Learning Language

Final Report

George Liao (gkl2104)
Joseanibal Colon Ramos (jc2373)
Stephen Robinson (sar2120)
Huabiao Xu(hx2104)

Contents

- Introduction
  1. Background
  2. Language Goals
- Project Plan
  1. Division of workload
  2. Timeline
- Lessons Learned
  1. George Liao
  2. Joseanibal Colon Ramos
  3. Stephen Robinson
  4. Huabiao Xu
- Language Reference Manual
  1. Lexical Conventions
  2. Basic Types
  3. Expressions
  4. Declarations
  5. Statements
  6. Input and Outpu
  7. Scope and Linkage
- Appendix
  1. Antlr Files
  2. Java Files
  3. Test Files

Introduction

1. Motivation

Learning Language is a programming language designed to be accessible to students prior to entering high school. Students at this age traditionally have no exposure to computer programming, despite the fact that many of them have the maturity and education to write procedural logic. For these students, existing programming languages can be overwhelming due to their abstract syntax and the difficulty of implementing I/O operations. Existing learning languages typically have large standard libraries so that the programmer has many high level tools at his or her disposal. While these libraries are certainly useful, they require that the programmer memorize more keywords. One of the central design tenets behind LL is to provide powerful functionality in an intuitive way.

Since computer science courses are typically not offered until high school, learning language is intended to be introduced in math class. For this reason, most features of LL are implemented using operators. LL is a tool to teach students the fundamentals of writing basic algorithms regardless of the underlying implementation. In LL, where possible, the tedious aspects of writing mathematical code are taken care of automatically, without loops. Similarly, LL provides seamless file and program window I/O. By dispelling the frustrating aspects of first learning to program, LL will foster interest in computer science.

2. Language Goals

The goal of the language is to simplify programming into its most basic building blocks so that children can learn to master those concepts before having to deal with more advanced concepts. This is to be done without a large standard library – in fact, functions are not supported at all. Every task is implemented intuitively using familiar operators, maintaining an easy to write, easy to read, and easy to understand language that can be picked up in a few minutes.

By providing an easy interface for beginners to learn to program, we hope to open up the field of computer science to a wider audience. Additionally, we hope that LL will provide young students with a solid foundation in basic programming concepts in the event that they pursue computer science further.

Project Plan

1. Division of workload

In preparation for this project, our team decided to further split ourselves into pairs so that we could improve our efficiency. George and Huabiao worked primarily on the front end. They were in charge of the lexer and parser. Stephen and Jose were responsible for the walker, semantic analysis and code generation. The team worked together on troubleshooting, debugging, writing sample code, and documentation.

Of course, these responsibilities were loosely defined. It was understood that all four members would be in some way involved with all aspects of the language design and implementation to ensure that everyone had a strong sense of how our compiler works.

2. Timeline

Our development timeline suffered from consistent delays throughout the entire project. We held weekly meetings on Wednesdays most weeks and in the last few weeks we met several extra times. Our scanner and preprocessor were completed before Spring Break. After Spring Break, development began on the parser and semantic analysis. These were mostly finished by mid-April at which point we began to frantically rush to complete code generation, the walker, and the support classes for generated code.   During the last week before our presentation, we rushed to correct broken features and code generation. Ultimately, testing suffered due to poor time management.

Lessons Learned

1. George Liao

While I learned a lot about programming languages, compilers, antlr, and other related subjects while working on this project, the most valuable lessons that I took away from this were those involved with teamwork. Working in a team that doesn't necessarily have a set meeting time and location, with other members that constantly have their own schedules and priorities, and simply making sure that each person was doing a fair amount of work turned out to be a huge obstacle we had to overcome.

I learned that design is crucial. When dealing with other people, you need to precisely put onto paper what your project should look like / operate / interact with itself and others. Verbal agreements simply aren't enough – you're going to need for everyone to have the same copy of the design in order for things to come out right. Not only that, but attention must be paid to every detail.

Also important is communication. A communication procedure needs to be set down and established so that no information is lost. Emails are good sometimes. Phone calls are nice. But by far and away is getting everyone together, however difficult that may be, to just meet and hammer out what needs to be done and how.

Finally, I also learned that planning ahead is crucial. Timeline and deadlines need to be followed – even if they are self imposed. They're there for a reason and keep you from scrambling to finish everything in the last second.

2. Joseanibal Colon Ramos

I quickly realized how making a language as simple and friendly as possible results in an exponential increase in inner complexity, since it takes very careful analysis to translate what looks like a very simple line of code to the real meaning on a lower lever language. One must analyze and carefully decide where intuition would lead a programmer using one's language. It is also important to take into consideration all possible interpretations of a code so that the goal of simplicity is truly achieved, not just what seems simple to the creator. Working on this project helped me gain significant experience with Java and its libraries, as well as with Java coding strategies. It also taught me how to better establish open and clear communication with fellow programmers, and finally, but very importantly, I became aware of the heavy work involved in the creation of a compiler and now I can better appreciate the beauty of broad languages.

3. Stephen Robinson

When I took CS1004 last fall, Professor Aho told the class about PLT. I promptly decided that PLT was a course I would never take (I would later learn that it is a computer engineering major requirement). Standing here having completed the course, I'm still not sure I believe that I wrote a compiler, even if an estimated 10% of the language doesn't work.

As team leader, I think that I did a poor job of motivating the group, setting and enforcing a reasonable development schedule and of distributing the workload. I don't typically see myself as much of a leader and I think that now I have a much better sense of what's involved in managing a project that I did before. Since our project was so rushed towards the end, LL didn't receive the amount of testing that it deserved. Every bug I found, I was able to fix but the language was too big to test in time.

Our language also suffered greatly from feature creep. As development progressed, I'd find a place in the semantic analysis where I was going to throw an error and I'd say to myself, "hey this statement has an intuitive meaning, let's implement it". The result is that the backend continued to grow even though LL is a relatively small language syntactically. Unfortunately, the language that LL has become would have been much better implemented as an interpreted language. Nonetheless, I'm happy that we did produce a compiled language because I think that the development experience is far more enriching. That said, I regret generating java code. A compiled language really ought to generate some sort of assembly.

Though a lot can be said to have gone wrong, there are several aspects of the project that went quite well. I think that we maintained good communication, especially towards the end of the semester. At times, I was sending updates on my progress several times a day and would receive an update from a teammate daily. Any confusion that occurred in our group was not for a lack of communication but for a lack of understanding of how the components written by other teammates worked. In the end, all the pieces of our compiler fit together and the result leaves me with a sense of accomplishment. This is the largest software engineering task that I have ever tackled and we emerged relatively successful.

4. Huabiao Xu

Your text here

Language Reference Manual

A. Lexical Conventions

A1. Comments

The character # introduces a comment, which terminates at the next newline. Comments do not occur within string literals.

A2. Identifiers

An identifier is a sequence of letters and digits and underscores. The first character must be a letter. Upper and lower case letters are the same. Identifiers may have any length.

A3. Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise. All keywords are case insensitive.

string	number	fraction
listof	openoutfile	openinfile
readinto	readfromfile	readlinefromfile
writetofile	writelinetofile	closeinfile
closeoutfile	display	displayline
endif	then	ifnot
repeat	endloop	times
until	if	called

A4. Constants

There are several kinds of constants.

Constant:
    Number-constant
    String-constant
    Fraction-constant
    List-constant

Number-Constant: A number constant consists of a sequence of digits which are taken to be an integer in decimal base. The sequence is optionally preceded by a '-' sign for negative integers.

String-Constant: A string constant is a sequence of one or more characters enclosed in double quotes, as in “foobar”. In order to represent the ‘ “ ’ character, newlines, and certain other characters, the following escape sequences may be used:

Newline	\n
Backslash	\\
Double quote	\”

Fraction-Constant: A fraction constant is two integers separated by ‘//’.

List-Constant: A list constant is a comma separated sequence of number, fraction or string constants surrounded by braces. Example: { 1, 2, 7 }

A5. Atom

Atoms are the building blocks of all data types in Learning Language – they can be a number, string, variable, array (list), or an expression treated as an atom for grouping.

Atom:
- *Num*
- *String*
- *Variable*
- *Array*
- *(expression)*

B. Basic Types

There are three basic types of variables in Learning Language. They are number, fraction and string. The number type is a 32 bit integer. The fraction type is two 32 bit integers (a numerator and a denominator). The string type is a dynamically allocated array of characters. The programmer does not need to specify the size of the string. The number type is used to represent Boolean values with zero representing false and all other values representing true.

There is one derived type in Learning Language. It is the list. Lists can be of one of the three basic variable types or of another list. Memory is dynamically allocated to the list as needed. Users do not need to specify the size of their list.

C. Conversions

In Learning Language, conversions between fractions and numbers are implicit. If a fraction is assigned to a number then it is rounded down (towards zero for negative fractions) to the nearest integer and if a number is assigned to a fraction it is automatically promoted. Additionally, programmers may promote two numbers to a fraction using the “//” operator.

D. Expressions

D1. Expressions

Primary expressions are identifiers, constants, strings, or expressions in parentheses.

Expression:
- *Mathematical Expression (plus, minus, multiply, divide, etc.)*
- *Unary Expression (unary plus, unary minus)*
- *String Expression (strings concatenated together)*
- *Logical Expression (and, or, not)*
- *Atom (basic types, explained previously)*

D2. Unary Operators

Unary Minus Operator ‘-’:

The operand of the unary minus operator must have type number or fraction. The result is the negative of the operand.
Example:
```
Number called b
b <- -4
b <- -b
```

Logical Magnitude Operator ‘|operand|’:

The operand must be of type string or list. The result is the length of the string or the number of elements in the list. The result is of type number.
Example:
```
Number called a
string called b
listof number called c
b <- “the cat”
a <- |b|      #a = 7
```

```
c[5] <- 3
a <- |c|     #a = 5
```

Absolute Value Operator '|operand|':

The operand must be of type number or fraction. The result is the length of the string. The result is of type number.

Example:

```
Number called a
number called b
a <- -8
b <- |a|      #b = 8
```

D3. Binary Arithmetic Operators

Exponential Operator '^':

The exponential operator groups right to left. The exponent must be of type number and positive. The left operand may be a fraction or a number.

exp_expr : atom (| EXP^ exp_expr) ;

Example:

```
Number called a
number called b
a <- 2
b <- 2 ^ a      #b = 4
```

Multiplicative Operators '*', '/', '%':

The multiplicative operators group left to right. The operands for '*' (multiplication) and '/' (division) must be numbers or fractions. The right operand for '%' (remainder) must be a number. If the second operand is zero for any of the above, the result is 0. If one operand is a fraction, the other is implicitly converted to a fraction.

mult_expr : exp_expr ((MULT^|DIV^|MOD^|FRACT^) exp_expr) ;*

Example:

```
Number called a
number called b
a <- 2 * 4 / 3  #a = 2
b <- a % 2      #b = 0
```

Additive Operators ‘+’, ‘-’ :

The additive operators group left to right. The operands for ‘+’ (addition) and ‘-’ (subtraction) must be numbers or fractions. If one operand is a fraction, the other is implicitly converted to a fraction

math_expr : mult_expr ((PLUS^|MINUS^) mult_expr) ;*

Example:
```
Number called a
a <- 2 + 4 - 3  #a = 3
```

D4. String Operators

Concatenation Operator ‘.’:

The ‘+’ operator can be used to concatenate two strings. It groups left to right.

The grammar for concatenation is the same as for addition.

Example:
```
string called a
a <- “cat”
a <- a + a   #a = “catcat”
```

D5. Relational Operators

The relational operators are ‘<’ (less than), ‘>’ (greater than), ‘<=’ (less than or equal), ‘>=’ (greater than or equal), ‘=’ (equals) and ‘not=’ (not equal to). These operators all yield 1 if the expression evaluates to true and 0 if it evaluates to false.

expression : string_expr ((EQ^|GT^|LT^|GE^|LE^|NE^) string_expr) ;*

D6. Assignment Operator

The assignment operator, ‘<-’, groups left to right. The operands must be of the same type with the exception of an integer being assigned to a fraction.

assignment : variable asgn_val

{ #assignment = #([ASSIGNMENT, "assignment"], assignment); } ;

asgn_val : ASGN! expression ;

Example:
```
Number called a
a <- 2    #a = 2
```

E. Declarations

Declarations are used to add identifiers to the name space. A declaration also specifies the data type of an identifier. All identifiers are case insensitive.

E1. Variable Declarations

Learning language supports the following types: number, fraction, string. Furthermore, lists of any of the above types may be declared. Memory is dynamically allocated for the list. The user does not specify the size of the list. Declarations have the form:

```
declaration : list_decl "called"! variable (asgn_val)?
        { #declaration = #([DECLARATION, "declaration"], declaration); } ;
list_decl   : KW_LIST^ list_decl | type ;
type        : (KW_STRING|KW_NUM|KW_FRACT) ;
```

Learning Language does not support user defined types.

F. Statements

F1. Statements

```
statement :
        ( declaration
       | assignment
       | repeat_stmt
       | if_stmt
```

| action_stmt
|)
NL!;

F2. Declaration Statements

Declares a new variable (see E1 above).

