_value_opt

*by_value_opt:*
/*epsilon*/
?

*default_value_opt:*
/*epsilon*/
= altlat

*altlat:*
atom
{expr} atom
altlat|altlat

*atom:*
expr
"`return`" expr
"`let`" expr

**Operator Precedence from highest to lowest:**

Name accessing operators:
```
# @
.
() [] {}
```
Unary Operators
`! –` *(unary minus)* `++ -- ? ^`

Arithmetic Operators:
```
*  /  %
```
`+  -` *(binary minus)*

`Comparison Operators:`
```
>  <  >=  <=
==  !=
```
Logical Operators:
```
&&  ||
```
Lattice Operators:
```
~
```
Assignment Operators
```
=  +=  -=   *=  /=   ?=   ~=   |=
```

## Appendix B – The Scanner

```
{ open Latparser }

rule token = parse
  [' ' '\t' '\r' '\n']  { token lexbuf }  (* Whitespace *)
| ".." { comment lexbuf }
| "**" { multi_line_comment lexbuf }

| '('  { LPAREN }          | ')'  { RPAREN }
| '{'  { LBRACE }          | '}'  { RBRACE }
| '['  { LSQBRACKET }      | ']'  { RSQBRACKET }

| '+'  { PLUS }            | '-'  { MINUS }
| '*'  { TIMES }           | '/'  { DIVIDE }
| '%'  { PERCENT }

| '|'  { PIPE }            | '~'  { TELDA }
| '!'  { EXCLAMATION }     | '?'  { QUEST }
| ';'  { SEMI }            | ':' { COLON }           | '^' { CARET }
| '@'  { AT }              | '.'  {DOT}              | '#' { HASH }

| '='  { ASSIGN }          | "+=" { PLUS_ASSIGN }  | "-=" { MINUS_ASSIGN }
| "*=" { TIMES_ASSIGN }    | "/=" { DIVIDE_ASSIGN }
| "?=" { QUEST_ASSIGN}     | "~=" { TELDA_ASSIGN } | "|=" { PIPE_ASSIGN }
| "++" { PLUS_PLUS }       | "--" { MINUS_MINUS }

| "==" { EQ }              | "!=" { NEQ }
| "&&" { AND }             | "||" { OR }
| '<'  { LT }              | '>'  { GT }
| "<=" { LEQ }             | ">=" { GEQ }

| "else"    { ELSE }
| "return"  { RETURN }     | "let" { LET }          | "this" { THIS }
| "true"    { TRUE }       | "false"   { FALSE }
| "nil"     { NIL }        | "epsilon" { EPSILON }
| "length"  { LENGTH }     | "count"   { COUNT }
| "clone"   { CLONE }      | "labels"  { LABELS }
| eof       { EOF }

| '\"'([^'\"'])*'\"' as lxm { STRING_LITERAL(lxm) }
| '`'([^'`'])+'`' as lxm { JAVA_CODE(lxm) }
| '/'([^'/'])+'/' as lxm { REGEX_MATCH(lxm) }

| ('-')?['0'-'9']+'.'['0'-'9']+ as lxm { FLOAT(float_of_string lxm) }
| ('-')?['0'-'9']+ as lxm { INTEGER(int_of_string lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

and comment = parse
 "\n" { token lexbuf }
| _   { comment lexbuf }

and multi_line_comment = parse
 "**" { token lexbuf }
| _   { comment lexbuf }
```

