NumLang

Coms W4115 - PLT
Final Project Report

Dan Aprahamian	dha2108
Damien Fenske-Corbiere	dpf2117
Siddhi Mittal	sm3210
Sahil Yakhmi	sy2348

1 Introduction

NumLang is a language designed to facilitate numerical computation. NumLang supports arithmetic operations without the loss of precision, computation with matrices, as well as user-defined mathematical functions to be manipulated, evaluated, and composed. Users can also create subroutines in NumLang and make use of the language's innovative match statements.

NumLang is designed to be intuitive, robust, and portable. It has a simple syntax that allows for most errors to be caught at compile time, and it compiles to Java source code. This makes it possible for NumLang to be used on any system where a Java Virtual Machine is available.

2 Language Tutorial

2.1 A short explanation telling a novice how to use your language

This is a short and friendly guide for NumLang:

2.1.1 Variables:

Declaration:

```
s = "hello";                    /* string of value "hello" */
x = 43;                         /* num of value 43 */
myl = [1,2,3];                  /* list of Num */
myll = [myl, [1,2]];                    /* list of list of num */
mat = m[0, 3, 2; 1, 4, 5]    ;  /* matrix  */
f = |x| -> |x^2|;               /* func */
diff = |x,y| -> |x-y|;                  /* func mapping two variables */
```

Notes:

- Once a variable is declared, it cannot be reassigned a new type.
- A variable may only be declared with a simultaneous assignment.
- Lists may only hold variables of the same type.
- Multiple-dimensional Lists must hold Lists of the same type, but can be jagged.
- A Matrix must be two-dimensionsal, can only hold Num values, and cannot be jagged.
- A Matrix is a separate type from a List. They are not compatibile

Assignment:

```
str = "now not hello";
str = 6;                        /* won't compile, invalid assignment */
myll[2][2] = 1;                 /* NumLang is 1-indexed, assigns 2nd element of 2nd list to 1
*/
myll[0][0] = 3;                 /* throws error, no 0 index */
```

```
mat[2][1] = 1;                  /* can only access individual elements of a matrix */
```

Built-in functions: log, ln, cos, sin, floor, ceil

```
lnx = ln(x);
```

Note: no variables or subroutines can be declared with the name of a built-in function

Manipulation:

```
x = myl[1] * myl[3];            /* basic arithmetic supported */
f = f(f(x));                    /* f=x^4 */
mat1 = m[1,2]
mat2 = m[0,0,2;6,9,2]
matr = mat1 # mat2              /*matrix multiplication*/
```

2.1.2 Subroutines:

Declaration:

```
sub callMe(num x, string list y) { y[x]; }
sub call2() { 34; }
```

Calling:

```
strList = [“fst”, “snd”];
str2 = callMe::(2, strList);
w = call2::();
```

Built-in Subroutines:

```
pop::(list L), rm::(list L), rmi::(num i, list L), len::(list L), str::(num N), str_func::(func F),
num::(string S), scanln::(), scan::(), println::(string S), print::(string S), m::(num r, num c)
```

None of these names can be used as identifiers in a NumLang program.

2.1.3 Match Statements:

```
match(w) {
        cont: w - (w % 10) ? {x = 1;}
        loop: > 22 ? {x =  - 1;}
        <= 12 % 4 ? pass;
        done: true ? pass;
}
```

2.1.4 IO:

```
input = scanln::(); /* Getting input from user*/
print::("str2"); /*Printing the value of str2*/
println::("str2"); /*Prints with a newline character*/
```

3 Language Manual

3.1 Group Members

Dan Aprahamian	dha2108
Damien Fenske-Corbiere	dpf2117
Siddhi Mittal	sm3210
Sahil Yakhmi	sy2348

3.2 Introduction

The **NUMLANG** programming language is designed to make numerical computation easy. One of the key features of this language is that it allows mathematical functions to be entered as literals. It allows computation with matrices and other common mathematical operations. The language is intended to be suitable for compilation as well as interpreting. The reference implementation is, however, a compiler.

3.3 Syntax notation

In the syntax notation used in the manual, syntactic categories are indicated by the *italic* type. Types and keywords are represented in **bold**.

3.4 Lexical Conventions

3.4.1) Comments

There is only one type of comment in this language, a block comment. A block comment is defined as anything in between the starting character sequence '/*', and the first occurrence of '*/'. Nothing within a comment is used by the compiler to generate code.

3.4.2) Identifiers

An identifier may be any alpha-numeric sequence of characters that begins with a letter character. An identifier terminates before the first white space character.

3.4.3) Keywords

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

match
done
cont
loop
any
pass
sub
const
->

3.5 Types

NUMLANG is a statically-typed language. It contains five fundamental types corresponding to the above literals. They are:

3.5.1 **num**

A rational number stored with arbitrary precision.

3.5.2 **string**

A sequence of zero or more ASCII characters.

3.5.3 **func**

A mathematical function representing a mapping from one or more numbers to single numerical value. The mapping will always provide a single output, except in the case of a mathematical error (divide by zero).

3.5.4 **list**

A **list** is a linear structure that can contain any type where every element needs to be of the same type. A multi-dimensional **list** is simply a **list** of variables of type **list**, and jagged multi-dimensional **list**'s are legal. **list**'s are indexed such that the first element has an index of 1 to follow common mathematical convention.

3.5.5 **matrix**

A **matrix** is a two-dimensional array-like data structure that contains only **num** types and has two dimensions with rows of consistent length.

3.5.6 Scalar vs. Non-scalar

In Numlang, **num**, **string**, and **func** are considered scalar types, while **list** and **matrix** are considered non-scalar. Non-scalar types may be included in left-hand-side expressions, and may contain scalar types; however, there is otherwise no semantic distinction between the two categories.

3.5.7 Type Conversions

There are no implicit type conversions in Numlang. Numerical values and string values can, however, be explicitly converted using the following built-in subroutines:

- str::(*num*|*func*|*list*|*matrix*)
  - Returns *a* **string** representing the passed in **num**, **func**, **list** or **matrix**.
- num::(*string*)
  - Returns the number value of a **string**

3.6 Literals

3.6.1 Numerical Literals

Numerical literals, corresponding to Numlang's **num** type, are specified as a sequence of decimal digits and at most one '.' character of arbitrary length. Optionally an 'E' character followed by a positive or negative integer exponent can be specified immediately following a numerical literal to multiply the number by 10 raised to the given integer exponent. A numerical literal may contain no white space. For example: '.00005332', '3234.0', '0.1', '1000.', '1.0E-4', and '.009E12' are valid numerical literals.

3.6.2 String Literals

A **string** literal is defined as anything between the first occurrence of a single quotation mark and the next occurrence of a single quotation mark that is not immediately preceded by a '\' escape character. Other special characters can also be escaped using the '\' character.

'\t'	tab character
'\r'	carriage return character
'\n'	newline character
'\\'	backslash
'\''	single quote character

3.6.3 Func Literals

func literals may be specified as literals using the following syntax:

function-literal:
| *function-parameter-list* | -> | function-expression |

function-parameter-list:
identifier
function-parameter-list, identifier

function-expression:
number-literal
identifier
(*function-expression*)
UNOP *function-expression*
function-expression BINOP *function-expression*
function-identifier(*function-expression*)

The operators allowed within *function-expression* is a subset of the operators in Numlang, including only numerical operators. Standard mathematical precedence rules apply. Additionally, certain function-identifiers are reserved and built-in to the Numlang language:

- log(x)
- ln(x)
- cos(x)
- sin(x)
- floor(x)
- ceil(x)

Examples:

- |x| -> |x + 1|
- |x, y| -> |x + y|
- |x, y, var1| -> |x + y / sin(var1)|

The **func**'s are mappings from the comma-delimited list of variables surrounded by '|' characters to a single numerical value, the value of the expression to the right of the -> keyword. The function expression is in a separate scope, such that identifiers in the *function-expression* are bound to identifiers in the *function-parameter-list* before other, previously declared variables.

3.6.4 List Literals

Variable-length **list'**s can be declared in-line by writing the comma-delimited element expressions in between brackets using the following syntax, where each expression in an list-expression-list must evaluate to the same type:

list-literal:
[ ]
[*expression-list*]

expression-list:
expression
expression-list, *expression*

Examples:
- [1, 2, 3, 4]
- [[1, 2, 3, 4], [5, 6, 7, 8], [1]]

Lists can be nested, and multi-dimensional lists may be jagged.