F3. Assignment Statements

Assigns the value of an expression to a predefined variable

Variable <- expression

F4. Selection Statements

Selection statements choose one of (up to) two flows of control. The expression in the if statement must have integer type and if it evaluates unequal to zero, then the first statement is executed. The second statement, if present, executes if the expression evaluates to zero.

selection-statement:
if *expression* then *statement* end
if *expression* then *statement* ifnot *statement* end

F5. Iteration Statements

iteration-statement:
repeat *expression* times *statement* end
repeat until (*expression) statement* end

In the first form of the loop, the expression is evaluated once and it must evaluate to a number. If that number is less than zero, then the loop will not execute. The statement is executed that number of times. The iteration count is held implicitly in the variable named nth, which is scoped within the loop. For nested loops, nth holds the iteration counter for the innermost loop.

In the second form of the loop, the expression must evaluate to a number, as in the first. If that expression evaluates equal to zero, the statement executes and then the expression is again executed. The loop will continue until the expression evaluates unequal to zero.

F6. Action Statements

Handles input and output to the GUI and to files. Explained in the following section.

G. Input and Output

G1. Standard I/O

Standard output takes on the following form:

Output:
    Displayline expression

The evaluation of expression is output to the LL GUI.

Input takes on the following form:

Input:
    Readinto expression

Where expression is a variable of type string.

G2. File I/O

To read or write a file, the file must first be opened. Only one file for reading and one file for writing may be open at a time:

```
openoutfile “file.txt”
openinfile “infile.txt”
```

To read from the file:

```
readfromfile someStrVar
```

```
readlinefromfile Somestrvar
```

The former reads space delimited words and the latter reads until the next new line.

To write to a file:

```
Writetofile string-expression
Writelinetofile string-expression
```

The above always appends to the file. When file I/O is complete, the file must be closed:

```
Closeinfile
Closeoutfile
```

G3. Default GUI

A default GUI is provided for all LL programs. Standard in and standard out are routed through the GUI, making it easier for kids used to working in windowed environments to interact with their program. The Default GUI is essentially a console for the program.

H. Scope and Linkage

Learning language does not support routines and therefore linkage is not necessary.

Learning Language is statically scoped. All variables declared inside conditional and iterative statements are scoped only within that statement. Variable names may not be reused within the same scope. However, as is the case with the nth variable, a name may be reused even if it is used in a higher scope. This is not the case with Java.

Appendix

Antlr Files

1. Grammar.g

```
/*
 * grammar.g : the lexer and the parser, in ANTLR grammar.
 * George Liao
 */

class LLAntlrLexer extends Lexer;

options {
    k=2;
    charVocabulary='\3'..'\377';
    caseSensitiveLiterals=false;
    exportVocab = LLAntlr;
}


protected
ALPHA   : 'a'..'z' | 'A'..'Z' | '_' ;

protected
DIGIT   : '0'..'9';

WS      : (' ' | '\t' | '\f')+
          { $setType(Token.SKIP); } ;

NL      : ( '\r' '\n'    // DOS/Windows
          | '\r'         // Macintosh
          | '\n'         // Unix
          )
          { newline(); } ;

COMMENT : '#' ( ~( '\r' | '\n' ) )*
          { $setType(Token.SKIP); } ;

ID         options { testLiterals = true; }
           : ALPHA ( ALPHA | DIGIT )* ;

INTLIT  : (DIGIT)+ ;

STRING  : '"'
             ( ~( '"' | '\\' | '\n' )
             | ('\\'! ( '"' | '\\' ))  /* escape characters */
             )*
             '"' ;

// bracketing
LPAREN  : '(' ;
RPAREN  : ')' ;
LBRACE  : '{' ;
RBRACE  : '}' ;
```

```
LBRK       : '[' ;
RBRK       : ']' ;

// math
PLUS    : '+' ;
MINUS   : '-' ;
MULT    : '*' ;
DIV        : '/' ;
MOD        : '%' ;
EXP        : '^' ;
FRACT   : "//" ;
DOT     : '.' ;

// comparison
NE      : "!=" ;
EQ    : '=' ;
GT      : '>' ;
LT      : '<' ;
GE      : ">=" ;
LE      : "<=" ;

// punctuation
SEMI    : ';' ;
COMMA      : ',' ;
COLON      : ':' ;

// misc
ASGN    : "<-" ;
PIPE    : '|' ;
MAPS    : "->" ;


// ---------------------
// ---- PARSER START ----
// ---------------------
class LLAntlrParser extends Parser;

options{
    buildAST = true;
}

tokens {
   // statement tokens
   PROGRAM ;
   ASSIGNMENT ;
   DECLARATION ;
   ARRAY ;
   BODY ;
   FUNCTIONS ;
   FUNC_CALL ;
   ARG_LIST ;
   DRIVER ;
   IDX ;
   UMINUS ;

   // control
   KW_LOOP = "repeat" ;
   KW_ENDLOOP = "endloop" ;
   KW_TIMES = "times" ;
   KW_UNTIL = "until" ;
   KW_IF = "if" ;
   KW_THEN = "then" ;
   KW_ENDIF = "endif" ;
   KW_IFNOT = "ifnot" ;
   KW_RETURN = "return";

   // action
   KW_OPENOUTFILE = "openoutfile" ;
   KW_OPENINFILE = "openinfile" ;
   KW_READINTO = "readinto" ;
   KW_READFROMFILE = "readfromfile" ;
   KW_READLINEFROMFILE = "readlinefromfile" ;
   KW_WRITETOFILE = "writetofile" ;
   KW_WRITELINETOFILE = "writelinetofile" ;
   KW_CLOSEINFILE = "closeinfile" ;
   KW_CLOSEOUTFILE = "closeoutfile" ;
   KW_DISPLAY = "display" ;
   KW_DISPLAYLINE = "displayline" ;

   // types
   KW_LIST = "listof" ;
   KW_STRING = "string" ;
   KW_NUM = "number" ;
   KW_FRACT = "fraction" ;

   // function
   KW_FUNC = "function" ;
   KW_ENDFUNC = "endfunction" ;
}

program       : //functions
               //driver
               ( statement )*
               EOF
              { #program = #([PROGRAM, "program"], program); } ;

//driver     : ( statement )*
//            { #driver = #([DRIVER, "driver"], driver); }
//            ;

statement   :
              ( declaration
            | assignment
            | repeat_stmt
            | if_stmt
            | action_stmt
            | )
            NL!;

// ---- function declaration ----
/*functions    : (function_decl)*
              { #functions = #([FUNCTIONS, "functions"], functions); };

function_decl : KW_FUNC^ "called"! ID arg_list MAPS! list_decl NL!
```

```
                     body
                KW_ENDFUNC! NL! ;

arg_list : LPAREN! arg_decl (COMMA! arg_decl)* RPAREN!
              { #arg_list = #([ARG_LIST, "arg_list"], arg_list); };
arg_decl : list_decl "called"! variable ;
*/
// ---- function call ----
//func_call    : ID LPAREN! (expression)? (COMMA! expression)* RPAREN!
  //          { #func_call = #([FUNC_CALL, "func_call"], func_call); } ;

// ---- declaration ----
declaration : list_decl "called"! variable (asgn_val)?
            { #declaration = #([DECLARATION, "declaration"], declaration); } ;

list_decl   : KW_LIST^ list_decl | type ;


// ---- assignment ----
assignment  : variable asgn_val
            { #assignment = #([ASSIGNMENT, "assignment"], assignment); } ;

asgn_val    : ASGN! expression ;


// ---- repeat ----
repeat_stmt : KW_LOOP! ( (expression KW_TIMES^) | (KW_UNTIL^ expression) ) NL!
                 body
              KW_ENDLOOP!;

// ---- if ----
if_stmt     : KW_IF^ expression KW_THEN! NL!
                 body
                 ( ifnot_stmt )?
              KW_ENDIF!;

ifnot_stmt  : KW_IFNOT! NL! body ;

// ---- action ----
action_stmt : ( KW_OPENOUTFILE^
              | KW_OPENINFILE^
              | KW_READINTO^
              | KW_READFROMFILE^
              | KW_READLINEFROMFILE^
              | KW_WRITETOFILE^
              | KW_WRITELINETOFILE^
              | KW_CLOSEINFILE^
              | KW_CLOSEOUTFILE^
              | KW_DISPLAY^
              | KW_DISPLAYLINE^
                | KW_RETURN^) expression ;

// ---- expression ----

expression  : string_expr ( (EQ^|GT^|LT^|GE^|LE^|NE^) string_expr )* ;

string_expr : unary_expr ( DOT^ unary_expr )* ;
```

```
unary_expr  :
               ( MINUS! math_expr
                     { #unary_expr = #([UMINUS, "uminus"], unary_expr) ; }
               | math_expr
               ) ;

math_expr   : mult_expr ( (PLUS^|MINUS^) mult_expr )* ;

mult_expr   : exp_expr ( (MULT^|DIV^|MOD^|FRACT^) exp_expr )* ;

exp_expr    : atom
               (
               | EXP^ exp_expr
               ) ;
  /* Huabiao : this should not be:
  *                atom (EXP^ atom)*
  *                because the precedence gets inverted in that case.
  *                ex: 5^2^3 is intuitively 5^(2^3) not (5^2)^3
  */

// ---- atom ----
atom        : INTLIT
            | STRING
            //| (ID LPAREN) => func_call
            | /*(ID (LBRK)?) =>*/ variable
            | array
            | LPAREN! expression RPAREN!
            | PIPE^ expression PIPE! ;


body        : (statement)*
            { #body = #([BODY, "body"], body); } ;

array       : LBRACE! (expression)? (COMMA! expression)* RBRACE!
            { #array = #([ARRAY, "array"], array); } ;


variable    : ID
         (
        | ( LBRK! expression RBRK! )+
        { #variable = #([IDX, "idx"], variable); }
        /* Huabiao: walker is changed to allow #(IDX arr 1 3 4)
        */
        );

type        : (KW_STRING|KW_NUM|KW_FRACT) ;
```

2. walker.g

```
/*
walker.g
Stephen Robinson
*/

{
import java.io.*;
import java.util.*;
}

class LLAntlrWalker extends TreeParser;
options{
  importVocab = LLAntlr ;
}

{
  LLContainer cntnr = new LLContainer();
}

//used for the left side of assignment statements
expl returns [ BaseType r ]
{
  BaseType a, b;
  r = null;
}

  : #(id:ID               { r = cntnr.getVariable("variable_"+id.getText() + "_");
})
  | #(PIPE a=expl)        { r = cntnr.leftSize(a); }
  | #(IDX a=expl
            (b=expr      { a = cntnr.leftIndex(a,b);})+

        { r = a; })

  ;

//used for the rest of the language
expr returns [ BaseType r ]
{
  BaseType a, b;
  r = null;
}

  : #(UMINUS a=expr)            { r = cntnr.uminus(a); }

  | #(IDX a=expr
            (b=expr      { a = cntnr.index(a,b);})+

        { r = a; })

  | #(DOT a=expr b=expr)                    { r = cntnr.append(a,b); }
  //***the left side of assignment goes to expl not expr ***
```

```
| #(ASSIGNMENT a=expl b=expr)       { r = cntnr.assign(a,b); }
| #(PIPE a=expr)                          { r = cntnr.size(a); }
| #(PLUS a=expr b=expr)                   { r = cntnr.plus(a,b); }
| #(MINUS a=expr b=expr)                  { r = cntnr.minus(a,b); }
| #(FRACT a=expr b=expr)                  { r = cntnr.fract(a,b); }
| #(MULT a=expr b=expr)                   { r = cntnr.mult(a,b); }
| #(DIV a=expr b=expr)                    { r = cntnr.intdiv(a,b); }
| #(MOD a=expr b=expr)                    { r = cntnr.remainder(a,b); }
| #(EXP a=expr b=expr)                    { r = cntnr.exp(a,b); }

| #(NE a=expr b=expr)                     { r = cntnr.ne(a,b); }
| #(EQ a=expr b=expr)                     { r = cntnr.eq(a,b); }
| #(GT a=expr b=expr)                     { r = cntnr.gt(a,b); }
| #(LT a=expr b=expr)                     { r = cntnr.lt(a,b); }
| #(GE a=expr b=expr)                     { r = cntnr.ge(a,b); }
| #(LE a=expr b=expr)                     { r = cntnr.le(a,b); }

| #(ARRAY   { r = new LLList(null); }
            (a=expr { r.append( a ); } )* )

| #(id:ID   { r = cntnr.getVariable("variable_"+id.getText() + "_"); })
      /* Huabiao: see #DECLARATION for details:
      *     change either none or both, not one or the other
      */

| i:INTLIT
{
      r = new LLNumber();
      r.setName(i.getText());
}

| str:STRING
{
      a = new LLString();
      a.setName(str.getText());
      r = a;
}

| #(STATEMENT r=expr)
| #(BODY (bdy:. {r = expr(#bdy); })*)
| #(PROGRAM (prgm:. {r = expr(#prgm); })*)
    { cntnr.printFooter(); r= new BaseType();}
| #(DECLARATION a=expr name:. (asgn:.)?)
{
      a.setName("variable_"+name.getText()+"_");
      a = cntnr.declareStmt(a, "variable_"+name.getText()+"_");
/* Huabiao: variable_ is prepended to avoid conflict with registers
*     change here AND in #id:ID
*/

      if(asgn != null)
      {
            b = expr(#asgn);
            r = cntnr.assign(a,b);
      }
      else
      {
            r = a;
      }
}

| #(KW_LIST a=expr) {r = cntnr.getListText(a);}
| #(KW_STRING (s:.)?) { r = new LLString();}
| #(KW_NUM (n:.)?) {r = new LLNumber();}
| #(KW_FRACT (f:.)?)
      {
            a = new LLFraction();
            a.setName("new Fraction()");
            r = a;
      }


| #(KW_IF a=expr thenstmt:. (elsestmt:.)?)
{
      if( !(a instanceof LLNumber))
      {
            a.error( "if: expression should evaluate to a number" );
      }
      cntnr.initIF(a);
      cntnr.LLOutput("{");
      b = expr(#thenstmt);
      cntnr.LLOutput("}");
      cntnr.leaveScope();
      if (elsestmt != null)
      {
      cntnr.LLOutput("else");
      cntnr.LLOutput("{");
      cntnr.enterScope();
      a = expr(#elsestmt);
      cntnr.LLOutput("}");
            cntnr.leaveScope();
      }
      r = new BaseType();
}

|#(KW_TIMES a=expr forbody:.)
{
      cntnr.initFor(a);
      b = expr(#forbody);
      cntnr.LLOutput("}");
      cntnr.leaveScope();
      r = a;
}

|#(KW_UNTIL cond:. untilbody:.)
{
      b = expr(#cond);
      cntnr.initUntil(b);
      a = expr(#untilbody);
      a = expr(#cond);
      cntnr.assign(b,a);
      cntnr.LLOutput("}");
      cntnr.leaveScope();
      r = b;
```

```
}

|#(KW_OPENOUTFILE a=expr)
{
      r = cntnr.openOutFile(a);
}
|#(KW_OPENINFILE a=expr)
{
      r = cntnr.openInFile(a);
}
|#(KW_READINTO a=expr)
{
      r = cntnr.readStdin(a);
}
|#(KW_READFROMFILE a=expr)
{
      r = cntnr.readFromFile(a);
}
|#(KW_READLINEFROMFILE a=expr)
{
      r = cntnr.readLineFromFile(a);
}
|#(KW_DISPLAY a=expr)
{
      r = cntnr.stdout(a);
}
|#(KW_DISPLAYLINE a=expr)
{
      r = cntnr.stdoutLine(a);
}
|#(KW_WRITETOFILE a=expr)
{
      r = cntnr.writeToFile(a);
}
|#(KW_WRITELINETOFILE a=expr)
{
      r = cntnr.writeLineToFile(a);
}
|#(KW_CLOSEINFILE (in:.)?)
{
      cntnr.LLOutput("LLIO.closeInFile()");
      r = new BaseType();
}
|#(KW_CLOSEOUTFILE (out:.)?)
{
      cntnr.LLOutput("LLIO.closeOutFile()");
      r = new BaseType();
}

;
```