## Appendix C – The Parser

These CFGs are unambiguous, they have no shift/reduce nor reduce/reduce conflicts. Only one **%prec** is used to solve the unary minus precedence issue: OCaml

```
%{ open Ast %}

%token LPAREN RPAREN LBRACE RBRACE LSQBRACKET RSQBRACKET EOF
%token PLUS MINUS TIMES DIVIDE PERCENT PLUS_PLUS MINUS_MINUS
%token PIPE TELDA QUEST SEMI COLON DOT AT HASH CARET
%token ASSIGN PLUS_ASSIGN MINUS_ASSIGN TIMES_ASSIGN DIVIDE_ASSIGN
%token TELDA_ASSIGN PIPE_ASSIGN QUEST_ASSIGN
%token AND OR EQ NEQ EXCLAMATION LT GT LEQ GEQ
%token EPSILON NIL TRUE FALSE
%token RETURN LET THIS LENGTH COUNT LABELS CLONE ELSE

%token <string> STRING_LITERAL
%token <string> JAVA_CODE
%token <string> REGEX_MATCH
%token <string> ID
%token <float> FLOAT
%token <int> INTEGER

/* Assignment Operators: */
%right ASSIGN QUEST_ASSIGN PLUS_ASSIGN MINUS_ASSIGN TIMES_ASSIGN DIVIDE_ASSIGN
PIPE_ASSIGN TELDA_ASSIGN
/* Lattice Operators: */
%left SEMI
%left PIPE
%left TILDA
/* Logical Operators: */
%left OR
%left AND
/* Comparison Operators: */
%left EQ NEQ
%left LT LEQ GT GEQ
/* Arithmetic Operators: */
%left PLUS MINUS
%left TIMES DIVIDE PERCENT
/* Unary Operators: */
%left UNI_MINUS EXCLAMATION PLUS_PLUS MINUS_MINUS QUEST CARET
/* Name Operators: */
%left LBRACE LSQBRACKET LPAREN
%left DOT
%left AT
%left HASH

%start lattice
%type <Ast.lattice> lattice

%%
```

```
/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/
/*=-=-=-=         Defining Sequential Lattices          =-=-=-=*/
/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/
lattice:
        /**/
      | labeled_altlat
      | labeled_altlat SEMI lattice


/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/
/*=-=-=-=         Defining Alternative Lattices         =-=-=-=*/
/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/
labeled_altlat: /* 'label: altlat' or ‘f^(x) = x++’ */
        altlat                /* '[condition] atom' */
      | name COLON altlat     /* 'label: [condition] atom' */
/* Constrained Variable Declaration: 'f^(x; y) = [condition] atom' */
      | name CARET LPAREN param_list RPAREN COLON  altlat /*label*/
      | name CARET LPAREN param_list RPAREN ASSIGN altlat /*local CV*/

param_list: /* 'x=4; y=(3; 5); ?z' */
        by_value_opt ID default_value_opt
      | param_list SEMI by_value_opt ID default_value_opt
by_value_opt:
        /**/
      | QUEST

default_value_opt:
        /**/
      | EQ altlat

altlat:
        atom
      | LSQBRACKET expr RSQBRACKET atom
      | altlat PIPE altlat

label:
/* 'x' or any string or #5 or #-3.5 or #x or #([x > 0] 7 | y.#3.2) */
        ID
      | HASH STRING_LITERAL
      | HASH number
      | HASH ID
      | HASH LPAREN altlat RPAREN

atom: /* 'expr' or 'return expr' or 'f(x; y) = [condition] atom' */
        expr
      | RETURN expr
      | LET expr
```

```
/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/
/*=-=-=-=             Defining Expressions              =-=-=-=*/
/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/
expr:
/* Left-Value Variable: */
        lvalue
/* Constants: 5, "abc", -4.3 */
      | STRING_LITERAL
      | number
      | EPSILON
      | NIL
      | TRUE
      | FALSE
      | REGEX_MATCH     /* /abc|d+/ */
      | JAVA_CODE       /* `System.out.println("Hi"+$myVar);` */
/* Altlat Spread: For each alternative in this 'expr' altlat ('expr'
   must evaluate to an alt lat): */
      | LBRACE altlat RBRACE
/* Right-Value Arithmatic INFIX operators: */
      | expr PLUS expr
      | expr MINUS expr
      | expr TIMES expr
      | expr DIVIDE expr
      | expr PERCENT expr
/* Right-Value Logical INFIX Operators */
      | expr OR expr
      | expr AND expr
/* Right-Value Comparison INFIX Operators */
      | expr EQ expr
      | expr NEQ expr
      | expr LT expr
      | expr LEQ expr
      | expr GT expr
      | expr GEQ expr
/* Right-Value Lattice INFIX Operators */
       | expr TILDA expr
/* Right-Value Arithmatic POSTFIX Operators */
      | expr PLUS_PLUS
      | expr MINUS_MINUS
/* Right-Value Arithmatic PREFIX Operators */
      | PLUS_PLUS expr
      | MINUS_MINUS expr
      | MINUS expr %prec UNI_MINUS
/* Right-Value Logical PREFIX Operators */
      | EXCLAMATION expr
/* Assignment Operators */
      | lvalue ASSIGN expr
      | lvalue QUEST_ASSIGN expr
      | lvalue PLUS_ASSIGN expr
      | lvalue MINUS_ASSIGN expr
      | lvalue TIMES_ASSIGN expr
      | lvalue DIVIDE_ASSIGN expr
      | lvalue PIPE_ASSIGN expr
      | lvalue TELDA_ASSIGN expr
/* Evaluation: */
      | QUEST LPAREN lattice RPAREN
/* Optimization: This means: Don't optimize this lattice (expr). */
      | CARET LPAREN lattice RPAREN
```

```
/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/
/*=-=-=-=          Defining Left-Value Names            =-=-=-=*/
/*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=*/
lvalue:
        name  /* This lvalue is a name of a place in memory */
      | LPAREN lattice RPAREN  /* This lvalue is a lattice of names */
      | QUEST name
      | CARET name  /* This means: Don't optimize this lattice (expr). */

/* Any name that represents a place in memory: */
/* e.g.: '@(this.x.y[z.#3.4]{w.#"str"[@-3].#@x.#(x.y[z.#([n>2] x+1 | 0)])}.length)'
*/
name:
        rest_of_name
      | THIS
      | THIS DOT rest_of_name
      | at_operation
      | at_operation DOT rest_of_name

rest_of_name:
        attribute
      | label
      | rest_of_name DOT rest_of_name
      | rest_of_name LSQBRACKET altlat RSQBRACKET
      | rest_of_name LBRACE altlat RBRACE
/* Applying a lattice to a lattice (application of rules or constraints): E.g.
function call, rule application */
      | rest_of_name LPAREN lattice RPAREN

attribute:
        LENGTH
      | COUNT
      | LABELS
      | CLONE

at_operation:
        AT INTEGER
      | AT ID
      | AT LPAREN altlat RPAREN

number:
        INTEGER
      | FLOAT
```