3.6.5 Matrices

matrix literals are specified starting with an m is directly before a left brace with no intervening white space. The matrix row element expression are comma-delimited and rows are separated with semicolons. A **matrix** may only contain **num** types, is required to have two dimensions, and must have rows of consistent length. **matrix** literals are specified with the following syntax:

matrix-literal:
m[matrix-row-list]

matrix-row-list:
matrix-row
matrix-row-list ; *matrix-row*

matrix-row:
expression
matrix-row, *expression*

In this grammar, the additional restrictions are applied that all expressions in *matrix-row* must evaluate to **num** types, and that all *matrix-row*s must contain the same number of comma-delimited expressions.

Examples:
- m[1, 2, 3 ; 1, 2, 3, 4] is invalid
- m[[[1, 2, 3], [1, 2, 3], [1, 2, 3]]; [[1, 2, 3], [1, 2, 3], [1, 2, 3]]; [[1, 2, 3], [1, 2, 3], [1, 2, 3]]] is invalid
- m[1, 2, 3; 1, 2, 3] is valid

Matrices can also be declared with default values by calling the built-in function m(rows, cols).

For example:
- m::(2, 3)

However, a **matrix** must always be two-dimensional.

3.7 Expressions

The precedence of expression operators is the same as the order of the major subsections of this section (highest precedence first). Within each subsection, the operators have the same precedence. Left- or right-associativity is specified in each subsection for the operators discussed therein. Otherwise the order of evaluation of expressions is undefined.

3.7.1 Unary operators

Expressions with unary operators group right-to-left.

3.7.1.1 − expression

This is the numerical negation operator.

- If expression evaluates to a **num**, the result is a **num** equal to the negative of the **num**.
- If expression evaluates to a **func**, the result is a **func** that represents the negation of the expression **func**.
- If expression evaluated to a **matrix**, the result is a **matrix** that represents the negated **matrix**.
- − applied to any other expression is illegal.

3.7.1.2 ! expression

This is the logical negation operator.

- If the expression evaluates to a **num**, the result is 0 if the **num** is non-zero, and 1 if the **num** is 0.
- If the expression evaluates to a **func**, the result is a new **func** that represents the logical negation of the **func**.
- ! applied to any other expression is illegal.

3.7.2 Multiplicative operators

The multiplicative operators *, / , and % group left-to-right.

3.7.2.1 expression * expression

The binary * operator indicates multiplication.

- If both expressions evaluate to **num** then the result is a **num**
- If either of the expression is a **matrix** and the other is **num**, the result is a matrix where each element is the element from the original **matrix** multiplied with **num**.
- If both operands are **func**, then the result is a **func**.
- If both expressions are of the type **matrix**, then the result is a **matrix** where each element at a location is the multiplication of elements from the original matrices at the same location, provided that the matrices are the same size.
- If one operand is a **num** and the other is **func**, the result is a **func**.
- * applied to any other pair of expressions is illegal.

3.7.2.2 expression / expression

The binary / operator indicates division.

- The same type considerations as for multiplication apply.
- For matrices this operation is element - wise.
- Attempting to divide by zero will also result in an error.

3.7.2.3 expression % expression

The binary % operator yields the remainder from the division of the first expression by the second.

- The same type considerations as for multiplication and division apply.

- For matrices this operation is element - wise.

3.7.2.4 expression # expression
The binary # operator yields the **matrix** multiplication of two matrices.
- If the first element is an nxm **matrix**, and the second element is an mxp **matrix**, then, then it returns an nxp **matrix** that is the result of the mathematical matrix multiplication.
- # applied to any other expression is illegal.

3.7.3 Additive operators
The additive operators + and − group left-to-right.
3.7.3.1 expression + expression
The result is the sum of the expressions.
- If both operands **num**, the result is also a **num**.
- If one operand is a **num**, and the other is a **func**, the result is a **func**.
- If both operands are **func**, the result is a **func**.
- If both expressions are of the type **matrix**, then the result is a **matrix** where each element at a location is the addition of elements from the original **matrices** at the same location, provided that the matrices are the same size.
- If one operand is a **num**, and the other is a **matrix**, then the result is a **matrix** with each element = *old-element + number*.
- No other type combinations are allowed.

3.7.3.2 expression − expression
The result is the difference of the expressions.
- If both operands **num**, the result is also a **num**.
- If one operand is a **num**, and the other is a **func**, the result is a **func**. If both operands are matrices, the result is a matrix if the operands are the same size, else an error occurs.
- If both operands are **func**, the result in a **func**.
- If the first element is a matrix, and the second element is a **num**, then the result is a new **matrix** where each element = *old-element - number* else if the first element is num and second element is a **matrix**, the result is the negated **matrix** plus the **num**.
- No other type combinations are allowed.

3.7.4) Exponential operator
The exponential operator ^ is right associative.
3.7.4.1 expression ^ expression
The result is first expression raised to the exponent of second expression.
- If both operands **num**, the result is also a **num**.
- If one operand is a **num**, and the other is a **func**, the result is a **func**.
- If both operands are **func**, the result is a **func**.
- If both operands are **matrix** then the result would be a **matrix** where each element at a location is the ^ of elements from the original **matrices** at the same location, provided that the matrices are the same size.
- If the first operand is a **matrix**, and the second is a **num**, then the result is a **matrix** with each element = *old-element ^ number*.
- If the first operand is a **num**, and the second is a **matrix**, then the result is a **matrix** with each element = *number ^ element at that location from matrix*.
- No other type combinations are allowed.

3.7.5) Relational operators

The relational operators group left-to-right, but this fact is not very useful; ‘‘a<b<c’’ does not mean what it seems to.

3.7.5.1 expression < expression

3.7.5.2 expression > expression

3.7.5.3 expression <= expression

3.7.5.4 expression >= expression

The operators < (less than), > (greater than), <= (less than or equal to) and >= (greater than or equal to) all yield 0 if the specified relation is false and 1 if it is true.

- Relational operators are only valid where the operands are either **num** or **func**.
- The result always is a **num**, unless one more more operand is a **func**, in which case the result is another **func**.

3.7.6) Equality operators

3.7.6.1 expression == expression

3.7.6.2 expression != expression

The == (equal to) and the != (not equal to) operators are exactly analogous to the relational operators except for their lower precedence. (Thus ‘‘a<b == c<d’’ is 1 whenever a<b and c<d have the same truth-value).

- Equality operators are only valid where the operands are either **num, string** or **func**.
- The result always is a **num**, unless one more more operand is a **func**, in which case the result is another **func**.
- If both the both the operands are **string**, the result is a **string**.
- No other combinations are possible.

3.7.7) Concatenation Operator

3.7.7.1 expression . expression

The concatenation operator ‘.’ is used to append the second expression to the first expression.

- If both operands are **strings** the result is always a **string**.
- If both operands are **list**s the result is always a new **list.**

3.7.8) Assignment operators

There is only one assignment operator, which groups right-to-left. It requires an lvalue (variable or list/matrix element) as its left operand. The value of the evaluated expression (right operand) is the value stored in the left operand after the assignment has taken place.

7.8.1 lvalue = expression

The value of the expression replaces the value stored in lvalue.

lvalue can either be an identifier representing a new or previously declared variable, or a **list** or **matrix** element.

For example:

- myVar = 0;
- myList[1] = 0;
- myMatrix[1][1] = 0;

3.8) Declarations and Initializations:
Variables in Numlang are statically typed; however, the type of a variable need not be explicitly specified. Rather, the first assignment determines the type of a variable. There can never be an uninitialized variable in Numlang.