Java Files

1. LLPreprocessor.java

```
import java.io.*;

//this class makes all tokens lower case
//pretty straightforward
//potentially could be used to extract file names of included function files
//v1.0
//Stephen Robinson
public class LLPreprocessor
{

	BufferedReader in;
	BufferedWriter out;


	public LLPreprocessor(String i, String o)
	{
		File In = new File(i);
		File Out = new File(o);
		try {
			in = new BufferedReader(new FileReader(In));
			out = new BufferedWriter(new FileWriter(Out));
		} catch (Exception ex)
		{
			throw new LLException(ex.toString());
		}
	}

	public final void process()
	{
		String line;
		try {
			line = null;
			while (( line = in.readLine()) != null)
			{
				line = processLine(line);
				out.write(line);
				out.newLine();
		    }
		}
		catch (Exception ex)
		{
		  System.out.println(ex.getMessage());
		}

		try
		{
			in.close();
			out.close();
		}
		catch (Exception ex)
		{
			System.out.println(ex.getMessage());
		}

	}

	public final String processLine(String line)
	{
		String out = "";
		boolean quotesOpen = false;
		boolean inComment = false;
		//Boolean pipeOpen = false;
		char a;
		char diff = (char)((int)'A' - (int)'a');
		for(int i = 0; i < line.length(); i++)
		{
			a = line.charAt(i);
			if(inComment == true)
			{
				out += a;
			}
			else if(a == '#')
			{
				inComment = true;
				out = out + a;
			}
			else if(a == '"')
			{
				quotesOpen = !quotesOpen;
				out += a;
			}
			else if(quotesOpen == true)
			{
				if(a == '\\')
				{
					out += a;
					i++;
					out += line.charAt(i);
				}
				else
				{
					out += a;
				}
			}
			/*else if(a == '|' && pipeOpen == false)
			{
				pipeOpen = true;
				out += "|(";
			}
			else if(a == '|' && pipeOpen == true)
			{
				pipeOpen = false;
				out += ")";
			} */
			else
			{
				if(a >= 'A' && a <= 'Z')
```

```
                        {
                              out += (char)((int)a - (int)diff);
                        }
                        else
                        {
                              out += a;
                        }
                  }
            }

            return out;
      }

}
```

2. BaseType.java

```
//LL base data type from which all other data types extend
//Stephen Robinson
//v 1.0

public class BaseType {

    String name;
    int scopeCnt;

    public BaseType()
    {
        name = null;
        scopeCnt = -5;
    }

    public void setScope(int n)
    {
        scopeCnt = n;
    }

    public static BaseType getCopy(BaseType a)
    {
        if(a instanceof LLNumber)
        {
              return new LLNumber();
        }
        else if(a instanceof LLString)
        {
              return new LLString();
        }
        else if(a instanceof LLFraction)
        {
              return new LLFraction();
        }
        if(a instanceof LLList)
        {
              return new LLList( getCopy(((LLList)a).type) );
        }
        return new BaseType();
    }

    public BaseType(String name)
    {
        this.name = name;
    }

    // finds the first level of the type of list or just this if there's no more
levels
    public BaseType getCoreType()
    {
        BaseType temp = this;
        while(temp instanceof LLList)
        {
            temp = ((LLList)temp).type;
        }
        return temp;
    }

    public void setName(String name)
    {
        this.name = name;
    }

    public String getName()
    {
        return this.name;
    }

    public String getTypeName()
    {
        return "unknown";
    }

    public void error(String msg)
    {
        String err = "Error: " + msg + " ( ";
        if(name != null)
        {
            err += name;
        }
        else
        {
            err += "noname";
        }
        err += " of type " + getTypeName() + " )";

        throw new LLException(err);
    }

    public void error(BaseType b, String msg)
    {
        if(b == null)
        {
            error(msg);
            return;
        }
        String err = "Error: " + msg + " ( ";
        if(name != null)
        {
            err += name;
        }
        else
        {
            err += "noname";
        }
        err += " of type " + getTypeName() + " )";
        err += " and (";
        if(b.name != null)
        {
            err += b.name;
        }
        else
        {
            err += "noname";
        }
        err += " of type " + b.getTypeName() + " )";

        throw new LLException(err);
    }

    public BaseType fract(BaseType b)
    {
        error(b, "//");
        return new BaseType("error");
    }

    public BaseType uminus() {
        error( "-" );
        return new BaseType("error");
    }

    public BaseType append(BaseType b) {
        error( b, "." );
        return new BaseType("error");
    }

    public BaseType assign(BaseType b) {
        error( b, "<-" );
        return new BaseType("error");
    }

    public BaseType size() {
        error( "magnitude" );
        return new BaseType("error");
    }

    public BaseType plus(BaseType b) {
        error( b, "+" );
        return new BaseType("error");
    }

    public BaseType minus( BaseType b ) {
        error( b, "-" );
        return new BaseType("error");
    }

    public BaseType exp( BaseType b ) {
        error( b, "^" );
        return new BaseType("error");
    }

    public BaseType mult( BaseType b ) {
        error( b, "*" );
        return new BaseType("error");
    }

    public BaseType index( BaseType b ) {
        error(b, "[]");
        return new BaseType("error");
    }

    public BaseType intdiv( BaseType b ) {
        error( b, "/" );
        return new BaseType("error");
    }

    public BaseType remainder( BaseType b ) {
        error( b, "%" );
        return new BaseType("error");
    }

    public BaseType gt( BaseType b ) {
        error( b, ">" );
        return new BaseType("error");
    }

    public BaseType ge( BaseType b ) {
        error( b, ">=" );
        return new BaseType("error");
    }

    public BaseType lt( BaseType b ) {
        error( b, "<" );
        return new BaseType("error");
    }

    public BaseType le( BaseType b ) {
        error( b, "<=" );
        return new BaseType("error");
    }

    public BaseType eq( BaseType b ) {
        error( b, "=" );
        return new BaseType("error");
    }

    public BaseType ne( BaseType b ) {
        error( b, "!=" );
        return new BaseType("error");
    }
}
```

