**COMS 4115 Programming Languages and Translators:**

**Final Project Report**

**Lattakia (A language for Lattices)**

**Team Members:**

Wael Salloum

**(Team Leader)**

Hebatallah Elfardy

Katherine Scott

Li Yifan

Table of Contents

Chapter 1 5

Introduction 5

1.1 A General Introduction to Word Lattices 5

1.2 Word Lattices in Lisp 6

1.3 Word Lattices in Lattakia 6

1.4 Operations on Lattices 7

Chapter 2 9

2.1 Creating Latte and Running “Hello World” 9

2.2 Tutorial 2 (Lattakia Factorial) 9

2.3 Tutorial 3 (GCD) 10

2.4 Tutorial 4 (quicksort) 10

Chapter 3 - Reference Manual 11

3. 1 Lexical conventions 11

3.1.1 Comments 11

3.1.2 Identifiers 11

3.1.3 Keywords 11

def Keyword 11

let Keyword 11

true Keyword 11

false Keyword 11

nil Keyword 11

epsilon Keyword 11

3.1.4 Constants 11

3.2 Lattakia Types 12

3.2.1 Atomic Types: 12

3.2.1 Composite Types (Lattices) 12

3.2.3.1 Arithmetic expressions 13

3.2.3.2 String Expressions 14

3.2.3.3 Boolean and Comparison Expressions 14

3.2.3.4 Assignment expressions 15

3.2.3.5 Evaluation Expressions 15

3.2.3.6 Function Application expressions: *name (lattice)* 15

3.2.3.7 Each expressions: *{x}* 16

3.2.3.8 Parenthetic expressions: *(a; b; c)* 16

3.2.3.9 Accessor expressions: *‘.’* 16

3.3. Declarations 16

3.4 Built-in Library Functions 17

Print: print(expression) 17

Chapter 4 Project Plan 18

4.1 Process used for planning, specification, development and testing 18

4.2 Programming style guide 18

4.3 Project timeline 19

4.4 Roles and responsibilities of each team member 19

4.5 Software development environment used (tools and languages) 20

4.5.1 Java Development 20

4.5.2 O’Caml Development 20

4.5.3 Source Control 20

4.6 Project log 20

Chapter 5: Architectural Design 26

5.1 Major Architecture Discussion 26

Table 5.1 : Lattakia Modules 28

5.2 Components and Interfaces 29

5.2.1 Scanner (Token Generation) 29

5.2.2 Parser (Lattice, Symbol Table, and AST Generation) 29

5.2.3 Lattice/Environment 29

5.2.4 Semantic Analysis 32

5.2.5 Evaluation 32

5.2.6 Latte (interpreter) 32

5.3 Work Breakdown 32

5.3.1 Kat’s Work 32

5.3.2 Heba’s Work 32

5.3.3 Wael’s Work 33

5.3.4 Li’s Work 33

Chapter 6: Test Plan 34

6.1 Testing Overview 34

6.2 Testing Examples 34

6.3 Test Case Burn Down 35

Chapter 7: Lessons Learned 36

7.1 General Lessons Learned 36

7.2 Kat Lessons Learned 36

7.3 Heba Lessons Learned 36

7.4 Wael Lessons Learned 36

7.5 Li Lessons Learned 37

Appendix A: Source Code 38

Ast.ml source code 38

Clone.ml Source Code 41

evaluate.ml Source Code 49

latparser.mly Source Code 61

latte.ml Source Code 65

latticeCreation.ml Source Code 67

Scanner.ml Source Code 76

semanticAnalysis.ml Source Code 78

# Chapter 1

# Introduction

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. The Lattakia Latte interpreter is used to execute Lattakia programs and determine the result of any given program.

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 supporting 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.*

## 1.1 A General Introduction to Word Lattices

A *lattice* (a partially ordered set) is a special type of directed acyclic graphs. A *word lattice* is a special kind of lattices which is defined recursively as consisting of:

1. A word represented by a single arc and corresponds to an expression.
2. Alternative Lattice (“X | Y”) where X and Y are lattices.
3. Sequence Lattice (“X; Y”) where X and Y are lattices.

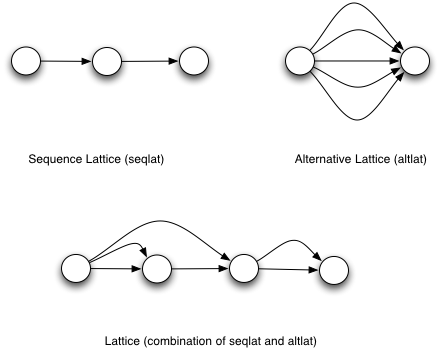


Figure 1.1: Word Lattices

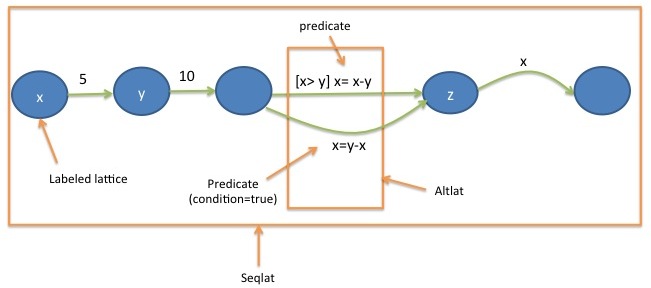


Figure 1.2: A lattice in Lattakia.

Figure 1.2 above gives an example of a lattice in Lattakia. In this example we an initial sequence lattice conditioned on the values x and y. The sequence lattice then feeds into an alternative lattice that contains a predicate conditioned on x and y. If the predicate is true x is then set to the value of y minus x.

## 1.2 Word Lattices in Lisp

A Common Lisp notation for word lattices includes three functions (WRD, OR, and SEQ) corresponding to the three constructing blocks of lattices above respectively. For example, the following notation corresponds to the third lattice in figure 1:

|  |
| --- |
| (SEQ  (OR  (WRD “”)  (SEQ  (OR  (WRD “”) (WRD “”))(WRD “”)  )  )  (OR  (WRD “”) (WRD “”)  )  ) |

Figure 1.3: Word Lattices in Lisp

Note that seqlats can also consist of other seqlats (not always altlats).

## 1.3 Word Lattices in Lattakia

Lattakia uses the same model used by Lisp for representing word lattices but with a different notation that is intended to make operation on lattices easier and more convenient.

A word in Lattakia is represented by a predicate which is a tuple of a condition and an expression. Predicates without a condition field have an implicit true condition while predicates without an expression correspond to an epsilon transition.

For example consider predicates consisting of integer constants and conditions.

1. An altlat is denoted as: “2 | 3 | 5”
2. A seqlat is denoted as: “2; 3; 5”
3. A seqlat lattice that constitutes words, and altlats is represented as

2; 3 | 5; 8; 4; 6 | 1 | 7; *where the “|” operator has a higher precedence over “;”*

*In the next sections we will explain in details the notation used by Lattakia to represent the different kind of word-lattices.*

## 1.4 Operations on Lattices

The section below gives an illustration of how arithmetic operations should function between lattices. Unfortunately, Lattakia does not support all of these functions, but we have included this tutorial for illustrative purposes.

For seqlat arithmetic operations the values in each lattice are simply added. For example:

(a; b; c) + (x; y; z) = (a+x; b+y; c+z)

(a; b; c) + (x; y) = (a+x; b+y; c)

(a; b) + (x; y; z) = (a+x; b+y; z)

For seqlat and altlat arithmetic operations the altlat translates to a seqlate where all of the matching elements evaluate to epsilon. That is to say we promote the altlat to me a seqlat:

(a; b; c) + (x | y | z) = (a; b; c) + ((x | y | z); epsilon; epsilon)

= (a + (x | y | z); b; c)

.. a + (x | y | z) is defined in WRD + OR

For operations between a sequence lattice and an integral type, the integral type is promoted to a seqlat where the first entry is the integral type, and all of the other entries are set to epsilon.

(a; b; c) + x = (a; b; c) + (x; epsilon; epsilon) = (a + x; b; c)

For arithmetic operations between integral types and seqlats we promote the integral type to seqlat where the first type is entry has a value and all of the other entries are epsilon

x + (a; b; c) = (x; epsilon; epsilon) + (a; b; c)  = (x + a; b; c)

For arithmetic operations between integral types and alternative lattices we demote the altlat to an expression and perform the arithmetic. For example:

x + (a | b | c) = x + (a + b + c)  = x + a + b + c

For arithmetic operations between an alternative lattice and a sequence lattice we promote the seqlat to an altlat where the first entry is the seqlat and all other matching entries are epsilon. For example:

(x | y | z) + (a; b; c) = (x | y | z) + ((a; b; c) | epsilon | epsilon)

= (x + (a; b; c) | y | z)

For the arithmetic operations on an altlat and an integral type we promote the type to a seqlat with the first entry equal to the integral type and all other types set to epsilon. For example:

(x | y | z) + a = (x | y | z) + (a | epsilon | epsilon) = (x + a | y | z)

# Chapter 2

## 2.1 Creating Latte and Running “Hello World”

In this section we will create and run a simple program that outputs “Hello World” in Lattakia programming language. To build the Lattakia compiler we first download the source code repository using SVN. The Lattakia source can be downloaded from:

<http://code.google.com/p/lattakia/>

Assuming that the OCaml compiler has been installed the Lattakia latte interpreter can be compiled by running the following commands from the command line:

>cd <svn home>/Lattakia/lat-compiler

>make all

After building the Lattakia latte interpreter we can test that it is functioning by running the test suite using the following command on the command line:

> make test

The Lattakia latte compiler processes .lat files written in the Lattakia language. For our “Hello World” program we will use a text editor to create a file called hello.lat, and put a single line of Lattakia code in the file like so:

\*\*

HelloWorld.lat

Prints “Hello World”

\*\*

print(“Hello World!\n”); .. print

To run our source file through the interpreter we will run the Lattakia Latte interpreter from the command line using the following command which will give us the following output:

> ./latte < HelloWorld.lat

Hello World!  
>

## 2.2 Tutorial 2 (Lattakia Factorial)

The following code demonstrates how to create a recursive function to calculate the factorial of a number in the Lattakia language.

\*\*

Calculate the factorial of a number.

Create a function f, with parameter n, if n < 1 return 1, otherwise return

n\*f(n-1)

\*\*

def f(n) = [n<1] 1 | n\*f(n-1);

a = f(5); .. apply the factorial to five and save the results

print(a); .. print the output

print("\n"); .. print an extra new line.

## 2.3 Tutorial 3 (GCD)

The following example demonstrates how to write a function that takes multiple parameters in the Lattakia language.

\*\*

GCD - calculate the greatest common denominator.

\*\*

def gcd(x; y) = ( [y == 0] x

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

);

a = 35;

b = 49;

print("GCD of 35 and 49 is ");

print(gcd(a; b));

print("\n");

## 2.4 Tutorial 4 (quicksort)

The following example demonstrates how to write a quicksort function that takes in a seqlat of numbers and returns the numbers sorted in descending order.

***NOTE: This example fails due to lack of the indexing operation.***

\*\*

Perform quicksort on a seqlat of integer values.

\*\*

def quicksort(input) =

(

let (less = (); greater = () ); .. create two local variables

[input.length <= 1] return = input | .. if we have a single value return

let pivot = input[0]; .. create a pivot

let input[0] = epsilon;

foreach(x; input; .. for each value in the array

[x < pivot] less = (less; x) .. create a less than lattice

| greater = (greater; x) .. and a greater than lattice

);

.. now recursively sort the new lattices and return a lattice

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

).return; .. return the input

values = (42;7;18;6;1;-3;15;30);

let sorted = quicksort(values);

# Chapter 3 - Reference Manual

### 3. 1 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.

### 3.1.1 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 “\*\*”.

### 3.1.2 Identifiers

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

### 3.1.3 Keywords

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

General keywords:  **let def**

Constant keywords:  **true false nil epsilon**

### def Keyword

defines a new variable and returns the value of the newly defined variable

### let Keyword

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

### true Keyword

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

### false Keyword

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

### nil Keyword

‘nil’ is a broken arc that you cannot pass through.

### epsilon Keyword

‘epsilon’ is an empty transition or a transition whose condition is always true.

### 3.1.4 Constants

There are several kinds of constants, as follows:

#### 3.1.4.1 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.

#### 3.1.4.2 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.

#### 3.1.4.3 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.

### 3.2 Lattakia 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, or a boolean value. 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.

### 3.2.1 Atomic Types:

Atomic types include integers, real numbers, strings and booleans.

### 3.2.1 Composite Types (Lattices)

Lattices are used to describe structured data and code. From a “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.

#### 3.2.1 Sequence Lattices

A seqlat in Lattakia is a sequence of blocks; each block is a tuple of a node and an altlat (a list of alternative lattices). These nodes can be assigned labels in order to access the corresponding altlats with these labels (similar to a hash table keys or an object’s fields). Each element in a seqlat may be given a ***label*** – a name to access it by. A Label is a name on a node of a seqlat. It is a variable that can be used to access the lattice between the label’s node and the next node in a seqlat. Elements of a seqlat are separated by a semi-colon

lat: (1; 2,;3) .. a seqlat of 3 elements

print(lat); ..prints (1;2;3)

#### 3.2.2 Alternative Lattices

Alternative lattices (altlats) are lists of alternatives. Each altlat is a sequence of one or more predicates. (explained in the next sub-section) Alternative lattices are processed sequentially one order at a time. Unlike sequence lattices though, elements are only processed if their condition is satisfied. Elements of an altlat are separated by a pipe. When evaluating an altlat, if the conditions of multiple predicates are true then the first of them is returned.

x: 1;

lat: [x>10] “GT 1”| [x == 10] “Eq.1” | “LT 1” .. an altlat of 3 elements

print(lat); ..prints LT 1

#### 3.2.3 Predicates

A predicate consists of an optional condition and an expression that will be executed if the condition holds. The condition is also an expression. Predicates without the condition field have an implicit true condition.

The expression is the basic building block of the language. To compose complex expressions, Lattakia includes a large range of operators, mainly for atomic types. Operators for composite types (lattices) aren’t currently supported. 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.

### 3.2.3.1 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. Applying any arithmetic operation to any operand and **nil** will evaluate to **nil** while applying any arithmetic operation to **epsilon** and any operand will evaluate that operand.

**Table 3.1 Lattakia Arithmetic Operators**

|  |  |  |
| --- | --- | --- |
| **Expression** | **Operator** | **Function** |
| Expression+expression | + | Sum of the two operands |
| Expression-Expression | - | Difference between the two operands |
| Expression\*Expression | \* | The product of two operands |
| Expression/Expression | / | The quotient between the two operators. The second operator cannot be zero. ( An runtime error will be thrown if the second operand evaluates to zero) |
| Expression%Expression | % | The result is the remainder of the two operands. The second operand must be positive |
| -Expression | - | The result is the negation of the operand. |

### 3.2.3.2 String Expressions

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

### 3.2.3.3 Boolean and Comparison 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.

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. The result is of boolean type.

Both operands of all logical expressions (except for logical negation which has one operand only) must evaluate to the same atomic type.

Comparing a non-nil value to nil will return nil (which is how we know that an error occurred more like null reference exception in Java) while comparing a nil value with nil returns true.

Epsilon is treated as being less than any value so comparing any value with epsilon will return true and comparing epsilon with epsilon returns true.

**Table 3.2 Boolean Expressions**

|  |  |  |
| --- | --- | --- |
| Operator | Example | Description |
| == | A==B | (integer, real, string or Boolean) The result is true if the two expressions evaluate to the same value and false otherwise.  Operands can be integer, real, string or Boolean or expressions evaluating to these types. |
| != | A!=B | The negation of the “==” operator  Operands can be integer, real, string or Boolean or expressions evaluating to these types. |
| <= | A<=B | The result is equal to the result of *(expression < expression) || (expression == expression)*.  Operands can be integer, real, string or expressions evaluating to these types. |
| < | A<B | The result is true if the numeric comparison is true, and false otherwise.  Operands can be integer, real, string or expressions evaluating to these types. |
| >= | A>=B | The result is equal to the result of *!(expression <= expression)*.  Operands can be integer, real, string or expressions evaluating to these types. |
| > | A>B | The result is equal to the result of *!(expression < expression)*.  Operands can be integer, real, string or expressions evaluating to these types. |
| && | A&&B | Logical and - The result is true if both operands are true, and false otherwise.  Operands should both be Booleans or expressions evaluating to Booleans |
| || | A||B | Logical or - the result is true if either operand is true, and false otherwise.  Operands should both be Booleans or expressions evaluating to Booleans |
| ! | !A | Logical negation - the result is true if the operand is false, and false otherwise.  Operand should be Booleans or expressions evaluating to a Boolean. |

### 3.2.3.4 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.

#### 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.

### 3.2.3.5 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.

#### Evaluation: ? expression

The result is simply the result of evaluating the operand.

def x = 3;

.. x is set 3 because 3 is a constant expression

def y = x;

.. y is bound to x

y = ?x;

.. but y = 3;

### 3.2.3.6 Function Application expressions: *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. 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

### 3.2.3.7 Each expressions: *{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. **‘{}’** overrides this behavior. Using “{}” 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 < {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.

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

Parentheses are used to surround elements of a lattice.

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)

### 3.2.3.9 Accessor expressions: *‘.’*

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

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)

## 3.3. Declarations

The only declarator that we use is the keyword def which is used to

1. Define variables (you have to define a variable before you use)
2. Define functions (whether these functions are labeled or defined using an assignment expression (local function declaration).

def x = 1 .. define a variable

def func1:(a; b; c ) .. labeled function declaration

def func2=(a;b;c) .. local function declaration

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

## 3.4 Built-in Library Functions

The standard library for Lattakia currently supports one functions (print).

### 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'

# Chapter 4 Project Plan

## 4.1 Process used for planning, specification, development and testing

Our project planning process involved a number of initial meetings where we a group first established a project concept and then white boarded the language syntax, semantics and functionality. Given the complex nature of the project Wael, the team leader took over the job of creating the parser and the initial AST structure. Wael also specified the Java interface and Kat took over the job of implementation. In the initial stages of the project we all collaborated remotely via instant message and a shared drop box folder. More than halfway through the semester it became obvious that we were all struggling to make sufficient progress on the project. At this point we opted to shift the focus of the language. Concurrently our team meetings shifted from informal post-class meetings to regular daily meetings in at the Center for Computational Learning Systems (CCLS). At CCLS we had access to a white board and projector where we could pair program or team code the more complex of the language. Having everyone in a single location allowed us to seamlessly hand off coding, testing, and writing to complete the project.

## 4.2 Programming style guide

Our group all had fairly similar coding styles to begin with so our coding style coalesced rather naturally. The table below gives some of the more formal coding style standards we used in OCaml.

**Table 4.1: Lattakia Style Guide**

|  |  |  |
| --- | --- | --- |
| **Style Object** | **Description** | **Example** |
| Variable Names | Camel Case | thisIsACamelVariable |
| Function Separators | Full line comment | (\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*) |
| Function Parameters | Append an underscore to avoid naming conflict | SeqLat(seqLat)-> doSomething seqLat  and doSomething \_seqLat |
| Function  Names | Use process to parse an expression.  Include return type in name if it makes it clearer | processLattice  processExprReturnType |
| Comments | Include a brief description at the top of each function. | <none> |
| Indentation | Use default for Java IDE | <none> |
| Types | Variables that represent types are appended by type. | dataTypeType |
| Global Data Types | Place shared data types into AST.ml | <none> |
| Main Interpreter | Main program should be minimal, subdivide each step of the process into another file. | semanticAnalysis.ml, evaluate.ml |
| Lattices | Everything is a lattice, use the a polymorphic type. | latticeType = altlat|seqlat|predicate|expr |
| Errors / Warmings | Use global error type and place into environment structure. Don’t throw in recursive code. |  |
| Bugs | Use the raise failure method to notify on bugs / todo’s | rf("BUG: SemanticAnalysis.processOpReturnType") |

## 4.3 Project timeline

The table below gives a summary of our project timeline as it actually occurred. Note the continuity break at task 9.

**Table 4.2 Lattakia Time Line**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Task/Week** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** |
| 1. Proposal | X | X |  |  |  |  |  |  |  |  |  |  |  |  |
| 2. Language Specification |  |  | X | X | X |  |  |  |  |  |  |  |  |  |
| 3. Lexer / Parser |  |  | X | X | X |  |  |  |  |  |  |  |  |  |
| 4. LRM |  |  |  | X | X |  |  |  |  |  |  |  |  |  |
| 5. AST Generation |  |  |  |  | X | X |  |  |  |  |  |  |  |  |
| 6. Java Code Interface |  |  |  |  | X | X | X |  |  |  |  |  |  |  |
| 7. Code Generation |  |  |  |  | X | X | X |  |  |  |  |  |  |  |
| 8. Java Code Implementation |  |  |  |  |  |  | X | X | X |  |  |  |  |  |
| 9. New Lattice Structure |  |  |  |  |  |  |  |  |  |  | X | X |  |  |
| 10. Semantic Check |  |  |  |  |  |  |  |  |  |  |  | X | X | X |
| 11. Lattice Evaluation |  |  |  |  |  |  |  |  |  |  |  | X | X | X |
| 12. Standard Library |  |  |  |  |  |  |  |  |  |  |  |  | X | X |
| 13. Testing |  |  |  |  |  |  |  |  |  |  |  |  |  | X |
| 14. Final Reporting |  |  |  |  |  |  |  |  |  |  |  |  |  | X |

## 4.4 Roles and responsibilities of each team member

**Table 4.3: Lattakia Roles and Duties.**

|  |  |  |
| --- | --- | --- |
| **Person** | **Development Duties** | **Project Duties** |
| Heba | Evaluation  Parser  Print Function  Old Semantic Analysis | Language Reference Manual  Final Report |
| Kat | Code Generation(Java)  Unit Test  Some Semantic Analysis  Print Function | Language Reference Manual (old group)  LRM Update  Final Report  Project Slides |
| Wael  (leader) | Parser  Lattice Creation  AST  Semantic Analysis  Code Generation (Java)  Print Function | Language Reference Manual  Final Report |
| Li |  |  |

## 4.5 Software development environment used (tools and languages)

### 4.5.1 Java Development

During the initial phase of development all java development used a stock distribution of the Eclipse Java IDE and java version 1.6. As discussed previously java development was scrapped in favor of developing an interpreter.

### 4.5.2 O’Caml Development

For O’Caml development the team primarily used Eclipse with the OcaIde plug-in for O’Caml development (<http://www.algo-prog.info/ocaide/>). For most of the project Kat opted to use emacs along with a package that allowed for O’Caml syntax highlighting and debugging (<http://www.cs.jhu.edu/~scott/pl/caml/emacs.html>). Kat notes that the auto-completion in eclipse makes life a lot easier when you are still learning O’Caml.

### 4.5.3 Source Control

The group initially used drop-box as the mechanism for source control. Once Kat came on board the decision was made to transition to a traditional source control mechanism such as SVN. Our SVN repository is hosted on Google code and can be found here: <http://code.google.com/p/lattakia/>. The main trunk of the project has four branches, docs – for documents, lat-compiler – the scanner/parser/interpreter, latte-java – the deprecated java back end, test-cases – the testing system.

## 4.6 Project log

Below is our project log pulled from our SVN repository.