3.8.1) Declaring and initializing a scalar:
A variable is declared when it is first assigned a value. Until this time, a variable may not be referenced in an expression. For example:

var1 = *num expr*; /* Declares *lvalue1* as a **num** and assigns value **num** */
var2 = *string expr*;/* Declares *lvalue2* as a **string** and assigns value **string** */
var3 = *func expr*; /* Declares *lvalue3* as a **func** and assigns value **func** */

An undeclared variable can also be declared as constant as such:

const *lvalue1* = *num*; /* Declares *lvalue1* as a **num**, and assigns value **num**. *lvalue1* can no longer change its value */

As long as a scalar has not been declared as const, its value can be changed. It cannot, however, be assigned a value that is of a different type than its first value:

var = 3; /* Declares *lvalue* as **num**, assigns 3 */
var = *lvalue* + 1; /* Assigns *lvalue* + 1 */
var = -1; /* Assigns -1 */
var = 'foo'; /* ERROR */
var = (x)->(x + 1); /* ERROR */
const *var* = 3; /* ERROR: variable cannot be re-declared as const */
const *var2* = |x| -> |x / 2|; /* Declares var2, assigns **func** value */
var2 = |x| -> |x / 3|; /* ERROR: const variable cannot be modified */

3.8.2) Declaring and initializing a list:
A **list** is a linear sequence of values. Each value can be a *scalar*, **matrix**, or another **list**, but all values contained in a list must be of the same type. The syntax for assigning a new or previously declared **list** variable is as follows.

lvalue = *list-literal*;

Example:

list1 = [1, 2, 3, 4]; /* Declares a size-4 **list** with the given values */

/* Declares a **list** of lists with the given values*/
list1 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]];

3.8.3) Declaring a matrix:
A **matrix** is akin to a mathematical matrix, a two-dimensional representation of a list of equally-sized vectors.

lvalue = *matrix-literal*;
-or-
lvalue = m(rows, cols);

3.8.4) Declaring a subroutine
To declare a subroutine:

subroutine:
sub *subroutine-name* **(***parameter-list***)** *statement*

parameter list:
parameter-type parameter-name
*parameter-name***,** *parameter-list*

3.9) List and Matrix accesses

list and **matrix** accesses can serve both as expressions as well as the left-hand side of an assignment.

3.9.1 List accesses
An element of a **list** may be accessed by appending an index within brackets to a **list** identifier with no intervening whitespace. Indices may be chained in order to perform accesses on multi-dimensional **list**s. The validity of a **list** access can generally not be determined until runtime.

list-access:
identifier [*integer-expression*]
list-access [*integer-expression*]

Examples:
- myList[1]
- myList[1][1][1]

3.9.2 Matrix accesses
An element of a **matrix** may be accessed by specifying two indices, each within brackets, directly following the identifier referring to the **matrix**, with no intervening whitespace. A **matrix** access will only result in an error if one or both of the indices are out of bounds.

matrix-access:
identifier [*integer-expression*] [*integer-expression*]

Examples:
- myMatrix[1][1]

3.10) Statements:
The program is made up of a series of statements. A statement is in the following format:
statement:
expression-statement
block-statement
match-statement
null-statement

Unless otherwise specified, in general statements are executed sequentially.

3.10.1) Expression Statement:
An expression statement simply consists of an expression and an expression terminator. Most statements are expression statements.

expression-statement:
expression**;**

3.10.2) Block Statement:
Block Statements allow one to group multiple statements into one statement, useful for when only one statement is expected. The Block Statement is defined as follows:

block-statement:
{statement-list**}**

statement-list:
ε
statement statement-list

3.10.3) Match Statement:
Match Statements are used for control flow. They incorporate features normally found in languages in if, switch, and while statements. They are defined as follows:

match-statement:
match($expression_a$**){***match-list***}**

match-list:
ε
match-command match-list

match-command:
flow-type match-condition **?** statement

flow-type:
ε
cont:
done:
loop:

match-condition:
$expression_b$
match-comparator $expression_b$
match-type

match-comparator:
>
>=
<
<=
!=

match-type:
SCALAR
STRING
FUNC
TRUE
ANY

The way the match works is as follows:

1) Start
2) For each *match-command* in the *match-list*, do the following:
   a. Determine if the condition matches
      i. If the -match-condition is $expression_b$, the condition matches if $expression_a == expression_b$
      ii. If the match-condition is *match-comparator* $expression_b$, the condition matches if **(**$expression_a$ *match-comparator* $expression_b$**)!= 0**
      iii. If the match-condition is a *match-type*, the condition matches in the following cases:
         1. **NUM**: $expression_a$ returns a **num**
         2. **STRING**: $expression_a$ returns a **string**
         3. **FUNC**: $expression_a$ returns a **func**
         4. **TRUE**: $expression_a$ returns a non-zero value
         5. **ANY**: always matches
   b. If the condition matches, do the following:
      i. Perform the *statement*
      ii. Depending on the *flow-type*, do the following:
         1. **cont:**: proceed to the next iteration of Step 2.
         2. **done:**: proceed to step 3
         3. **loop:**: proceed to step 1
         4. **ε:**: treat as cont
3) Finish

3.10.4) Null Statement:

The Null Statement is useful for places where you need a placeholder that does nothing. It is defined as follows:

null-statement:
pass;

3.11) Scope rules

Variables declared in the top level of a file are in the global scope. Otherwise, the language implements block level scope. For example, if a variable is first declared in a match statement, it will not be accessible once the match statement has finished.

A subroutine may only be declared in the top level of the program, and cannot be nested within another subroutine.

3.12) More on Types

3.12.1) Scalar Types

3.12.1.1. **num**

- A **num** is a basic floating point or integer number. Basic arithmetic rules apply.
  - a + b: add b to a
  - a - b: subtract b from a
  - a * b: multiply a by b
  - a / b: divide a by b. b cannot equal 0
  - a % b: returns the remainder of a / b. b cannot equal 0.
  - -a: returns the negative value of a.
- In addition, **num** is also used as boolean type. 0 is false, non-zero is true.
  - Integer to boolean operations
    - a == b: returns 1 if a is equal to b, 0 otherwise
    - a != b: returns 0 if a is equal to b, 1 otherwise
    - a > b: returns 1 if a is greater than b, 0 otherwise
    - a >= b: returns 1 if a is greater than or equal to b, 0 otherwise
    - a < b: returns 1 if a is less than b, 0 otherwise
    - a <= b: returns 1 if a is less than or equal to b, 0 otherwise
  - Boolean to boolean operations
    - !a: returns 0 if 1, 1 if 0
    - to achieve AND and OR operations, use * and + respectively
      - ex: a + b === a OR b
      - ex: a * b === a AND b

3.12.1.2**string**

- A **string** is a series of characters (ex: "Hello", "Goodbye")

3.12.1.3**func**

- A **func** is a mathematical function that takes in certain values and returns a **num**
- Literal: (*input-params*) -> *function-of-input-params*
  - Ex: (x) -> 2x + 3;
- Assigning function to variable: *lvalue* = *literal*
  - Ex: f = (x) -> 2x + 3;
- Evaluating function at value: *function*(*value*)
  - Ex: f(3); /*Returns 9*/
- Operators on functions all return new functions that combine both operands.
  - Valid operations: +, -, *, /, %, >, >=, <, <=, ==, !=
  - Ex:
    - f = |x| -> |x + 1|;
    - g = f + 1; /* g == |x| -> |x + 1 + 1| */
    - h = f * g; /* h == |x, y| -> |(x + 1) * (y + 2)| */
- Can combine functions
  - Ex:
    - f = |x| -> |2x - 3|;
    - g = |x| -> |x + 1|;
    - h = f(g); /* h == |x|-> |2(x + 1) - 3|*/

3.12.4) Subroutines

- Subroutines must be defined on a global level.
- Defining a subroutine: sub *subroutine-name* (*parameter-list*) *statement*
- Ex:
  - sub mySum(num a, num b) return a + b;

- sub lotsofstuff(num a, num b, num c)
  {

      a = b + c;
      b = b + b;
      c = a + a;
      return c;   }
- mySum(5,2); /*Calling the subroutine mySum*/
- lotsofstuff(1,2,3) /*Calling the subroutine lotsofstuff*/

3.13) Input and Output

3.13.1 Printing
The built-in subroutines print::(string|num|func|list|matrix)) and println::(string|num|func|list|matrix) are used for printing to the console.

3.13.2 Scanning
The built-in subroutine scanln::() is used for getting input from the console. This subroutine returns a value of type **string**, corresponding to all inputted characters until the next newline.
The built-in subroutine scan::() also gets input from the console, and returns a value of type **string** corresponding to to all inputted characters until the next whitespace.