3. LLNumber.java

```
//LL number type semantic rules and code gen.  Pretty robust.
//the code to be output is stored in the name of the returned instance
//of basetype.  Code generation could have been planned out better but
//with our existing class structure this was the best solution
//Stephen Robinson
//v 1.0

import java.util.*;

public class LLNumber extends BaseType {


    public LLNumber()
    {

    }

    public String getTypeName()
    {
        return "number";
    }

    public BaseType assign(BaseType b)
    {
        if (b instanceof LLList)
        {
        error(b,"<-");
        }
        else if (b instanceof LLNumber)
        {
        if(scopeCnt == -1)
        {
              String s = name + b.getName() + ")";
              BaseType ret = new LLNumber();
              ret.setName(s);
              return ret;
        }
        else
        {
              String s = name + " = " + b.getName();
              BaseType ret = new LLNumber();
              ret.setName(s);
              return ret;
        }
        }
        else if (b instanceof LLString)
        {
        error(b,"<-");
        }
        else if (b instanceof LLFraction)
        {
        if(scopeCnt == -1)
        {
              String s = name + b.getName() + ".FracToNum())";
              BaseType ret = new LLNumber();
              ret.setName(s);
              return ret;
        }
        else
        {
              String s = name + " = " + b.getName() + ".FracToNum()";
              BaseType ret = new LLNumber();
              ret.setName(s);
              return ret;
        }
        }
        return new BaseType();
    }

    //this is absolute value
    public BaseType size()
    {
        String s = "(" + name + "<0?-" + name + ":" + name + ")";
        BaseType ret = new LLNumber();
        ret.setName(s);
        return ret;
    }

    public BaseType append(BaseType b)
    {
        if (b instanceof LLList)
        {
            if(((LLList)b).type instanceof LLNumber)
            {
              String s = b.getName() + ".prependPrimitive(new Integer(";
              s += this.name + "))";
              BaseType ret = new LLList(new LLNumber());
              ret.setName(s);
              return ret;
            }
            else if(((LLList)b).type instanceof LLFraction)
            {
              String s = b.getName() + ".prependPrimitive(new Fraction(";
              s += this.name + ",1))";
              BaseType ret = new LLList(new LLFraction());
              ret.setName(s);
              return ret;
            }
            else
            {
              error(b,".");
            }
        }
        else if(b instanceof LLNumber)
        {
        String s = "new Listof(\"number\", new Integer(" + name;
        s += "), new Integer(" + b.getName() + "))";
        BaseType ret = new LLList(new LLNumber());
```

```
        ret.setName(s);
        return ret;
        }
        else if (b instanceof LLString)
        {
        error(b,".");
        }
        else if (b instanceof LLFraction)
        {
        String s = "new Listof(\"fraction\", " + "new Fraction(";
        s += name + ", 1)";
        s += ", " + b.getName() + ")";
        BaseType ret = new LLList(new LLFraction());
        ret.setName(s);
        return ret;
        }
        return new BaseType("error");
    }

    public BaseType fract(BaseType b)
    {
        if (b instanceof LLList)
        {
        BaseType tmp = b.getCoreType();
        if(tmp instanceof LLNumber)
        {
              String s = b.getName() + ".prefract(";
              s += name + ")";
              BaseType ret = getCopy(b);
              ret.setName(s);
              return ret;
        }
        else
        {
              error(b,"//");
        }
        }
        else if (b instanceof LLNumber)
        {
        String s = "new Fraction(" + name + ", ";
        s += b.getName() + ")";
        BaseType ret = new LLFraction();
        ret.setName(s);
        return ret;
        }
        else if (b instanceof LLString)
        {
        error(b,"//");
        }
        else if (b instanceof LLFraction)
        {
        error(b,"//");
        }
        return new BaseType("error");
    }

    public BaseType exp(BaseType b)
```

```
{
    if (b instanceof LLList)
    {
    String s = b.getName() + ".preexp(" + name + ")";
    BaseType ret = getCopy(b);
    ret.setName(s);
    return ret;
    }
    else if (b instanceof LLNumber)
    {
    String s = "(int)Math.pow(" + name + ", ";
    s += b.getName() + ")";
    BaseType ret = new LLNumber();
    ret.setName(s);
    return ret;
    }
    else if (b instanceof LLString)
    {
    error(b,"^");
    }
    else if (b instanceof LLFraction)
    {
    error(b,"^");
    }
    return new BaseType("error");
}


public BaseType uminus()
{
    String s = "-" + name;
    BaseType ret = new LLNumber();
    ret.setName(s);
    return ret;
}

public BaseType plus(BaseType b)
{
    if (b instanceof LLList)
    {
        BaseType tmp = b.getCoreType();

        if(tmp.getTypeName() == "number")
        {
          String s = b.getName() + ".add(";
          s += name + ")";
          BaseType ret = getCopy(b);
          ret.setName(s);
          return ret;
        }
        else if(tmp.getTypeName() == "fraction")
        {
          String s = b.getName() + ".add(new Fraction(";
          s += name + ", 1))";
          BaseType ret = getCopy(b);
          ret.setName(s);
          return ret;
```

```
            }
            else
            {
              error(b,"+");
            }
        }
        else if (b instanceof LLNumber)
        {
        String s = name + " + " + b.getName();
        BaseType ret = new LLNumber();
        ret.setName(s);
        return ret;
        }
        else if (b instanceof LLString)
        {
        error(b,"+");
        }
        else if (b instanceof LLFraction)
        {
        String s = "(new Fraction(" + name;
        s += ", 1)).add(" + b.getName() + ")";
        BaseType ret = new LLFraction();
        ret.setName(s);
        return ret;
        }
        return new BaseType("error");
    }

    public BaseType minus(BaseType b)
    {
        if (b instanceof LLList)
        {
            BaseType tmp = b.getCoreType();
            if(tmp.getTypeName() == "number")
            {
              String s = b.getName() + ".preminus(";
              s += name + ")";
              BaseType ret = getCopy(b);
              ret.setName(s);
              return ret;
            }
            else if(tmp.getTypeName() == "fraction")
            {
              String s = b.getName() + ".preminus(new Fraction(";
              s += name + ",1))";
              BaseType ret = getCopy(b);
              ret.setName(s);
              return ret;
            }
            else
            {
              error(b,"-");
            }
        }
        else if (b instanceof LLNumber)
        {
        String s = name + " - " + b.getName();
        BaseType ret = new LLNumber();
              ret.setName(s);
        return ret;
        }
        else if (b instanceof LLString)
        {
        error(b,"-");
        }
        else if (b instanceof LLFraction)
        {
        String s = "(new Fraction(" + name + ", 1)).subtract(";
        s +=  b.getName() + ")";
        BaseType ret = new LLFraction();
              ret.setName(s);
        return ret;
        }
        return new BaseType("error");
    }

    public BaseType mult(BaseType b)
    {
        if (b instanceof LLList)
        {
            BaseType tmp = b.getCoreType();

            if(tmp.getTypeName() == "number")
            {
              String s = b.getName() + ".multiply(";
              s += name + ")";
              BaseType ret = getCopy(b);
              ret.setName(s);
              return ret;
            }
            else if(tmp.getTypeName() == "fraction")
            {
              String s = b.getName() + ".multiply(new Fraction(";
              s += name + ", 1))";
              BaseType ret = getCopy(b);
              ret.setName(s);
              return ret;
            }
            else
            {
              error(b,"*");
            }
        }
        else if (b instanceof LLNumber)
        {
        String s = name + " * " + b.getName();
        BaseType ret = new LLNumber();
        ret.setName(s);
        return ret;
        }
        else if (b instanceof LLString)
        {
        error(b,"*");
        }
        else if (b instanceof LLFraction)
        {
        String s = "(new Fraction(" + name;
        s += ", 1)).multiply(" + b.getName() + ")";
        BaseType ret = new LLFraction();
        ret.setName(s);
        return ret;
        }
        return new BaseType("error");
    }

    public BaseType intdiv(BaseType b)
    {
        if (b instanceof LLList)
        {
            BaseType tmp = b.getCoreType();
            if(tmp.getTypeName() == "number")
            {
              String s = b.getName() + ".predivide(";
              s += name + ")";
              BaseType ret = getCopy(b);
              ret.setName(s);
              return ret;
            }
            else if(tmp.getTypeName() == "fraction")
            {
              String s = b.getName() + ".predivide(new Fraction(";
              s += name + ",1))";
              BaseType ret = getCopy(b);
              ret.setName(s);
              return ret;
            }
            else
            {
              error(b,"/");
            }
        }
        else if (b instanceof LLNumber)
        {
        String s = name + " / " + b.getName();
        BaseType ret = new LLNumber();
              ret.setName(s);
        return ret;
        }
        else if (b instanceof LLString)
        {
        error(b,"/");
        }
        else if (b instanceof LLFraction)
        {
        String s = "(new Fraction(" + name + ", 1)).divide(";
        s +=  b.getName() + ")";
        BaseType ret = new LLFraction();
              ret.setName(s);
        return ret;
        }
        return new BaseType("error");
```

```
}

public BaseType remainder(BaseType b)
{
    if (b instanceof LLList)
    {
        BaseType tmp = b.getCoreType();
        if(tmp.getTypeName() == "number")
        {
          String s = b.getName() + ".premodulus(";
          s += name + ")";
          BaseType ret = getCopy(b);
          ret.setName(s);
          return ret;
        }
        else
        {
          error(b,"%");
        }
    }
    else if (b instanceof LLNumber)
    {
    String s = name + " % " + b.getName();
    BaseType ret = new LLNumber();
          ret.setName(s);
    return ret;
    }
    else if (b instanceof LLString)
    {
    error(b,"%");
    }
    else if (b instanceof LLFraction)
    {
    error(b,"%");
    }
    return new BaseType("error");
}

public BaseType gt(BaseType b)
{
    if (b instanceof LLList)
    {
    error(b,">");
    }
    else if (b instanceof LLNumber)
    {
    String s = "((" + name + " > " + b.getName() + ")?1:0)";
    BaseType ret = new LLNumber();
    ret.setName(s);
    return ret;
    }
    else if (b instanceof LLString)
    {
    error(b,">");
    }
    else if (b instanceof LLFraction)
    {
    String s = "((new Fraction(" + name + ",1)).gt(" + b.getName() + "))";
    BaseType ret = new LLNumber();
    ret.setName(s);
    return ret;
    }
    return new BaseType();
}

public BaseType ge(BaseType b)
{
    if (b instanceof LLList)
    {
    error(b,">=");
    }
    else if (b instanceof LLNumber)
    {
    String s = "((" + name + " >= " + b.getName() + ")?1:0)";
    BaseType ret = new LLNumber();
    ret.setName(s);
    return ret;
    }
    else if (b instanceof LLString)
    {
    error(b,">=");
    }
    else if (b instanceof LLFraction)
    {
    String s = "((new Fraction(" + name + ",1)).ge(" + b.getName() + "))";
    BaseType ret = new LLNumber();
    ret.setName(s);
    return ret;
    }
    return new BaseType();
}

public BaseType lt(BaseType b)
{
    if (b instanceof LLList)
    {
    error(b,"<");
    }
    else if (b instanceof LLNumber)
    {
    String s = "((" + name + " < " + b.getName() + ")?1:0)";
    BaseType ret = new LLNumber();
    ret.setName(s);
    return ret;
    }
    else if (b instanceof LLString)
    {
    error(b,"<");
    }
    else if (b instanceof LLFraction)
    {
    String s = "((new Fraction(" + name + ",1)).lt(" + b.getName() + "))";
    BaseType ret = new LLNumber();
    ret.setName(s);
    return ret;
    }
    return new BaseType();
}

public BaseType le(BaseType b)
{
    if (b instanceof LLList)
    {
    error(b,"<=");
    }
    else if (b instanceof LLNumber)
    {
    String s = "((" + name + " <= " + b.getName() + ")?1:0)";
    BaseType ret = new LLNumber();
    ret.setName(s);
    return ret;
    }
    else if (b instanceof LLString)
    {
    error(b,"<=");
    }
    else if (b instanceof LLFraction)
    {
    String s = "((new Fraction(" + name + ",1)).le(" + b.getName() + "))";
    BaseType ret = new LLNumber();
    ret.setName(s);
    return ret;
    }
    return new BaseType();
}

public BaseType eq(BaseType b)
{
    if (b instanceof LLList)
    {
    error(b,"=");
    }
    else if (b instanceof LLNumber)
    {
    String s = "((" + name + " == " + b.getName() + ")?1:0)";
    BaseType ret = new LLNumber();
    ret.setName(s);
    return ret;
    }
    else if (b instanceof LLString)
    {
    error(b,"=");
    }
    else if (b instanceof LLFraction)
    {
    String s = "((new Fraction(" + name + ",1)).eq(" + b.getName() + "))";
    BaseType ret = new LLNumber();
    ret.setName(s);
    return ret;
    }
    return new BaseType();
}

public BaseType ne(BaseType b)
{
    if (b instanceof LLList)
    {
    error(b,"!=");
    }
    else if (b instanceof LLNumber)
    {
    String s = "((" + name + " != " + b.getName() + ")?1:0)";
    BaseType ret = new LLNumber();
    ret.setName(s);
    return ret;
    }
    else if (b instanceof LLString)
    {
    error(b,"!=");
    }
    else if (b instanceof LLFraction)
    {
    String s = "((new Fraction(" + name + ",1)).ne(" + b.getName() + "))";
    BaseType ret = new LLNumber();
    ret.setName(s);
    return ret;
    }
    return new BaseType();
}

}
```