* r143 | katherineAScott@gmail.com | 2011-12-22 14:02:02 -0500 (Thu, 22 Dec 2011)
  + fixed test cases
* r142 | katherineAScott@gmail.com | 2011-12-22 13:23:38 -0500 (Thu, 22 Dec 2011)
  + Added code signing
* r141 | katherineAScott@gmail.com | 2011-12-22 11:25:21 -0500 (Thu, 22 Dec 2011)
  + moar
* r140 | wael.salloum@gmail.com | 2011-12-22 11:22:42 -0500 (Thu, 22 Dec 2011)
  + some eval bugs solved. most of evaluation is done, except NameList, DirectAccess, Assign, Define, FuncCall, Tilde. Semantic still have bugs. -3
* r139 | katherineAScott@gmail.com | 2011-12-22 11:22:00 -0500 (Thu, 22 Dec 2011)
  + added source to document
* r138 | katherineAScott@gmail.com | 2011-12-21 21:47:37 -0500 (Wed, 21 Dec 2011)
  + more documents
* r137 | heba.elfardy@gmail.com | 2011-12-21 19:23:18 -0500 (Wed, 21 Dec 2011)
  + changing the LRM according to the final specs
* r136 | heba.elfardy@gmail.com | 2011-12-21 19:14:07 -0500 (Wed, 21 Dec 2011)
  + process-name-list
* r135 | katherineAScott@gmail.com | 2011-12-21 19:08:29 -0500 (Wed, 21 Dec 2011)
  + added pretty print
* r134 | wael.salloum@gmail.com | 2011-12-21 18:03:12 -0500 (Wed, 21 Dec 2011) | 1 line
* r134 | wael.salloum@gmail.com | 2011-12-21 18:03:12 -0500 (Wed, 21 Dec 2011)
  + some eval bugs solved. most of evaluation is done, except NameList, DirectAccess, Assign, Define, FuncCall, Tilde. Semantic still have bugs. -3
* r133 | wael.salloum@gmail.com | 2011-12-21 16:53:36 -0500 (Wed, 21 Dec 2011)
  + some eval bugs solved. most of evaluation is done, except NameList, DirectAccess, Assign, Define, FuncCall, Tilde. Semantic still have bugs. -2
* r132 | katherineAScott@gmail.com | 2011-12-21 14:36:01 -0500 (Wed, 21 Dec 2011)
  + report figures
* r131 | katherineAScott@gmail.com | 2011-12-21 14:33:19 -0500 (Wed, 21 Dec 2011)
  + added block diagrams for ast
* r130 | wael.salloum@gmail.com | 2011-12-21 08:18:56 -0500 (Wed, 21 Dec 2011)
  + some eval bugs solved. most of evaluation is done, except NameList, DirectAccess, Assign, Define, FuncCall, Tilde. Semantic still have bugs.
* r129 | wael.salloum@gmail.com | 2011-12-21 06:49:55 -0500 (Wed, 21 Dec 2011)
  + most of evaluation is done, except NameList, DirectAccess, Assign, Define, FuncCall, Tilde. Semantic still have bugs
* r128 | katherineAScott@gmail.com | 2011-12-20 23:06:21 -0500 (Tue, 20 Dec 2011)
  + added quick sort
* r127 | katherineAScott@gmail.com | 2011-12-20 23:05:54 -0500 (Tue, 20 Dec 2011)
  + uhg calling it a night
* r126 | wael.salloum@gmail.com | 2011-12-20 23:05:14 -0500 (Tue, 20 Dec 2011)
  + semantic analysis version 1 finished
* r125 | katherineAScott@gmail.com | 2011-12-20 19:59:54 -0500 (Tue, 20 Dec 2011)
  + safety commit
* r124 | wael.salloum@gmail.com | 2011-12-20 17:34:50 -0500 (Tue, 20 Dec 2011)
  + symbol table bugs were solved
* r123 | wael.salloum@gmail.com | 2011-12-20 15:32:04 -0500 (Tue, 20 Dec 2011)
  + two symbol Table bugs were solved. there is still more :(
* r122 | wael.salloum@gmail.com | 2011-12-20 14:52:16 -0500 (Tue, 20 Dec 2011)
  + semantic check: Parameter types checked (param type inference from FuncCall) + most ops handled (only Assign and Define left) + NameListExpr is not handled yet.
* r121 | katherineAScott@gmail.com | 2011-12-20 13:42:59 -0500 (Tue, 20 Dec 2011)
  + small changes to the tester
* r120 | wael.salloum@gmail.com | 2011-12-20 02:39:42 -0500 (Tue, 20 Dec 2011)
  + semantic check: Parameter types checked (param type inference from FuncCall) + most ops handled (only Assign and Define left) + NameListExpr is not handled yet
* r119 | heba.elfardy@gmail.com | 2011-12-20 00:38:13 -0500 (Tue, 20 Dec 2011)
  + more evaluate stuff
* r118 | katherineAScott@gmail.com | 2011-12-19 21:58:27 -0500 (Mon, 19 Dec 2011)
  + Need to fill in the functional tests.
* r117 | katherineAScott@gmail.com | 2011-12-19 21:54:11 -0500 (Mon, 19 Dec 2011)
  + Added functional tests
* r116 | wael.salloum@gmail.com | 2011-12-19 21:21:59 -0500 (Mon, 19 Dec 2011)
  + semantic check -- 3
* r115 | katherineAScott@gmail.com | 2011-12-19 21:12:49 -0500 (Mon, 19 Dec 2011)
  + This looks good
* r114 | katherineAScott@gmail.com | 2011-12-19 21:07:15 -0500 (Mon, 19 Dec 2011)
  + cleaning up test names
* r113 | wael.salloum@gmail.com | 2011-12-19 20:40:02 -0500 (Mon, 19 Dec 2011)
  + semantic check -- 3
* r112 | wael.salloum@gmail.com | 2011-12-19 20:37:45 -0500 (Mon, 19 Dec 2011)
  + semantic check -- 4
* r111 | wael.salloum@gmail.com | 2011-12-19 20:35:44 -0500 (Mon, 19 Dec 2011)
  + semantic check -- 3
* r110 | wael.salloum@gmail.com | 2011-12-19 20:20:08 -0500 (Mon, 19 Dec 2011)
  + semantic check -- 2
* r109 | katherineAScott@gmail.com | 2011-12-19 20:06:50 -0500 (Mon, 19 Dec 2011)
  + we really need to work in the same directory
* r108 | katherineAScott@gmail.com | 2011-12-19 19:55:56 -0500 (Mon, 19 Dec 2011)
  + okay this should get us closer to god
* r107 | katherineAScott@gmail.com | 2011-12-19 19:08:22 -0500 (Mon, 19 Dec 2011)
  + this is a start
* r106 | katherineAScott@gmail.com | 2011-12-19 18:18:06 -0500 (Mon, 19 Dec 2011)
  + adding output
* r105 | katherineAScott@gmail.com | 2011-12-19 17:19:45 -0500 (Mon, 19 Dec 2011)
  + got some basic tests going
* r104 | wael.salloum@gmail.com | 2011-12-18 21:49:59 -0500 (Sun, 18 Dec 2011)
  + semantic check -- 2
* r103 | wael.salloum@gmail.com | 2011-12-18 19:09:26 -0500 (Sun, 18 Dec 2011)
  + semantic check – 1
* r102 | katherineAScott@gmail.com | 2011-12-18 17:57:31 -0500 (Sun, 18 Dec 2011)
  + semantics are broken I suck
* r101 | wael.salloum@gmail.com | 2011-12-18 17:21:10 -0500 (Sun, 18 Dec 2011)
  + expressions finished. still working on a bug in SymbolTable. error handling for both parsing and scanning is done. printLattice is done. -- 4
* r100 | katherineAScott@gmail.com | 2011-12-18 16:54:15 -0500 (Sun, 18 Dec 2011)
  + working on semantic analysis
* r99 | wael.salloum@gmail.com | 2011-12-18 14:32:37 -0500 (Sun, 18 Dec 2011)
  + Expressions finished. still working on a bug in SymbolTable. error handling for both parsing and scanning is done. printLattice is done. -- 2
* r98 | wael.salloum@gmail.com | 2011-12-18 11:22:18 -0500 (Sun, 18 Dec 2011)
  + Expressions finished. still working on a bug in SymbolTable. error handling for both parsing and scanning is done. printLattice is done.
* r97 | wael.salloum@gmail.com | 2011-12-18 05:32:46 -0500 (Sun, 18 Dec 2011)
  + updated expression and namelist (but not final) -- 3
* r96 | wael.salloum@gmail.com | 2011-12-18 02:05:00 -0500 (Sun, 18 Dec 2011)
  + updated expression and namelist (but not final) -- 2
* r95 | heba.elfardy@gmail.com | 2011-12-17 19:56:55 -0500 (Sat, 17 Dec 2011)
  + process-altlat
* r94 | heba.elfardy@gmail.com | 2011-12-17 19:46:46 -0500 (Sat, 17 Dec 2011)
  + process-seqlat almost working
* r93 | heba.elfardy@gmail.com | 2011-12-17 19:08:59 -0500 (Sat, 17 Dec 2011)
  + fixing indentation
* r92 | heba.elfardy@gmail.com | 2011-12-17 19:02:58 -0500 (Sat, 17 Dec 2011)
  + evaluation of some expressions
* r91 | katherineAScott@gmail.com | 2011-12-17 17:42:02 -0500 (Sat, 17 Dec 2011)
  + forgot svn is directory specific
* r90 | katherineAScott@gmail.com | 2011-12-17 17:41:26 -0500 (Sat, 17 Dec 2011)
  + changed tester to spit out stderr, working on semantic analysis make test! note change to latte.ml
* r89 | katherineAScott@gmail.com | 2011-12-17 14:56:30 -0500 (Sat, 17 Dec 2011)
  + okay tests done
* r88 | katherineAScott@gmail.com | 2011-12-17 14:56:05 -0500 (Sat, 17 Dec 2011)
  + Okay, testing should now work from makefile
* r87 | katherineAScott@gmail.com | 2011-12-17 14:41:43 -0500 (Sat, 17 Dec 2011)
  + testing almost done
* r86 | katherineAScott@gmail.com | 2011-12-17 14:36:43 -0500 (Sat, 17 Dec 2011)
  + adding diff output files
* r85 | katherineAScott@gmail.com | 2011-12-17 14:26:19 -0500 (Sat, 17 Dec 2011)
  + close to done, just gotta put in the make file
* r84 | katherineAScott@gmail.com | 2011-12-17 14:19:02 -0500 (Sat, 17 Dec 2011)
  + unit tests are close, just gotta capture diff output and do makefile integration
* r83 | katherineAScott@gmail.com | 2011-12-17 13:23:19 -0500 (Sat, 17 Dec 2011)
  + working on unit tests - sed is not right
* r82 | wael.salloum@gmail.com | 2011-12-17 12:38:29 -0500 (Sat, 17 Dec 2011)
  + updated expression and namelist (but not final)
* r81 | wael.salloum@gmail.com | 2011-12-17 12:36:37 -0500 (Sat, 17 Dec 2011)
* r80 | wael.salloum@gmail.com | 2011-12-17 10:48:56 -0500 (Sat, 17 Dec 2011)
* r79 | katherineAScott@gmail.com | 2011-12-16 21:56:07 -0500 (Fri, 16 Dec 2011)
  + print is kinda working
* r78 | katherineAScott@gmail.com | 2011-12-16 21:50:02 -0500 (Fri, 16 Dec 2011)
  + added print lattice back
* r77 | katherineAScott@gmail.com | 2011-12-16 21:32:07 -0500 (Fri, 16 Dec 2011)
  + still working on the print function
* r76 | katherineAScott@gmail.com | 2011-12-16 21:17:35 -0500 (Fri, 16 Dec 2011)
  + working on dump function
* r75 | katherineAScott@gmail.com | 2011-12-16 19:59:06 -0500 (Fri, 16 Dec 2011)
  + lattice has output now
* r74 | heba.elfardy@gmail.com | 2011-12-16 19:45:25 -0500 (Fri, 16 Dec 2011)
  + delete extra files
* r73 | heba.elfardy@gmail.com | 2011-12-16 19:45:13 -0500 (Fri, 16 Dec 2011)
  + delete extra files
* r72 | wael.salloum@gmail.com | 2011-12-16 19:02:53 -0500 (Fri, 16 Dec 2011)
* r71 | katherineAScott@gmail.com | 2011-12-16 18:02:54 -0500 (Fri, 16 Dec 2011)
  + moving lattice tools
* r70 | katherineAScott@gmail.com | 2011-12-16 17:58:34 -0500 (Fri, 16 Dec 2011)
  + notes from meeting and dump functions
* r69 | wael.salloum@gmail.com | 2011-12-16 17:57:57 -0500 (Fri, 16 Dec 2011) | 1 lin
* r68 | heba.elfardy@gmail.com | 2011-12-16 16:48:42 -0500 (Fri, 16 Dec 2011)
  + delete ml file
* r67 | wael.salloum@gmail.com | 2011-12-16 16:33:03 -0500 (Fri, 16 Dec 2011)
* r66 | wael.salloum@gmail.com | 2011-12-16 16:17:12 -0500 (Fri, 16 Dec 2011)
* r65 | katherineAScott@gmail.com | 2011-12-15 13:26:52 -0500 (Thu, 15 Dec 2011)
  + moved stray test files
* r64 | katherineAScott@gmail.com | 2011-12-15 13:25:15 -0500 (Thu, 15 Dec 2011)
  + okay we're done with dropbox
* r63 | katherineAScott@gmail.com | 2011-12-15 13:17:30 -0500 (Thu, 15 Dec 2011)
  + going to help Heba
* r62 | katherineAScott@gmail.com | 2011-12-15 13:15:19 -0500 (Thu, 15 Dec 2011)
  + working on getting this running, still don't have the compiler
* r61 | wael.salloum@gmail.com | 2011-12-15 13:02:06 -0500 (Thu, 15 Dec 2011)
* r60 | wael.salloum@gmail.com | 2011-12-15 12:58:32 -0500 (Thu, 15 Dec 2011)
  + interpreter
* r59 | katherineAScott@gmail.com | 2011-12-15 12:56:40 -0500 (Thu, 15 Dec 2011)
  + getting testing going
* r58 | heba.elfardy@gmail.com | 2011-12-08 14:15:14 -0500 (Thu, 08 Dec 2011)
* r57 | heba.elfardy@gmail.com | 2011-12-08 14:11:46 -0500 (Thu, 08 Dec 2011)
* r56 | heba.elfardy@gmail.com | 2011-12-07 23:17:16 -0500 (Wed, 07 Dec 2011)
* r55 | heba.elfardy@gmail.com | 2011-12-07 14:55:00 -0500 (Wed, 07 Dec 2011)
* r54 | heba.elfardy@gmail.com | 2011-12-07 14:28:52 -0500 (Wed, 07 Dec 2011)
* r53 | heba.elfardy@gmail.com | 2011-12-07 13:19:24 -0500 (Wed, 07 Dec 2011)
* r52 | heba.elfardy@gmail.com | 2011-12-07 13:18:04 -0500 (Wed, 07 Dec 2011)
* r51 | heba.elfardy@gmail.com | 2011-12-07 12:13:45 -0500 (Wed, 07 Dec 2011)
* r50 | heba.elfardy@gmail.com | 2011-12-07 02:25:46 -0500 (Wed, 07 Dec 2011)
* r49 | heba.elfardy@gmail.com | 2011-12-07 01:37:48 -0500 (Wed, 07 Dec 2011)
* r48 | heba.elfardy@gmail.com | 2011-12-07 01:37:22 -0500 (Wed, 07 Dec 2011)
* r47 | heba.elfardy@gmail.com | 2011-12-07 01:28:06 -0500 (Wed, 07 Dec 2011)
* r46 | heba.elfardy@gmail.com | 2011-12-06 17:25:20 -0500 (Tue, 06 Dec 2011)
* r45 | katherineAScott@gmail.com | 2011-12-05 23:07:46 -0500 (Mon, 05 Dec 2011)
  + added operations
* r44 | katherineAScott@gmail.com | 2011-12-05 22:14:09 -0500 (Mon, 05 Dec 2011)
  + moved this over from the drop box
* r43 | heba.elfardy@gmail.com | 2011-12-05 09:54:28 -0500 (Mon, 05 Dec 2011)
* r42 | heba.elfardy@gmail.com | 2011-12-01 19:24:05 -0500 (Thu, 01 Dec 2011)
* r41 | heba.elfardy@gmail.com | 2011-11-24 20:49:34 -0500 (Thu, 24 Nov
* 2011)
* r40 | heba.elfardy@gmail.com | 2011-11-24 20:40:27 -0500 (Thu, 24 Nov 2011)
  + translate
* r39 | heba.elfardy@gmail.com | 2011-11-24 20:38:17 -0500 (Thu, 24 Nov 2011)
* code-generation-version 1.0
* r38 | heba.elfardy@gmail.com | 2011-11-23 21:56:35 -0500 (Wed, 23 Nov 2011)
* code-generation-version 0.0
* r37 | heba.elfardy@gmail.com | 2011-11-20 21:23:38 -0500 (Sun, 20 Nov 2011)
  + Copied the code in mli to ml
* r36 | heba.elfardy@gmail.com | 2011-11-20 21:20:13 -0500 (Sun, 20 Nov 2011)
* r35 | heba.elfardy@gmail.com | 2011-11-14 19:05:26 -0500 (Mon, 14 Nov 2011)
* r34 | heba.elfardy@gmail.com | 2011-11-14 19:05:19 -0500 (Mon, 14 Nov 2011)
* r33 | heba.elfardy@gmail.com | 2011-11-14 19:05:02 -0500 (Mon, 14 Nov 2011)
* r32 | heba.elfardy@gmail.com | 2011-11-14 19:04:28 -0500 (Mon, 14 Nov 2011)
* r31 | heba.elfardy@gmail.com | 2011-11-14 19:04:21 -0500 (Mon, 14 Nov 2011)
* r30 | heba.elfardy@gmail.com | 2011-11-14 19:04:15 -0500 (Mon, 14 Nov 2011)
* r29 | heba.elfardy@gmail.com | 2011-11-14 19:04:06 -0500 (Mon, 14 Nov 2011)
* r28 | heba.elfardy@gmail.com | 2011-11-14 19:02:31 -0500 (Mon, 14 Nov 2011)
  + ocaml
* r27 | heba.elfardy@gmail.com | 2011-11-14 18:41:01 -0500 (Mon, 14 Nov 2011)
* r26 | heba.elfardy@gmail.com | 2011-11-14 18:40:40 -0500 (Mon, 14 Nov 2011)
* r25 | heba.elfardy@gmail.com | 2011-11-14 18:36:11 -0500 (Mon, 14 Nov 2011)
  + java
* r24 | heba.elfardy@gmail.com | 2011-11-14 18:34:47 -0500 (Mon, 14 Nov
* 2011)
  + java
* r23 | heba.elfardy@gmail.com | 2011-11-14 18:34:21 -0500 (Mon, 14 Nov 2011)
  + java
* r22 | heba.elfardy@gmail.com | 2011-11-14 18:33:42 -0500 (Mon, 14 Nov 2011)
* r21 | heba.elfardy@gmail.com | 2011-11-14 18:33:35 -0500 (Mon, 14 Nov 2011)
* r20 | heba.elfardy@gmail.com | 2011-11-14 18:32:29 -0500 (Mon, 14 Nov
* 2011)
  + test-case 2
* r19 | heba.elfardy@gmail.com | 2011-11-14 18:32:15 -0500 (Mon, 14 Nov 2011)
  + test-case 1
* r18 | heba.elfardy@gmail.com | 2011-11-14 18:20:20 -0500 (Mon, 14 Nov 2011)
* r17 | heba.elfardy@gmail.com | 2011-11-14 18:19:48 -0500 (Mon, 14 Nov 2011)
* r16 | heba.elfardy@gmail.com | 2011-11-14 18:19:33 -0500 (Mon, 14 Nov 2011)
* r15 | heba.elfardy@gmail.com | 2011-11-14 18:19:17 -0500 (Mon, 14 Nov 2011)
* r14 | heba.elfardy@gmail.com | 2011-11-14 18:19:11 -0500 (Mon, 14 Nov 2011)
* r13 | heba.elfardy@gmail.com | 2011-11-14 18:18:57 -0500 (Mon, 14 Nov 2011)
* r12 | heba.elfardy@gmail.com | 2011-11-14 18:18:22 -0500 (Mon, 14 Nov 2011)
* r11 | heba.elfardy@gmail.com | 2011-11-14 18:18:14 -0500 (Mon, 14 Nov 2011)
* r10 | heba.elfardy@gmail.com | 2011-11-14 18:18:08 -0500 (Mon, 14 Nov 2011)
* r9 | heba.elfardy@gmail.com | 2011-11-14 18:17:57 -0500 (Mon, 14 Nov 2011)
* r8 | heba.elfardy@gmail.com | 2011-11-14 18:17:52 -0500 (Mon, 14 Nov 2011)
* r7 | heba.elfardy@gmail.com | 2011-11-14 18:17:42 -0500 (Mon, 14 Nov 2011)
* r6 | heba.elfardy@gmail.com | 2011-11-14 18:17:33 -0500 (Mon, 14 Nov 2011)
* r5 | heba.elfardy@gmail.com | 2011-11-14 18:17:20 -0500 (Mon, 14 Nov 2011)
* r4 | heba.elfardy@gmail.com | 2011-11-14 18:17:12 -0500 (Mon, 14 Nov 2011)
* r3 | heba.elfardy@gmail.com | 2011-11-14 18:14:21 -0500 (Mon, 14 Nov 2011)

# Chapter 5: Architectural Design

## 5.1 Major Architecture Discussion

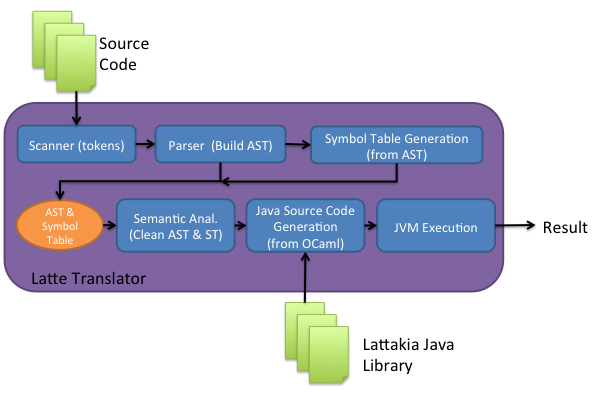
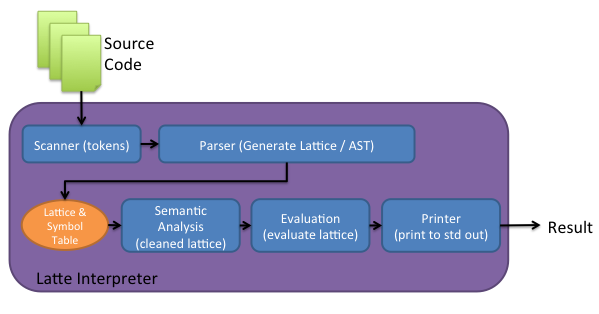
Our initial project goal was to create a translator that would take in a valid Lattakia program and translate it into a java lattice implementation that could be executed. This design proved to be too technically challenging and complex for the scope of this project. This initial design is displayed in Figure 5.1. This design was challenging for a number of reasons, but one of the most serious concerns was the vast amount of code that needed to be generated to implement a working lattice structure in java. This project begin with a working Java lattice library from which we could work, and there were no open source packages upon which we could draw. This meant that in addition to writing our translator we would need to build an entire java lattice library from scratch. One of the major concerns we encountered as we were developing our Java library was the number of arithmetic operations that we had to write. Initially, we intended for lattice arithmetic operations to support a complex type promotion system where lesser types would be promoted to superior types. For example, the addition of the float seqlat (1.0;2.0,3.0) with a string seqlat (“A”;”B”;”C”) would result in the float implementation being promoted to strings and result in (“1A”;”2B”;”3C”). To realize this in a java library would require that for each integral type, and each lattice type, we would need to define a complex arithmetic function. With three base types (int,float,string), three lattice types (seq,alt,and word), and over a dozen operation (+,-,\*,%,/,unary -, ==,!=,>=,>,<,<=) the combinatorics of this arrangement were prohibitive.