3.14) Reserved Subroutines

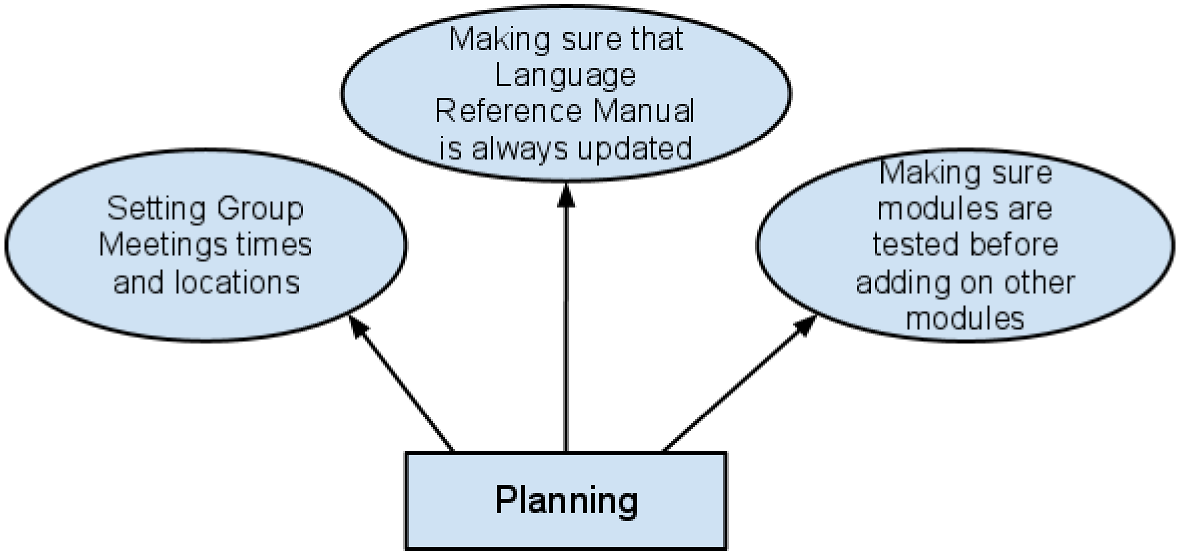
The names of the following built-in subroutines are reserved and a program that attempts to use any of them as an identifier will not compile:

pop::(list L) removes the first element of L and returns it.
rm::(list L) removes the last element of L and returns it.
rmi::(num i, list L) removes the ith element of L (1-indexed) and returns it.
len::(list L) returns the length of L as a **num**.
str::(num N) returns the **string** representation of N.
str_func::(func F) returns the **string** representation of F.
num::(string S) returns the **num** representation of S.
scanln::() returns the subsequent **string** terminating in a newline character from standard input.
scan::() returns the subsequent whitespace delimited token from standard input as a **string**.
println::(string S) prints S to standard output followed by a newline character.
print::(string S) prints S to standard output.
m::(num r, num c) returns a **matrix** of r rows and c columns with the value at every index set to zero.

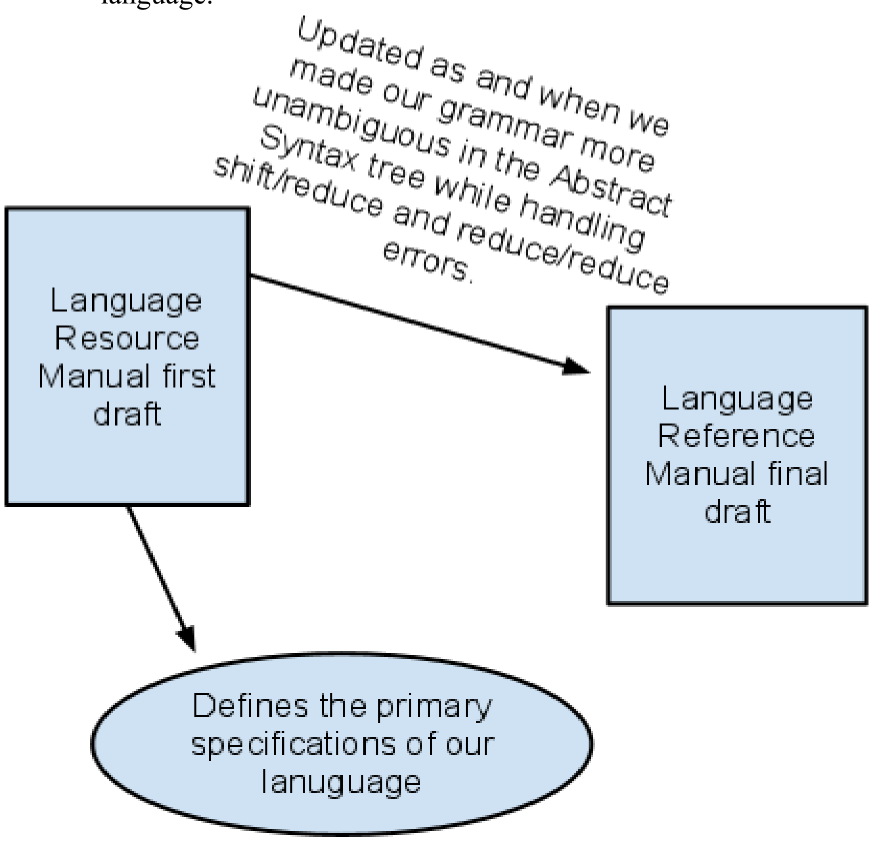
4 Project Plan

4.1 Identify process used for planning, specification, development and testing

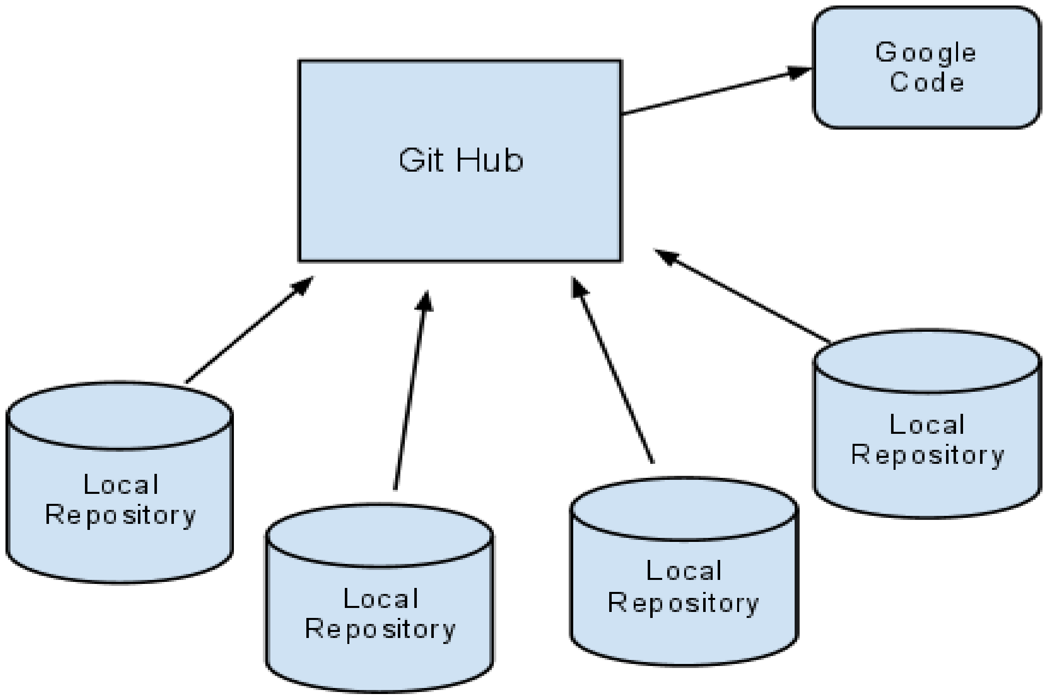
4.1.1 Planning - This is the most important part of the entire project. The main things we focused on are shown below.