4. LLFraction.java

```
//LL fraction type semantic rules and code gen
//Stephen Robinson
//v 1.0

public class LLFraction extends BaseType {


    public String getTypeName()
    {
        return "fraction";
    }

    public BaseType assign(BaseType b) {
        if (b instanceof LLList)
        {
        error(b,"<-");
        }
        else if (b instanceof LLNumber)
        {
        //indicates that we are assigning with a function so just
        //append the parameter (b)
        if(scopeCnt == -1)
        {
              String s = name + "new Fraction(" + b.getName() + ", 1))";
              BaseType ret = new LLNumber();
              ret.setName(s);
              return ret;
        }
        else
        {
              String s = name + " = new Fraction(" + b.getName() + ", 1)";
              BaseType ret = new LLFraction();
              ret.setName(s);
              return ret;
        }
        }
        else if (b instanceof LLString)
        {
        error(b,"<-");
        }
        else if (b instanceof LLFraction)
        {
        if(scopeCnt == -1)
        {
              String s = name + b.getName() + ")";
              BaseType ret = new LLNumber();
              ret.setName(s);
              return ret;
        }
        else
        {
              String s = name + " = " + b.getName();
              BaseType ret = new LLFraction();
              ret.setName(s);
              return ret;
        }
        }
        return new BaseType();
    }

    public BaseType append(BaseType b)
    {
        if (b instanceof LLList)
        {
            if(((LLList)b).type instanceof LLNumber)
            {
              String s = b.getName() + ".toFracList().prependPrimitive(" + name +
")";
              BaseType ret = new LLList(new LLFraction());
              ret.setName(s);
              return ret;
            }
            else if(((LLList)b).type instanceof LLFraction)
            {
              String s = b.getName() + "prependPrimitive(" + name + ")";
              BaseType ret = new LLList(new LLFraction());
              ret.setName(s);
              return ret;
            }
            else
```

```
            {
              error(b,".");
            }
        }
        else if (b instanceof LLNumber)
        {
         String s = "(new Listof(\"fraction\", " + name + ", ";
         s += "new Fraction(" + b.getName() + ", 1)))";
         BaseType ret = new LLList(new LLFraction());
         ret.setName(s);
         return ret;
        }
        else if (b instanceof LLString)
        {
         error(b,".");
        }
        else if (b instanceof LLFraction)
        {
         String s = "(new Listof(\"fraction\", " + name + ", ";
         s += b.getName() + "))";
         BaseType ret = new LLList(new LLFraction());
         ret.setName(s);
         return ret;
        }
        return new BaseType();
    }

    public BaseType exp(BaseType b)
    {
        if (b instanceof LLList)
        {
            BaseType tmp = b.getCoreType();

            if(tmp.getTypeName() == "number")
            {
              String s = b.getName() + ".preexp(" + name + ")";
              BaseType ret = getCopy(b);
              ret.setName(s);
              return ret;
            }
            else
            {
              error(b,"^");
            }
        }
        else if (b instanceof LLNumber)
        {
         String s = name + ".exp(" + b.getName() + ")";
         BaseType ret = new LLFraction();
         ret.setName(s);
         return ret;
        }
        else if (b instanceof LLString)
        {
         error(b,"^");
        }
        else if (b instanceof LLFraction)
        {
         error(b,"^");
        }
        return new BaseType("error");
    }

    public BaseType uminus()
    {
         String s = name + ".uminus();";
         BaseType ret = new LLFraction();
         ret.setName(s);
        return ret;
    }

    public BaseType remainder(BaseType b)
    {
        if (b instanceof LLList)
        {
            BaseType tmp = b.getCoreType();

            if(tmp.getTypeName() == "number")
            {
              String s = b.getName() + ".premodulus(" + name + ")";
              BaseType ret = getCopy(b);
              ret.setName(s);
              return ret;
            }
            else
            {
              error(b,"%");
            }
        }
        else if (b instanceof LLNumber)
        {
         String s = name + ".modulus(" + b.getName() + ")";
         BaseType ret = new LLFraction();
         ret.setName(s);
         return ret;
        }
        else if (b instanceof LLString)
        {
         error(b,"%");
        }
        else if (b instanceof LLFraction)
        {
         error(b,"%");
        }
        return new BaseType();
    }

    public BaseType plus(BaseType b)
    {
        if (b instanceof LLList)
        {
            BaseType tmp = b.getCoreType();

            if(tmp.getTypeName() == "number")
            {
              String s = b.getName() + ".toFracList().add(" + name + ")";
              BaseType ret = getCopy(b);
              ret.setName(s);
              return ret;
            }
            else if(tmp.getTypeName() == "fraction")
            {
              String s = b.getName() + ".add(" + name + ")";
              BaseType ret = getCopy(b);
              ret.setName(s);
              return ret;
            }
            else
            {
              error(b,"+");
            }
        }
        else if (b instanceof LLNumber)
        {
        String s = name + ".add(new Fraction(" + b.getName() + ",1))";
        BaseType ret = new LLFraction();
        ret.setName(s);
        return ret;
        }
        else if (b instanceof LLString)
        {
        error(b,"+");
        }
        else if (b instanceof LLFraction)
        {
        String s = name + ".add(" + b.getName() + ")";
        BaseType ret = new LLFraction();
        ret.setName(s);
        return ret;
        }
        return new BaseType();
    }

    public BaseType minus(BaseType b)
    {
        if (b instanceof LLList)
        {
            BaseType tmp = b.getCoreType();

            if(tmp.getTypeName() == "fraction")
            {
              String s = b.getName() + ".preminus(" + name + ")";
              BaseType ret = getCopy(b);
              ret.setName(s);
              return ret;
            }
            else if(tmp.getTypeName() == "number")
            {
              String s = b.getName() + ".toFracList().preminus(" + name + ")";
              BaseType ret = getCopy(b);
              ret.setName(s);
```

```
                return ret;
            }
            else
            {
                error(b,"-");
            }
        }
        else if (b instanceof LLNumber)
        {
         String s = name + ".subtract(new Fraction(" + b.getName() + ",1))";
         BaseType ret = new LLFraction();
         ret.setName(s);
         return ret;
        }
        else if (b instanceof LLString)
        {
         error(b,"-");
        }
        else if (b instanceof LLFraction)
        {
         String s = name + ".subtract(" + b.getName() + ")";
         BaseType ret = new LLFraction();
         ret.setName(s);
         return ret;
        }
        return new BaseType();
    }

    public BaseType mult(BaseType b)
    {
        if (b instanceof LLList)
        {
            BaseType tmp = b.getCoreType();

            if(tmp.getTypeName() == "number")
            {
                String s = b.getName() + ".toFracList().multiply(" + name + ")";
                BaseType ret = getCopy(b);
                ret.setName(s);
                return ret;
            }
            else if(tmp.getTypeName() == "fraction")
            {
                String s = b.getName() + ".multiply(" + name + ")";
                BaseType ret = getCopy(b);
                ret.setName(s);
                return ret;
            }
            else
            {
                error(b,"*");
            }
        }
        else if (b instanceof LLNumber)
        {
         String s = name + ".multiply(new Fraction(" + b.getName() + ",1))";
         BaseType ret = new LLFraction();
        ret.setName(s);
        return ret;
        }
        else if (b instanceof LLString)
        {
        error(b,"*");
        }
        else if (b instanceof LLFraction)
        {
        String s = name + ".multiply(" + b.getName() + ")";
        BaseType ret = new LLFraction();
        ret.setName(s);
        return ret;
        }
        return new BaseType();
    }

    public BaseType intdiv(BaseType b)
    {
        if (b instanceof LLList)
        {
            BaseType tmp = b.getCoreType();

            if(tmp.getTypeName() == "fraction")
            {
              String s = b.getName() + ".predivide(" + name + ")";
              BaseType ret = getCopy(b);
              ret.setName(s);
              return ret;
            }
            else if(tmp.getTypeName() == "number")
            {
              String s = b.getName() + ".toFracList().predivide(" + name + ")";
              BaseType ret = getCopy(b);
              ret.setName(s);
              return ret;
            }
            else
            {
              error(b,"/");
            }
        }
        else if (b instanceof LLNumber)
        {
        String s = name + ".divide(new Fraction(" + b.getName() + ",1))";
        BaseType ret = new LLFraction();
        ret.setName(s);
        return ret;
        }
        else if (b instanceof LLString)
        {
        error(b,"/");
        }
        else if (b instanceof LLFraction)
        {
        String s = name + ".divide(" + b.getName() + ")";
        BaseType ret = new LLFraction();
        ret.setName(s);
        return ret;
        }
        return new BaseType();
    }

    public BaseType gt(BaseType b)
    {
        if(b instanceof LLList)
        {
        error(b,">");
        }
        else if (b instanceof LLNumber)
        {
        String s = name + ".gt(new Fraction(" + b.getName() + ",1))";
        BaseType ret = new LLNumber();
        ret.setName(s);
        return ret;
        }
        else if (b instanceof LLString)
        {
        error(b,">");
        }
        else if (b instanceof LLFraction)
        {
        String s = name + ".gt(" + b.getName() + ")";
        BaseType ret = new LLNumber();
        ret.setName(s);
        return ret;
        }
        return new BaseType();
    }

    public BaseType ge(BaseType b)
    {
        if (b instanceof LLList)
        {
        error(b,">=");
        }
        else if (b instanceof LLNumber)
        {
        String s = name + ".ge(new Fraction(" + b.getName() + ",1))";
        BaseType ret = new LLNumber();
        ret.setName(s);
        return ret;
        }
        else if (b instanceof LLString)
        {
        error(b,">=");
        }
        else if (b instanceof LLFraction)
        {
        String s = name + ".ge(" + b.getName() + ")";
        BaseType ret = new LLNumber();
        ret.setName(s);
        return ret;
        }
        return new BaseType();
    }

    public BaseType lt(BaseType b)
    {
        if (b instanceof LLList)
        {
        error(b,"<");
        }
        else if (b instanceof LLNumber)
        {
        String s = name + ".lt(new Fraction(" + b.getName() + ",1))";
        BaseType ret = new LLNumber();
        ret.setName(s);
        return ret;
        }
        else if (b instanceof LLString)
        {
        error(b,"<");
        }
        else if (b instanceof LLFraction)
        {
        String s = name + ".lt(" + b.getName() + ")";
        BaseType ret = new LLNumber();
        ret.setName(s);
        return ret;
        }
        return new BaseType();
    }

    public BaseType le(BaseType b)
    {
        if (b instanceof LLList)
        {
        error(b,"<=");
        }
        else if (b instanceof LLNumber)
        {
        String s = name + ".le(new Fraction(" + b.getName() + ",1))";
        BaseType ret = new LLNumber();
        ret.setName(s);
        return ret;
        }
        else if (b instanceof LLString)
        {
        error(b,"<=");
        }
        else if (b instanceof LLFraction)
        {
        String s = name + ".le(" + b.getName() + ")";
        BaseType ret = new LLNumber();
        ret.setName(s);
        return ret;
        }
        return new BaseType();
    }
```

```
    public BaseType eq(BaseType b)
    {
        if (b instanceof LLList)
        {
        error(b,"=");
        }
        else if (b instanceof LLNumber)
        {
        String s = name + ".eq(new Fraction(" + b.getName() + ",1))";
        BaseType ret = new LLNumber();
        ret.setName(s);
        return ret;
        }
        else if (b instanceof LLString)
        {
        error(b,"=");
        }
        else if (b instanceof LLFraction)
        {
        String s = name + ".eq(" + b.getName() + ")";
        BaseType ret = new LLNumber();
        ret.setName(s);
        return ret;
        }
        return new BaseType();
    }

    public BaseType ne(BaseType b)
    {
        if (b instanceof LLList)
        {
        error(b,"!=");
        }
        else if (b instanceof LLNumber)
        {
        String s = name + ".ne(new Fraction(" + b.getName() + ",1))";
        BaseType ret = new LLNumber();
        ret.setName(s);
        return ret;
        }
        else if (b instanceof LLString)
        {
        error(b,"!=");
        }
        else if (b instanceof LLFraction)
        {
        String s = name + ".ne(" + b.getName() + ")";
        BaseType ret = new LLNumber();
        ret.setName(s);
        return ret;
        }
        return new BaseType();
    }
}
```

5. LLString.java

```
//LL string type semantic rules and code gen.  pretty robust
//Stephen Robinson
//v 1.0

public class LLString extends BaseType {


    public String getTypeName() {
        return "string";
    }

    public BaseType append(BaseType b)
    {
         if (b instanceof LLList)
        {
                if(((LLList)b).type instanceof LLString)
                {
                      String s = b.getName() + ".prependPrimitive(";
                      s += name + ")";
                      BaseType ret = new LLList(new LLString());
                      ret.setName(s);
                      return ret;
                }
                else
                {
                       error(b,".");
                }
        }
        else if (b instanceof LLNumber)
        {
         error(b,".");
        }
        else if (b instanceof LLString)
        {
         String s = "(new Listof(\"string\", " + name;
         s += ", " + b.getName() + "))";
         BaseType ret = new LLList(new LLString());
         ret.setName(s);
         return ret;
        }
        else if (b instanceof LLFraction)
        {
               error(b,".");
        }
        return new BaseType();
    }

    public BaseType plus(BaseType b)
    {
        if (b instanceof LLList)
        {
         BaseType tmp = b.getCoreType();
         if(tmp instanceof LLString)
         {
               String s = b.getName() + ".preadd(";
               s += name + ")";
               BaseType ret = getCopy(b);
               ret.setName(s);
               return ret;
         }
         else
         {
               error(b,"+");
         }
        }
        else if (b instanceof LLNumber)
        {
         error(b,"+");
        }
        else if (b instanceof LLString)
        {
         String s = name + " + " + b.getName();
         BaseType ret = new LLString();
         ret.setName(s);
         return ret;
        }
        else if (b instanceof LLFraction)
        {
         error(b,"+");
        }
        return new BaseType();
    }

    public BaseType index(BaseType b)
    {
        if (b instanceof LLList)
        {
         error(b,"[]");
        }
        else if (b instanceof LLNumber)
        {
         String s = name + ".charAt(" + b.getName() + ")";
         BaseType ret = new LLString();
         ret.setName(s);
         return ret;
        }
        else if (b instanceof LLString)
        {
         error(b,"[]");
        }
        else if (b instanceof LLFraction)
        {
         error(b,"[]");
        }
        return new BaseType();
    }

    public BaseType assign(BaseType b)
    {
        if (b instanceof LLList)
        {
         error(b,"<-");
```

```
    }
    else if (b instanceof LLNumber)
    {
     error(b,"<-");
    }
    else if (b instanceof LLString)
    {
     String s = name + " = " + b.getName();
     BaseType ret = new LLString();
     ret.setName(s);
     return ret;
    }
    else if (b instanceof LLFraction)
    {
     error(b,"<-");
    }
    return new BaseType();
}

public BaseType size()
{
     String s = name + ".length()";
     BaseType ret = new LLNumber();
     ret.setName(s);
     return ret;
}

public BaseType eq(BaseType b)
{
    if (b instanceof LLList)
    {
     error(b,"=");
    }
    else if(b instanceof LLNumber)
    {
     error(b,"=");
    }
    else if (b instanceof LLString)
    {
     String s = "((" + name + " == " + b.getName() + ")?1:0)";
     BaseType ret = new LLNumber();
     ret.setName(s);
     return ret;
    }
    else if (b instanceof LLFraction)
    {
     error(b,"=");
    }
    return new BaseType();
}

public BaseType ne(BaseType b)
{
    if (b instanceof LLList)
    {
     error(b,"!=");
    }
```

```
        else if (b instanceof LLNumber)
        {
         error(b,"!=");
        }
        else if (b instanceof LLString)
        {
         String s = "((" + name + " != " + b.getName() + ")?1:0)";
         BaseType ret = new LLNumber();
         ret.setName(s);
         return ret;
        }
        else if (b instanceof LLFraction)
        {
         error(b,"!=");
        }
        return new BaseType();
    }
}
```