Figure : Original Lattakia Block Diagram – translation from Lattakia source to java source code.

The template method we concocted to perform a lattice arithmetic operation was almost 200 lines of code, given that Lattakia has three integral types and a dozen operations we would need to write six permutations of each for each of the dozen comparison and arithmetic operations. Implemented naively this would require roughly 14000 lines of code. We discussed a number of solutions to simplify this process, the first being the use of a polymorphic type to encapsulate operations, and then applying it to our template function. While this approach would have sped up the development process significantly, it was vetoed for fear of the run-time polymorphism being too computationally expensive. A second approach was to have the java code generated automatically using either a Python or Perl script. However there were fears that this approach might be error prone and difficult.

In addition to Java Library concerns, our original design has a number of components that would be difficult to integrate into a final translator. For example, this design required separate code for generating the AST and the symbol table, followed by code to generate the lattice structures in Java. This java source code generation in OCaml is a nightmare from a development process standpoint; as it would be difficult to maintain the code “inside” of our translator. One potential solution would have been for OCaml to parse our java source files directly to generate lattice code, but we were unsure of how to do this. Another possible solution was to use fixed java code snippets within our OCaml code generation module, but this approach would be difficult to debug and update.

At this point in the development process we decided to radically change of project architecture and to dramatically limit the scope of the Lattakia language. A revised system block diagram is given in figure 5.2. As evident in the figure our most significant change was to transition Lattakia from being a language translator to an interpreter. This step allowed us to collapse AST generation, symbol table generation, and lattice formation down to a single step. This change allowed us to treat our AST as our lattice structure and evaluate it directly. Since Lattakia is a functional language doing this allowed us to use the functional tools built into OCaml versus trying to impose functional constructs onto Java. The complex list iteration that we would have had to perform in Java could be replaced with recursion in OCaml and dramatically reduce the amount of code we would need to write. By switching to an interpreter we also avoided the hassle of having a code generation step that would require we place java code snippets into OCaml and the complex testing and debugging process that would require. We also imposed a number of constraints on Lattakia that differ from the original Language Reference Manual. For example, types are no longer automatically promoted, which makes Lattakia more strongly typed than we would like, but also simplified our development process.

Figure : Revised Lattakia Interpreter Block Diagram

An overview of the revised Lattakia architecture is given in Figure 5.2. The main entry point for the interpreter is the lattte module. The latte module holds the O’Caml modules for the scanner, parser and lexer to translate source files into our abstract syntax tree and symbol table. The scanner is contained in scanner.mll and uses the lattice parser and lattice creation package to generate a single lattice structure that functions as both our AST, an initial symbol table, and a set of warning tables, which we collectively call the environment. If scanning proceeds successfully then the environment is passed to the semantic analysis module. If there are any errors during scanning they are reported via the error table in the environment. After scanning is completed the core lattice is passed to the semantic analysis module that performs a limited set of sematic checks, such as making sure all predicates evaluate to a Boolean type, comparing the return types of expressions, and scope errors. If an error is found, an error message is pushed onto the lattice error table and it is printed at the end of semantic analysis. After semantic analysis the lattice structure is passed to evaluation, which traverses and evaluates the lattice structure and reports the results of evaluation. During evaluation the print statement makes heavy use of the print library to print any user output. This output is used to heavily by the testing system.

As discussed previously the entirety of a Lattakia program is treated as a lattice structure (i.e. a seqlat of atomic expressions and labels). This greatly simplifies the evaluation process of the Lattakia language by allowing us to work recursively with a single lattice structure that holds all of the lattices in any given program as well as the global symbol table. When the lattice structure is traversed scoping is performed by creating local copies of the global symbol table. Our implementation of Lattakia is very simplistic at this point , and the entire code base can be contained in only a couple of files. Table 5.1 summarizes all of the Lattakia O’Caml source files (we have omitted the Java files that were deprecated). Each of these modules is discussed in details in the next section.

## Table 5.1 : Lattakia Modules

|  |  |  |
| --- | --- | --- |
| **Source File** | **Modules Defined** | **Function Description** |
| ast.ml | Lattice (the ast) Symbol Table (env) | AST and Symbol Table definition and helper functions. |
| evaluate.ml | Eval lattice | Evaluates a lattice structure and returns the result. |
| latparser.ml/latparser.mly | Parser | Lattice semantic definition  Expression semantic definition |
| latte.ml | Latte Interpreter | Main entry point for all modules. |
| latticeCreation.ml | Lattice manipulation and creation. | Creates lattices from the latte parser. |
| printLattice.ml | Helper function for debugging and printing output. | Text output. |
| scanner.mll | O’Caml Scanner | Translates raw code files into tokens. |
| semanticAnalysis.ml | semanticAnalysis | Does basic semantic analysis via type checking. |

## 5.2 Components and Interfaces

### 5.2.1 Scanner (Token Generation)

The Lattakia scanner is a vanilla scanner implementation in OCaml. The scanner takes the basic Lattakia input file and converts it to a set of tokens that are passed to the parser.

### 5.2.2 Parser (Lattice, Symbol Table, and AST Generation)

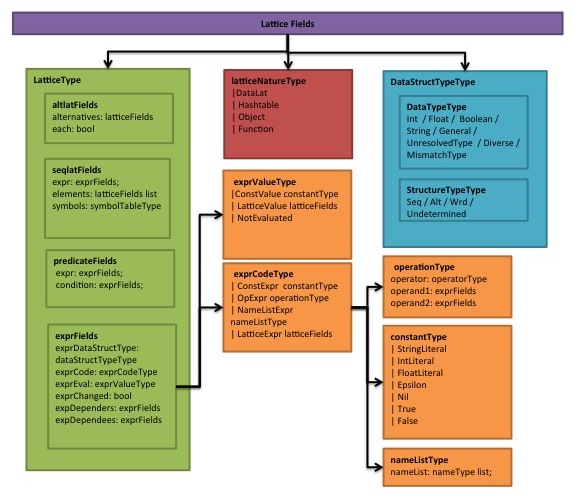
In our interpreter, parsing, AST generation, and AST conversion to a lattice structure are all done in a single pass. Since all code and data in Lattakia is contained in lattices, these lattices are equivalent to the abstract syntax tree, and we are able to bypass the translation step. Specifically, the code in latparser.mly matches Lattakia patterns and generates an equivalent lattice data structure (defined in ast.ml) using the helper functions defined in latticeCreation.ml. During parsing we use a global environment data structure, of type “envType” ; this environment data structure holds a local symbol table, the current lattice, a set of lists for error and warning handling, and assorted book- keeping variables for tracing our path back up the parse tree. When a parse error is encountered we create an errorType data structure, report the results, place the error in the environment, and continue to parse the file. When parsing completes we check the error table in the environment and report any errors. Errors during parsing are specified using a errorCategoryType and errorMessageType, the category of the error will be either a warning, syntactic, or lexical, while the message type will either be a SynParseError, or a LexUndefinedSymbol. These parse errors are handled in the main latte.ml function and are printed to the screen.

### 5.2.3 Lattice/Environment

The Lattakia parser returns an environment data type which contains a list of lattice fields and, the symbol table, and various other data types. After parsing this data type is piped to the semantic analysis, evaluation, and printing modules. Within the environment data type, the lattice field list data structure serves as a representation of the source code (see Figure 5.2). The lattice field list data structure contains the dataStructureTypeType which is a tuple of the dataTypeType (e.g. float, int, string, Boolean, or general) and the structureTypeType (either word, altlat, seqlat, or unknown). The lattice field data structure also contains latticeType which is a polymorphic data type that holds either a seqlat, an altlat, an expression, or a predicate. The seqlat lattice type holds a list of lattice fields for each element, while the altlat lattice holds a list of lattice fields for each alternative. The expression lattice type holds the expression type (e.g. constant, an operation, a lookup, lattice operation), and the return type of the expression (constant, lattice, or unknown). Predicate lattice types hold two expression types, one for the conditional, and one for the resulting condition if action evaluates to true. The latEval member of a lattice field holds a list of other lattice fields which may be part of the current lattice. Finally the lattice field has a set of list for parameters and types if the lattice represents a function. .

The symbol table data structure contains a string map between symbol names and a symbolType data structure (see Figure 5.3). The symbolType data structure holds a description of the dataType ( lattice type, and integral type), the symbol value, a value indicating the scope of the symbol, and the symbol name. This symbol table is then used in the environment data type. The environment data structure holds a symbol table, a stack identification number generator, a table of warnings and errors, a pointer to the symbol table entry of the current lattice, and a number for the current code line and column.

Figure .2 The Lattice Field Data Structure take from ast.ml



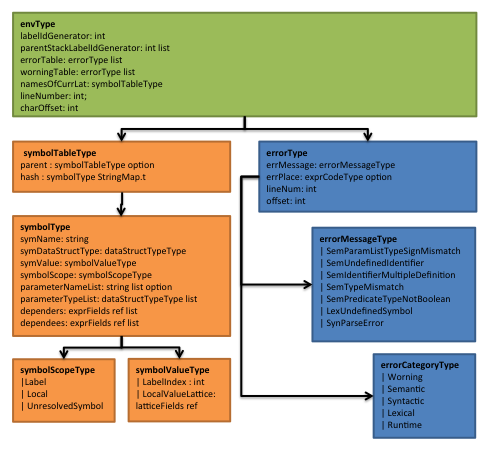


Figure .3: The envType data structure – this data structure holds the lattice, the symbol table, and the warning and error tables.

### 5.2.4 Semantic Analysis

All of the Lattakia semantic analysis is performed in the semanticAnalysis.ml file, using the root lattice data structure and the environment data type. The majority of the semantic checks performed by Lattakia fall into the category of type match checking. Semantic type checking is performed by descending the lattice structure until only constant types are encountered and then comparing the returned types used in every expression. For example, in arithmetic operations all of the variables and constants must have the same type, and this value must match the value on the left hand side of an assignment. Any deviation from rule will result in semantic error that will be placed in the environment’s error table. We perform other, additional, semantic checks on the lattice during the recursion for basic type checking. For example, the Lattakia interpreter checks that all predicates evaluate to a Boolean type, and that all variables are defined only once in the scope of the symbol table. Lattakia also checks the semantics of function calls to validate that functions are defined and that the number of parameters passed to a function are correct.

### 5.2.5 Evaluation

After semantic analysis Lattakia proceeds to evaluate the lattice structure by descending the lattice structure and processing each lattice until the end of the tree is reached. Lattakia evaluates each lattice expression and returns the results of the operation as values in the symbol table. Print statements are handled by the Lattakia print utility. During evaluation if any type mismatch occurs between the components in an expression Lattakia throws a run-time type mismatch exception.

### 5.2.6 Latte (interpreter)

The latte interpreter integrates all of the other Lattakia modules together into a single executable interpreter. Once compiled the Latte, or the Lattakia interpreter takes in a single Lattakia file, attempts to parse it, and if this succeeds executes the program.

## 5.3 Work Breakdown

## 5.3.1 Kat’s Work

For my old group I did a lot of the initial language design and I wrote the vast majority of the LRM. With this group I started off by doing the Java implementation of the lattice library, and I brought the problems with lattice arithmetic to Wael’s attention. When the team decided to pivot from making an interpreter to a translator I wrote the first draft of the semantic analysis and the print function. I also wrote the testing scripts and all of the test case and the majority of the final report and final presentation.

## 5.3.2 Heba’s Work

At the beginning of the project, I worked on testing the parser by tracing the output of each rule.(To make sure it's working and to give me a better understanding of the language). Then I worked on building the AST, code-generation and type checking in ocaml.(Basically calling the functions that Kat was working on in Java) However we then decided that we will not continue in this direction and will implement an ocaml interpreter instead so I worked on implementing this interpreter given the new lattice data-structure.  I also worked on writing the Language Reference Manual and the final report.

## 5.3.3 Wael’s Work

I worked on defining the language, writing and solving the grammar, writing backend Java code for the compiler, writing the new Lattice data structure for the interpreter, creating the lattice in the parser, writing code for semantic analysis and evaluation.

## 5.3.4 Li’s Work

I wrote an annotated grammar, in the format that is similar to the grammar of the C-language-manual. And I tested part of the java-code of our language, to find if the java code correctly captures the format of the basic element of our language-the lattice. And in the last half of our period, I took an independent task of writing a test editor which can highlight the tokens of our grammar, but haven’t got a working version on it yet.

# Chapter 6: Test Plan

## 6.1 Testing Overview

&> cd ./lattakia/test-cases/

&> make tests

For this project we put together a very primitive black-box unit testing suite similar to the one discussed in class. Each test looked solely at the interpreter output and attempted to pick apart a single application of the language. The testing suite consists of a body of source code file each with a “ground truth” text output file (e.g. foo.lat has a matching foo.txt file), and a testing shell script. The testing suite was integrated with our project Makefiles so that testing could be performed after each build. The testing shell script works by consuming all of the lattika source files in a directory one at a time and applying the files to the latte interpreter. If the interpreter fails to parse the file this result is sent to a log file and printed to the screen. If the interpreter does parse the file, the output is placed in a file with the “.out” extension. This “.out” file is then diffed with the ground truth text file. If there is no difference between the two files the test passes and this is noted on the screen and log file. If the test fails this is also noted on the screen and in the log file; in this case the diff file and out files are also kept for debugging purposes. Our test cases fall into three main categories: functional tests written directly from the LRM, ad-hoc tests written to specifically examine bugs, and integration or system testing intended to look at the function of the system as a whole. The testing suite can be run from the command line by following examples:

## 6.2 Testing Examples

let z = 1;

def foo(x,y) = 1;

def bar(x,y) = x\*y;

def fnord(x,y) = x\*y\*z;

let a = foo(1,2);

let b = bar(1,2);

let c = fnord(2,3);

print(a);

print(b);

print(c);

The code below gives and example of unit test 16 (TEST16Functions.lat) which tests the application of simple functions. This code segmented is provided to give and example of a representative unit test. The output of this code is diffed against the TEST16Functions.txt file which contains the following values.

1

2

6

Once the test is applied the following information is printed to the log file.

--------------------------------

Starting test

Base File Name: ./TEST16Functions

Truth File: ./TEST16Functions.txt

Output File: ./TEST16Functions.out

Diff File: ./TEST16Functions.diff

--->FAILED: latte terminated - DUMPING

## 6.3 Test Case Burn Down

The following section contains three tables that summarizes each of the tests found in our testing suite. Table 6.1 provides our unit tests used to validate our language reference manual. Table 6.2 gives a list of the ad-hoc unit tests we used to debug our code in process. Table 6.3 gives complex functional tests and example programs we created to demonstrate our language

**Table 6.1 Lattakia Functional Tests**

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Name** | **Description** | **Pass** | **Author** |
| TEST01BasicLat | Creation of basic alt and seq lats | Yes | KAS |
| TEST02Expressions | Complex mathematical expressions on ints and floats | NO | KAS |
| TEST03Concatenation | Concatenating lattices | NO | KAS |
| TEST04DepPred | Lattices with dependent predicates | NO | KAS |
| TEST05ForEach | For each looping constructs | NO | KAS |
| TEST06Floats | Floating point number lattice representations | NO | KAS |
| TEST07IndepPred | Parsing lattice predicates | NO | KAS |
| TEST08Indexing | Indexing into lattices | NO | KAS |
| TEST09Ints | Basic ints and int lattices | NO | KAS |
| TEST10BadID | Test basic identifiers | Yes | KAS |
| TEST11BadString | Catch malformed strings | Yes | KAS |
| TEST12Looping | More looping constructs | NO | KAS |
| TEST13Print | Test the print() method | Yes | KAS |
| TEST14Comment | Test parsing multi-line comments. | Yes | KAS |
| TEST15strings | Test parsing strings. | Yes | KAS |
| TEST16Functions | Test basic single parameter functions. | NO | KAS |
| TEST17Functions | Test multiple parameter functions | NO | KAS |
| TEST18Identifiers | Test parsing of valid identifiers. | Yes | KAS |
| TEST19Variables | Test parsing of int/float/string variables. | Yes | KAS |
| TEST20DepVars | Test basic variable expressions (e.g. x=y\*3) | NO | KAS |
| TEST21LatAdd | Basic Lattice arithmetic operations. | NO | KAS |
| TEST22Labels | Lattice Labels | NO | KAS |
| TEST23ParseError | Parse Error | YES | KAS |
| TEST24MultipleDef | Multiply defined variables | YES | KAS |
| TEST25PredicateTest | Basic predicates tests | YES | KAS |
| TEST26TypeMismatch | Comparison between two types | YES | KAS |
| TEST27Undef | Variable not defined | YES | KaS |
| TEST28Functions | Really really basic functions | YES | KAS |
| TEST29Identifiers | Basic Identifiers | YES | KAS |
| TEST30NestedLats | Test nested lattices | YES | KAS |
| TEST31Predicates | Test predicates | YES | KAS |
| TEST32WordOps | Test operations on strings | YES | KAS |

**Table 6.2 Lattakia Functional Tests**

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Name** | **Description** | **Pass** | **Author** |
| FTEST01-Heba | ???? | YES | Heba |
| FTEST02-GCD | Greatest common denominator of two digits | YES | Wael |
| FTEST03-FIB | Calculate the nth digit of the Fibonnaci sequence | YES | Wael |
| FTEST04-POS | Parts of speech tagger. | YES | Wael |
| FTEST05-Factorial | Calculate the factorial of n | YES | Kat |
| FTEST06-quicksort | Quicksort example | YES | Wael |

# Chapter 7: Lessons Learned

## 7.1 General Lessons Learned

The group consensus is that fundamentally our language was much too difficult for the scope of a semester project. It was difficult enough working in O’Caml to build a compiler and the complexities of the lattice structure further slowed our progress. Additionally, we tried to work independently on the project for too long and we would have made much more progress on the project working together to team from the outset.

## 7.2 Kat Lessons Learned

I think the general lessons I learned for this project were in team management. This project would have been manageable if the language was straight forward, but with such a complex language it was important to have everyone in a room working together. My initial gut reaction upon joining group was the project was not being managed correctly. For example, there was no source control, and it was extremely difficult to get everyone in a room to work together. I should have been more assertive about getting the project moving forward. Furthermore, not everyone was always clear about the language constructs. This made it difficult for me to move forward with doing semantic analysis or code generation. Aside from language difficulties, I could have been more assertive about getting a subset of the language working first and then expanding upon it. Our development process was such that semantic analysis and code generation were done concurrently and I would have liked to see evaluation completed before moving on to semantic analysis. I devoted a significant amount of time to generating a testing framework that never really got used because we didn’t have an easy way to remove the copious debugging information printed by the translator. I think if I could have gotten Wael to use the test cases we could have made more progress.

## 7.3 Heba Lessons Learned

It's always better to start with a simple kernel that captures the basic idea, get it working then work on adding more pieces to it because overwhleming yourself with many details will at some point make you lose the big picture. Define your language in such a way that you spend more time learning concepts needed to build a compiler rather than spend much time learning your own language. Make sure all team-members have the same target

## 7.4 Wael Lessons Learned

I should’ve learned more about functional language theory before defining the language and should’ve looked at different interesting functional languages. Now, I find functional languages more interesting especially OCaml and it’s type inference (because I faced some similar problems). I learned that I should always think a lot of what I’m doing before implementing and do always what make more sense not the easy way out, for example, defining a powerful lattice data structure in OCaml make my life easier when I advance to the next steps. I used SVN for the first time, and I gained knowledge and experience in many theoretical and technical issues.

## 7.5 Li Lessons Learned

The compiler that we build let me learn that write a language is not easy. And actually what we built likes a subversion of Ocaml, which is quite complicated. Before I haven’t done like thousands of programs like my teammates, I only did some in hardware, or in C or Java because I am an EE student. We have several meetings before I get the independent work, and we argue on it and discuss on it and made improvements. And an editor with the functionality to highlight the grammar looks simple but isn’t actually that simple because of the new language has a new interface that is not easy to implement.

# Appendix A: Source Code

## Ast.ml source code

*(\*\**

*\*\* Lattakia Compiler*

*\*\* Columbia University COMS-4115*

*\*\* Programming Languages and Translators*

*\*\* Authors:*

*\*\* Wael Salloum*

*\*\*)*

**module** **StringMap** = **Map**.**Make**(**String**);;

**type** operatorType =

**Plus** | **Minus** | **Times** | **Divide** | **Percent** | **Or** | **And** | **Equal** | **NotEqual** | **Less** | **LessOrEqual** | **Greater** | **GreaterOrEqual** | **Tilde**

| **UniMinus** | **Not** | **Eval** | **DontOpt** | **Let** | **DirectAccess** | **Assign** | **FuncCall** | **Define**

**type** latticeNatureType = **DataLat** *(\* No label or conditions \*)* | **Hashtable** *(\*at least on hash label \*)*

| **Object** *(\*at least on ID label \*)* | **Funcion** *(\*at least on condition \*)*

**type** structureTypeType = **Seq** | **Alt** | **Wrd** | **UndeterminedStruct**

**type** dataTypeType = **Int** | **Float** | **Boolean** | **String** | **General** *(\*for epsilon and nil\*)*

| **UnresolvedType** | **Diverse** | **MismatchType**

**and** dataStructTypeType = dataTypeType \* structureTypeType

**type** predefinedLabelType = **LENGTH** | **COUNT** | **LABELS** | **CLONE** | **REVERSE** | **RETURN** | **LOOP** | **GET\_LOST** | **INDEX** | **SUBJECT**

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

**type** latticeFields = {

**mutable** latDataStructType: dataStructTypeType;

**mutable** latNature: latticeNatureType; *(\*Not used yet\*)*

**mutable** latEval: exprValueType list option;

**mutable** lattice: latticeType; *(\* Seqlat | Altlat | Predicate | Expr \*)*

**mutable** paramNameList: string list option; *(\*used when created in parser\*)*

**mutable** paramTypeList: dataStructTypeType list option; *(\*used when processed in semantic analysis\*)*

**mutable** hasChanged: bool; *(\*Not used yet\*)* *(\*true if at least one dependee has changed\*)*

**mutable** optimize: bool;

}

**and** latticeType = **Seqlat** **of** seqlatFields

| **Altlat** **of** altlatFields

| **Predicate** **of** predicateFields

| **Expr** **of** exprFields

**and** predicateFields = {

**mutable** expr: exprFields;

**mutable** condition: exprFields;

}

**and** altlatFields = {

**mutable** alternatives: latticeFields list;

**mutable** each: bool;

}

**and** seqlatFields = {

**mutable** elements: latticeFields list;

**mutable** symbols: symbolTableType;

}

**and** symbolTableType = {

**mutable** parent : symbolTableType option;

**mutable** hash : symbolType **StringMap**.t;

}

**and** symbolType = {

**mutable** symName: string;

**mutable** symDataStructType: dataStructTypeType;

**mutable** symValue: symbolValueType;

**mutable** symbolScope: symbolScopeType;

**mutable** parameterNameList: string list option; *(\*used when created in parser\*)*

**mutable** parameterTypeList: dataStructTypeType list option; *(\*used when processed in semantic analysis\*)*

**mutable** dependers: exprFields ref list; *(\*Not used yet\*)*

**mutable** dependees: exprFields ref list; *(\*Not used yet\*)*

}

**and** symbolValueType = **LabelIndex** **of** int | **LocalValueLattice** **of** latticeFields ref

**and** symbolScopeType = **Label** | **Local** | **UnresolvedSymbol**

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

**and** exprFields = {

**mutable** exprDataStructType: dataStructTypeType;

**mutable** exprCode: exprCodeType; *(\*ConstExpr | LatticeExpr | OpExpr | NameListExpr \*)*

**mutable** exprEval: exprValueType; *(\*ConstValue | LatticeValue | NotEvaluated \*)*

**mutable** exprChanged: bool; *(\*Not used yet\*)* *(\*true if at least one dependee has changed\*)*

**mutable** expDependers: exprFields list; *(\*Not used yet\*)*

**mutable** expDependees: exprFields list; *(\*Not used yet\*)*

}

**and** exprCodeType = **ConstExpr** **of** constantType | **OpExpr** **of** operationType | **NameListExpr** **of** nameListType | **LatticeExpr** **of** latticeFields