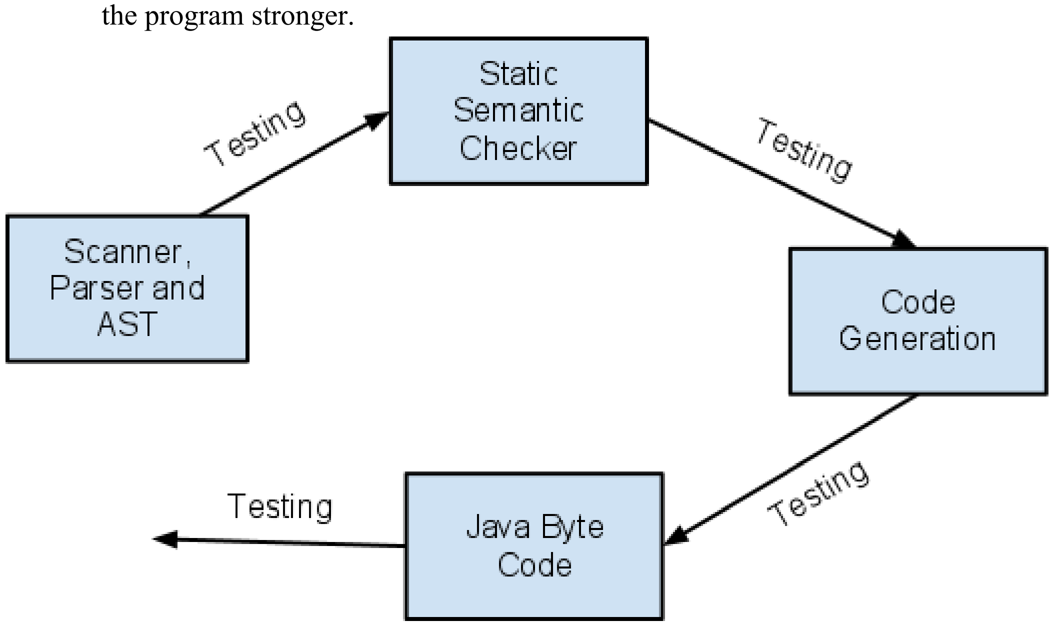
4.1.2 Specification - The Language Reference Manual contains all the specifications of the language.



4.1.3 Development - The following is the development environment that we used.



4.1.4 Testing Testing has been adopted at every stage between every Module. Alongwith that testing also occurs after our finished compiler to keep flushing out the inconsistencies and making the program stronger.



4.2 Include a programming style guide used by the team
Our language has a very similar programming style to C.

4.2.1 Indentation - Assists in identifying control flow, blocks of code and the meaning of the program. This is a matter of style and not a strict enforced structure.
4.2.2 Vertical Alignment - Aligning similar objects vertically. This again like C, is a matter of

style and not a strict enforced structure.
4.2.3 Spaces - This is again a stylistic choice. It is used to enhance readibility.

4.3 Show your project timeline
The following timeline was set for this project.

September 24	Concept of Language and features discussed
September 26	Language Whitepaper, core languages features defined
October 31	Language Reference Manual and Grammar complete
November 15	Development environment setup/ Future team Meetings timeline decided
November 21	Scanner, Parser and AST complete
December 5	Static Sematic Checker and SAST complete
December 13	Code Generation working
December 17	Project Complete

4.4 Identify roles and responsibilities of each team member
Along with the following primary responsibilities, all the team members were responsible for debugging and testing individually and as a group.

Dan Aprahamian	Java Run time Code implementation and Ocaml Compiler
Damien Fenske-Corbiere	Ocaml (Static Semantic Checker and Compiler)
Sahil Yakmi	Ocaml (Static Semantic Checker and Compiler)
Siddhi Mittal	Ocaml (Static Semantic Checker and Compiler)

4.5 Describe the software development environment used (tools and languages)
The following tools and languages have been used in this project -

4.5.1 Git Hub - is the version control system that we used. We used this in conjunction with Google Code to make sure that all versions of code were getting stored, were visible to everyone at all times and were easily trackable.
4.5.2 Ocamlyacc - This was used to write the AST
4.5.3 Ocaml - This language is used to scan, parse, static semantically check and compile the program into Java Source Code.
4.5.4 Java - This language is used to take in all the produced source code from Ocaml and convert it to Java ByteCode.
4.5.5 Google Documents - This application has been used extensively for collaborative coding in this project where 2 or more members were working on the same file at the same time.
4.5.6 Eclipse - To test our programs by running compiled Java Byte Code

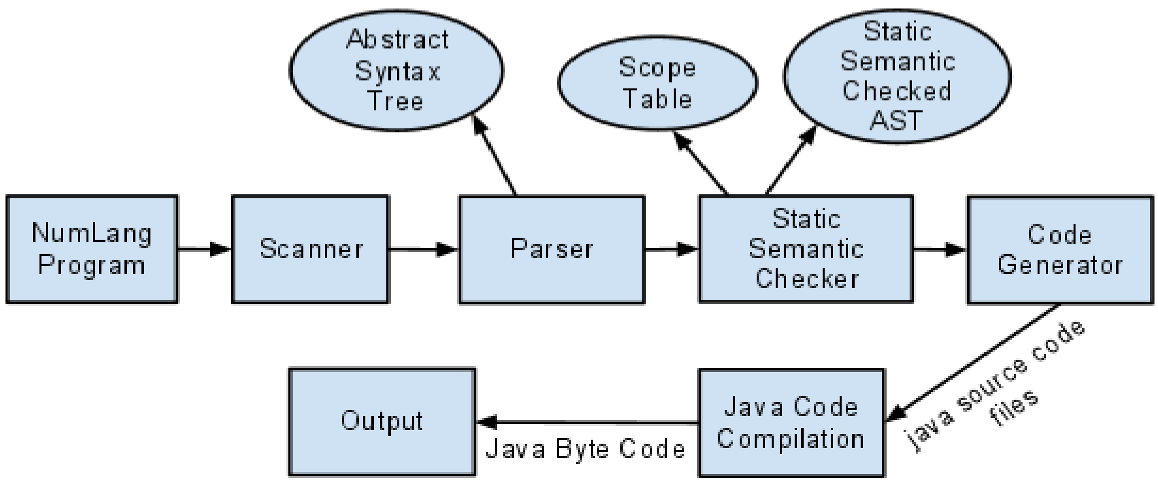
4.6 Include your project log
The following timeline was set and followed for this project.

September 18	Conversations initiated. First group meeting time decided.
September 24	Concept of Language and features discussed
September 26	Core languages features defined
October 31	Language Reference Manual First Draft
November 15	Development environment setup/ Future team Meetings timeline decided

November 21	AST, Scanner and Parser created
November 24	Scanner, Parser and AST complete
December 5	Language Reference Manual Updated
December 8	Static Semantic Checker created
December 9	Java Run Time classes created
December 10	SAST Created
December 12	Static Semantic Checker Debugging
December 15	Static Semantic Checker Debugging
December 17	Static Semantic Checker Compiled
December 20	Compiler created
December 21	Compiler Debugging and Code Generation
December 22	Code Generation and Rigorous Testing

5 Architectural Design

5.1 Give block diagram showing the major components of your translator



5.2 Describe the interfaces between the components
Below is the description of interfaces between all the components of the compiler for NumLang.

5.2.1 Scanner and Parser The Scanner scans the inputted program and sends the result to the parser which creates a Abstract Syntax tree and removes all the useless tokens retaining just the right amount of information needed to process the program.
5.2.2 Parser and Static Semantic Checker The Parser then sends the code to the Static Semantic Checker which checks for all the types of the functions, expressions, statements and makes sure nothing illegal is getting passed. It then creates another Static Semantic Abstract Syntax Tree which stores this additional information of their types.
5.3.3 Static Semantic Checker and Code Generator The input from Static Semantic Checker then goes through the Code Generator which converts all the Ocaml code into Java Source Code.
5.3.4 Code Generator and Java Code All the Java Source code is then compiled to Java Byte Code and an output is produced.

5.3 State who implemented each component

We took a slightly different approach to dividing the modules in this assignment. instead of diving roles by modules, we divided roles primarily by the language the code was to be written in - Java and Ocaml.

5.3.1 Dan primarily handled the Java code responsible for compiling all Java Source Code into Java Byte Code along with contribution to the Scanner, Parser and the AST.
5.3.2 Damien, Sahil and Siddhi adopted the method of collaborative coding using Google Docs for the Scanner, Parser, AST, Static Semantic Checker and Code Generator since all these modules depend on each other and we wanted one module to completely work before moving onto the next.