6. LLList.java

```
//LL list type semantic rules and code gen
//the code gen in this class is ugly and wasn't working too well
//this is partly due to some strange implementations in Listof and partly
//due to the complexity of lists in LL.
//
//Stephen Robinson
//v 1.1  Code gen is working well for 1-Dimensional lists.  Multidimensional lists
//are a lost cause.

public class LLList extends BaseType {

    BaseType type;

    public LLList(BaseType t)
    {
        type = t;
    }

    public String getTypeName()
    {
        return "listof";
    }

    public BaseType uminus()
    {
         BaseType l = type.getCoreType();
         if(l instanceof LLNumber)
         {
               String s = name + ".uminus()";
               BaseType ret = getCopy(this);
               ret.setName(s);
               return ret;
         }
         else if(l instanceof LLFraction)
         {
            String s = name + ".uminus()";
            BaseType ret = getCopy(this);
            ret.setName(s);
            return ret;
        }
        else
        {
            error("-");
        }
        return new BaseType();
    }
  //this should be allowed if the core type of the lists match
  //and this has the same or more dimensions than b
    public BaseType assign(BaseType b)
    {
        //cheap hack to determine if we don't need an equals sign because
        //we're calling a function
        if(scopeCnt == -1)
        {
            if(type.getTypeName() == b.getTypeName())
            {
                BaseType retType = getCopy(b);
                retType.setName(name + b.getName() + ")");
                return retType;
            }
            else if(type instanceof LLNumber && b instanceof LLFraction)
            {
                BaseType retType = getCopy(b);
                retType.setName(name + b.getName() + ".getNumerator() / " +
b.getName() +".getDenominator())");
                return retType;
            }
            else if(type instanceof LLFraction && b instanceof LLNumber)
            {
                BaseType retType = getCopy(b);
                retType.setName(name + "new Fraction(" +  b.getName() +
",1)");

                return retType;
            }
        }

        BaseType localType = this;
        BaseType bType = b;
        while(localType instanceof LLList && bType instanceof LLList)
        {
            localType = ((LLList)localType).type;
            bType = ((LLList)bType).type;
        }

        if(bType instanceof LLList)
        {
            error(b,"<-");
            return new BaseType();
        }

            localType = localType.getCoreType();
            if(!(b instanceof LLList))
```

```
                {
                        if(b.getTypeName() == localType.getTypeName())
                        {
                                String s = name + ".set(" + b.getName() + ")";
                                BaseType ret = getCopy(b);
                                ret.setName(s);
                                return ret;
                        }
                        else
                        {
                                error(b,"<-");
                        return new BaseType();
              }
                }
                else if(localType.getTypeName() == bType.getTypeName())
        {
                        String s = name + ".content = " + b.getName() + ".content";
                        BaseType ret = getCopy(b);
                        ret.setName(s);
                        return ret;
        }
        else if(localType instanceof LLNumber && bType instanceof LLFraction)
        {
                        String s = name + ".content = " + b.getName() +
".toNumList().content";
                        BaseType ret = getCopy(b);
                        ret.setName(s);
                        return ret;
        }
        else if(localType instanceof LLFraction && bType instanceof LLNumber)
        {
                String s = name + ".content = " + b.getName() +
".toFracList().content";
                        BaseType ret = getCopy(b);
                        ret.setName(s);
                        return ret;
        }
        else
        {
                error(b,"<-");
        }
       return new BaseType();
    }

    public BaseType size()
    {
        String s = name + ".size()";
        BaseType ret = new LLNumber();
       ret.setName(s);
       return ret;
    }

    public BaseType append(BaseType b)
    {
        //we can append two things that have a difference in dimension of one or
less
        //For example, {3,4} . 5 = {3,4,5}
```

```
//and {3,4} . {5,6} = {3,4,5,6}
//and {1,2;3,4} . {5,6} = {1,2;3,4;5,6} note that it is not {1,2,5;3,4,6}

BaseType localType = this;
BaseType bType = b;
boolean localHasMoreDims = true;
boolean sameDims = true;
while(localType instanceof LLList && bType instanceof LLList)
{
      localType = ((LLList)localType).type;
      bType = ((LLList)bType).type;
}

if(bType instanceof LLList)
{
      if(((LLList)bType).type instanceof LLList)
      {
            error(b, ".");
            return new BaseType();
      }
      else
      {
            localHasMoreDims = false;
            bType = bType.getCoreType();
      }
}
else if(localType instanceof LLList)
{
      if(((LLList)localType).type instanceof LLList)
      {
            error(b, ".");
            return new BaseType();
      }
      else
      {
            localHasMoreDims = true;
            localType = localType.getCoreType();
      }
}
else
{
      sameDims = true;
}

//return deeper list structure if types match
if(localType.getTypeName() == bType.getTypeName())
{
      if(localHasMoreDims)
      {
            if(b instanceof LLList)
            {
            String s = name + ".appendList(" + b.getName() + ")";
            BaseType ret = getCopy(this);
            ret.setName(s);
            return ret;
            }
            else
```

```
                    {
                    String s = name + ".appendPrimitive(" +
(bType.getTypeName().equals("number") ? "new Integer(" + b.getName() + ")" :
b.getName()) + ")";
                    BaseType ret = getCopy(this);
                    ret.setName(s);
                    return ret;
                    }
              }
              else
              {
                    String s = b.getName() + ".prependList(" + name + ")";
                    BaseType ret = getCopy(b);
                    ret.setName(s);
                    return ret;
              }
        }
        else if(localType instanceof LLNumber && bType instanceof LLFraction)
        {
              if(localHasMoreDims)
              {
                    if(b instanceof LLList)
                    {
                    String s = name + ".toFracList().appendList(" + b.getName() +
")";
                    BaseType ret = getListAsFractionType(this);
                    ret.setName(s);
                    return ret;
                    }
                    else
                    {
                    String s = name + ".toFracList().appendPrimitive(" +
b.getName() + ")";
                    BaseType ret = getListAsFractionType(this);
                    ret.setName(s);
                    return ret;
                    }
              }
              else
              {
                    String s = b.getName() + ".prependList(" + name +
".toFracList())";
                    BaseType ret = getCopy(b);
                    ret.setName(s);
                    return ret;
              }
        }
        else if(localType instanceof LLFraction && bType instanceof LLNumber)
        {
              if(localHasMoreDims)
              {
                    if(b instanceof LLList)
                    {
                    String s = name + ".appendList(" + b.getName() +
".toFracList())";
                    BaseType ret = getCopy(this);
                    ret.setName(s);
                    return ret;
                    }
                    else
                    {
                    String s = name + ".appendPrimitive(" + b.getName() +
".toFracList())";
                    BaseType ret = getCopy(this);
                    ret.setName(s);
                    return ret;
                    }
                }
                else
                {
                    String s = b.getName() + ".toFracList().prependList(" + name +
")";
                    BaseType ret = getCopy(b);
                    ret.setName(s);
                    return ret;
                }
            }
            else
            {
                b.error(".");
            }
            return new BaseType();
        }


        public BaseType plus(BaseType b)
        {

            //pairwise add.  If one list is longer than the other, the sum
            //will have length of the longer and will assume that the shorter
            //had zeroes in all missing entries

            //this should be allowed if the core type of the lists match
            BaseType localType = this;
            BaseType bType = b;
            boolean localHasMoreDims = true;
            while(localType instanceof LLList && bType instanceof LLList)
            {
                localType = ((LLList)localType).type;
                bType = ((LLList)bType).type;
            }
            if(bType instanceof LLList)  //b has more dimensions
            {
                localHasMoreDims = false;
                bType = bType.getCoreType();
            }
            else if(localType instanceof LLList)  //local has more dimensions
            {
                localHasMoreDims = true;
                localType = localType.getCoreType();
            }

            //return deeper list structure if types match
            if(localType.getTypeName() == bType.getTypeName())
        {
              if(localHasMoreDims)
              {
                    String s = name + ".add(" + b.getName() + ")";
                    BaseType ret = getCopy(this);
                    ret.setName(s);
                    return ret;
              }
              else
              {
                    String s = b.getName() + ".add(" + name + ")";
                    BaseType ret = getCopy(b);
                    ret.setName(s);
                    return ret;
              }
        }
        else if(localType instanceof LLNumber && bType instanceof LLFraction)
        {
              if(localHasMoreDims)
              {
                    String s = name + ".toFracList().append(" + b.getName() + ")";
                    BaseType ret = getListAsFractionType(this);
                    ret.setName(s);
                    return ret;
              }
              else
              {
                    String s = b.getName() + ".prepend(" + name +
".toFracList())";
                    BaseType ret = getCopy(b);
                    ret.setName(s);
                    return ret;
              }
        }
        else if(localType instanceof LLFraction && bType instanceof LLNumber)
        {
              if(localHasMoreDims)
              {
                    String s = name + ".append(" + b.getName() + ".toFracList())";
                    BaseType ret = getCopy(this);
                    ret.setName(s);
                    return ret;
              }
              else
              {
                    String s = b.getName() + ".toFracList().prepend(" + name +
")";
                    BaseType ret = getCopy(b);
                    ret.setName(s);
                    return ret;
              }
        }
        else
        {
              b.error("+");
        }
        return new BaseType();
```

```
    }

    public BaseType exp(BaseType b)
    {
        //this should be allowed if the core type of the first list is a number or 
fraction
        //and the second is a number
        BaseType localType = this;
        BaseType bType = b;
        boolean localHasMoreDims = true;
        while(localType instanceof LLList && bType instanceof LLList)
        {
              localType = ((LLList)localType).type;
              bType = ((LLList)bType).type;
        }
        if(bType instanceof LLList)  //b has more dimensions
        {
              localHasMoreDims = false;
              bType = bType.getCoreType();
        }
        else if(localType instanceof LLList)  //local has more dimensions
        {
              localHasMoreDims = true;
              localType = localType.getCoreType();
        }

        //return deeper list structure if types match
        if(bType instanceof LLNumber)
        {
              if(localType instanceof LLNumber)
              {
                    if(localHasMoreDims)
                    {
                          String s = name + ".exp(" + b.getName() + ")";
                          BaseType ret = getCopy(this);
                    ret.setName(s);
                          return ret;
                    }
              else
              {
                          String s = b.getName() + ".preexp(" + name + ")";
                          BaseType ret = getCopy(b);
                    ret.setName(s);
                          return ret;
              }
              }
              else if(localType instanceof LLFraction)
              {
                    if(localHasMoreDims)
                    {
                          String s = name + ".exp(" + b.getName() + ")";
                          BaseType ret = getCopy(this);
                    ret.setName(s);
                          return ret;
                    }
              else
              {
```

```
                        String s = b.getName() + ".preexp(" + name + ")";
                        BaseType ret = getListAsFractionType(b);
                    ret.setName(s);
                        return ret;
                }
                }
                else
                {
                    error(b, "^");
                }
            }
            else
            {
                error(b, "^");
            }
            return new BaseType();
        }

        public BaseType minus(BaseType b)
        {
            //this should be allowed if the core type of the first list is a number or
fraction
            //and the second is a number or fraction
            BaseType localType = this;
            BaseType bType = b;
            boolean localHasMoreDims = true;
            while(localType instanceof LLList && bType instanceof LLList)
            {
                localType = ((LLList)localType).type;
                bType = ((LLList)bType).type;
            }
            if(bType instanceof LLList)  //b has more dimensions
            {
                localHasMoreDims = false;
                bType = bType.getCoreType();
            }
            else if(localType instanceof LLList)  //local has more dimensions
            {
                localHasMoreDims = true;
                localType = localType.getCoreType();
            }

            //return deeper list structure if types match
            if(localType.getTypeName() == bType.getTypeName() && bType.getTypeName()
!= "string")
            {
                if(localHasMoreDims)
                {
                    String s = name + ".minus(" + b.getName() + ")";
                    BaseType ret = getCopy(this);
                    ret.setName(s);
                    return ret;
                }
                else
                {
                    String s = b.getName() + ".preminus(" + name + ")";
                    BaseType ret = getCopy(b);
```

```
                        ret.setName(s);
                        return ret;
                }
        }
        else if(localType instanceof LLNumber && bType instanceof LLFraction)
        {
                if(localHasMoreDims)
                {
                        String s = name + ".toFracList().minus(" + b.getName() + ")";
                        BaseType ret = getListAsFractionType(this);
                        ret.setName(s);
                        return ret;
                }
                else
                {
                        String s = b.getName() + ".preminus(" + name +
".toFracList())";
                        BaseType ret = getCopy(b);
                        ret.setName(s);
                        return ret;
                }
        }
        else if(localType instanceof LLFraction && bType instanceof LLNumber)
        {
                if(localHasMoreDims)
                {
                        if(b instanceof LLNumber)
                        {
                        String s = name + ".minus(new Fraction(" + b.getName() +
",1))";

                        BaseType ret = getCopy(this);
                        ret.setName(s);
                        return ret;
                        }
                        else
                        {
                                String s = name + ".minus(" + b.getName() +
".toFracList())";
                        BaseType ret = getCopy(this);
                        ret.setName(s);
                        return ret;
                        }
                }
                else
                {
                        String s = b.getName() + ".toFracList().preminus(" + name +
")";
                        BaseType ret = getListAsFractionType(b);
                        ret.setName(s);
                        return ret;
                }
        }
        else
        {
                error(b, "-");
        }
        return new BaseType();
```

```
    }

    public static BaseType getListAsFractionType(BaseType a)
    {
        if(a instanceof LLList)
        {
            return new LLList( getListAsFractionType(((LLList)a).type) );
        }
        else
        {
            return new LLFraction();
        }
    }

    public BaseType mult(BaseType b)
    {
        //this should be allowed if the core type of the first list is a number or
fraction
        //and the second is a number or fraction
        BaseType localType = this;
        BaseType bType = b;
        boolean localHasMoreDims = true;
        while(localType instanceof LLList && bType instanceof LLList)
        {
            localType = ((LLList)localType).type;
            bType = ((LLList)bType).type;
        }
        if(bType instanceof LLList)  //b has more dimensions
        {
            localHasMoreDims = false;
            bType = bType.getCoreType();
        }
        else if(localType instanceof LLList)  //local has more dimensions
        {
            localHasMoreDims = true;
            localType = localType.getCoreType();
        }

        //return deeper list structure if types match
        if(localType.getTypeName() == bType.getTypeName() && bType.getTypeName()
!= "string")
        {
            if(localHasMoreDims)
            {
                String s = name + ".multiply(" + b.getName() + ")";
                BaseType ret = getCopy(this);
                ret.setName(s);
                return ret;
            }
            else
            {
                String s = b.getName() + ".multiply(" + name + ")";
                BaseType ret = getCopy(b);
                ret.setName(s);
                return ret;
            }
        }
        else if(localType instanceof LLNumber && bType instanceof LLFraction)
        {
              if(localHasMoreDims)
              {
                    String s = name + ".toFracList().multiply(" + b.getName() +
")";
                    BaseType ret = getListAsFractionType(this);
                    ret.setName(s);
                    return ret;
              }
              else
              {
                    String s = b.getName() + ".multiply(" + name +
".toFracList())";
                    BaseType ret = getCopy(b);
                    ret.setName(s);
                    return ret;
              }
        }
        else if(localType instanceof LLFraction && bType instanceof LLNumber)
        {
              if(localHasMoreDims)
              {
                    if(b instanceof LLNumber)
                    {
                    String s = name + ".multiply(new Fraction(" + b.getName() +
",1))";
                    BaseType ret = getCopy(this);
                    ret.setName(s);
                    return ret;
                    }
                    else
                    {
                          String s = name + ".multiply(" + b.getName() +
".toFracList())";
                    BaseType ret = getCopy(this);
                    ret.setName(s);
                    return ret;
                    }
              }
              else
              {
                    String s = b.getName() + ".toFracList().multiply(" + name +
")";
                    BaseType ret = getListAsFractionType(b);
                    ret.setName(s);
                    return ret;
              }
        }
        else
        {
              error(b, "*");
        }
        return new BaseType();
    }

    public BaseType index(BaseType b)
    {
        if (b instanceof LLList)
        {
        error(b, "[]");
        }
        else if (b instanceof LLNumber)
        {
        String s = "";
        if(type instanceof LLList)
        {
              s = "(Listof)";
        }
        else if(type instanceof LLNumber)
        {
              s = "(Integer)";
        }
        else if(type instanceof LLString)
        {
              s = "(String)";
        }
        else if(type instanceof LLFraction)
        {
              s = "(Fraction)";
        }
        s += name + ".content.get(" + b.getName() + ")";
        BaseType ret = getCopy(type);
        ret.setName(s);
        return ret;
        }
        else if (b instanceof LLString)
        {
        error(b, "[]");
        }
        else if (b instanceof LLFraction)
        {
        error(b, "[]");
        }
        return new BaseType();
    }

    public BaseType intdiv(BaseType b)
    {
        //this should be allowed if the core type of the first list is a number or
fraction
        //and the second is a number or fraction
        BaseType localType = this;
        BaseType bType = b;
        boolean localHasMoreDims = true;
        while(localType instanceof LLList && bType instanceof LLList)
        {
              localType = ((LLList)localType).type;
              bType = ((LLList)bType).type;
        }
        if(bType instanceof LLList)  //b has more dimensions
        {
              localHasMoreDims = false;
              bType = bType.getCoreType();
        }
        else if(localType instanceof LLList)  //local has more dimensions
        {
              localHasMoreDims = true;
              localType = localType.getCoreType();
        }

        //return deeper list structure if types match
        if(localType.getTypeName() == bType.getTypeName() && bType.getTypeName()
!= "string")
        {
              if(localHasMoreDims)
              {
                    String s = name + ".divide(" + b.getName() + ")";
                    BaseType ret = getCopy(this);
                    ret.setName(s);
                    return ret;
              }
              else
              {
                    String s = b.getName() + ".predivide(" + name + ")";
                    BaseType ret = getCopy(b);
                    ret.setName(s);
                    return ret;
              }
        }
        else if(localType instanceof LLNumber && bType instanceof LLFraction)
        {
              if(localHasMoreDims)
              {
                    String s = name + ".toFracList().divide(" + b.getName() + ")";
                    BaseType ret = getListAsFractionType(this);
                    ret.setName(s);
                    return ret;
              }
              else
              {
                    String s = b.getName() + ".predivide(" + name +
".toFracList())";
                    BaseType ret = getCopy(b);
                    ret.setName(s);
                    return ret;
              }
        }
        else if(localType instanceof LLFraction && bType instanceof LLNumber)
        {
              if(localHasMoreDims)
              {
                    if(b instanceof LLNumber)
                    {
                    String s = name + ".divide(new Fraction(" + b.getName() +
",1))";
                    BaseType ret = getCopy(this);
                    ret.setName(s);
                    return ret;
                    }
                    else
                    {
                          String s = name + ".divide(" + b.getName() +
".toFracList())";
                    BaseType ret = getCopy(this);
                    ret.setName(s);
                    return ret;
                    }
              }
              else
              {
                    String s = b.getName() + ".toFracList().predivide(" + name +
")";
                    BaseType ret = getListAsFractionType(b);
                    ret.setName(s);
                    return ret;
              }
        }
        else
        {
              error(b, "/");
        }
        return new BaseType();
    }

    public BaseType remainder(BaseType b)
    {
        //this should be allowed if the core type of the first list is a number or
fraction
        //and the second is a number
        BaseType localType = this;
        BaseType bType = b;
        boolean localHasMoreDims = true;
        while(localType instanceof LLList && bType instanceof LLList)
        {
              localType = ((LLList)localType).type;
              bType = ((LLList)bType).type;
        }
        if(bType instanceof LLList)  //b has more dimensions
        {
              localHasMoreDims = false;
              bType = bType.getCoreType();
        }
        else if(localType instanceof LLList)  //local has more dimensions
        {
              localHasMoreDims = true;
              localType = localType.getCoreType();
        }

        //return deeper list structure if types match
        if(bType instanceof LLNumber)
        {
              if(localType instanceof LLNumber)
              {
                    if(localHasMoreDims)
                    {
                          String s = name + ".modulus(" + b.getName() + ")";
                          BaseType ret = getCopy(this);
```

```
                        ret.setName(s);
                              return ret;
                        }
                  else
                  {
                              String s = b.getName() + ".premodulus(" + name + ")";
                              BaseType ret = getCopy(b);
                        ret.setName(s);
                              return ret;
                  }
                  }
                  else if(localType instanceof LLFraction)
                  {
                        if(localHasMoreDims)
                        {
                              String s = name + ".modulus(" + b.getName() + ")";
                              BaseType ret = getCopy(this);
                        ret.setName(s);
                              return ret;
                        }
                  else
                  {
                              String s = b.getName() + ".premodulus(" + name + ")";
                              BaseType ret = getListAsFractionType(b);
                        ret.setName(s);
                              return ret;
                  }
                  }
                  else
                  {
                        error(b, "%");
                  }
            }
            else
            {
                  error(b, "%");
            }
            return new BaseType();
      }

      public BaseType fract(BaseType b)
      {
            //this should be allowed if both types are int
            BaseType localType = this;
            BaseType bType = b;
            boolean localHasMoreDims = true;
            while(localType instanceof LLList && bType instanceof LLList)
            {
                  localType = ((LLList)localType).type;
                  bType = ((LLList)bType).type;
            }
            if(bType instanceof LLList)  //b has more dimensions
            {
                  localHasMoreDims = false;
                  bType = bType.getCoreType();
            }
            else if(localType instanceof LLList)  //local has more dimensions
```

```
        {
              localHasMoreDims = true;
              localType = localType.getCoreType();
        }

        //return deeper list structure if types match
        if(bType instanceof LLNumber && localType instanceof LLNumber)
        {
              if(localHasMoreDims)
              {
                    String s = name + ".fract(" + b.getName() + ")";
                    BaseType ret = getListAsFractionType(this);
                    ret.setName(s);
                    return ret;
              }
              else
              {
                    String s = name + ".prefract(" + b.getName() + ")";
                    BaseType ret = getListAsFractionType(b);
                    ret.setName(s);
                    return ret;
              }
        }
        else
        {
              error(b, "//");
        }
        return new BaseType();
    }
}
```