**and** exprValueType = **ConstValue** **of** constantType | **LatticeValue** **of** latticeFields | **NotEvaluated**

**and** constantType = **StringLiteral** **of** string | **IntLiteral** **of** int | **FloatLiteral** **of** float | **Epsilon** | **Nil** | **True** | **False**

**and** operationType = {

**mutable** operator: operatorType;

**mutable** operand1: exprFields;

**mutable** operand2: exprFields option;

}

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

**and** nameListType = {

**mutable** nameList: nameType list;

}

**and** nameType = **This** | **PredefinedLabel** **of** predefinedLabelType

| **IndexingOp** **of** structureTypeType \* latticeType *(\*uses: Seq (for []), Alt (for {}), and UndeterminedStruct (for @)\*)*

| **HashOp** **of** exprFields | **Id** **of** string

| **Each** **of** latticeType

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

**and** errorCategoryType = **Worning** | **Semantic** | **Syntactic** | **Lexical** | **Runtime**

**and** errorMessageType =

**SemParamListTypeSignMismatch**

| **SemUndefinedIdentifier** **of** string

| **SemIdentifierMultipleDefinition**

| **SemTypeMismatch**

| **SemPredicateTypeNotBoolean**

| **LexUndefinedSymbol** **of** string

| **SynParseError**

**and** errorType = {

**mutable** errMessage: errorMessageType;

**mutable** errPlace: exprCodeType option;

**mutable** lineNum: int;

**mutable** offset: int;

}

**and** envType = {

**mutable** labelIdGenerator: int;

**mutable** parentStackLabelIdGenerator: int list;

**mutable** errorTable: errorType list;

**mutable** worningTable: errorType list;

**mutable** namesOfCurrLat: symbolTableType; *(\*The SymbolTable of the current lattice \*)*

**mutable** poppedSymbolTable: symbolTableType;

**mutable** lineNumber: int; *(\*cursor position in source code\*)*

**mutable** charOffset: int; *(\*cursor position in source code\*)*

}

**let** env = {

labelIdGenerator = -1;

parentStackLabelIdGenerator = [];

errorTable = [];

worningTable = [];

namesOfCurrLat = {parent = **None**; hash = **StringMap**.empty};

poppedSymbolTable = {parent = **None**; hash = **StringMap**.empty};

lineNumber = 0;

charOffset = 0;

*(\*pos = {pos\_fname=""; pos\_lnum=0; pos\_bol=0; pos\_cnum=0}\*)*

};;

**let** **rec** symbolTableSearchFor (scope : symbolTableType) name =

**try**

**StringMap**.find name scope.hash

**with** **Not\_found** ->

**match** scope.parent **with**

**Some**(parent) -> symbolTableSearchFor parent name

| \_ -> raise **Not\_found**

**let** **rec** symbolTableSearchForLabel (scope : symbolTableType) name =

**let** sym = **StringMap**.find name scope.hash **in**

**match** sym.symbolScope **with**

| **Label** -> sym

| \_ -> raise **Not\_found**

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

**type** lattice = (latticeFields list) \* envType

## Clone.ml Source Code

*(\*\**

*\*\* Lattakia Compiler*

*\*\* Columbia University COMS-4115*

*\*\* Programming Languages and Translators*

*\*\* Authors:*

*\*\* Wael Salloum*

*\*\*)*

*(\* Clone Lattice \*)*

**open** **Ast**

**open** **LatticeCreation**

**open** **PrintLattice**

*(\*------------------------------------------------------------------------------------------------\*)*

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

**type** cloneEnv = {

**mutable** currentLat: latticeFields;

**mutable** parentStack: latticeFields list;

}

**let** clnEnv = {currentLat = **LatticeCreation**.createEmptyLattice []; parentStack = []}

**let** **rec** cloneLattice \_lattice \_doCloneCode = *(\*return an error string -if it is empty we are good\*)*

processLattice \_lattice \_doCloneCode

**and** processLattice (\_lattice: latticeFields) \_doCloneCode = *(\*input a latticeFields, which is our AST\*)*

{

latDataStructType = \_lattice.latDataStructType;

latNature = \_lattice.latNature;

latEval = \_lattice.latEval;

paramNameList = \_lattice.paramNameList;

paramTypeList = \_lattice.paramTypeList;

hasChanged = \_lattice.hasChanged;

optimize = \_lattice.optimize;

lattice = **if** \_doCloneCode = false **then** \_lattice.lattice

**else** (**match** \_lattice.lattice **with**

| **Seqlat**(seqlat) -> (

clnEnv.parentStack <- clnEnv.currentLat :: clnEnv.parentStack *(\*PUSH\*)*;

clnEnv.currentLat <- \_lattice;

**let** seqlatClone = **Seqlat**(processSeqlat seqlat \_doCloneCode) **in**

clnEnv.currentLat <- **List**.hd clnEnv.parentStack *(\*POP\*)*;

clnEnv.parentStack <- **List**.tl clnEnv.parentStack *(\*Remove\*)*;

seqlatClone

)

| **Altlat**(altlat) -> **Altlat**(processAltlat altlat \_doCloneCode)

| **Predicate**(pred) -> **Predicate**(processPredicate pred \_doCloneCode)

| **Expr**(expr) -> **Expr**(processExpr expr \_doCloneCode)

)

}

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

**and** processSeqlat \_seqlat \_doCloneCode = *(\*do the seq lat \*)*

*(\*List.map (processLattice) \_seqlat.elements\*)*

rf(**"Not Implemented."**)

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

**and** processAltlat \_altlat \_doCloneCode = *(\*do the alt lat \*)*

*(\*List.map (processLattice) \_altlat.alternatives\*)*

rf(**"Not Implemented."**)

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

*(\* go through a lattice list and make sure all of the types are the same \*)*

**and** getLatticeListType(\_latticeList, (\_prevType: dataStructTypeType), (\_mismatchType: dataTypeType)) \_doCloneCode=

*(\*match \_latticeList with*

*| [] -> \_prevType*

*| hd::tl -> let (hdType: dataStructTypeType) = processLattice hd in*

*if ((fst \_prevType) = (fst hdType)) && ((snd \_prevType) = (snd hdType)) then (getLatticeListType(tl, hdType, \_mismatchType))*

*else (\_mismatchType, UndeterminedStruct)\*)*

rf(**"Not Implemented."**)

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

**and** processPredicate \_pred \_doCloneCode =

*(\* Use the symbol table to compare predicate types - i.e no int < string \*)*

*(\*let cond = (processExpr \_pred.condition) in*

*let pred = processExpr \_pred.expr in*

*match cond with*

*| (Boolean, Wrd) -> pred*

*| (\_, \_) -> addToErrorTable(SemPredicateTypeNotBoolean, None); (MismatchType, UndeterminedStruct)\*)*

rf(**"Not Implemented."**)

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

*(\* Do a expression return a type\*)*

**and** processExpr (\_expr: exprFields) \_doCloneCode =

{

exprDataStructType = \_expr.exprDataStructType;

exprCode = **if** \_doCloneCode = false **then** \_expr.exprCode **else** rf(**"Not Implemented."**)

*(\*match \_expr.exprCode with*

*| ConstExpr(const) -> processConst const*

*| OpExpr(op) -> processOp op (\*lookup from symbol table\*)*

| **NameListExpr**(nameList) -> processNameList(clnEnv.currentLat, nameList.nameList)

| **LatticeExpr**(lattice) -> processLattice lattice \_doCloneCode

\*)

; *(\*ConstExpr | LatticeExpr | OpExpr | NameListExpr \*)*

exprEval = cloneExprEval \_expr.exprEval; *(\*ConstExpr | LatticeExpr \*)*

exprChanged = \_expr.exprChanged; *(\*a constant cannot change\*)*

expDependers = \_expr.expDependers;

expDependees = \_expr.expDependees;

}

**and** cloneExprEval \_exprEval =

**match** \_exprEval **with**

| **ConstValue**(x) -> **ConstValue**(x)

| **LatticeValue**(lat) -> **LatticeValue**(processLattice lat true)

| **NotEvaluated** -> **NotEvaluated**

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

**and** isWordOperator = **function**

| **Plus** -> true | **Minus** -> true | **Times** -> true | **Divide** -> true | **Percent** -> true | **Or** -> true | **And** -> true | **Less** -> true

| **UniMinus** -> true | **Not** -> true | **LessOrEqual** -> true | **Greater** -> true | **GreaterOrEqual** -> true

| \_ -> false

*(\*Tilde | Eval | DontOpt | Let | DirectAccess | Assign | FuncCall | Define | Equal | NotEqual\*)*

**and** doCheckOperandsBeforeCheckingOperator = **function**

| **Plus** -> true | **Minus** -> true | **Times** -> true | **Divide** -> true | **Percent** -> true | **Or** -> true | **And** -> true | **Less** -> true

| **UniMinus** -> true | **Not** -> true | **LessOrEqual** -> true | **Greater** -> true | **GreaterOrEqual** -> true

| **Tilde** -> true | **Eval** -> true | **DontOpt** -> true | **Let** -> true | **DirectAccess** -> false | **Assign** -> true

| **FuncCall** -> true | **Define** -> false | **Equal** -> true | **NotEqual** -> true

*(\* given an operation expression return the type it \_should\_ return \*)*

**and** processOp op =

rf(**"Not Implemented."**)

*(\* let op1type = if doCheckOperandsBeforeCheckingOperator op.operator then processExpr op.operand1 else (General, Wrd) in*

*let op2type = if doCheckOperandsBeforeCheckingOperator op.operator then*

*(match op.operand2 with*

*| None -> (General, Wrd) (\*Apply to all: most general case\*)*

| **Some**(opr) -> (processExpr opr)) **else** (**General**, **Wrd**) **in**

**if** (isWordOperator op.operator)

&& (**match** (snd op1type, snd op2type) **with**

| (**Wrd**, **Wrd**) -> false *(\*No problem\*)*

| (\_,\_) -> true)

**then** (**MismatchType**, **UndeterminedStruct**)

**else**

(**match** op.operator **with**

*(\* Arithmatic Operations \*)*

| **Plus** -> (

**match** (fst op1type, fst op2type) **with**

| (**Int**, **Int**) -> (**Int**, **Wrd**)

| (**Float**, **Float**) -> (**Float**, **Wrd**)

| (**String**, **String**) -> (**String**, **Wrd**)

| (**General**, t) -> (t, **Wrd**)

| (t, **General**) -> (t, **Wrd**)

| (\_,\_) -> (**MismatchType**, **Wrd**)

)

| **Minus** -> (

**match** (fst op1type, fst op2type) **with**

| (**Int**, **Int**) -> (**Int**, **Wrd**)

| (**Float**, **Float**) -> (**Float**, **Wrd**)

| (**General**, t) -> (t, **Wrd**)

| (t, **General**) -> (t, **Wrd**)

| (\_,\_) -> (**MismatchType**, **Wrd**)

)

| **Times** -> (

**match** (fst op1type, fst op2type) **with**

| (**Int**, **Int**) -> (**Int**, **Wrd**)

| (**Float**, **Float**) -> (**Float**, **Wrd**)

| (**General**, t) -> (t, **Wrd**)

| (t, **General**) -> (t, **Wrd**)

| (\_,\_) -> (**MismatchType**, **Wrd**)

)

| **Divide** -> (

**match** (fst op1type, fst op2type) **with**

| (**Int**, **Int**) -> (**Int**, **Wrd**)

| (**Float**, **Float**) -> (**Float**, **Wrd**)

| (**General**, t) -> (t, **Wrd**)

| (t, **General**) -> (t, **Wrd**)

| (\_,\_) -> (**MismatchType**, **Wrd**)

)

| **Percent** -> (

**match** (fst op1type, fst op2type) **with**

| (**Int**, **Int**) -> (**Int**, **Wrd**)

| (**Float**, **Float**) -> (**Float**, **Wrd**)

| (**General**, t) -> (t, **Wrd**)

| (t, **General**) -> (t, **Wrd**)

| (\_,\_) -> (**MismatchType**, **Wrd**)

)

| **UniMinus** -> (

**match** (fst op1type) **with**

| (**Int**) -> (**Int**, **Wrd**)

| (**Float**) -> (**Float**, **Wrd**)

| (**General**) -> (**General**, **Wrd**)

| (\_) -> (**MismatchType**, **Wrd**)

)

*(\* Logical Operations \*)*

| **Or** -> (

**match** (fst op1type, fst op2type) **with**

| (**Boolean**, **Boolean**) -> (**Boolean**, **Wrd**)

| (**General**, t) -> (t, **Wrd**)

| (t, **General**) -> (t, **Wrd**)

| (\_,\_) -> (**MismatchType**, **Wrd**)

)

| **And** -> (

**match** (fst op1type, fst op2type) **with**

| (**Boolean**, **Boolean**) -> (**Boolean**, **Wrd**)

| (**General**, t) -> (t, **Wrd**)

| (t, **General**) -> (t, **Wrd**)

| (\_,\_) -> (**MismatchType**, **Wrd**)

)

| **Not** -> (

**match** (fst op1type) **with**

| (**Boolean**) -> (**Boolean**, **Wrd**)

| (**General**) -> (**General**, **Wrd**)

| (\_) -> (**MismatchType**, **Wrd**)

)

*(\* Comparison Operations \*)*

| **Equal** -> (

**match** (fst op1type, fst op2type) **with**

| (**Int**, **Int**) -> (**Boolean**, **Wrd**)

| (**Float**, **Float**) -> (**Boolean**, **Wrd**)

| (**String**, **String**) -> (**Boolean**, **Wrd**)

| (**General**, t) -> (**Boolean**, **Wrd**)

| (t, **General**) -> (**Boolean**, **Wrd**)

| (\_,\_) -> (**MismatchType**, **Wrd**)

)

| **NotEqual** -> (

**match** (fst op1type, fst op2type) **with**

| (**Int**, **Int**) -> (**Boolean**, **Wrd**)

| (**Float**, **Float**) -> (**Boolean**, **Wrd**)

| (**String**, **String**) -> (**Boolean**, **Wrd**)

| (**General**, t) -> (**Boolean**, **Wrd**)

| (t, **General**) -> (**Boolean**, **Wrd**)

| (\_,\_) -> (**MismatchType**, **Wrd**)

)

| **Less** -> (

**match** (fst op1type, fst op2type) **with**

| (**Int**, **Int**) -> (**Boolean**, **Wrd**)

| (**Float**, **Float**) -> (**Boolean**, **Wrd**)

| (**String**, **String**) -> (**Boolean**, **Wrd**)

| (**General**, t) -> (**Boolean**, **Wrd**)

| (t, **General**) -> (**Boolean**, **Wrd**)

| (\_,\_) -> (**MismatchType**, **Wrd**)

)

| **LessOrEqual** -> (

**match** (fst op1type, fst op2type) **with**

| (**Int**, **Int**) -> (**Boolean**, **Wrd**)

| (**Float**, **Float**) -> (**Boolean**, **Wrd**)

| (**String**, **String**) -> (**Boolean**, **Wrd**)

| (**General**, t) -> (**Boolean**, **Wrd**)

| (t, **General**) -> (**Boolean**, **Wrd**)

| (\_,\_) -> (**MismatchType**, **Wrd**)

)

| **Greater** -> (

**match** (fst op1type, fst op2type) **with**

| (**Int**, **Int**) -> (**Boolean**, **Wrd**)

| (**Float**, **Float**) -> (**Boolean**, **Wrd**)

| (**String**, **String**) -> (**Boolean**, **Wrd**)

| (**General**, t) -> (**Boolean**, **Wrd**)

| (t, **General**) -> (**Boolean**, **Wrd**)

| (\_,\_) -> (**MismatchType**, **Wrd**)

)

| **GreaterOrEqual** -> (

**match** (fst op1type, fst op2type) **with**

| (**Int**, **Int**) -> (**Boolean**, **Wrd**)

| (**Float**, **Float**) -> (**Boolean**, **Wrd**)

| (**String**, **String**) -> (**Boolean**, **Wrd**)

| (**General**, t) -> (**Boolean**, **Wrd**)

| (t, **General**) -> (**Boolean**, **Wrd**)

| (\_,\_) -> (**MismatchType**, **Wrd**)

)

*(\* Misc Operations: Tilde | Eval | DontOpt | Let | DirectAccess | Assign | FuncCall | Define | Equal | NotEqual\*)*

| **Tilde** ->

**let** structType = (**match** (snd op1type, snd op2type) **with**

| (**Wrd**, **Wrd**) -> **Seq**

| (**Wrd**, x) -> x

| (**Seq**, \_) -> **Seq**

| (**Alt**, \_) -> **Alt**

| (**UndeterminedStruct**, \_) -> **UndeterminedStruct**

) **in**

**let** dataType = (**match** (fst op1type, fst op2type) **with**

| (**Int**, **Int**) -> **Int**

| (**Float**, **Float**) -> **Float**

| (**String**, **String**) -> **String**

| (**Boolean**, **Boolean**) -> **Boolean**

| (**General**, t) -> t

| (t, **General**) -> t

| (**UnresolvedType**, \_) -> **UnresolvedType**

| (\_, **UnresolvedType**) -> **UnresolvedType**

| (**MismatchType**, \_) -> **MismatchType**

| (\_, **MismatchType**) -> **MismatchType**

| (\_,\_) -> **Diverse**

) **in**

(**match** (dataType, structType) **with**

| (**Diverse**, **Alt**) -> (**MismatchType**, **Alt**)

| (\_, \_) -> (dataType, structType))

| **Eval** -> (

**match** (fst op1type, snd op1type) **with**

| (x, **Wrd**) -> (x, **Wrd**)

| (x, **Seq**) -> (x, **UndeterminedStruct**)

| (x, **Alt**) -> (x, **UndeterminedStruct**)

| (\_,\_) -> (**MismatchType**, **Wrd**)

)

| **DontOpt** -> op1type

| **Let** -> (**General**, **Wrd**)

| **DirectAccess** ->

**let** nameList = (**match** (gs op.operand2 **""**).exprCode **with** **NameListExpr**(nameListEx) -> nameListEx.nameList | \_ -> rf(**"BUG: SemanticAnalysis.processOp(): DirectAccesss: nameList!"**)) **in**

**let** lattice = (**match** op.operand1.exprCode **with** **LatticeExpr**(lat) -> lat | \_ -> rf(**"BUG: SemanticAnalysis.processOp(): DirectAccesss: lattice!"**)) **in**

processNameList(lattice, nameList)

| **FuncCall** ->

**let** funcNameList = (**match** op.operand1.exprCode **with** **NameListExpr**(nameListEx) -> nameListEx.nameList | \_ -> rf(**"BUG: SemanticAnalysis.processOp(): FuncCall: funcLattice!"**)) **in**

**let** (idSymOpt, idLatOpt) = **try** **LatticeCreation**.findNameListSymbolInLatticeFields(clnEnv.currentLat, funcNameList)

**with** **Not\_found** -> addToErrorTable(**SemUndefinedIdentifier**(**PrintLattice**.processNameList {parent = **None**; hash = **StringMap**.empty} funcNameList), **None**); (**None**, **None**)

**in**

(**match** idLatOpt **with**

| **None** -> (**UnresolvedType**, **UndeterminedStruct**)

| **Some**(funcLattice) -> (

**let** expr = (gs op.operand2 **""**) **in**

**let** lattice = (**match** expr.exprCode **with** **LatticeExpr**(lat) -> lat | \_ -> createEmptyLattice [createLatticeFieldsFromLatticeType (**Expr**(expr))]) **in**

**if** funcLattice.paramNameList = **None** **then** (**MismatchType**, **UndeterminedStruct**)

**else** **if** funcLattice.paramTypeList = **None** **then** (

**if** calcParamTypeList(lattice, funcLattice) **then**

processLattice(funcLattice)

**else**

(**MismatchType**, **UndeterminedStruct**)

)

**else**

**if** checkParameterTypes(funcLattice.paramTypeList, lattice) **then**

funcLattice.latDataStructType

**else**

(**MismatchType**, **UndeterminedStruct**)

))

| **Assign** ->

**let** nameList = (**match** op.operand1.exprCode **with** **NameListExpr**(n) -> n.nameList | \_ -> rf(**"BUG: SymanticAnalysis: Assign: NameList-2!"**)) **in**

**let** (idSymOpt, idLatOpt) = **try** **LatticeCreation**.findNameListSymbolInLatticeFields(clnEnv.currentLat, nameList)

**with** **Not\_found** -> addToErrorTable(**SemUndefinedIdentifier**(**PrintLattice**.processNameList {parent = **None**; hash = **StringMap**.empty} nameList), **None**); (**None**, **None**)

**in**

**let** idSymbolType = (**match** idSymOpt **with**

| **None** -> (**UnresolvedType**, **UndeterminedStruct**)

| **Some**(idSymbol) -> idSymbol.symDataStructType) **in**

**let** rvalueType =

(**match** (fst idSymbolType, fst op2type) **with**

| (**General**, t) -> (t, snd op2type)

| (t, **General**) -> (t, snd op2type)

| (x, y) -> **if** x = y **then** (x, snd op2type) **else** (**MismatchType**, snd op2type)

) **in** rvalueType

| **Define** ->

**let** idSymbolOpt = **LatticeCreation**.findInSymbolTable((**LatticeCreation**.getSymbolTableFromSeqlat clnEnv.currentLat),

(**match** op.operand1.exprCode **with** **NameListExpr**(n) ->

(**match** **List**.hd n.nameList **with** **Id**(x) -> x | \_ -> rf(**"BUG: SymanticAnalysis: Define: NameList-1!"**))

| \_ -> rf(**"BUG: SymanticAnalysis: Define: NameList-2!"**))) **in**

**let** rvalueType = processExpr (gs op.operand2 **""**) **in**

**match** idSymbolOpt **with**

| **None** -> rf(**"BUG: SymanticAnalysis: Define: NameList-3!"**)

| **Some**(idSymbol) -> (idSymbol.symDataStructType <- rvalueType; rvalueType)

)\*)

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

**and** processConst = **function** *(\*return the type so we can do type checking\*)*

| **StringLiteral**(\_) -> (**String**, **Wrd**)

| **IntLiteral**(\_) -> (**Int**, **Wrd**)

| **FloatLiteral**(\_) -> (**Float**, **Wrd**)

| **Epsilon** -> (**General**, **Wrd**)

| **Nil** -> (**General**, **Wrd**)

| **True** -> (**Boolean**, **Wrd**)

| **False** -> (**Boolean**, **Wrd**)

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

*(\* make sure that x.y.z is in the right symbol table scope\*)*

**and** processNameList(\_lattice, \_nameList) =

*(\* let (idSymOpt, idLatOpt) = try LatticeCreation.findNameListSymbolInLatticeFields(\_lattice, \_nameList)*

*with Not\_found -> addToErrorTable(SemUndefinedIdentifier(PrintLattice.processNameList {parent = None; hash = StringMap.empty} \_nameList), None); (None, None)*

*in*

*let idSymbolType = (match idSymOpt with*

*| None -> (UnresolvedType, UndeterminedStruct)*

*| Some(idSymbol) -> idSymbol.symDataStructType*

*) in idSymbolType*

*\*)*

rf(**"Not Implemented."**)

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

**and** checkParameterTypes(\_paramTypeList, \_paramValueLattice) =

*(\*let elements = getSeqlatElementsFromLatticeFields \_paramValueLattice in*

*List.fold\_left2 (fun a b c -> a && (b == c.latDataStructType)) true (gs \_paramTypeList "BUG: checkParameterTypes!") elements\*)*

rf(**"Not Implemented."**)