6 Test Plan

6.1 Show two or three representative source language programs along with the target language program generated for each

Program 1

NumLang Program -

```
sub mySub(num x)
{
        x + 1;
}

sub world_this_string(string x)
{
        x . " World";
}

sub mod_func_plus_x(func x)
{
        temp = |x| -> |x|;
        x + temp;
}

sub gettwobytwomatrixofones()
{
        mat = m::(2, 2);
        mat[1][1] = 1;
        mat[1][2] = 1;
        mat[2][1] = 1;
        mat[2][2] = 1;
        mat;
}
mm = gettwobytwomatrixofones::();
println::("" . str::(mm[1][1]) . ", " . str::(mm[1][2]) . ", " . str::(mm[2][1]) . ", " . str::(mm[2][2]));
println::("Done");
```

Java Source Code Program -

```
import com.numlang.*;
```

```
public class Runner
{
public static void main(String[] args)
{
NumLang.init();
final Subroutine mySub = new Subroutine() {
public Object invoke(Object... args) {
final Var<NumValue> _x = new Var<NumValue>((NumValue) args[0] );
return (_x.value().add((new NumValue(new BigRational("1")))));
}
};
final Subroutine world_this_string = new Subroutine() {
public Object invoke(Object... args) {
final Var<StringValue> _x = new Var<StringValue>((StringValue) args[0] );
return (_x.value().concat((new StringValue(" World"))));
}
};
final Subroutine mod_func_plus_x = new Subroutine() {
public Object invoke(Object... args) {
final Var<FuncValue> _x = new Var<FuncValue>((FuncValue) args[0] );
final Var<FuncValue> _temp = new Var<FuncValue>((FuncValue)(new FuncValue (1,(new
FuncValue.Func(0)))));
return (_x.value().add(_temp.value()));
}
};
final Subroutine gettwobytwomatrixofones = new Subroutine() {
public Object invoke(Object... args) {
final Var<MatrixValue> _mat = new Var<MatrixValue>((MatrixValue)(new MatrixValue((new
NumValue(new BigRational("2"))), (new NumValue(new BigRational("2"))))));
_mat.value().set(((NumValue)(new NumValue(new BigRational("1")))).subtract(new
NumValue(new BigRational(1))), ((NumValue)(new NumValue(new BigRational("1")))).subtract(new
NumValue(new BigRational(1))), ((NumValue)(new NumValue(new BigRational("1"))))
);_mat.value().set(((NumValue)(new NumValue(new BigRational("2")))).subtract(new NumValue(new
BigRational(1))), ((NumValue)(new NumValue(new BigRational("1")))).subtract(new NumValue(new
BigRational(1))), ((NumValue)(new NumValue(new BigRational("1")))));_mat.value().set(((NumValue)
(new NumValue(new BigRational("1")))).subtract(new NumValue(new BigRational(1))), ((NumValue)
(new NumValue(new BigRational("2")))).subtract(new NumValue(new BigRational(1))), ((NumValue)
(new NumValue(new BigRational("1")))));_mat.value().set(((NumValue)(new NumValue(new
BigRational("2")))).subtract(new NumValue(new BigRational(1))), ((NumValue)(new NumValue(new
BigRational("2")))).subtract(new NumValue(new BigRational(1))), ((NumValue)(new NumValue(new
BigRational("1")))));return _mat.value();
}
};
final Var<MatrixValue> mm = new Var<MatrixValue>((MatrixValue)
gettwobytwomatrixofones.invoke());
NumLang.IO.println(new StringValue((((((((new StringValue("")).concat((new
StringValue(mm.value().get(((NumValue)(new NumValue(new BigRational("1")))).subtract(new
NumValue(new BigRational(1))), ((NumValue)(new NumValue(new BigRational("1")))
).subtract(new NumValue(new BigRational(1)))))))).concat((new StringValue(", ")))).concat((new
StringValue(mm.value().get(((NumValue)(new NumValue(new BigRational("1")))).subtract(new
NumValue(new BigRational(1))), ((NumValue)(new NumValue(new BigRational("2")))
).subtract(new NumValue(new BigRational(1)))))))).concat((new StringValue(", ")))).concat((new
StringValue(mm.value().get(((NumValue)(new NumValue(new BigRational("2")))).subtract(new
NumValue(new BigRational(1))), ((NumValue)(new NumValue(new BigRational("1")))
).subtract(new NumValue(new BigRational(1)))))))).concat((new StringValue(", ")))).concat((new
```

```
StringValue(mm.value().get(((NumValue)(new NumValue(new BigRational("2")))).subtract(new
NumValue(new BigRational(1))), ((NumValue)(new NumValue(new BigRational("2")))).subtract(new
NumValue(new BigRational(1))))))))));
	NumLang.IO.println(new StringValue((new StringValue("Done"))));
	}
	}
```

Output of program:

1, 1, 1, 1
Done

Program 2

NumLang Program -

```
	x = -10;
	y = 1;
	match(x)
	{
		any ? match(y)
		{
			any ? println::("" . str::(x) . " + " . str::(y) . " = " . str::(x + y));
			any ? println::("" . str::(x) . " - " . str::(y) . " = " . str::(x - y));
			any ? println::("" . str::(x) . " * " . str::(y) . " = " . str::(x * y));
			any ? println::("" . str::(x) . " / " . str::(y) . " = " . str::(x / y));
			any ? println::("" . str::(x) . " ^ " . str::(y) . " = " . str::(x ^ y));
			any ? println::("" . str::(x) . " % " . str::(y) . " = " . str::(x % y));
			any ? println::("" . str::(x) . " == " . str::(y) . " = " . str::(x == y));
			any ? println::("" . str::(x) . " != " . str::(y) . " = " . str::(x != y));
			any ? println::("" . str::(x) . " < " . str::(y) . " = " . str::(x < y));
			any ? println::("" . str::(x) . " <= " . str::(y) . " = " . str::(x <= y));
			any ? println::("" . str::(x) . " > " . str::(y) . " = " . str::(x > y));
			any ? println::("" . str::(x) . " >= " . str::(y) . " = " . str::(x >= y));
			loop: < 10 ? y = y + 1;
		}
		loop: < 10 ? x = x + 1;
	}
```

Java Source Code Program -

```
	import com.numlang.*;

	public class Runner
	{
	public static void main(String[] args)
	{
	NumLang.init();
	final Var<NumValue> x = new Var<NumValue>((NumValue)((new NumValue(new
BigRational("10"))).neg()));
	final Var<NumValue> y = new Var<NumValue>((NumValue)(new NumValue(new
BigRational("1"))));
	while(true){
	{while(true){
	{NumLang.IO.println(new StringValue(((((((new StringValue("")).concat((new
StringValue(x.value())))).concat((new StringValue(" + ")))).concat((new StringValue(y.value())))
).concat((new StringValue(" = ")))).concat((new StringValue((x.value().add(y.value())))))))));
```

```
            }
            {NumLang.IO.println(new StringValue(((((((new StringValue("")).concat((new
StringValue(x.value())))).concat((new StringValue(" - ")))).concat((new StringValue(y.value())))
).concat((new StringValue(" = ")))).concat((new StringValue((x.value().subtract(y.value())))))))));
            }
            {NumLang.IO.println(new StringValue(((((((new StringValue("")).concat((new
StringValue(x.value())))).concat((new StringValue(" * ")))).concat((new StringValue(y.value())))
).concat((new StringValue(" = ")))).concat((new StringValue((x.value().multiply(y.value())))))))));
            }
            {NumLang.IO.println(new StringValue(((((((new StringValue("")).concat((new
StringValue(x.value())))).concat((new StringValue(" / ")))).concat((new StringValue(y.value())))
).concat((new StringValue(" = ")))).concat((new StringValue((x.value().divide(y.value())))))))));
            }
            {NumLang.IO.println(new StringValue(((((((new StringValue("")).concat((new
StringValue(x.value())))).concat((new StringValue(" ^ ")))).concat((new StringValue(y.value())))
).concat((new StringValue(" = ")))).concat((new StringValue((x.value().exp(y.value())))))))));
            }
            {NumLang.IO.println(new StringValue(((((((new StringValue("")).concat((new
StringValue(x.value())))).concat((new StringValue(" % ")))).concat((new StringValue(y.value())))
).concat((new StringValue(" = ")))).concat((new StringValue((x.value().mod(y.value())))))))));
            }
            {NumLang.IO.println(new StringValue(((((((new StringValue("")).concat((new
StringValue(x.value())))).concat((new StringValue(" == ")))).concat((new StringValue(y.value())))
).concat((new StringValue(" = ")))).concat((new StringValue((x.value().eq(y.value())))))))));
            }
            {NumLang.IO.println(new StringValue(((((((new StringValue("")).concat((new
StringValue(x.value())))).concat((new StringValue(" != ")))).concat((new StringValue(y.value())))
).concat((new StringValue(" = ")))).concat((new StringValue((x.value().neq(y.value())))))))));
            }
            {NumLang.IO.println(new StringValue(((((((new StringValue("")).concat((new
StringValue(x.value())))).concat((new StringValue(" < ")))).concat((new StringValue(y.value())))
).concat((new StringValue(" = ")))).concat((new StringValue((x.value().lt(y.value())))))))));
            }
            {NumLang.IO.println(new StringValue(((((((new StringValue("")).concat((new
StringValue(x.value())))).concat((new StringValue(" <= ")))).concat((new StringValue(y.value())))
).concat((new StringValue(" = ")))).concat((new StringValue((x.value().leq(y.value())))))))));
            }
            {NumLang.IO.println(new StringValue(((((((new StringValue("")).concat((new
StringValue(x.value())))).concat((new StringValue(" > ")))).concat((new StringValue(y.value())))
).concat((new StringValue(" = ")))).concat((new StringValue((x.value().gt(y.value())))))))));
            }
            {NumLang.IO.println(new StringValue(((((((new StringValue("")).concat((new
StringValue(x.value())))).concat((new StringValue(" >= ")))).concat((new StringValue(y.value())))
).concat((new StringValue(" = ")))).concat((new StringValue((x.value().geq(y.value())))))))));
            }
            if(!(y.value().lt((new NumValue(new BigRational("10")))).getValue().isZero())){
            y.assign((NumValue)(y.value().add((new NumValue(new BigRational("1"))))));
            continue;}
            break;
            }
            }
            if(!(x.value().lt((new NumValue(new BigRational("10")))).getValue().isZero())){
            x.assign((NumValue)(x.value().add((new NumValue(new BigRational("1"))))));
            continue;}
            break;
            }
```

```
        }
        }
```