7. SymbolTable.java

```
//LL symbol table class.  pretty straightforward
//Stephen Robinson
//v 1.0

import java.util.*;

public class SymbolTable extends HashMap {

  SymbolTable parentScope;
  int scopeCnt;

    public final SymbolTable parent() {
        return parentScope;
    }

    public final int cnt()
    {
        return scopeCnt;
    }

    public final void setParent(SymbolTable parent) {
        parentScope = parent;
    }
    public SymbolTable(SymbolTable parent, int n)
    {
        parentScope = parent;
        scopeCnt = n;
    }

    public SymbolTable() {
        parentScope = null;
        scopeCnt = 0;
    }

    // get in current scope and in all parents' scopes
    public final BaseType getVar(String name) {
         SymbolTable st = this;

         Object x = st.get(name);

         while(x == null && st.parent() != null )
         {
                st = st.parent();
                x = st.get(name);
         }
         if(x != null)
         {
                ((BaseType)x).setScope(st.cnt());
         }
         return (BaseType) x;
   }
    // get in current scope
    public final BaseType getLocalVar(String name)
    {
         SymbolTable st = this;
         Object x = st.get(name);
         return (BaseType) x;
   }

    public final void addVar(String name, BaseType dt)
    {
         if(this.containsKey(name))
         {
                throw new LLException("variable " + name + " already declared in
this scope!");
         }
         else
         {
                this.put(name, dt);
         }
    }
}
```