**and** calcParamTypeList(\_paramLattice, \_funcLattice) =

*(\*let elements = getSeqlatElementsFromLatticeFields \_paramLattice in*

*if (List.length elements) <> (List.length (gs \_funcLattice.paramNameList "BUG: calcParamTypeList")) then false*

*else let paramTypeList = List.map (fun x -> x.latDataStructType) elements in*

*\_funcLattice.paramTypeList <- Some(paramTypeList); true\*)*

rf(**"Not Implemented."**)

## evaluate.ml Source Code

**open** **Ast**

**open** **LatticeCreation**

**open** **PrintLattice**

**open** **SemanticAnalysis**

**open** **Clone**

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

**type** comparison = **Eq** | **Lt** | **Gt** | **NilEncountered**

**type** evalEnvType = {

**mutable** parentStack: latticeFields list;

**mutable** currentLat: latticeFields;

}

**let** evalEnv = {

parentStack = [];

currentLat = **LatticeCreation**.createEmptyLattice [];

}

**let** preEvalEnv = {

parentStack = [];

currentLat = **LatticeCreation**.createEmptyLattice [];

}

**let** isValueEpsilon = **function**

| **ConstValue**(**Epsilon**) -> true

| \_ -> false

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

**let** isBooleanValue = **function** **ConstValue**(**True**) -> true | **ConstValue**(**False**) -> true | **ConstValue**(**Nil**) -> true | **ConstValue**(**Epsilon**) -> true | \_ -> false

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

*(\*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=\*)*

**let** **rec** eval lattice =

evalEnv.currentLat <- lattice;

processLattice lattice

**and** processLattice lattice = *(\*input a latticeFields, which is our AST\*)*

**let** listOfEval = (**match** lattice.lattice **with**

| **Seqlat**(seqlat) ->

evalEnv.parentStack <- evalEnv.currentLat :: evalEnv.parentStack *(\*PUSH\*)*;

evalEnv.currentLat <- lattice;

**let** listOfEval = processSeqlat seqlat **in**

(**if** (**List**.length evalEnv.parentStack) > 0 **then**

evalEnv.currentLat <- **List**.hd evalEnv.parentStack *(\*POP\*)*

**else** rf(**"BUG: Stack is not being built Correctly!!!\n"**);

evalEnv.parentStack <- **List**.tl evalEnv.parentStack; *(\*Remove\*)*

listOfEval)

| **Altlat**(altlat) ->

**let** listOfEval = processAltlat altlat **in** listOfEval

| **Predicate**(pred) -> [processPredicate pred]

| **Expr**(expr) -> [processExpr expr]

) **in**

**let** optListOfEval = **if** lattice.optimize **then**

**List**.fold\_left (**fun** x y -> x @ (**if** isValueEpsilon(y) **then** [] **else** [y])) [] listOfEval

**else**

listOfEval

**in** (

**let** latClone = **Clone**.cloneLattice lattice false **in**

latClone.latEval <- **Some**(optListOfEval);

**if** (**List**.length optListOfEval)>1 **then** **LatticeValue**(latClone) **else** **if** (**List**.length optListOfEval)>0 **then** (**List**.hd optListOfEval)

**else** **ConstValue**(**Epsilon**)

)

*(\*--------------------------------------------------------------------\*)*

**and** processSeqlat \_seqlat = *(\*do the seq lat \*)*

**if** (**List**.length \_seqlat.elements) > 0 **then**

**List**.map (processLattice) \_seqlat.elements

**else** rf (**"BUG: Seqlat has no elements!"**)

*(\*--------------------------------------------------------------------\*)*

**and** processAlternatives = **function**

| [] -> []

| hd::tl ->

**let** headEval = (processLattice hd) **in**

*(\*match headEval.lattice with*

*| Expr(expr) ->\*)* (**match** headEval **with**

| **ConstValue**(**Nil**) -> (processAlternatives tl)

| \_ -> [headEval]

)

*(\*| \_ -> [headEval]*

*\*)*

*(\*--------------------------------------------------------------------\*)*

**and** processAltlat \_altlat = *(\*do the alt lat \*)*

**if** \_altlat.each **then**

**List**.map processLattice \_altlat.alternatives

**else**

processAlternatives \_altlat.alternatives

*(\*--------------------------------------------------------------------\*)*

**and** processPredicate \_pred =

**let** condExpr = processExpr \_pred.condition **in**

\_pred.condition.exprEval <- condExpr;

**let** condVal = (**match** condExpr **with**

| **ConstValue**(constVal) -> ( **match** constVal **with**

| **True** -> true

| **False** -> false

| \_ -> false

)

| \_ -> false *(\*Lattice\*)*

) **in**

**if** condVal = true **then**

processExpr \_pred.expr

**else**

**ConstValue**(**Nil**)

*(\*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=\*)*

**and** processExpr \_expr =

\_expr.exprEval <-

(**match** \_expr.exprCode **with**

| **ConstExpr**(const) -> processConst const

| **OpExpr**(op) -> processOperation op

| **NameListExpr**(nameList) -> processNameList nameList.nameList

| **LatticeExpr**(latticeFields) -> processLattice latticeFields

);

\_expr.exprEval

*(\*--------------------------------------------------------------------\*)*

**and** processConst const =

**match** const **with**

| **StringLiteral**(stringLit) -> **ConstValue**(**StringLiteral**(stringLit))

| **IntLiteral**(intLit) -> **ConstValue**(**IntLiteral**(intLit))

| **FloatLiteral**(floatLit) -> **ConstValue**(**FloatLiteral**(floatLit))

| **Epsilon** -> **ConstValue**(**Epsilon**)

| **Nil** -> **ConstValue**(**Nil**)

| **True** -> **ConstValue**(**True**)

| **False** -> **ConstValue**(**False**)

*(\*--------------------------------------------------------------------\*)*

**and** processOperation op =

**let** op1Val = **if** doCheckOperandsBeforeCheckingOperator op.operator **then** processExpr op.operand1 **else** **ConstValue**(**Nil**) **in**

**let** op2Val = **if** doCheckOperandsBeforeCheckingOperator op.operator **then** (

**match** op.operand2 **with**

| **None** -> **ConstValue**(**Nil**)

| **Some**(opr) -> processExpr opr

) **else** **ConstValue**(**Nil**) **in**

(**match** op.operator **with**

| **Plus** -> (**match** (op1Val, op2Val) **with**

| (**ConstValue**(**IntLiteral**(intLit1)),**ConstValue**(**IntLiteral**(intLit2))) -> **ConstValue**(**IntLiteral**(intLit1+intLit2))

| (**ConstValue**(**StringLiteral**(strLit1)),**ConstValue**(**StringLiteral**(strLit2))) -> **ConstValue**(**StringLiteral**(strLit1^strLit2))

| (**ConstValue**(**FloatLiteral**(floatLit1)),**ConstValue**(**FloatLiteral**(floatLit2))) -> **ConstValue**(**FloatLiteral**(floatLit1 +. floatLit2))

| (**ConstValue**(**Nil**),x) -> **ConstValue**(**Nil**)

| (x,**ConstValue**(**Nil**)) -> **ConstValue**(**Nil**)

| (**ConstValue**(**Epsilon**),x) -> x

| (x,**ConstValue**(**Epsilon**)) -> x

| \_ -> raise (**Failure**(**"Runtime Exception: Type Mismatch.\n"**)) )

*(\*.......................................................................................\*)*

| **Minus** -> (**match** (op1Val, op2Val) **with**

| (**ConstValue**(**IntLiteral**(intLit1)),**ConstValue**(**IntLiteral**(intLit2))) -> **ConstValue**(**IntLiteral**(intLit1-intLit2))

| (**ConstValue**(**FloatLiteral**(floatLit1)),**ConstValue**(**FloatLiteral**(floatLit2))) -> **ConstValue**(**FloatLiteral**(floatLit1 -. floatLit2))

| (**ConstValue**(**Nil**),x) -> **ConstValue**(**Nil**)

| (x,**ConstValue**(**Nil**)) -> **ConstValue**(**Nil**)

| (**ConstValue**(**Epsilon**),x) -> x

| (x,**ConstValue**(**Epsilon**)) -> x

| \_ ->raise (**Failure**(**"Runtime Exception: Type Mismatch.\n"**)) )

*(\*.......................................................................................\*)*

| **Times** -> (**match** (op1Val, op2Val) **with**

| (**ConstValue**(**IntLiteral**(intLit1)),**ConstValue**(**IntLiteral**(intLit2))) -> **ConstValue**(**IntLiteral**(intLit1\*intLit2))

| (**ConstValue**(**FloatLiteral**(floatLit1)),**ConstValue**(**FloatLiteral**(floatLit2))) -> **ConstValue**(**FloatLiteral**(floatLit1 \*. floatLit2))

| (**ConstValue**(**Nil**),x) -> **ConstValue**(**Nil**)

| (x,**ConstValue**(**Nil**)) -> **ConstValue**(**Nil**)

| (**ConstValue**(**Epsilon**),x) -> x

| (x,**ConstValue**(**Epsilon**)) -> x

| \_ ->raise (**Failure**(**"Runtime Exception: Type Mismatch.\n"**)) )

*(\*.......................................................................................\*)*

| **Divide** -> (**match** (op1Val, op2Val) **with**

| (**ConstValue**(**IntLiteral**(intLit1)),**ConstValue**(**IntLiteral**(intLit2))) -> **if** intLit2 = 0 **then** raise (**Division\_by\_zero**) **else** **ConstValue**(**IntLiteral**(intLit1/intLit2))

| (**ConstValue**(**FloatLiteral**(floatLit1)),**ConstValue**(**FloatLiteral**(floatLit2))) -> **if** floatLit2 = 0.0 **then** raise (**Division\_by\_zero**) **else** **ConstValue**(**FloatLiteral**(floatLit1 /. floatLit2))

| (**ConstValue**(**Nil**),x) -> **ConstValue**(**Nil**)

| (x,**ConstValue**(**Nil**)) -> **ConstValue**(**Nil**)

| (**ConstValue**(**Epsilon**),x) -> x

| (x,**ConstValue**(**Epsilon**)) -> x

| \_ ->raise (**Failure**(**"Runtime Exception:Type Mismatch.\n"**)) )

*(\*.......................................................................................\*)*

| **Percent** -> (**match** (op1Val, op2Val) **with**

| (**ConstValue**(**IntLiteral**(intLit1)),**ConstValue**(**IntLiteral**(intLit2))) -> **ConstValue**(**IntLiteral**(intLit1 mod intLit2))

| (**ConstValue**(**Nil**),x) -> **ConstValue**(**Nil**)

| (x,**ConstValue**(**Nil**)) -> **ConstValue**(**Nil**)

| (**ConstValue**(**Epsilon**),x) -> x

| (x,**ConstValue**(**Epsilon**)) -> x

| \_ ->raise (**Failure**(**"Runtime Exception: Type Mismatch.\n"**)) )

*(\*.......................................................................................\*)*

| **UniMinus** -> (**match** (op1Val) **with**

| (**ConstValue**(**IntLiteral**(intLit1))) -> **ConstValue**(**IntLiteral**(-intLit1))

| (**ConstValue**(**FloatLiteral**(floatLit1))) -> **ConstValue**(**FloatLiteral**(0.0 -. floatLit1))

| (**ConstValue**(**Nil**)) -> **ConstValue**(**Nil**)

| (**ConstValue**(**Epsilon**)) -> **ConstValue**(**Epsilon**)

| \_ ->raise (**Failure**(**"Runtime Exception: Type Mismatch in '-' (unary minus) operand!\n"**)) )

*(\*.......................................................................................\*)*

| **Less** -> (**match** compareExprValues(op1Val, op2Val) **with**

| **NilEncountered** -> **ConstValue**(**Nil**)

| **Gt** -> **ConstValue**(**False**)

| **Eq** -> **ConstValue**(**False**)

| **Lt** -> **ConstValue**(**True**)

)

*(\*.......................................................................................\*)*

| **LessOrEqual** -> (**match** compareExprValues(op1Val, op2Val) **with**

| **NilEncountered** -> **ConstValue**(**Nil**)

| **Gt** -> **ConstValue**(**False**)

| **Eq** -> **ConstValue**(**True**)

| **Lt** -> **ConstValue**(**True**)

)

*(\*.......................................................................................\*)*

| **Greater** -> (**match** compareExprValues(op1Val, op2Val) **with**

| **NilEncountered** -> **ConstValue**(**Nil**)

| **Gt** -> **ConstValue**(**True**)

| **Eq** -> **ConstValue**(**False**)

| **Lt** -> **ConstValue**(**False**)

)

*(\*.......................................................................................\*)*

| **GreaterOrEqual** -> (**match** compareExprValues(op1Val, op2Val) **with**

| **NilEncountered** -> **ConstValue**(**Nil**)

| **Gt** -> **ConstValue**(**True**)

| **Eq** -> **ConstValue**(**True**)

| **Lt** -> **ConstValue**(**False**)

)

*(\*.......................................................................................\*)*

| **Equal** -> (**match** compareExprValues(op1Val, op2Val) **with**

| **NilEncountered** -> **ConstValue**(**Nil**)

| **Gt** -> **ConstValue**(**False**)

| **Eq** -> **ConstValue**(**True**)

| **Lt** -> **ConstValue**(**False**)

)

*(\*.......................................................................................\*)*

| **NotEqual** -> (**match** compareExprValues(op1Val, op2Val) **with**

| **NilEncountered** -> **ConstValue**(**Nil**)

| **Gt** -> **ConstValue**(**True**)

| **Eq** -> **ConstValue**(**False**)

| **Lt** -> **ConstValue**(**True**)

)

*(\*.......................................................................................\*)*

| **And** -> **if** isBooleanValue(op1Val) && isBooleanValue(op2Val) **then**

(**match** (op1Val, op2Val) **with**

| (**ConstValue**(**Nil**),x) -> **ConstValue**(**Nil**)

| (x,**ConstValue**(**Nil**)) -> **ConstValue**(**Nil**)

| (**ConstValue**(**Epsilon**), x) -> x

| (x, **ConstValue**(**Epsilon**)) -> x

| (**ConstValue**(**False**),\_) | (\_,**ConstValue**(**False**)) -> **ConstValue**(**False**)

| \_-> **ConstValue**(**True**)

) **else** raise (**Failure**(**"Runtime Exception: Type Mismatch: '&&' operands must be of type boolean!\n"**))

*(\*.......................................................................................\*)*

| **Or** -> **if** isBooleanValue(op1Val) && isBooleanValue(op2Val) **then**

(**match** (op1Val, op2Val) **with**

| (**ConstValue**(**Nil**),x) -> **ConstValue**(**Nil**)

| (x,**ConstValue**(**Nil**)) -> **ConstValue**(**Nil**)

| (**ConstValue**(**Epsilon**), x) -> x

| (x, **ConstValue**(**Epsilon**)) -> x

| (**ConstValue**(**False**),**ConstValue**(**False**)) -> **ConstValue**(**False**)

| \_-> **ConstValue**(**True**)

) **else** raise (**Failure**(**"Runtime Exception: Type Mismatch: '||' operands must be of type boolean!\n\t Found: "**^(gs (processExprValue op1Val) **"FATAL ERROR: NOT EVALUATED"**)^**" || "**^(gs (processExprValue op2Val) **"FATAL ERROR: NOT EVALUATED"**)))

*(\*.......................................................................................\*)*

| **Not** -> (**match** (op1Val) **with**

| (**ConstValue**(**Nil**)) -> **ConstValue**(**Nil**)

| (**ConstValue**(**Epsilon**)) -> **ConstValue**(**Epsilon**)

| **ConstValue**(**False**) -> **ConstValue**(**True**)

| **ConstValue**(**True**) -> **ConstValue**(**False**)

| \_ -> raise (**Failure**(**"Runtime Exception: Type Mismatch: '!' (negation) operand must be of type boolean!\n"**)) )

*(\*.......................................................................................\*)*

| **Tilde** -> raise (**Failure**(**"Err2.\n"**)) *(\*op1Val ^ "~" ^ op2Val \*)*

*(\*.......................................................................................\*)*

| **Eval** -> op1Val *(\*Maybe here we should clone the expression\*)*

*(\*.......................................................................................\*)*

| **DontOpt** -> raise (**Failure**(**"Err5.\n"**)) *(\*"^" ^ op1Val\*)*

*(\*.......................................................................................\*)*

| **Let** -> **ConstValue**(**Epsilon**)

*(\*.......................................................................................\*)*

| **DirectAccess** -> raise (**Failure**(**"Err11.\n"**))

*(\*.......................................................................................\*)*

| **FuncCall** ->

**let** funcNameList = (**match** op.operand1.exprCode **with** **NameListExpr**(nameListEx) -> nameListEx.nameList | \_ -> rf(**"BUG: SemanticAnalysis.processOpReturnType(): FuncCall: funcLattice!"**)) **in**

**let** (idSymOpt, idLatOpt) = **try** findNameListSymbolAndLatticeInLatticeFields(ref (evalEnv.currentLat), funcNameList)

**with** **Not\_found** -> addToErrorTable(**SemUndefinedIdentifier**(**PrintLattice**.processNameList {parent = **None**; hash = **StringMap**.empty} funcNameList), **None**); (**None**, **None**)

**in**

(**match** idLatOpt **with**

| **None** -> raise (**Failure** (**"Runtime Exception: error in function call: function not declared:\n\t"**^**PrintLattice**.processOperation (op)))

| **Some**(funcLatticeRef) -> ( **let** funcLattice = !funcLatticeRef **in**

**let** expr = (gs op.operand2 **""**) **in**

**let** lattice = (**match** expr.exprCode **with** **LatticeExpr**(lat) -> lat | \_ -> createEmptyLattice [createLatticeFieldsFromLatticeType (**Expr**(expr))]) **in**

**let** funcLattice = (**if** funcLattice.paramNameList = **None** **then**

(**match** funcLattice.lattice **with** **Expr**(expr) ->

(**match** expr.exprCode **with** **LatticeExpr**(lat) -> lat | \_ -> funcLattice)

| \_ -> funcLattice)

**else** funcLattice) **in**

**let** doRunFunc = (**if** funcLattice.paramNameList = **None** **then** *(\*no param list\*)*

raise (**Failure** (**"Runtime Exception: error in function call: Not a function:\n\t"**^**PrintLattice**.processOperation (op)))

**else** **if** funcLattice.paramTypeList = **None** **then** ( *(\*first call\*)*

**if** calcParamTypeList(lattice, funcLattice) **then**

true

**else**

raise (**Failure** (**"Runtime Exception: error in function call: Parameter List Signature Length Difference:\n\t"**^**PrintLattice**.processOperation (op)))

)

**else** false) **in**

**if** (doRunFunc) **then** (

**if** **SemanticAnalysis**.checkParameterTypes(funcLattice.paramTypeList, lattice) **then** (

funcLattice.lattice <- (!(setSymbolTableInLattice(ref funcLattice, (getSymbolTableFromSeqlat funcLattice).hash)).lattice);

processLattice (funcLattice)

)

**else**

raise (**Failure** (**"Runtime Exception: error in function call: Parameter List Signature Type Mismatch:\n\t"**^**PrintLattice**.processOperation (op)))

)

**else**

**ConstValue**(**Nil**)

))

*(\*.......................................................................................\*)*

| **Assign** -> assignExprToNameList(op.operand1, op.operand2); processExpr op.operand1

*(\*.......................................................................................\*)*

| **Define** -> assignExprToNameList(op.operand1, op.operand2); processExpr op.operand1

*(\*.......................................................................................\*)*

)

**and** calcParamTypeList(\_paramLattice, \_funcLattice) =

**let** elements = getSeqlatElementsFromLatticeFields \_paramLattice **in**

**if** (**List**.length elements) <> (**List**.length (gs \_funcLattice.paramNameList **"BUG: calcParamTypeList"**)) **then** false

**else** **let** paramTypeList = **List**.map (**fun** x -> x.latDataStructType) elements **in**

\_funcLattice.paramTypeList <- **Some**(paramTypeList); true

**and** compareExprValues(\_val1, \_val2) =

**match** (\_val1, \_val2) **with**

| (**ConstValue**(**IntLiteral**(val1)),**ConstValue**(**IntLiteral**(val2))) ->

**if** val1 < val2 **then** **Lt** **else** **if** val1 > val2 **then** **Gt** **else** **Eq**

| (**ConstValue**(**FloatLiteral**(val1)),**ConstValue**(**FloatLiteral**(val2))) ->

**if** val1 < val2 **then** **Lt** **else** **if** val1 > val2 **then** **Gt** **else** **Eq**

| (**ConstValue**(**StringLiteral**(val1)),**ConstValue**(**StringLiteral**(val2))) ->

**let** res = **String**.compare val1 val2 **in** **if** res < 0 **then** **Lt** **else** **if** res > 0 **then** **Gt** **else** **Eq**

| (**ConstValue**(**Nil**),**ConstValue**(**Nil**)) -> **Eq**

| (**ConstValue**(**Nil**),x) -> **NilEncountered**

| (x,**ConstValue**(**Nil**)) -> **NilEncountered**

| (**ConstValue**(**Epsilon**),**ConstValue**(**Epsilon**)) -> **Eq**

| (**ConstValue**(**Epsilon**),x) -> **Lt**

| (x,**ConstValue**(**Epsilon**)) -> **Gt**

| \_ -> raise (**Failure**(**"Runtime Exception: Type Mismatch.\n"**))

**and** processNameList \_nameList =

**let** (idSymOpt, idLatOpt) = **try** findNameListSymbolAndLatticeInLatticeFields(ref (evalEnv.currentLat), \_nameList)

**with** **Not\_found** -> addToErrorTable(**SemUndefinedIdentifier**(**PrintLattice**.processNameList {parent = **None**; hash = **StringMap**.empty} \_nameList), **None**); (**None**, **None**)

**in**

(**match** idLatOpt **with**

| **None** -> **ConstValue**(**Nil**)

| **Some**(lat) -> processLattice (!lat)

)

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Pre-Evaluate Lattice \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

**and** preEvalLattice lattice =

**if** lattice.paramNameList = **None** **then**

**match** lattice.lattice **with**

| **Seqlat**(seqlat) -> (

preEvalEnv.parentStack <- preEvalEnv.currentLat :: preEvalEnv.parentStack *(\*PUSH\*)*;

preEvalEnv.currentLat <- lattice;

**ignore**(**List**.map (preEvalLattice) seqlat.elements);

**if** (**List**.length preEvalEnv.parentStack) > 0 **then**

preEvalEnv.currentLat <- **List**.hd preEvalEnv.parentStack *(\*POP\*)*

**else** rf(**"BUG: preEvalLattice(): Stack is not being built Correctly!!!\n"**);

preEvalEnv.parentStack <- **List**.tl preEvalEnv.parentStack; *(\*Remove\*)*

)

| **Altlat**(altlat) -> **ignore**(**List**.map (preEvalLattice) altlat.alternatives)

| **Predicate**(pred) -> **ignore**(preEvalExpr pred.condition); **ignore**(preEvalExpr pred.expr)

| **Expr**(expr) -> **ignore**(preEvalExpr expr)

*(\*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=\*)*

**and** preEvalExpr \_expr =

\_expr.exprEval <-

(**match** \_expr.exprCode **with**

| **OpExpr**(op) -> preEvalOperation op

| \_ -> **NotEvaluated**

);