Output:

```
-10 + 1 = -9
-10 - 1 = -11
-10 * 1 = -10
-10 / 1 = -10
-10 ^ 1 = -10
-10 % 1 = 1
-10 == 1 = 0
-10 != 1 = 1
-10 < 1 = 1
-10 <= 1 = 1
-10 > 1 = 0
-10 >= 1 = 0
-10 + 2 = -8
-10 - 2 = -12
-10 * 2 = -20
-10 / 2 = -5
-10 ^ 2 = 100
-10 % 2 = 2
-10 == 2 = 0
-10 != 2 = 1
-10 < 2 = 1
-10 <= 2 = 1
-10 > 2 = 0
-10 >= 2 = 0
-10 + 3 = -7
-10 - 3 = -13
-10 * 3 = -30
-10 / 3 = -10/3
-10 ^ 3 = -1000
-10 % 3 = 2
-10 == 3 = 0
-10 != 3 = 1
-10 < 3 = 1
-10 <= 3 = 1
-10 > 3 = 0
-10 >= 3 = 0
-10 + 4 = -6
-10 - 4 = -14
-10 * 4 = -40
-10 / 4 = -5/2
-10 ^ 4 = 10000
-10 % 4 = 2
-10 == 4 = 0
-10 != 4 = 1
-10 < 4 = 1
-10 <= 4 = 1
-10 > 4 = 0
-10 >= 4 = 0
-10 + 5 = -5
-10 - 5 = -15
-10 * 5 = -50
-10 / 5 = -2
```

-10 ^ 5 = -100000
-10 % 5 = 5
-10 == 5 = 0
-10 != 5 = 1
-10 < 5 = 1
-10 <= 5 = 1
-10 > 5 = 0
-10 >= 5 = 0
-10 + 6 = -4
-10 - 6 = -16
-10 * 6 = -60
-10 / 6 = -5/3
-10 ^ 6 = 1000000
-10 % 6 = 2
-10 == 6 = 0
-10 != 6 = 1
-10 < 6 = 1
-10 <= 6 = 1
-10 > 6 = 0
-10 >= 6 = 0
-10 + 7 = -3
-10 - 7 = -17
-10 * 7 = -70
-10 / 7 = -10/7
-10 ^ 7 = -10000000
-10 % 7 = 4
-10 == 7 = 0
-10 != 7 = 1
-10 < 7 = 1
-10 <= 7 = 1
-10 > 7 = 0
-10 >= 7 = 0
-10 + 8 = -2
-10 - 8 = -18
-10 * 8 = -80
-10 / 8 = -5/4
-10 ^ 8 = 100000000
-10 % 8 = 6
-10 == 8 = 0
-10 != 8 = 1
-10 < 8 = 1
-10 <= 8 = 1
-10 > 8 = 0
-10 >= 8 = 0
-10 + 9 = -1
-10 - 9 = -19
-10 * 9 = -90
-10 / 9 = -10/9
-10 ^ 9 = -1000000000
-10 % 9 = 8
-10 == 9 = 0
-10 != 9 = 1
-10 < 9 = 1
-10 <= 9 = 1
-10 > 9 = 0
-10 >= 9 = 0

-10 + 10 = 0
-10 - 10 = -20
-10 * 10 = -100
-10 / 10 = -1
-10 ^ 10 = 10000000000
-10 % 10 = 10
-10 == 10 = 0
-10 != 10 = 1
-10 < 10 = 1
-10 <= 10 = 1
-10 > 10 = 0
-10 >= 10 = 0
-9 + 10 = 1
-9 - 10 = -19
-9 * 10 = -90
-9 / 10 = -9/10
-9 ^ 10 = 3486784401
-9 % 10 = 1
-9 == 10 = 0
-9 != 10 = 1
-9 < 10 = 1
-9 <= 10 = 1
-9 > 10 = 0
-9 >= 10 = 0
-8 + 10 = 2
-8 - 10 = -18
-8 * 10 = -80
-8 / 10 = -4/5
-8 ^ 10 = 1073741824
-8 % 10 = 2
-8 == 10 = 0
-8 != 10 = 1
-8 < 10 = 1
-8 <= 10 = 1
-8 > 10 = 0
-8 >= 10 = 0
-7 + 10 = 3
-7 - 10 = -17
-7 * 10 = -70
-7 / 10 = -7/10
-7 ^ 10 = 282475249
-7 % 10 = 3
-7 == 10 = 0
-7 != 10 = 1
-7 < 10 = 1
-7 <= 10 = 1
-7 > 10 = 0
-7 >= 10 = 0
-6 + 10 = 4
-6 - 10 = -16
-6 * 10 = -60
-6 / 10 = -3/5
-6 ^ 10 = 60466176
-6 % 10 = 4
-6 == 10 = 0
-6 != 10 = 1

-6 < 10 = 1
-6 <= 10 = 1
-6 > 10 = 0
-6 >= 10 = 0
-5 + 10 = 5
-5 - 10 = -15
-5 * 10 = -50
-5 / 10 = -1/2
-5 ^ 10 = 9765625
-5 % 10 = 5
-5 == 10 = 0
-5 != 10 = 1
-5 < 10 = 1
-5 <= 10 = 1
-5 > 10 = 0
-5 >= 10 = 0
-4 + 10 = 6
-4 - 10 = -14
-4 * 10 = -40
-4 / 10 = -2/5
-4 ^ 10 = 1048576
-4 % 10 = 6
-4 == 10 = 0
-4 != 10 = 1
-4 < 10 = 1
-4 <= 10 = 1
-4 > 10 = 0
-4 >= 10 = 0
-3 + 10 = 7
-3 - 10 = -13
-3 * 10 = -30
-3 / 10 = -3/10
-3 ^ 10 = 59049
-3 % 10 = 7
-3 == 10 = 0
-3 != 10 = 1
-3 < 10 = 1
-3 <= 10 = 1
-3 > 10 = 0
-3 >= 10 = 0
-2 + 10 = 8
-2 - 10 = -12
-2 * 10 = -20
-2 / 10 = -1/5
-2 ^ 10 = 1024
-2 % 10 = 8
-2 == 10 = 0
-2 != 10 = 1
-2 < 10 = 1
-2 <= 10 = 1
-2 > 10 = 0
-2 >= 10 = 0
-1 + 10 = 9
-1 - 10 = -11
-1 * 10 = -10
-1 / 10 = -1/10