\_expr.exprEval

*(\*--------------------------------------------------------------------\*)*

**and** preEvalOperation op =

**let** \_ = preEvalExpr op.operand1 **in**

**let** \_ = ( **match** op.operand2 **with**

| **None** -> **NotEvaluated**

| **Some**(opr) -> preEvalExpr opr

) **in**

(**match** op.operator **with**

*(\*.......................................................................................\*)*

| **Assign** ->

assignExprToNameList(op.operand1, op.operand2); **NotEvaluated**

*(\*.......................................................................................\*)*

| **Define** ->

assignExprToNameList(op.operand1, op.operand2); **NotEvaluated**

*(\*.......................................................................................\*)*

| **Eval** -> processExpr op.operand1

*(\*.......................................................................................\*)*

| \_ -> **NotEvaluated**

)

*(\*--------------------------------------------------------------------\*)*

**and** assignExprToNameList(\_lvalueExpr, \_rvalueExpr) =

**let** nameList = (**match** \_lvalueExpr.exprCode **with** **NameListExpr**(n) -> n.nameList | \_ -> rf(**"BUG: Evaluate: preEvalOperation(): Assign: NameList-2!"**)) **in**

**let** (idSymOpt, idLatOpt) = **try** findNameListSymbolAndLatticeInLatticeFields(ref (evalEnv.currentLat), nameList)

**with** **Not\_found** -> addToErrorTable(**SemUndefinedIdentifier**(**PrintLattice**.processNameList {parent = **None**; hash = **StringMap**.empty} nameList), **None**); (**None**, **None**)

**in**

(**match** idSymOpt **with**

| **None** -> ()

| **Some**(idSymbol) -> *(\*let exprF = createExprFieldsFromExprCodeType(convertExprValueTypeToExprCodeType op2Val) in\*)*

(**match** (!idSymbol).symValue **with**

| **LabelIndex**(index) -> (

**try** setSeqlatElementForLatticeFields(ref evalEnv.currentLat, index, createLatticeFieldsFromLatticeType (**Expr**(gs \_rvalueExpr **""**)))

**with** **Invalid\_argument**(\_) -> rf(**"BUG: preEvalOperation! Not a seqlat"**) )

| **LocalValueLattice**(latFRef) -> (!idSymbol).symValue <- (**LocalValueLattice**(ref (createLatticeFieldsFromLatticeType (**Expr**(gs \_rvalueExpr **""**)))))

; **ignore**(setNameListSymbolInLatticeFields(ref (evalEnv.currentLat), nameList, (!idSymbol)) )

)

)

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

**and** setSeqlatElementForLatticeFields(\_latticeRef, \_index, (\_value: latticeFields)) =

**let** seqlatRef = **try** getSeqlatFromLatticeFields(\_latticeRef) **with** e -> raise (**Invalid\_argument**(**"Not found!!"**)) **in**

**let** array = **Array**.of\_list (!seqlatRef).elements **in**

**Array**.set array \_index \_value;

(!seqlatRef).elements <- **Array**.to\_list array;

\_latticeRef := !(setSeqlatFromLatticeFields(\_latticeRef, seqlatRef))

*(\*--------------------------------------------------------------------\*)*

**and** findNameListSymbolAndLatticeInLatticeFields(\_latticeF, \_nameList) =

**match** \_nameList **with**

| [hd] -> findNameTypeSymbolAndLatticeInLatticeFields(\_latticeF, hd)

| hd::tl -> **let** (symOpt, latOpt) = findNameTypeSymbolAndLatticeInLatticeFields(\_latticeF, hd) **in**

(**match** latOpt **with**

| **None** -> (symOpt, **None**)

| **Some**(lat) -> findNameListSymbolAndLatticeInLatticeFields(lat, tl)

)

| [] -> rf(**"BUG: getNameListFromSymbolTable!"**)

**and** findNameTypeSymbolAndLatticeInLatticeFields(\_latticeF, \_nameType) =

**let** symbolTable = **try** getSymbolTableFromSeqlat (!\_latticeF) **with** e -> raise **Not\_found** **in**

**match** \_nameType **with**

| **This** -> rf(**"Not Implemented: 'This'!"**)

| **PredefinedLabel**(predefinedLabel) -> rf(**"Not Implemented: 'This'!"**)

| **IndexingOp**(structureType, latticeT) -> *(\*uses: Seq (for []), Alt (for {}), and UndeterminedStruct (for @)\*)*

rf(**"Not Implemented: 'This'!"**)

| **HashOp**(exprF) -> rf(**"Not Implemented: 'Hash'!"**)

| **Id**(str) ->

**let** symbolOpt = findRefInSymbolTable(symbolTable, str) **in**

**let** latticeOpt = (**match** symbolOpt **with**

| **None** -> raise (**Not\_found**)

| **Some**(symbolRef) -> (**match** (!symbolRef).symValue **with**

| **LabelIndex**(ind) -> **Some**(ref (getLatticeFieldsByIndex(!\_latticeF, ind)))

| **LocalValueLattice**(latFRef) -> **Some**(latFRef)

)) **in**

(symbolOpt, latticeOpt)

| **Each**(latticeT) -> rf(**"Not Implemented: 'This'!"**)

*(\*--------------------------------------------------------------------\*)*

**and** setNameListSymbolInLatticeFields(\_latticeF, \_nameList, \_symbol) =

**match** \_nameList **with**

| [hd] -> setNameTypeSymbolInLatticeFields(\_latticeF, hd, \_symbol, true)

| hd::tl -> **let** (symOpt, latOpt) = setNameTypeSymbolInLatticeFields(\_latticeF, hd, \_symbol, false) **in**

(**match** latOpt **with**

| **None** -> (symOpt, **None**)

| **Some**(lat) -> setNameListSymbolInLatticeFields(lat, tl, \_symbol)

)

| [] -> rf(**"BUG: getNameListFromSymbolTable!"**)

**and** setNameTypeSymbolInLatticeFields(\_latticeF, \_nameType, \_symbol, \_doSave) =

**let** symbolTable = **try** getSymbolTableFromSeqlat (!\_latticeF) **with** e -> raise **Not\_found** **in**

**match** \_nameType **with**

| **This** -> rf(**"Not Implemented: 'This'!"**)

| **PredefinedLabel**(predefinedLabel) -> rf(**"Not Implemented: 'This'!"**)

| **IndexingOp**(structureType, latticeT) -> *(\*uses: Seq (for []), Alt (for {}), and UndeterminedStruct (for @)\*)*

rf(**"Not Implemented: 'This'!"**)

| **HashOp**(exprF) -> rf(**"Not Implemented: 'Hash'!"**)

| **Id**(str) ->

**let** symbolOpt = findRefInSymbolTable(symbolTable, str) **in**

**if** (\_doSave) **then** (

symbolTable.hash <- **StringMap**.add str \_symbol symbolTable.hash;

**ignore**(\_latticeF = setSymbolTableInLattice(\_latticeF, symbolTable.hash));

(**None**, **None**)

)

**else**

**let** latticeOpt = (**match** symbolOpt **with**

| **None** -> raise (**Not\_found**)

| **Some**(symbolRef) -> (**match** (!symbolRef).symValue **with**

| **LabelIndex**(ind) -> **Some**(ref (getLatticeFieldsByIndex(!\_latticeF, ind)))

| **LocalValueLattice**(latFRef) -> **Some**(latFRef)

)) **in**

(symbolOpt, latticeOpt)

| **Each**(latticeT) -> rf(**"Not Implemented: 'This'!"**)

**and** setSymbolTableInLattice(\_latticeF, \_symbolTableHash) =

**match** (!\_latticeF).lattice **with**

| **Seqlat**(seqlat) -> (seqlat.symbols.hash <- \_symbolTableHash; (!\_latticeF).lattice <- **Seqlat**(seqlat)); \_latticeF

| **Expr**(expr) -> (**match** expr.exprCode **with**

| **LatticeExpr**(lat) -> **let** \_ = setSymbolTableInLattice(ref lat, \_symbolTableHash) **in** \_latticeF

| \_ -> rf(**"BUG: getSymbolTableFromSeqlat()!"**))

| \_ -> rf(**"BUG: getSymbolTableFromSeqlat()!"**)

*(\*Deprecated\*)*

*(\*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=\*)*

**and** processCurNameList \_curNamesList =

**match** \_curNamesList **with**

| [] -> raise (**Failure**(**"Bug"**))

| hd::tl -> **if** (**List**.length tl > 0) **then**

(

**match** hd **with**

| **Id**(id) -> (

**match** evalEnv.currentLat.lattice **with**

| **Seqlat**(seqlat) -> (

**let** mySymbol = **StringMap**.find id seqlat.symbols.hash **in**

**match** mySymbol.symValue **with**

| **LabelIndex**(index) -> (

evalEnv.parentStack <- evalEnv.currentLat :: evalEnv.parentStack *(\*PUSH\*)*;

**let** curLat = **List**.nth seqlat.elements index **in**

evalEnv.currentLat <- curLat;

**let** res = processCurNameList tl **in** res

*(\* evalEnv.currentLat <- List.hd evalEnv.parentStack; (\*POP\*)*;

evalEnv.parentStack <- **List**.tl evalEnv.parentStack; *(\*Remove\*)*

\*)

)

| **LocalValueLattice**(newlatFields) ->

(

*(\* let nextLatFields = !newlatFields in*

*processCurNameList tl \*)*

raise (**Failure**(**"Local"**))

)

)

| \_ -> raise (**Failure**(**"Bug"**))

)

| \_ -> raise (**Failure**(**"Still Not Handled"**))

)

**else**

(

**match** hd **with**

| **Id**(id) -> (

**match** evalEnv.currentLat.lattice **with**

| **Seqlat**(seqlat) -> (

**let** mySymbol = **StringMap**.find id seqlat.symbols.hash **in**

**match** mySymbol.symValue **with**

| **LabelIndex**(index) ->

**let** curElement = **List**.nth seqlat.elements index **in**

**let** result = (**match** curElement.lattice **with**

| **Seqlat**(seq) -> raise (**Failure**(**"Seq "**))

| **Altlat**(alt) -> raise (**Failure**(**"Alt"**))

| **Predicate**(pred) -> raise (**Failure**(**"Pred"**))

| **Expr**(expr) -> processExpr expr

) **in** result

| **LocalValueLattice**(newlatFields) ->

**let** nextLatFields = !newlatFields **in**

**let** result = ( **match** nextLatFields.lattice **with**

| **Seqlat**(seq) -> raise (**Failure**(**"Seq-local"**))

| **Altlat**(alt) -> raise (**Failure**(**"Alt-Local"**))

| **Predicate**(pred) -> raise (**Failure**(**"Pred-Local"**))

| **Expr**(expr) -> processExpr expr

) **in** result

)

| **Altlat**(altlat) -> raise (**Failure**(**"altlat"**))

| **Predicate**(pred) -> raise (**Failure**(**"pred"**))

| **Expr**(expr) -> raise (**Failure**(**"expr"**))

*(\*let res = createExprFieldsFromExprCodeType(ConstExpr(IntLiteral(2))) in res \*)*

)

| **IndexingOp**(structType, lattice) -> raise (**Failure**(**"Still Not Handled"**))

| \_ -> raise (**Failure**(**"Still Not Handled"**))

)

## latparser.mly Source Code

%{

**open** **Ast** ;;

**open** **LatticeCreation**

%}

%token **LPAREN** **RPAREN** **LBRACE** **RBRACE** **LSQBRACKET** **RSQBRACKET** **EOF**

%token **PLUS** **MINUS** **TIMES** **DIVIDE** **PERCENT** **PLUS\_PLUS** **MINUS\_MINUS**

%token **PIPE** **TILDE** **QUEST** **SEMI** **COLON** **DOT** **AT** **CARET**

%token **ASSIGN** **QUEST\_ASSIGN** **DEF**

%token **AND** **OR** **EQ** **NEQ** **EXCLAMATION** **LT** **GT** **LEQ** **GEQ**

%token **EPSILON** **NIL** **TRUE** **FALSE**

%token **LET** **THIS** **NO\_TOKEN** **FUNC\_CALL**

/\***LENGTH** **COUNT** **LABELS** **CLONE** **REVERSE** **RETURN** **LOOP** **GET\_LOST**\*/

%token <int> **PREDEFINED\_LABEL**

%token <string> **STRING\_LITERAL**

%token <string> **ID**

%token <float> **FLOAT**

%token <int> **INTEGER**

%left **NO\_TOKEN**

%left **COLON**

%left **LET**

/\* **Assignment** **Operators**: \*/

%right **ASSIGN** **QUEST\_ASSIGN**

/\* **Lattice** **Operators**: \*/

%left **SEMI**

%left **PIPE**

%left **TILDE**

/\* **Logical** **Operators**: \*/

%left **OR**

%left **AND**

/\* **Comparison** **Operators**: \*/

%left **EQ** **NEQ**

%left **LT** **LEQ** **GT** **GEQ**

/\* **Arithmatic** **Operators**: \*/

%left **PLUS** **MINUS**

%left **TIMES** **DIVIDE** **PERCENT**

/\* **Unary** **Operators** **Operators**: \*/

%left **UNI\_MINUS** **EXCLAMATION** **PLUS\_PLUS** **MINUS\_MINUS** **QUEST** **CARET**

/\* **Name** **Operators**: \*/

%left **LBRACE** **LSQBRACKET** **LPAREN**

%left **DOT**

%left **AT**

%left **DEF**

%start lattice

%**type** <**Ast**.lattice> lattice

%%

/\*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=\*/

/\*=-=-=-=-= **DEFINING** **LATTICEs** =-=-=-=-=-=\*/

/\*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=\*/

lattice:

labeled\_altlat { ([(createLatticeFieldsFromLatticeType $1)], env) }

| labeled\_altlat **SEMI** lattice { (appendListForSeqlat((createLatticeFieldsFromLatticeType $1), (fst $3)), env) }

| error **SEMI** lattice { addToErrorTable(**SynParseError**, **None**); (appendListForSeqlat((createLatticeFieldsFromLatticeType(**Expr**(newExprFields(**ConstExpr**(**Nil**))))), (fst $3)), env) }

| error { addToErrorTable(**SynParseError**, **None**); ([createLatticeFieldsFromLatticeType(**Expr**(newExprFields(**ConstExpr**(**Nil**))))], env) }

labeled\_altlat:

altlat { incLabelIdGen env.labelIdGenerator; $1 }

| label **COLON** altlat { incLabelIdGen env.labelIdGenerator; createLabeledAltlat($1, $3) }

| function\_header **COLON** altlat { incLabelIdGen env.labelIdGenerator; createParamLabeledAltlat(fst $1, snd $1, $3) }

function\_header:

**DEF** **ID** **LPAREN** param\_list **RPAREN** { popOldSymbolTable(env); ($2, $4) }

param\_list:

**ID** { [$1] }

| param\_list **SEMI** **ID** { $3::$1 }

altlat:

/\*epsilon\*/ { **Expr**(newExprFields(**ConstExpr**(**Epsilon**))) }

| expr { **Expr**($1) }

| **LSQBRACKET** expr **RSQBRACKET** expr { createPredicateLattice($2, $4) }

| altlat **PIPE** altlat { appendListForAltlat($1, $3) }

label:

**ID** { $1 }

/\*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=\*/

/\*=-=-=-=-=-= **Defining** **Expressions** =-=-=-=-=\*/

/\*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=\*/

expr:

lvalue { $1 }

| name **QUEST\_ASSIGN** expr { newExprFields(createBinOp($1,**Assign**,newExprFields(createUnaryOp(**Eval**, $3)))) }

| lvalue **ASSIGN** expr { newExprFields(createBinOp($1,**Assign**,$3)) }

| **DEF** **ID** **ASSIGN** expr { createLocal($2, $4) }

| function\_header **ASSIGN** expr { createParamLocal(fst $1, snd $1, $3) }

| **QUEST** lvalue { newExprFields(createUnaryOp(**Eval**, $2)) }

| **LET** expr { newExprFields(createUnaryOp(**Let**, $2))}

| **LPAREN** lattice **RPAREN** **DOT** rest\_of\_name { createDirectAccessExprFields($2, $5) }

| name **LPAREN** lattice **RPAREN** { newExprFields(createBinOp($1, **FuncCall**, createParenthisizedLattice($3))) }

| number { $1 }

| **STRING\_LITERAL** { newExprFields(**ConstExpr**(**StringLiteral**($1))) }

| **EPSILON** { newExprFields(**ConstExpr**(**Epsilon**)) }

| **NIL** { newExprFields(**ConstExpr**(**Nil**)) }

| **TRUE** { newExprFields(**ConstExpr**(**True**)) }

| **FALSE** { newExprFields(**ConstExpr**(**False**)) }

| expr **PLUS** expr { newExprFields(createBinOp($1,**Plus**,$3)) }

| expr **MINUS** expr { newExprFields(createBinOp($1,**Minus**,$3)) }

| expr **TIMES** expr { newExprFields(createBinOp($1,**Times**,$3)) }

| expr **DIVIDE** expr { newExprFields(createBinOp($1,**Divide**,$3)) }

| expr **PERCENT** expr { newExprFields(createBinOp($1,**Percent**,$3)) }

| expr **OR** expr { newExprFields(createBinOp($1,**Or**,$3)) }

| expr **AND** expr { newExprFields(createBinOp($1,**And**,$3)) }

| expr **EQ** expr { newExprFields(createBinOp($1,**Equal**,$3)) }

| expr **NEQ** expr { newExprFields(createBinOp($1,**NotEqual**,$3)) }

| expr **LT** expr { newExprFields(createBinOp($1,**Less**,$3)) }

| expr **LEQ** expr { newExprFields(createBinOp($1,**LessOrEqual**,$3)) }

| expr **GT** expr { newExprFields(createBinOp($1,**Greater**,$3)) }

| expr **GEQ** expr { newExprFields(createBinOp($1,**GreaterOrEqual**,$3)) }

| expr **TILDE** expr { newExprFields(createBinOp($1,**Tilde**,$3)) }

| **MINUS** expr %prec **UNI\_MINUS** { newExprFields(createUnaryOp(**UniMinus**, $2)) }

| **EXCLAMATION** expr { newExprFields(createUnaryOp(**Not**, $2)) }

| **CARET** **LPAREN** lattice **RPAREN** { newExprFields(createUnaryOp(**DontOpt**, createParenthisizedLattice($3))) }

/\*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=\*/

/\*=-=-=-=-= **Defining** **Left**-**Value** **Names** =-=-=-=-=-=\*/

/\*=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=\*/

lvalue:

name { $1 }

| **LPAREN** lattice **RPAREN** { createParenthisizedLattice($2) }

| **CARET** name { newExprFields(createUnaryOp(**DontOpt**, $2)) }

name:

rest\_of\_name { newExprFields(**NameListExpr**($1)) }

| at\_operation { newExprFields(**NameListExpr**($1)) }

| at\_operation **DOT** rest\_of\_name { newExprFields(**NameListExpr**(combineTwoNameLists($1, $3))) }

rest\_of\_name:

**PREDEFINED\_LABEL** { {nameList = [**PredefinedLabel**(getPredefinedLabel($1))]} }

| label %prec **NO\_TOKEN** { {nameList = [**Id**($1)]} }

| rest\_of\_name **DOT** rest\_of\_name { combineTwoNameLists($1, $3) }

| rest\_of\_name **LSQBRACKET** altlat **RSQBRACKET** { combineTwoNameLists($1, {nameList=[**IndexingOp**(**Seq**, $3)]}) }

| rest\_of\_name **LBRACE** altlat **RBRACE** { combineTwoNameLists($1, {nameList=[**IndexingOp**(**Alt**, $3)] }) }

| **LBRACE** altlat **RBRACE** { {nameList = [**Each**($2)]} }

at\_operation:

**AT** **INTEGER** { createAtOperationOfInt($2) }

| **AT** **ID** { createAtOperationOfId($2) }

| **AT** **LPAREN** altlat **RPAREN** { createAtOperationOfAltlat($3) }

number:

**INTEGER** { newExprFields(**ConstExpr**(**IntLiteral**($1))) }

| **FLOAT** { newExprFields(**ConstExpr**(**FloatLiteral**($1))) }

## latte.ml Source Code

*(\*\**

*\*\* Lattakia Compiler*

*\*\* Columbia University COMS-4115*

*\*\* Programming Languages and Translators*

*\*\* Authors:*

*\*\* Wael Salloum*

*\*\* Heba ElFardy*

*\*\* Katherine Scott*

*\*\*)*

**open** **Ast**

**open** **LatticeCreation**

**open** **SemanticAnalysis**

**open** **Evaluate**

**open** **PrintLattice**

**let** testMode = false;;

**let** \_ =

*(\* Lexical Analysis \*)*

**let** lexbuf = **Lexing**.from\_channel stdin **in**

*(\* Syntactic Analysis \*)*

**let** latticeAndEnv = **Latparser**.lattice **Scanner**.token lexbuf **in**

**let** env = snd latticeAndEnv **in**

**let** lattice = createNewLatFromLatEnvTuple(latticeAndEnv, env.namesOfCurrLat) **in**

**if** testMode **then**

print\_endline (**"\n\t>>>>> Parser Output <<<<<\n"** ^ (**PrintLattice**.toString lattice {printEnv.options **with** doProcessSymbolTable = false; doAnnotateTypes=false}));

*(\* Semantic Analysis \*)*

**let** (lattice, latticeType) = **SemanticAnalysis**.analyzeSemantics lattice **in**

**if** testMode **then** ( print\_endline (**"\n\t>>>>> Semantic Output <<<<<\n"** ^ (**PrintLattice**.toString lattice {printEnv.options **with** doProcessSymbolTable=true; doAnnotateTypes=true}));

print\_endline (**if** **List**.length env.errorTable != 0 **then** (**"\n\t>>>>> Errors <<<<<\n"**^(stringOfErrorTable env.errorTable)) **else** **"No Errors"**)

);

**if** (**match** latticeType **with**

| (**MismatchType**, \_) -> print\_endline(**"Semantic Analysis >> Error: Type Mismatch: "**^(**PrintLattice**.dataStructTypeToString latticeType)); true

| (**UnresolvedType**, \_) -> **if** testMode **then** print\_endline(**"Semantic Analysis >> Ambiguity: Unresolved Type: "**^(**PrintLattice**.dataStructTypeToString latticeType)); false

| (x, y) -> **if** testMode **then** print\_endline(**"Semantic Analysis >> Succeeded! Lattice Type: "**^(**PrintLattice**.dataStructTypeToString latticeType)); false

) **then** () **else** *(\*Continue\*)*

**if** (**List**.length env.errorTable) > 0 **then** () **else** (

*(\* Execution (The Interpreter) \*)*

**let** exprValue = **Evaluate**.eval lattice **in**

*(\* Output \*)*

**PrintLattice**.printEnv.options <- {processCode=false; doProcessSymbolTable = false; doAnnotateTypes=false};

**let** result = gs (**PrintLattice**.processExprValue exprValue) **"FATAL ERROR: COULD NOT EVALUATE THE LATTICE!"** **in**

print\_endline ( (**if** testMode **then** **"\n\t>>>>> Final Output <<<<<\n"** **else** **""**)^ result ^ **"\n"**)

)

## latticeCreation.ml Source Code

**open** **Ast**

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Helpers (Auxiliary Function) \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

**let** rf *(\*raise failure\*)* str = raise (**Failure**(str))

**let** gs *(\*get Some\*)* opt msg = **match** opt **with** **Some**(value) -> value | \_ -> (rf(**if**(**String**.length msg) = 0 **then** **"BUG"** **else** msg))

**let** incr\_linenum lexbuf =

**let** pos = lexbuf.**Lexing**.lex\_curr\_p **in**

lexbuf.**Lexing**.lex\_curr\_p <- { pos **with**

**Lexing**.pos\_lnum = pos.**Lexing**.pos\_lnum + 1;

**Lexing**.pos\_bol = pos.**Lexing**.pos\_cnum;

};

env.lineNumber <- pos.**Lexing**.pos\_lnum;

env.charOffset <- pos.**Lexing**.pos\_cnum;

;;

**let** addToErrorTable ((\_errorMessage: errorMessageType), (\_errorPlace: exprCodeType option)) =

*(\* let start\_pos = Parsing.rhs\_start\_pos 3 in Parsing.*

*let end\_pos = Parsing.rhs\_end\_pos 3 in*

*printf "%d.%d-%d.%d: division by zero"*

*start\_pos.pos\_lnum (start\_pos.pos\_cnum - start\_pos.pos\_bol)*

*end\_pos.pos\_lnum (end\_pos.pos\_cnum - end\_pos.pos\_bol);\*)*

env.errorTable <- {errMessage = \_errorMessage; errPlace = \_errorPlace; lineNum = env.lineNumber; offset = env.charOffset} :: env.errorTable;

;;

**let** addToWorningTable ((\_errorMessage: errorMessageType), (\_errorPlace: exprCodeType option)) =

env.worningTable <- {errMessage = \_errorMessage; errPlace = \_errorPlace; lineNum = env.lineNumber; offset = env.charOffset} :: env.worningTable

;;

**let** createEmptyLattice \_elements =

{

latDataStructType = (**UnresolvedType**, **Seq**);

latNature = **Object**;

latEval = **None**;

paramNameList = **None**;

paramTypeList = **None**;

hasChanged = true;

optimize = true;

lattice = **Seqlat** {elements = \_elements; symbols = {parent=**None**; hash=**StringMap**.empty}};

}

;;

**let** createLatticeFieldsFromLatticeType latticeTypeRecord=

{

latDataStructType = (**UnresolvedType**, **UndeterminedStruct**);

latNature = **Object**;

latEval = **None**;

paramNameList = **None**;

paramTypeList = **None**;

hasChanged = true;

optimize = true;

lattice = latticeTypeRecord;

}

;;

**let** createNewLatFromLatEnvTuple(\_latticeEnvTuple, (\_symbolTable: symbolTableType)) =

**let** (latticeElements, latEnv) = \_latticeEnvTuple **in**

**if** (**List**.length latticeElements <= 1) && (**StringMap**.is\_empty \_symbolTable.hash) && false **then**

**let** innerLatFields = **List**.hd latticeElements **in** (**match** innerLatFields.lattice **with**

| **Seqlat**(innerSeqlat) -> innerSeqlat.symbols.parent <- \_symbolTable.parent

| \_ -> ());

innerLatFields

**else**

**let** innerSeqlat =

**Seqlat** {

elements = latticeElements;

symbols = \_symbolTable;

} **in**

createLatticeFieldsFromLatticeType innerSeqlat

**let** createExprFieldsFromLatticeFields (\_latFields: **Ast**.latticeFields) =

{

exprDataStructType = \_latFields.latDataStructType;

exprCode = **LatticeExpr**(\_latFields); *(\*ConstExpr | LatticeExpr | OpExpr | NameListExpr \*)*

exprEval = **NotEvaluated**; *(\*ConstExpr | LatticeExpr \*)*

exprChanged = true; *(\*a constant cannot change\*)*

expDependers = [];

expDependees = [];

}

**let** createExprFieldsFromExprCodeType(exprCode: **Ast**.exprCodeType) =

{

exprDataStructType = (**UnresolvedType**, **UndeterminedStruct**);

exprCode = exprCode; *(\*ConstExpr | LatticeExpr | OpExpr | NameListExpr \*)*

exprEval = **NotEvaluated**; *(\*ConstExpr | LatticeExpr \*)*

exprChanged = true; *(\*a constant cannot change\*)*

expDependers = [];

expDependees = [];

}

**let** convertExprValueTypeToExprCodeType(exprValue: **Ast**.exprValueType) =

(**match** exprValue **with**

| **ConstValue**(const) -> **ConstExpr**(const)

| **LatticeValue**(lat) -> **LatticeExpr**(lat)

| **NotEvaluated** -> **ConstExpr**(**Nil**)

)

*(\*A shorter name only\*)*

**let** newExprFields (exprCode: **Ast**.exprCodeType) = createExprFieldsFromExprCodeType exprCode

**let** getDataTypeForPredefinedLabel id = **match** id **with**

0 -> **Int** *(\*LENGTH\*)*

| 1 -> **Int** *(\*COUNT\*)*

| 2 -> **UnresolvedType** *(\*CLONE\*)*

| 3 -> **UnresolvedType** *(\*REVERSE\*)*

| 4 -> **UnresolvedType** *(\*RETURN\*)*

| 5 -> **UnresolvedType***(\*LOOP\*)*

| 6 -> **General** *(\*GET\_LOST\*)*

| 10 -> **UnresolvedType** *(\*LABELS\*)*

| \_ -> raise (**Failure** (**"BUG: getPredefinedLabel(): id our of range: id = "**^string\_of\_int id))

;;

**let** getPredefinedLabel id = **match** id **with**

0 -> **LENGTH**

| 1 -> **COUNT**

| 2 -> **CLONE**

| 3 -> **REVERSE**

| 4 -> **RETURN**

| 5 -> **LOOP**

| 6 -> **GET\_LOST**

| 10 -> **LABELS**

| \_ -> raise (**Failure** (**"BUG: getPredefinedLabel(): id our of range: id = "**^string\_of\_int id))

**let** **rec** createNewSymbol (\_name, \_scope, \_paramList, \_dependers, \_dependees, (\_altlatFields: **Ast**.latticeFields)) =

**let** symbolValue = (**match** \_scope **with**

| **Local** -> **LocalValueLattice**(ref (\_altlatFields))

| **Label** -> **LabelIndex**(env.labelIdGenerator)

| **UnresolvedSymbol** -> rf(**"Runtime Exception: createNewSymbol(): UnresolvedSymbol!"**)

) **in**

{

symName = \_name;

symDataStructType = \_altlatFields.latDataStructType;

symValue = symbolValue;

symbolScope = \_scope;

parameterNameList = \_paramList;

parameterTypeList = **None**;

dependers = \_dependers;

dependees = \_dependees;

}

**and** addParamListToSeqlatSymbols(\_paramList, \_symbolTable) =

**let** symTable = (**match** \_paramList **with**

| [] -> \_symbolTable

| paramName::tl -> ((

**if** **StringMap**.mem paramName \_symbolTable.hash **then**

addToErrorTable(**SemIdentifierMultipleDefinition**, **None**)

**else**

**let** paramSymbol = createNewSymbol (paramName, **Local**, **None**, [], [], (createEmptyLattice [])) **in**

\_symbolTable.hash <- **StringMap**.add paramName paramSymbol \_symbolTable.hash);

addParamListToSeqlatSymbols(tl, \_symbolTable)

)

)

**in** symTable

**and** createNewIntLiteralLattice i =

{

exprDataStructType = (**Int**, **Wrd**);

exprCode = **ConstExpr**(**IntLiteral**(i)); *(\*ConstExpr | LatticeExpr | OpExpr | NameListExpr \*)*

exprEval = **ConstValue**(**IntLiteral**(i)); *(\*ConstExpr | LatticeExpr \*)*

exprChanged = false; *(\*a constant cannot change\*)*

expDependers = [];

expDependees = [];

}

**and** getSymbolTableFromSeqlat \_latticeFieldsSeqlat = **match** \_latticeFieldsSeqlat.lattice **with**

| **Seqlat**(seqlat) -> seqlat.symbols

| **Expr**(expr) -> (**match** expr.exprCode **with** **LatticeExpr**(lat) -> getSymbolTableFromSeqlat lat | \_ -> rf(**"BUG: getSymbolTableFromSeqlat()!"**))

| \_ -> rf(**"BUG: getSymbolTableFromSeqlat()!"**)

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

*(\*For Evaluation\*)*

**and** getLatticeFieldsByIndex(\_latticeFields, \_index) =

**match** \_latticeFields.lattice **with**

| **Seqlat**(seqlat) -> **List**.nth seqlat.elements \_index

| **Expr**(expr) -> (**match** expr.exprCode **with** **LatticeExpr**(lat) -> getLatticeFieldsByIndex(lat, \_index) | \_ -> raise(**Not\_found**))

| \_ -> raise(**Not\_found**)

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

*(\*Symbol Table\*)*

**and** findInCurrentSymbolTable(\_symbolTable, \_symName) =

**if** **StringMap**.mem \_symName \_symbolTable.hash **then**

**Some**( **StringMap**.find \_symName \_symbolTable.hash )

**else**

**None**

**and** findInSymbolTable(\_symbolTable, \_symName) =

**if** **StringMap**.mem \_symName \_symbolTable.hash **then**

**Some**((**StringMap**.find \_symName \_symbolTable.hash))

**else**

**match** \_symbolTable.parent **with**

| **None** -> **None**

| **Some**(\_\_symbolTable) -> findInSymbolTable(\_\_symbolTable, \_symName)

**and** findRefInSymbolTable(\_symbolTable, \_symName) =

**if** **StringMap**.mem \_symName \_symbolTable.hash **then**

**Some**(ref (**StringMap**.find \_symName \_symbolTable.hash))

**else**

**match** \_symbolTable.parent **with**

| **None** -> **None**

| **Some**(\_\_symbolTable) -> findRefInSymbolTable(\_\_symbolTable, \_symName)

*(\*-------------------------------------------------------------------------------------\*)*

**and** findNameTypeSymbolInLatticeFields(\_latticeF, \_nameType) =

**let** symbolTable = **try** getSymbolTableFromSeqlat \_latticeF **with** e -> raise **Not\_found** **in**

**match** \_nameType **with**

| **This** -> rf(**"Not Implemented: 'This'!"**)

| **PredefinedLabel**(predefinedLabel) -> (**None**, **None**)

| **IndexingOp**(structureType, latticeT) -> *(\*uses: Seq (for []), Alt (for {}), and UndeterminedStruct (for @)\*)*

(**None**, **None**)

| **HashOp**(exprF) -> rf(**"Not Implemented: 'Hash'!"**)

| **Id**(str) ->

**let** symbolOpt = findInSymbolTable(symbolTable, str) **in**

**let** latticeOpt = (**match** symbolOpt **with**

| **None** -> raise (**Not\_found**)

| **Some**(symbol) -> (**match** (symbol).symValue **with**

| **LabelIndex**(ind) -> **Some**(getLatticeFieldsByIndex(\_latticeF, ind))

| **LocalValueLattice**(latFRef) -> **Some**(!latFRef)

)) **in**

(symbolOpt, latticeOpt)

| **Each**(latticeT) -> (**None**, **None**)

**and** findNameListSymbolInLatticeFields(\_latticeF, \_nameList) =

**match** \_nameList **with**

| [hd] -> findNameTypeSymbolInLatticeFields(\_latticeF, hd)

| hd::tl -> **let** (symOpt, latOpt) = findNameTypeSymbolInLatticeFields(\_latticeF, hd) **in**

(**match** latOpt **with**

| **None** -> (symOpt, **None**)

| **Some**(lat) -> findNameListSymbolInLatticeFields(lat, tl)

)

| [] -> rf(**"BUG: getNameListFromSymbolTable!"**)

*(\*deprecated\*)*

**and** findNameListSymbolInLatticeFieldsOpt(\_latticeF, \_nameList) =

**try** **Some**(findNameListSymbolInLatticeFields(\_latticeF, \_nameList))

**with** \_ -> **None**

*(\*-------------------------------------------------------------------------------------\*)*

**and** addLocalToSymbolTable(\_symbolTable, \_symName, \_paramList, (expr: **Ast**.exprFields)) =

**let** found = (**match** findInCurrentSymbolTable(\_symbolTable, \_symName) **with**

| **None** -> false

| **Some**(\_) -> true) **in**

**if** found **then**

addToErrorTable(**SemIdentifierMultipleDefinition**, **None**)

**else**

**let** symbol = createNewSymbol(\_symName, **Local**, \_paramList, [], [],

createLatticeFieldsFromLatticeType (**Expr**(expr))) **in**

\_symbolTable.hash <- **StringMap**.add \_symName symbol \_symbolTable.hash

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Lattice Creation Functions \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

**let** incLabelIdGen lastId =

env.labelIdGenerator <- lastId + 1

;;

*(\* >>>>> lattice: Append lattice list for building a Seqlat <<<<< \*)*

**let** appendListForSeqlat(\_labAltlat, \_lattice) =

(\_labAltlat::\_lattice)

;;

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

*(\* >>>>> labeled\_altlat: Create Labeled Altlat <<<<< \*)*

**let** createLabeledAltlat(\_label, (\_altlat: **Ast**.latticeType)) =

**let** altlatFields = createLatticeFieldsFromLatticeType \_altlat **in**

**ignore** (**match** findInCurrentSymbolTable(env.namesOfCurrLat, \_label) **with**

| **None** ->

**let** newLabel = createNewSymbol(\_label, **Label**, **None**, [], [ref (createExprFieldsFromLatticeFields altlatFields)], altlatFields) **in**

env.namesOfCurrLat.hash <- **StringMap**.add \_label newLabel env.namesOfCurrLat.hash;

newLabel

| **Some**(oldLabel) ->

addToErrorTable(**SemIdentifierMultipleDefinition**, **Some**(**LatticeExpr**(altlatFields)));

oldLabel

);

\_altlat

;;

*(\* >>>>> labeled\_altlat: Create Parameterized Labeled Altlat <<<<< \*)*

**let** createParamLabeledAltlat(\_label, \_paramList, (\_altlat: **Ast**.latticeType)) =

**let** altlatFields = createLatticeFieldsFromLatticeType \_altlat **in**

**ignore** (**match** findInCurrentSymbolTable(env.namesOfCurrLat, \_label) **with**

| **None** ->

**let** newLabel = createNewSymbol(\_label, **Label**, **Some**(\_paramList), [], [ref (createExprFieldsFromLatticeFields altlatFields)], altlatFields) **in**

env.namesOfCurrLat.hash <- **StringMap**.add \_label newLabel env.namesOfCurrLat.hash;

newLabel

| **Some**(oldLabel) ->

addToErrorTable(**SemIdentifierMultipleDefinition**, **Some**(**LatticeExpr**(altlatFields)));

oldLabel

);

**let** newParamSeqlat = createEmptyLattice [altlatFields] **in**

**let** innerSeqlat = (**match** newParamSeqlat.lattice **with**

| **Seqlat**(seqlat) ->

seqlat.symbols <- (addParamListToSeqlatSymbols(\_paramList, seqlat.symbols)); seqlat

| \_ -> rf(**"BUG: createParamLabeledAltlat()"**)

) **in**

newParamSeqlat.lattice <- **Seqlat**(innerSeqlat);

newParamSeqlat.paramNameList <- **Some**(\_paramList);

**Expr**(createExprFieldsFromLatticeFields(newParamSeqlat))

;;

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

*(\* >>>>> altlat: Create Parameterized Labeled Altlat <<<<< \*)*

**let** createPredicateLattice(\_condition, \_expr) = **Predicate**

{

expr = \_expr;

condition = \_condition;

}

;;

*(\* >>>>> altlat: Concatenate two list of alternatives to create an Altlat <<<<< \*)*

**let** appendListForAltlat((\_alt1: latticeType), (\_alt2: latticeType)) =

**let** list1 = (**match** \_alt1 **with**

| **Altlat**(alt1) -> alt1.alternatives

| \_ -> [createLatticeFieldsFromLatticeType(\_alt1)]) **in**

**let** list2 = (**match** \_alt2 **with**

| **Altlat**(alt2) -> alt2.alternatives

| \_ -> [createLatticeFieldsFromLatticeType(\_alt2)]) **in**

**Altlat** {

alternatives = list1 @ list2;

each = false;

}

;;

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Expression Creation Functions \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

**let** processSymbolTableKeys \_symbolTable =

**let** keyList = **StringMap**.fold(**fun** x y z -> x ^**", "** ^ z) \_symbolTable.hash **""** **in**

keyList

**let** pushNewSymbolTable \_ =

**let** newSymTab = {parent = **Some**(env.namesOfCurrLat); hash = **StringMap**.empty} **in**

env.namesOfCurrLat <- newSymTab;

env.parentStackLabelIdGenerator <- env.labelIdGenerator :: env.parentStackLabelIdGenerator;

env.labelIdGenerator <- -1

**let** popOldSymbolTable \_ =

env.poppedSymbolTable <- env.namesOfCurrLat;

(**match** env.namesOfCurrLat.parent **with**

| **None** -> rf(**"BUG: createParenthisizedLattice(): No Parent Symbol Table!"**)

| **Some**(parent) -> env.namesOfCurrLat <- parent);

env.labelIdGenerator <- (**List**.hd env.parentStackLabelIdGenerator);

env.parentStackLabelIdGenerator <- (**List**.tl env.parentStackLabelIdGenerator)

;;

**let** createParenthisizedLattice(lattice) =

popOldSymbolTable(env);

**let** parenthisizedLat = createNewLatFromLatEnvTuple(lattice, env.poppedSymbolTable) **in**

**let** latExpr = newExprFields(**match** parenthisizedLat.lattice **with**

| **Expr**(ex) -> ex.exprCode

| \_ -> **LatticeExpr**(parenthisizedLat)

) **in**

latExpr

*(\* >>>>> expr: <<<<< \*)*

**let** createBinOp(op1, op, op2) =

**OpExpr**({operator = op; operand1 = op1; operand2 = **Some**(op2)})

**let** createUnaryOp(\_operator, \_operand) =

**OpExpr**({operator = \_operator; operand1 = \_operand; operand2 = **None**})

**let** createParamLocal (\_id, \_paramList, (expr: **Ast**.exprFields)) =

addLocalToSymbolTable(env.namesOfCurrLat, \_id, **Some**(\_paramList), expr);

**let** newParamSeqlat = createEmptyLattice [createLatticeFieldsFromLatticeType(**Expr**(expr))] **in**

**let** innerSeqlat = (**match** newParamSeqlat.lattice **with**

| **Seqlat**(seqlat) ->

seqlat.symbols <- (addParamListToSeqlatSymbols(\_paramList, seqlat.symbols)); seqlat

| \_ -> rf(**"BUG: createParamLabeledAltlat()"**)

) **in**

newParamSeqlat.lattice <- **Seqlat**(innerSeqlat);

newParamSeqlat.paramNameList <- **Some**(\_paramList);

**let** newExpr = (createExprFieldsFromLatticeFields(newParamSeqlat)) **in**

newExprFields(createBinOp(newExprFields(**NameListExpr**({nameList = [**Id**(\_id)]})), **Define**, newExpr))

**let** createLocal ((\_id: string), (expr: **Ast**.exprFields)) =

addLocalToSymbolTable(env.namesOfCurrLat, \_id, **None**, expr);

newExprFields(createBinOp(newExprFields(**NameListExpr**({nameList = [**Id**(\_id)]})), **Define**, expr))

**let** createDirectAccessExprFields(\_lattice, \_nameList) =

**let** latticeExpr = createParenthisizedLattice(\_lattice) **in**

**let** seqlatExpr = **match** latticeExpr.exprCode **with**

| **LatticeExpr**(\_) -> latticeExpr

| \_ -> newExprFields(**LatticeExpr**(createEmptyLattice [createLatticeFieldsFromLatticeType (**Expr**(latticeExpr))]))

**in** newExprFields(createBinOp(seqlatExpr, **DirectAccess**, newExprFields(**NameListExpr**(\_nameList))))

*(\* >>>>> lvalue: <<<<< \*)*

*(\* >>>>> name: AND rest\_of\_name: <<<<< \*)*

**let** combineTwoNameLists(first, second) = first.nameList <- first.nameList @ second.nameList; first

*(\* >>>>> at\_operation: <<<<< \*)*

**let** createAtOperation (\_offsetExprFields: exprFields)=

**let** (offsetExprFields: exprFields) = \_offsetExprFields **in**

**let** currentIndex = newExprFields(**NameListExpr**({nameList = [**PredefinedLabel**(**INDEX**)]})) **in**

**let** calculatedIndex = newExprFields(createBinOp(currentIndex,**Plus**,offsetExprFields)) **in**

{ nameList = [**PredefinedLabel**(**SUBJECT**); **IndexingOp**(**UndeterminedStruct**, **Expr**(calculatedIndex))] }

**let** createAtOperationOfInt (\_offset: int) =

**let** (offsetExprFields: exprFields) = newExprFields(**ConstExpr**(**IntLiteral**(\_offset))) **in**

createAtOperation(offsetExprFields)

**let** createAtOperationOfId (\_offset: string) =

**let** (offsetExprFields: exprFields) = newExprFields(**NameListExpr**({nameList = [**Id**(\_offset)]})) **in**

createAtOperation(offsetExprFields)

**let** createAtOperationOfAltlat (\_offset: latticeType) =

**let** (offsetExprFields: exprFields) =

newExprFields(**LatticeExpr**(createLatticeFieldsFromLatticeType(\_offset))) **in**

createAtOperation(offsetExprFields)

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

**let** **rec** getSeqlatFromLatticeFields \_latticeFRef = **match** (!\_latticeFRef).lattice **with**

| **Seqlat**(seqlat) -> ref seqlat

| **Expr**(expr) -> (**match** expr.exprCode **with** **LatticeExpr**(lat) -> getSeqlatFromLatticeFields (ref lat) | \_ -> rf(**"BUG: getSymbolTableFromSeqlat()!"**))

| \_ -> rf(**"BUG: Expected Seqlat but not found!"**)

**let** **rec** setSeqlatFromLatticeFields(\_latticeFRef, \_seqlat) = **match** (!\_latticeFRef).lattice **with**

| **Seqlat**(seqlat) -> ((!\_latticeFRef).lattice <- **Seqlat**(!\_seqlat); \_latticeFRef)

| **Expr**(expr) -> (**match** expr.exprCode **with**

| **LatticeExpr**(lat) -> (**let** resLat = setSeqlatFromLatticeFields((ref lat), \_seqlat) **in**

expr.exprCode <- **LatticeExpr**(!resLat); (!\_latticeFRef).lattice <- **Expr**(expr); \_latticeFRef)

| \_ -> rf(**"BUG: getSymbolTableFromSeqlat()!"**))

| \_ -> rf(**"BUG: Expected Seqlat but not found!"**)

**let** getSeqlatElementsFromLatticeFields \_latticeF = **match** \_latticeF.lattice **with**

| **Seqlat**(seqlat) -> seqlat.elements

| \_ -> [\_latticeF]

## Scanner.ml Source Code

*(\*\**

*\*\* Lattakia Compiler*

*\*\* Columbia University COMS-4115*

*\*\* Programming Languages and Translators*

*\*\* Authors:*

*\*\* Wael Salloum*

*\*\*)*

{ **open** **Ast**

**open** **LatticeCreation**

**open** **Latparser**

**let** length=0;; **let** count=1;; **let** clone=2;; **let** reverse=3;; **let** return=4;; **let** loop=5;; **let** getlost=6;;