-1 ^ 10 = 1
-1 % 10 = 9
-1 == 10 = 0
-1 != 10 = 1
-1 < 10 = 1
-1 <= 10 = 1
-1 > 10 = 0
-1 >= 10 = 0
0 + 10 = 10
0 - 10 = -10
0 * 10 = 0
0 / 10 = 0
0 ^ 10 = 0
0 % 10 = 0
0 == 10 = 0
0 != 10 = 1
0 < 10 = 1
0 <= 10 = 1
0 > 10 = 0
0 >= 10 = 0
1 + 10 = 11
1 - 10 = -9
1 * 10 = 10
1 / 10 = 1/10
1 ^ 10 = 1
1 % 10 = 1
1 == 10 = 0
1 != 10 = 1
1 < 10 = 1
1 <= 10 = 1
1 > 10 = 0
1 >= 10 = 0
2 + 10 = 12
2 - 10 = -8
2 * 10 = 20
2 / 10 = 1/5
2 ^ 10 = 1024
2 % 10 = 2
2 == 10 = 0
2 != 10 = 1
2 < 10 = 1
2 <= 10 = 1
2 > 10 = 0
2 >= 10 = 0
3 + 10 = 13
3 - 10 = -7
3 * 10 = 30
3 / 10 = 3/10
3 ^ 10 = 59049
3 % 10 = 3
3 == 10 = 0
3 != 10 = 1
3 < 10 = 1
3 <= 10 = 1
3 > 10 = 0
3 >= 10 = 0

4 + 10 = 14
4 - 10 = -6
4 * 10 = 40
4 / 10 = 2/5
4 ^ 10 = 1048576
4 % 10 = 4
4 == 10 = 0
4 != 10 = 1
4 < 10 = 1
4 <= 10 = 1
4 > 10 = 0
4 >= 10 = 0
5 + 10 = 15
5 - 10 = -5
5 * 10 = 50
5 / 10 = 1/2
5 ^ 10 = 9765625
5 % 10 = 5
5 == 10 = 0
5 != 10 = 1
5 < 10 = 1
5 <= 10 = 1
5 > 10 = 0
5 >= 10 = 0
6 + 10 = 16
6 - 10 = -4
6 * 10 = 60
6 / 10 = 3/5
6 ^ 10 = 60466176
6 % 10 = 6
6 == 10 = 0
6 != 10 = 1
6 < 10 = 1
6 <= 10 = 1
6 > 10 = 0
6 >= 10 = 0
7 + 10 = 17
7 - 10 = -3
7 * 10 = 70
7 / 10 = 7/10
7 ^ 10 = 282475249
7 % 10 = 7
7 == 10 = 0
7 != 10 = 1
7 < 10 = 1
7 <= 10 = 1
7 > 10 = 0
7 >= 10 = 0
8 + 10 = 18
8 - 10 = -2
8 * 10 = 80
8 / 10 = 4/5
8 ^ 10 = 1073741824
8 % 10 = 8
8 == 10 = 0
8 != 10 = 1

```
8 < 10 = 1
8 <= 10 = 1
8 > 10 = 0
8 >= 10 = 0
9 + 10 = 19
9 - 10 = -1
9 * 10 = 90
9 / 10 = 9/10
9 ^ 10 = 3486784401
9 % 10 = 9
9 == 10 = 0
9 != 10 = 1
9 < 10 = 1
9 <= 10 = 1
9 > 10 = 0
9 >= 10 = 0
10 + 10 = 20
10 - 10 = 0
10 * 10 = 100
10 / 10 = 1
10 ^ 10 = 10000000000
10 % 10 = 0
10 == 10 = 1
10 != 10 = 0
10 < 10 = 0
10 <= 10 = 1
10 > 10 = 0
10 >= 10 = 1
```

6.2 Show the test suites used to test your translator

```
/* Type conversions */
x = “8”;
y = num(x) == 8
println::(str::(y))

/* Tests print and scan*/
print::("What is your multiplying factor?");
w = num::(scan::());

/* Tests scanln*/
print::("What is your input?");
v = num::(scan::());

/*Creates a string*/
mystr = "Hello";

/*Creates a new function*/
a = |x| -> |x * w|;

/*Creates a new function based off of a*/
b = a - 7;
```

```
/*Nests functions*/
c = |x, y| -> |(x + b(y) ) < 50|;

/*Nests functions*/
d = |x| -> |c(x, x)|;

/*A piecewise function*/
e = |x| -> |((x < 0) * -1) + ((x > 0) * 1)|;

/*Testing special functions*/
f = |x| -> |ceil(x)|;
f = |x| -> |floor(x)|;
f = |x| -> |ln(x)|;
f = |x| -> |log(x)|;
f = |x| -> |cos(x) ^ 2 + sin(x) ^ 2|;

q = a(v);
r = d(v);

x = 0;

/*Tests that all operators can work*/
x = 1 + 2 * 3 - 4 ^ 5 / 6;

/*Tests subroutines and lists*/
sub mySub(num par1, num par2, num par3)
{
		lst = [par1];
		lst = lst . par2;
		lst = par3 . lst;
		println::("Length of lst: " . str::(len::(lst)));
		println::("Middle of lst: " . str::(lst[2]));
		println::("First of lst: " . str::(pop::(lst)));
		println::("Last of lst: " . str::(rm::(lst)));
}

/*Tests calling subroutines*/
mySub::(1, 2, 55555);
t = mySub::(1, 2, 55555);

/*Tests printing, and checks all values*/
println::("Testing t = " . t);
println::("my x = " . str::(x));
println::("mult = " . str::(w));
println::("input = " . str::(v));
println::("a(x) = " . str_func::(a));
println::("a(input) = " . str::(q));
println::("d(x) = " . str_func::(d));
println::("d(input) = " . str::(r));
println::("piecewise(x) = " . str_func::(e));
println::("testsin(x) = " . str_func::(f));
```

6.3 Explain why and how these test cases were chosen - The test cases are not meant to be a completely exhaustive list but somewhat close. Our test statements were chosen to cover cases that are not completely obvious and might have resulted in a bad output. The test cases have been added slowly and gradually as we correct more inaccuracies and erros in our compiler. It is an evolving process and as of now covers a lot of cases that we could come up with. Running all these cases together make sure that cases don't break once new cases are added.

6.4 State who did what - Dan provided many test cases. Collaborative Google Coding.

7 Lessons Learned

7.1 Each team member should explain his or her most important learning
Here is a compiled list of lessons learned from all the team members that created NumLang.

> **7.1.1 Dan Aprahamian -** This was my first time working with a functional programming language. It helped me refine my understanding of matching, states, grammars, and such. It also exposed me to Ocaml.
> **7.1.2 Damien Fenske-Corbiere -** I learned to think of compilation as string translation broken down in several steps. It was very satisfying when we reached the compiler stage and started returning actual strings.
> **7.1.3 Sahil Yakmi -** I learned how to make a compiler! I learnt my way out of reduce/reduce conflicts, shift/reduce conflicts and the benefits of Ocaml.
> **7.1.4. Siddhi Mittal -** I learned how all the modules within the compiler fit with each other. I also learned the need for umambiguous grammar and the need for a good debugger (thank you Ocamldebug).

7.2 Include any advice the team has for future teams
This might be a cliched advice but start early. This is a project where you really learn a lot - from creating unambiguous grammar to resolving shift reduce conflicts all while trying to struggle with coding using a functional language. As is rightly said, it takes hours to write a simple program in OCaml, but once it compiles - it runs and its beautiful. There will be more time to experiment with this, play around with features if you start early.

8 Appendix

> 9.1 Attach a complete code listing of your translator with each module signed by its author
> 9.2 Do *not* include any ANTLR-generated files, only the .g sources.

Please find the attached folder with all of our source code.

Note:

<u>In order to build the compiler:</u> In the folder where you have the numlang files
cd ../java
make
cd ..
make

In order to compile a file: In the folder where you installed numlang:
./cnumlang.sh *filename*

In order to run the file you just compiled: In the folder where you installed numlang and compiled your file
./rnumlang.sh

**you can only run the file you just compiled with this script. cnumlang.sh generates the java class file Runner.class. If you wish to save a program you built, save this file.