**let** labels=10;;

}

rule token = parse

[**' '** **'\t'**] { token lexbuf } *(\* Whitespace \*)*

| [**'\r'** **'\n'**] { incr\_linenum lexbuf; token lexbuf } *(\* Whitespace \*)*

| **".."** { singleLineComment lexbuf }

| **"\*\*"** { multiLineComment lexbuf }

| **'('** { pushNewSymbolTable(env); **LPAREN** } | **')'** { **RPAREN** } *(\* Punctuation \*)*

| **'{'** { **LBRACE** } | **'}'** { **RBRACE** }

| **'['** { **LSQBRACKET** } | **']'** { **RSQBRACKET** }

| **'^'** {**CARET**}

| **'+'** { **PLUS** } | **'-'** { **MINUS** }

| **'\*'** { **TIMES** } | **'/'** { **DIVIDE** }

| **'%'** { **PERCENT** }

| **'|'** { **PIPE** } | **'~'** { **TILDE** }

| **'!'** { **EXCLAMATION** } | **'?'** { **QUEST** }

| **';'** { **SEMI** } | **':'** { **COLON** }

| **'@'** { **AT** }

| **'.'** {**DOT**}

| **"def"** {**DEF**}

| **'='** { **ASSIGN** } | **"?="** { **QUEST\_ASSIGN** }

| **"=="** { **EQ** } | **"!="** { **NEQ** }

| **"&&"** { **AND** } | **"||"** { **OR** }

| **'<'** { **LT** } | **'>'** { **GT** }

| **"<="** { **LEQ** } | **">="** { **GEQ** }

| **"let"** { **LET** }

| **"true"** { **TRUE** } | **"false"** { **FALSE** }

| **"this"** { **THIS** } | **"nil"** { **NIL** }

| **"epsilon"** { **EPSILON** }

| **"length"** { **PREDEFINED\_LABEL**(length) } | **"count"** { **PREDEFINED\_LABEL**(count) }

| **"clone"** { **PREDEFINED\_LABEL**(clone) } | **"labels"** { **PREDEFINED\_LABEL**(labels) }

| **"return"** { **PREDEFINED\_LABEL**(return) } | **"loop"** { **PREDEFINED\_LABEL**(loop) }

| **"getlost"** { **PREDEFINED\_LABEL**(getlost) }

| eof { **EOF** }

| **'\"'**([^**'\"'**])\***'\"'** **as** lxm { **STRING\_LITERAL**(**String**.sub lxm 1 ((**String**.length lxm)-2)) }

| (**'-'**)?[**'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 { addToErrorTable(**LexUndefinedSymbol**(**Char**.escaped char), **None**); **NIL**}

**and** singleLineComment = parse

**"\n"** { token lexbuf }

| eof { **EOF** }

| \_ { singleLineComment lexbuf }

**and** multiLineComment = parse

**"\*\*"** { token lexbuf }

| eof { **EOF** }

| \_ { multiLineComment lexbuf }

## semanticAnalysis.ml Source Code

*(\*\**

*\*\* Lattakia Compiler*

*\*\* Columbia University COMS-4115*

*\*\* Programming Languages and Translators*

*\*\* Authors:*

*\*\* Wael Salloum*

*\*\* Katherine Scott*

*\*\*)*

*(\* Semantic Analysis \*)*

**open** **Ast**

**open** **LatticeCreation**

**open** **PrintLattice**

*(\**

*\*\*\* Basic concept here:*

*\*\*\* descend down the AST and drag the envType with us*

*\*\*\* - if we can find a check to do, do it, if we hit an error*

*\*\*\* add it to the environment*

*\*)*

*(\*\**

*\*\*\* FOR THE REPORT*

*\*\*\* Each lattakia type is made up of:*

*\*\*\* dataStructTypeType = dataTypeType \* structureTypeType*

*\*\*\* We need to have multiple levels of semantic checking*

*\*\*\* We have three primitive types (structureTypeType)*

*\*\*\* (seqlat, altlat, wrd) These can only do operations*

*\*\*\* between ones of the same type*

*\*\*\* i.e.*

*\*\*\* seqlat(op)seqlat ...*

*\*\*\**

*\*\*\* Additionally inside each primitive type there is what*

*\*\*\* we call a specific type (dataTypeType) which is int, float, string, boolean.*

*\*\*\* If the primitive type matches then we need to on and check that*

*\*\*\* each specific type also matches.*

*\*\*\**

*\*\*)*

**let** dataTypeToString \_dataType =

**match** \_dataType **with** **Int** -> **"int"** | **Float** -> **"float"** | **Boolean** -> **"boolean"** | **String** -> **"string"** | **General** -> **"general"**

| **UnresolvedType** -> **"unresolved"** | **Diverse** -> **"diverse"** | **MismatchType** -> **"mismatch"**

**let** structTypeToString \_dataType =

**match** \_dataType **with** **Seq** -> **"seqlat"** | **Alt** -> **"altlat"** | **Wrd** -> **"predicate"** | **UndeterminedStruct** -> **"undetermined"**

**let** dataStructTypeToString \_dataStructTuple = **"("**^(dataTypeToString(fst \_dataStructTuple))^**", "**^(structTypeToString(snd \_dataStructTuple)) ^ **")"**

*(\*------------------------------------------------------------------------------------------------\*)*

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

**type** symanticEnv = {

**mutable** currentLat: latticeFields;

**mutable** parentStack: latticeFields list;

}

**let** symEnv = {currentLat = **LatticeCreation**.createEmptyLattice []; parentStack = []}

**let** **rec** analyzeSemantics \_lattice = *(\*return an error string -if it is empty we are good\*)*

**let** latticeType = processLatticeReturnType \_lattice **in** (\_lattice, latticeType)

**and** processLatticeReturnType (\_lattice: latticeFields) = *(\*input a latticeFields, which is our AST\*)*

**let** latticeType = (**match** \_lattice.lattice **with**

| **Seqlat**(seqlat) -> (

symEnv.parentStack <- symEnv.currentLat :: symEnv.parentStack *(\*PUSH\*)*;

symEnv.currentLat <- \_lattice;

**let** seqlatString = processSeqlatReturnType seqlat **in**

symEnv.currentLat <- **List**.hd symEnv.parentStack *(\*POP\*)*;

symEnv.parentStack <- **List**.tl symEnv.parentStack *(\*Remove\*)*;

seqlatString

)

| **Altlat**(altlat) -> processAltlatReturnType altlat

| **Predicate**(pred) -> processPredicateReturnType pred

| **Expr**(expr) -> processExprReturnType expr

) **in** \_lattice.latDataStructType <- latticeType; latticeType

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

**and** processSeqlatReturnType \_seqlat = *(\*do the seq lat \*)*

**let** hd = (**List**.hd \_seqlat.elements) **in**

getLatticeListType(\_seqlat.elements, processLatticeReturnType hd, **Diverse**)

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

**and** processAltlatReturnType \_altlat = *(\*do the alt lat \*)*

**let** hd = (**List**.hd \_altlat.alternatives) **in**

**let** resultType = getLatticeListType(\_altlat.alternatives, processLatticeReturnType hd, **MismatchType**) **in**

**if** (fst resultType) = **MismatchType** **then** addToErrorTable(**SemTypeMismatch**, **None**) **else** (); resultType

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

*(\* go through a lattice list and make sure all of the types are the same \*)*

**and** getLatticeListType(\_latticeList, (\_prevType: dataStructTypeType), (\_mismatchType: dataTypeType)) =

**match** \_latticeList **with**

| [] -> \_prevType

| hd::tl -> **let** (hdType: dataStructTypeType) = processLatticeReturnType hd **in**

**if** ((fst \_prevType) = (fst hdType)) && ((snd \_prevType) = (snd hdType)) **then** (getLatticeListType(tl, hdType, \_mismatchType))

**else** (\_mismatchType, **UndeterminedStruct**)

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

**and** processPredicateReturnType \_pred =

*(\* Use the symbol table to compare predicate types - i.e no int < string \*)*

**let** condReturnType = (processExprReturnType \_pred.condition) **in**

**let** predReturnType = processExprReturnType \_pred.expr **in**

**match** condReturnType **with**

| (**Boolean**, **Wrd**) -> predReturnType

| (\_, \_) -> addToErrorTable(**SemPredicateTypeNotBoolean**, **None**); (**MismatchType**, **UndeterminedStruct**)

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

*(\* Do a expression return a type\*)*

**and** processExprReturnType (\_expr: exprFields) =

**let** exprType = (**match** \_expr.exprCode **with**

| **ConstExpr**(const) -> processConstReturnType const

| **OpExpr**(op) -> processOpReturnType op *(\*lookup from symbol table\*)*

| **NameListExpr**(nameList) -> processNameListReturnType(symEnv.currentLat, nameList.nameList)

| **LatticeExpr**(lattice) -> processLatticeReturnType lattice

) **in** \_expr.exprDataStructType <- exprType;

exprType

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

**and** isWordOperator = **function**

| **Plus** -> true | **Minus** -> true | **Times** -> true | **Divide** -> true | **Percent** -> true | **Or** -> true | **And** -> true | **Less** -> true

| **UniMinus** -> true | **Not** -> true | **LessOrEqual** -> true | **Greater** -> true | **GreaterOrEqual** -> true

| \_ -> false

*(\*Tilde | Eval | DontOpt | Let | DirectAccess | Assign | FuncCall | Define | Equal | NotEqual\*)*

**and** doCheckOperandsBeforeCheckingOperator = **function**

| **Plus** -> true | **Minus** -> true | **Times** -> true | **Divide** -> true | **Percent** -> true | **Or** -> true | **And** -> true | **Less** -> true

| **UniMinus** -> true | **Not** -> true | **LessOrEqual** -> true | **Greater** -> true | **GreaterOrEqual** -> true

| **Tilde** -> true | **Eval** -> true | **DontOpt** -> true | **Let** -> true | **DirectAccess** -> false | **Assign** -> true

| **FuncCall** -> true | **Define** -> false | **Equal** -> true | **NotEqual** -> true

*(\* given an operation expression return the type it \_should\_ return \*)*

**and** processOpReturnType op =

**let** op1type = **if** doCheckOperandsBeforeCheckingOperator op.operator **then** processExprReturnType op.operand1 **else** (**General**, **Wrd**) **in**

**let** op2type = **if** doCheckOperandsBeforeCheckingOperator op.operator **then**

(**match** op.operand2 **with**

| **None** -> (**General**, **Wrd**) *(\*Apply to all: most general case\*)*

| **Some**(opr) -> (processExprReturnType opr)) **else** (**General**, **Wrd**) **in**

**if** (isWordOperator op.operator)

&& (**match** (snd op1type, snd op2type) **with**

| (**Wrd**, **Wrd**) -> false *(\*No problem\*)*

| (\_,\_) -> true)

**then** (**MismatchType**, **UndeterminedStruct**)

**else**

(**match** op.operator **with**

*(\* Arithmatic Operations \*)*

| **Plus** -> (

**match** (fst op1type, fst op2type) **with**

| (**Int**, **Int**) -> (**Int**, **Wrd**)

| (**Float**, **Float**) -> (**Float**, **Wrd**)

| (**String**, **String**) -> (**String**, **Wrd**)

| (**General**, t) -> (t, **Wrd**)

| (t, **General**) -> (t, **Wrd**)

| (\_,\_) -> (**MismatchType**, **Wrd**)

)

| **Minus** -> (

**match** (fst op1type, fst op2type) **with**

| (**Int**, **Int**) -> (**Int**, **Wrd**)

| (**Float**, **Float**) -> (**Float**, **Wrd**)

| (**General**, t) -> (t, **Wrd**)

| (t, **General**) -> (t, **Wrd**)

| (\_,\_) -> (**MismatchType**, **Wrd**)

)

| **Times** -> (

**match** (fst op1type, fst op2type) **with**

| (**Int**, **Int**) -> (**Int**, **Wrd**)

| (**Float**, **Float**) -> (**Float**, **Wrd**)

| (**General**, t) -> (t, **Wrd**)

| (t, **General**) -> (t, **Wrd**)

| (\_,\_) -> (**MismatchType**, **Wrd**)

)

| **Divide** -> (

**match** (fst op1type, fst op2type) **with**

| (**Int**, **Int**) -> (**Int**, **Wrd**)

| (**Float**, **Float**) -> (**Float**, **Wrd**)

| (**General**, t) -> (t, **Wrd**)

| (t, **General**) -> (t, **Wrd**)

| (\_,\_) -> (**MismatchType**, **Wrd**)

)

| **Percent** -> (

**match** (fst op1type, fst op2type) **with**

| (**Int**, **Int**) -> (**Int**, **Wrd**)

| (**Float**, **Float**) -> (**Float**, **Wrd**)

| (**General**, t) -> (t, **Wrd**)

| (t, **General**) -> (t, **Wrd**)

| (\_,\_) -> (**MismatchType**, **Wrd**)

)

| **UniMinus** -> (

**match** (fst op1type) **with**

| (**Int**) -> (**Int**, **Wrd**)

| (**Float**) -> (**Float**, **Wrd**)

| (**General**) -> (**General**, **Wrd**)

| (\_) -> (**MismatchType**, **Wrd**)

)

*(\* Logical Operations \*)*

| **Or** -> (

**match** (fst op1type, fst op2type) **with**

| (**Boolean**, **Boolean**) -> (**Boolean**, **Wrd**)

| (**General**, t) -> (t, **Wrd**)

| (t, **General**) -> (t, **Wrd**)

| (\_,\_) -> (**MismatchType**, **Wrd**)

)

| **And** -> (

**match** (fst op1type, fst op2type) **with**

| (**Boolean**, **Boolean**) -> (**Boolean**, **Wrd**)

| (**General**, t) -> (t, **Wrd**)

| (t, **General**) -> (t, **Wrd**)

| (\_,\_) -> (**MismatchType**, **Wrd**)

)

| **Not** -> (

**match** (fst op1type) **with**

| (**Boolean**) -> (**Boolean**, **Wrd**)

| (**General**) -> (**General**, **Wrd**)

| (\_) -> (**MismatchType**, **Wrd**)

)

*(\* Comparison Operations \*)*

| **Equal** -> (

**match** (fst op1type, fst op2type) **with**

| (**Int**, **Int**) -> (**Boolean**, **Wrd**)

| (**Float**, **Float**) -> (**Boolean**, **Wrd**)

| (**String**, **String**) -> (**Boolean**, **Wrd**)

| (**General**, t) -> (**Boolean**, **Wrd**)

| (t, **General**) -> (**Boolean**, **Wrd**)

| (\_,\_) -> (**MismatchType**, **Wrd**)

)

| **NotEqual** -> (

**match** (fst op1type, fst op2type) **with**

| (**Int**, **Int**) -> (**Boolean**, **Wrd**)

| (**Float**, **Float**) -> (**Boolean**, **Wrd**)

| (**String**, **String**) -> (**Boolean**, **Wrd**)

| (**General**, t) -> (**Boolean**, **Wrd**)

| (t, **General**) -> (**Boolean**, **Wrd**)

| (\_,\_) -> (**MismatchType**, **Wrd**)

)

| **Less** -> (

**match** (fst op1type, fst op2type) **with**

| (**Int**, **Int**) -> (**Boolean**, **Wrd**)

| (**Float**, **Float**) -> (**Boolean**, **Wrd**)

| (**String**, **String**) -> (**Boolean**, **Wrd**)

| (**General**, t) -> (**Boolean**, **Wrd**)

| (t, **General**) -> (**Boolean**, **Wrd**)

| (\_,\_) -> (**MismatchType**, **Wrd**)

)

| **LessOrEqual** -> (

**match** (fst op1type, fst op2type) **with**

| (**Int**, **Int**) -> (**Boolean**, **Wrd**)

| (**Float**, **Float**) -> (**Boolean**, **Wrd**)

| (**String**, **String**) -> (**Boolean**, **Wrd**)

| (**General**, t) -> (**Boolean**, **Wrd**)

| (t, **General**) -> (**Boolean**, **Wrd**)

| (\_,\_) -> (**MismatchType**, **Wrd**)

)

| **Greater** -> (

**match** (fst op1type, fst op2type) **with**

| (**Int**, **Int**) -> (**Boolean**, **Wrd**)

| (**Float**, **Float**) -> (**Boolean**, **Wrd**)

| (**String**, **String**) -> (**Boolean**, **Wrd**)

| (**General**, t) -> (**Boolean**, **Wrd**)

| (t, **General**) -> (**Boolean**, **Wrd**)

| (\_,\_) -> (**MismatchType**, **Wrd**)

)

| **GreaterOrEqual** -> (

**match** (fst op1type, fst op2type) **with**

| (**Int**, **Int**) -> (**Boolean**, **Wrd**)

| (**Float**, **Float**) -> (**Boolean**, **Wrd**)

| (**String**, **String**) -> (**Boolean**, **Wrd**)

| (**General**, t) -> (**Boolean**, **Wrd**)

| (t, **General**) -> (**Boolean**, **Wrd**)

| (\_,\_) -> (**MismatchType**, **Wrd**)

)

*(\* Misc Operations: Tilde | Eval | DontOpt | Let | DirectAccess | Assign | FuncCall | Define | Equal | NotEqual\*)*

| **Tilde** ->

**let** structType = (**match** (snd op1type, snd op2type) **with**

| (**Wrd**, **Wrd**) -> **Seq**

| (**Wrd**, x) -> x

| (**Seq**, \_) -> **Seq**

| (**Alt**, \_) -> **Alt**

| (**UndeterminedStruct**, \_) -> **UndeterminedStruct**

) **in**

**let** dataType = (**match** (fst op1type, fst op2type) **with**

| (**Int**, **Int**) -> **Int**

| (**Float**, **Float**) -> **Float**

| (**String**, **String**) -> **String**

| (**Boolean**, **Boolean**) -> **Boolean**

| (**General**, t) -> t

| (t, **General**) -> t

| (**UnresolvedType**, \_) -> **UnresolvedType**

| (\_, **UnresolvedType**) -> **UnresolvedType**

| (**MismatchType**, \_) -> **MismatchType**

| (\_, **MismatchType**) -> **MismatchType**

| (\_,\_) -> **Diverse**

) **in**

(**match** (dataType, structType) **with**

| (**Diverse**, **Alt**) -> (**MismatchType**, **Alt**)

| (\_, \_) -> (dataType, structType))

| **Eval** -> (

**match** (fst op1type, snd op1type) **with**

| (x, **Wrd**) -> (x, **Wrd**)

| (x, **Seq**) -> (x, **UndeterminedStruct**)

| (x, **Alt**) -> (x, **UndeterminedStruct**)

| (\_,\_) -> (**MismatchType**, **Wrd**)

)

| **DontOpt** -> op1type

| **Let** -> (**General**, **Wrd**)

| **DirectAccess** ->

**let** nameList = (**match** (gs op.operand2 **""**).exprCode **with** **NameListExpr**(nameListEx) -> nameListEx.nameList | \_ -> rf(**"BUG: SemanticAnalysis.processOpReturnType(): DirectAccesss: nameList!"**)) **in**

**let** lattice = (**match** op.operand1.exprCode **with** **LatticeExpr**(lat) -> lat | \_ -> rf(**"BUG: SemanticAnalysis.processOpReturnType(): DirectAccesss: lattice!"**)) **in**

processNameListReturnType(lattice, nameList)

| **FuncCall** ->

**let** funcNameList = (**match** op.operand1.exprCode **with** **NameListExpr**(nameListEx) -> nameListEx.nameList | \_ -> rf(**"BUG: SemanticAnalysis.processOpReturnType(): FuncCall: funcLattice!"**)) **in**

**let** (idSymOpt, idLatOpt) = **try** **LatticeCreation**.findNameListSymbolInLatticeFields(symEnv.currentLat, funcNameList)

**with** **Not\_found** -> addToErrorTable(**SemUndefinedIdentifier**(**PrintLattice**.processNameList {parent = **None**; hash = **StringMap**.empty} funcNameList), **None**); (**None**, **None**)

**in**

(**match** idLatOpt **with**

| **None** -> (**UnresolvedType**, **UndeterminedStruct**)

| **Some**(funcLattice) -> (

**let** expr = (gs op.operand2 **""**) **in**

**let** lattice = (**match** expr.exprCode **with** **LatticeExpr**(lat) -> lat | \_ -> createEmptyLattice [createLatticeFieldsFromLatticeType (**Expr**(expr))]) **in**

**if** funcLattice.paramNameList = **None** **then** (**MismatchType**, **UndeterminedStruct**)

**else** **if** funcLattice.paramTypeList = **None** **then** (

**if** calcParamTypeList(lattice, funcLattice) **then**

processLatticeReturnType(funcLattice)

**else**

(**MismatchType**, **UndeterminedStruct**)

)

**else**

**if** checkParameterTypes(funcLattice.paramTypeList, lattice) **then**

funcLattice.latDataStructType

**else**

(**MismatchType**, **UndeterminedStruct**)

))

| **Assign** ->

**let** nameList = (**match** op.operand1.exprCode **with** **NameListExpr**(n) -> n.nameList | \_ -> rf(**"BUG: SymanticAnalysis: Assign: NameList-2!"**)) **in**

**let** (idSymOpt, idLatOpt) = **try** **LatticeCreation**.findNameListSymbolInLatticeFields(symEnv.currentLat, nameList)

**with** **Not\_found** -> addToErrorTable(**SemUndefinedIdentifier**(**PrintLattice**.processNameList {parent = **None**; hash = **StringMap**.empty} nameList), **None**); (**None**, **None**)

**in**

**let** idSymbolType = (**match** idSymOpt **with**

| **None** -> (**UnresolvedType**, **UndeterminedStruct**)

| **Some**(idSymbol) -> idSymbol.symDataStructType) **in**

**let** rvalueType =

(**match** (fst idSymbolType, fst op2type) **with**

| (**General**, t) -> (t, snd op2type)

| (t, **General**) -> (t, snd op2type)

| (x, y) -> **if** x = y **then** (x, snd op2type) **else** (**MismatchType**, snd op2type)

) **in** rvalueType

| **Define** ->

**let** idSymbolOpt = **LatticeCreation**.findInSymbolTable((**LatticeCreation**.getSymbolTableFromSeqlat symEnv.currentLat),

(**match** op.operand1.exprCode **with** **NameListExpr**(n) ->

(**match** **List**.hd n.nameList **with** **Id**(x) -> x | \_ -> rf(**"BUG: SymanticAnalysis: Define: NameList-1!"**))

| \_ -> rf(**"BUG: SymanticAnalysis: Define: NameList-2!"**))) **in**

**let** rvalueType = processExprReturnType (gs op.operand2 **""**) **in**

**match** idSymbolOpt **with**

| **None** -> rf(**"BUG: SymanticAnalysis: Define: NameList-3!"**)

| **Some**(idSymbol) -> (idSymbol.symDataStructType <- rvalueType; rvalueType)

)

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

**and** processConstReturnType = **function** *(\*return the type so we can do type checking\*)*

| **StringLiteral**(\_) -> (**String**, **Wrd**)

| **IntLiteral**(\_) -> (**Int**, **Wrd**)

| **FloatLiteral**(\_) -> (**Float**, **Wrd**)

| **Epsilon** -> (**General**, **Wrd**)

| **Nil** -> (**General**, **Wrd**)

| **True** -> (**Boolean**, **Wrd**)

| **False** -> (**Boolean**, **Wrd**)

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

*(\* make sure that x.y.z is in the right symbol table scope\*)*

**and** processNameListReturnType(\_lattice, \_nameList) =

**let** (idSymOpt, idLatOpt) = **try** **LatticeCreation**.findNameListSymbolInLatticeFields(\_lattice, \_nameList)

**with** **Not\_found** -> addToErrorTable(**SemUndefinedIdentifier**(**PrintLattice**.processNameList {parent = **None**; hash = **StringMap**.empty} \_nameList), **None**); (**None**, **None**)

**in**

**let** idSymbolType = (**match** idSymOpt **with**

| **None** -> (**UnresolvedType**, **UndeterminedStruct**)

| **Some**(idSymbol) -> idSymbol.symDataStructType

) **in** idSymbolType

*(\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*)*

**and** checkParameterTypes(\_paramTypeList, \_paramValueLattice) =

**let** elements = getSeqlatElementsFromLatticeFields \_paramValueLattice **in**

**List**.fold\_left2 (**fun** a b c -> a && (b == c.latDataStructType)) true (gs \_paramTypeList **"BUG: checkParameterTypes!"**) elements

**and** calcParamTypeList(\_paramLattice, \_funcLattice) =

**let** elements = getSeqlatElementsFromLatticeFields \_paramLattice **in**

**if** (**List**.length elements) <> (**List**.length (gs \_funcLattice.paramNameList **"BUG: calcParamTypeList"**)) **then** false

**else** **let** paramTypeList = **List**.map (**fun** x -> x.latDataStructType) elements **in**

\_funcLattice.paramTypeList <- **Some**(paramTypeList); true