8. LLContainer.java

```
import java.util.*;
import java.io.*;
```

```java
import antlr.CommonAST;
import antlr.collections.AST;
import antlr.RecognitionException;
import antlr.TokenStreamException;
import antlr.TokenStreamIOException;

//This is the class that implements all the semantic analysis and code generation
//It is used by the walker
//All the data type classes and the symbol table are called through this bloated
class.
//Stephen Robinson
//v1.0
public class LLContainer
{

    SymbolTable symt;
    int scopeIndex;
    int regIndex;
    FileOutputStream out;

    public LLContainer()
    {
        scopeIndex = 0;
        regIndex = 0;
        symt = new SymbolTable(null, scopeIndex);
        try
        {
            out = new FileOutputStream("out.java");
        }
        catch (Exception e)
        {
            throw new LLException(e.toString());
        }
        printHeader();
    }

    public void enterScope()
    {
        symt = new SymbolTable(symt, ++scopeIndex);
    }

    public void leaveScope()
    {
        if(symt.parent() == null)
        {
            throw new LLException("cannot leave scope");
        }
        else
        {
            symt = symt.parent();
            scopeIndex--;
        }
    }

    public void declareVar(String name, BaseType x)
    {
        symt.addVar(name, x);
    }

    public static int getNumber(String s)
    {
        return Integer.parseInt(s);
    }

    public static BaseType getDataType(String s)
    {
        if (s.equals("number"))
            return new LLNumber();
        else if (s.equals("fraction"))
            return new LLFraction();
        else if (s.equals("string"))
            return new LLString();

        throw new LLException("Unknown data type");
    }

    public BaseType getVariable(String s)
    {
        // default static scoping
         //System.out.println("in get variable:" + s);
        BaseType x = symt.getVar(s);
        //System.out.println(x.toString());
        if (null == x)
        {
            throw new LLException(s + " not declared");
        }
        return x;
    }

    public void printHeader()
    {
         LLOutput("public class out {");
         LLOutput("public static void main(String[] args){");
         LLOutput("LLIO llio = new LLIO();");
    }

    public void printFooter()
    {
         LLOutput("}}");
    }

    public void LLOutput(String s)
    {
         try
         {
               out.write(s.getBytes());
               out.write('\n');
         }
         catch (Exception e)
         {
               throw new LLException(e.toString());
         }
    }
```

```
public BaseType assign(BaseType a, BaseType b)
{
	BaseType bt;
	if((bt = symt.getVar(a.getName())) == null)
	{
		//throw new LLException("cannot assign to a literal");
	}
	else
	{
		a.setName( a.getName() + bt.scopeCnt );
	}

	if((bt = symt.getVar(b.getName())) != null)
	{
		//b is a variable so append a scope number
		b.setName(b.getName() + bt.scopeCnt);
	}

	//verify that these can be assigned
	BaseType retType = a.assign(b);

	LLOutput(retType.getName() + ";");
	return b;
}

//used when assigning to a temp variable
public void assignOutput(BaseType a, BaseType b)
{
	//test retType
	if(a instanceof LLNumber)
	{
		LLOutput(a.getName() +  " = " + b.getName() + ";");
	}
	else if(a instanceof LLFraction)
	{
		LLOutput(a.getName() + " = " + b.getName() + ";");
	}
	else if(a instanceof LLList)
	{
		LLOutput(a.getName() + " = " + b.getName() + ";");
	}
	else if(a instanceof LLString)
	{
		LLOutput(a.getName() +  " = " + b.getName() + ";");
	}
}

public void initIF(BaseType a)
{
	String s;
	BaseType tmp;
	if((tmp = symt.getVar(a.name)) != null)
	{
		s = a.getName() + tmp.scopeCnt;
	}
	else
	{
```

```
            s = a.getName();
        }
        LLOutput("if(" + s + " != 0)");
        enterScope();
    }

    public void initUntil(BaseType a)
    {
        String s;
        BaseType tmp = getCopy(a);
        if((tmp = symt.getVar(a.name)) != null)
        {
            s = a.getName() + tmp.scopeCnt;
        }
        else
        {
            s = a.getName();
        }

        LLOutput("while(" + s + " == 0){");
        enterScope();
    }

    public void initFor(BaseType a)
    {
        String s;
        BaseType tmp = getCopy(a);
        if((tmp = symt.getVar(a.name)) != null)
        {
            s = a.getName() + tmp.scopeCnt;
        }
        else
        {
            s = a.getName();
        }
        enterScope();
        LLNumber n = new LLNumber();
        n.setName("variable_nth_");
        declareVar("variable_nth_", n);
        String cntName = "variable_nth_" + Integer.toString(scopeIndex);
        String arg = "for(int " + cntName;
        arg += " = 0; " + cntName + " < " + s;
        arg += "; " + cntName + "++){";
        LLOutput(arg);
    }

    public BaseType regClone(BaseType a)
    {
        if(a instanceof LLNumber)
        {
            return new LLNumber();
        }
        else if(a instanceof LLString)
        {
            return new LLString();
        }
        else if(a instanceof LLFraction)
        {
            return new LLFraction();
        }
        if(a instanceof LLList)
        {
            return new LLList( regClone(((LLList)a).type) );
        }
        return new BaseType();
    }

    //used to declare temp variables
    public void declareOutput(BaseType b)
    {
        if(b instanceof LLNumber)
        {
            LLOutput("int " + b.getName() + ";");
        }
        else if(b instanceof LLFraction)
        {
            LLOutput("Fraction " + b.getName() + ";");
        }
        else if(b instanceof LLString)
        {
            LLOutput("String " + b.getName() + ";");
        }
        else if(b instanceof LLList)
        {
            LLOutput("Listof " + b.getName() + ";");
        }
    }

    public String opGetName(BaseType a)
    {
        String s1 = "";
        BaseType tmp;
        if((tmp = symt.getVar(a.getName())) == null)
            s1 = a.getName();
        else
            s1 = a.getName() + tmp.scopeCnt;

        return s1;
    }

    public BaseType doRegStuff(BaseType retType)
    {
        BaseType register = regClone(retType);
        String name;
        name = "reg" + Integer.toString(regIndex) + "x";
        register.setName(name);
        declareVar(name, register);
        register.setName(name + Integer.toString(scopeIndex));
        declareOutput(register);
        assignOutput(register, retType);
        regIndex ++;
        register.setName(name);
        return register;
    }
```

```
public BaseType plus(BaseType a, BaseType b)
{
    a.setName(opGetName(a));
    b.setName(opGetName(b));

    //rettype.name is the string for the addition
    BaseType retType = a.plus(b);

    return doRegStuff(retType);
}

public BaseType minus(BaseType a, BaseType b)
{
    a.setName(opGetName(a));
    b.setName(opGetName(b));

    BaseType retType = a.minus(b);

    return doRegStuff(retType);
}

public BaseType mult(BaseType a, BaseType b)
{
    a.setName(opGetName(a));
    b.setName(opGetName(b));

    BaseType retType = a.mult(b);

    return doRegStuff(retType);
}

public BaseType intdiv(BaseType a, BaseType b)
{
    a.setName(opGetName(a));
    b.setName(opGetName(b));

    BaseType retType = a.intdiv(b);

    return doRegStuff(retType);
}

public BaseType exp(BaseType a, BaseType b)
{
    a.setName(opGetName(a));
    b.setName(opGetName(b));

    BaseType retType = a.exp(b);

    return doRegStuff(retType);
}

public BaseType remainder(BaseType a, BaseType b)
{
    a.setName(opGetName(a));
    b.setName(opGetName(b));
```

```
        BaseType retType = a.remainder(b);

        return doRegStuff(retType);
    }

    public BaseType fract(BaseType a, BaseType b)
    {
        a.setName(opGetName(a));
        b.setName(opGetName(b));

        BaseType retType = a.fract(b);

        return doRegStuff(retType);
    }

    public BaseType append(BaseType a, BaseType b)
    {
        a.setName(opGetName(a));
        b.setName(opGetName(b));

        BaseType retType = a.append(b);

        return doRegStuff(retType);
    }

    public BaseType eq(BaseType a, BaseType b)
    {
        a.setName(opGetName(a));
        b.setName(opGetName(b));

        BaseType retType = a.eq(b);

        return doRegStuff(retType);
    }

    public BaseType lt(BaseType a, BaseType b)
    {
        a.setName(opGetName(a));
        b.setName(opGetName(b));

        BaseType retType = a.lt(b);

        return doRegStuff(retType);
    }

    public BaseType le(BaseType a, BaseType b)
    {
        a.setName(opGetName(a));
        b.setName(opGetName(b));

        BaseType retType = a.le(b);

        return doRegStuff(retType);
    }

    public BaseType ge(BaseType a, BaseType b)
    {
        a.setName(opGetName(a));
        b.setName(opGetName(b));

        BaseType retType = a.ge(b);

        return doRegStuff(retType);
    }

    public BaseType ne(BaseType a, BaseType b)
    {
        a.setName(opGetName(a));
        b.setName(opGetName(b));

        BaseType retType = a.ne(b);

        return doRegStuff(retType);
    }

    public BaseType gt(BaseType a, BaseType b)
    {
        a.setName(opGetName(a));
        b.setName(opGetName(b));

        BaseType retType = a.gt(b);

        return doRegStuff(retType);
    }


    public BaseType index(BaseType a, BaseType b)
    {
        a.setName(opGetName(a));
        b.setName(opGetName(b));

        BaseType retType = a.index(b);

        return doRegStuff(retType);
    }

    public BaseType leftIndex(BaseType a, BaseType b)
    {
        a.setName(opGetName(a));
        b.setName(opGetName(b));

        BaseType retType;
        if(b instanceof LLNumber)
        {
        if(a instanceof LLList)
        {
              retType = getCopy(((LLList)a).type);
              String reg = "reg" + regIndex + "x" + scopeIndex;
              LLOutput("int[] " + reg + " = new int[1];");
              LLOutput(reg + "[0] = " + b.getName() + ";");
              String s = a.getName() + ".realset(" + reg + ", ";
              retType.setName(s);
              retType.setScope(-1);
              regIndex++;
```

```
                return retType;
            }

            }
            throw new LLException("cannot assign to the index of that");
            //return new BaseType();
        }

        public BaseType leftSize(BaseType a)
        {
            a.setName(opGetName(a));

            if(a instanceof LLList)
            {
                BaseType retType = new LLNumber();
                String s = a.getName() + " = " + a.getName() + ".content.subList(0,
";
                retType.setName(s);
                retType.setScope(-1);
                return retType;
            }
            throw new LLException("cannot assign to the size of that");

        }

        public BaseType size(BaseType a)
        {
            a.setName(opGetName(a));

            BaseType retType = a.size();

            return doRegStuff(retType);
        }

        public BaseType uminus(BaseType a)
        {
            a.setName(opGetName(a));

            BaseType retType = a.uminus();

            return doRegStuff(retType);
        }

        public BaseType getListText(BaseType a)
        {
            if(a instanceof LLList)
            {
                String s = "new Listof(\"" + a.getTypeName() + "\", (" + a.getName()
+ ") )";
                BaseType ret = new LLList(getCopy(a));
                ret.setName(s);
                return ret;
            }
            else
            {
                String s = "new Listof(\"" + a.getTypeName() + "\")";
                BaseType ret = new LLList(getCopy(a));
                ret.setName(s);
                return ret;
            }

        }

        public static BaseType getCopy(BaseType a)
        {
            if(a instanceof LLNumber)
            {
                return new LLNumber();
            }
            else if(a instanceof LLString)
            {
                return new LLString();
            }
            else if(a instanceof LLFraction)
            {
                return new LLFraction();
            }
            if(a instanceof LLList)
            {
                return new LLList( getCopy(((LLList)a).type) );
            }
            return new BaseType();
        }

        // declares variable named b as type a
        public BaseType declareStmt(BaseType a, String b)
        {
            String s = "";
            String name = b + Integer.toString(scopeIndex);
            if(a instanceof LLNumber)
            {
                s = "int " + name + " = 0;";
            }
            else if(a instanceof LLString)
            {
                s = "String " + name + " = \"\";";
            }
            else if(a instanceof LLFraction)
            {
                s = "Fraction " + name + ";";
            }
            else if(a instanceof LLList)
            {

                s = "Listof " + name + " = new Listof(";
                if(((LLList)a).type instanceof LLNumber)
                {
                    s += "\"number\")";
                }
                else if(((LLList)a).type instanceof LLFraction)
                {
                    s += "\"fraction\")";
                }
                else if(((LLList)a).type instanceof LLString)
                {
                      s += "\"string\")";
                }
                else if(((LLList)a).type instanceof LLList)
                {
                      s += "\"listof\", " + a.getName() + ")";
                }
                s += ";";

          }
          a.setName(b);
          declareVar(b, a);

          LLOutput(s);
          BaseType ret = getCopy(a);
          ret.setName(b);
          return ret;
}

public BaseType openOutFile(BaseType a)
{
      if(a instanceof LLString)
      {
            BaseType b;
            if((b = symt.getVar(a.name)) == null)
            {
                  LLOutput("llio.openOutFile(" + a.name + ");");
            }
            else
            {
                  LLOutput("llio.openOutFile(" + a.name + b.scopeCnt + ");");
            }
      }
      else
      {
            throw new LLException("file name must be a string");
      }
      return new BaseType();
}

public BaseType openInFile(BaseType a)
{
      if(a instanceof LLString)
      {
            BaseType b;
            if((b = symt.getVar(a.name)) == null)
            {
                  LLOutput("llio.openInFile(" + a.name + ");");
            }
            else
            {
                  LLOutput("llio.openInFile(" + a.name + b.scopeCnt + ");");
            }
      }
      else
      {
            throw new LLException("file name must be a string");
```

```
    }
    return new BaseType();
}

public BaseType readStdin(BaseType a)
{
    if(a instanceof LLString)
    {
          BaseType b;
          if((b = symt.getVar(a.name)) == null)
          {
                throw new LLException(a.getName() + " not declared");
          }
          else
          {
                LLOutput(a.name + b.scopeCnt + "= llio.read();");
          }
    }
    else
    {
          throw new LLException("variable must be a string");
    }
    return new BaseType();
}

public BaseType readFromFile(BaseType a)
{
    if(a instanceof LLString)
    {
          BaseType b;
          if((b = symt.getVar(a.name)) == null)
          {
                throw new LLException(a.getName() + " not declared");
          }
          else
          {
                LLOutput(a.name + b.scopeCnt + "= llio.readWordFromFile();");
          }
    }
    else
    {
          throw new LLException("variable must be a string");
    }
    return new BaseType();
}

public BaseType readLineFromFile(BaseType a)
{
    if(a instanceof LLString)
    {
          BaseType b;
          if((b = symt.getVar(a.name)) == null)
          {
                throw new LLException(a.getName() + " not declared");
          }
          else
          {
                LLOutput(a.name + b.scopeCnt + "= llio.readLineFromFile();");
          }
    }
    else
    {
          throw new LLException("variable must be a string");
    }
    return new BaseType();
}

public BaseType stdout(BaseType a)
{
    BaseType b;
    if((b = symt.getVar(a.name)) == null)
    {
          LLOutput("llio.output(" + a.name + ");");
    }
    else
    {
          LLOutput("llio.output(" + a.name + b.scopeCnt + ");");
    }
    return new BaseType();
}

public BaseType stdoutLine(BaseType a)
{
    BaseType b;
    if((b = symt.getVar(a.name)) == null)
    {
          LLOutput("llio.outputLine(" + a.name + ");");
    }
    else
    {
          LLOutput("llio.outputLine(" + a.name + b.scopeCnt + ");");
    }
    return new BaseType();
}

public BaseType writeToFile(BaseType a)
{
    BaseType b;
    if((b = symt.getVar(a.name)) == null)
    {
          LLOutput("llio.outputToFile(" + a.name + ");");
    }
    else
    {
          LLOutput("llio.outputToFile(" + a.name + b.scopeCnt + ");");
    }
    return new BaseType();
}

public BaseType writeLineToFile(BaseType a)
{
    BaseType b;
    if((b = symt.getVar(a.name)) == null)
    {
                LLOutput("llio.outputLineToFile(" + a.name + ");");
    }
    else
    {
                LLOutput("llio.outputLineToFile(" + a.name + b.scopeCnt + ");");
    }
    return new BaseType();
}
}
```

9. LLCompiler.java

```
import java.io.*;
import antlr.*;

//The actual compiler
//Stephen Robinson
public class LLCompiler
{
  public static void main(String[] args)
  {
        if(args.length < 1)
                System.out.println("invalid args");

        try
        {
                LLPreprocessor pre = new LLPreprocessor(args[0], "out.ll");
                pre.process();
                InputStream input = (InputStream) new FileInputStream("out.ll");

            LLAntlrLexer lexer = new LLAntlrLexer(input);
            LLAntlrParser parser = new LLAntlrParser(lexer);

            parser.program();

            CommonAST tree = (CommonAST)parser.getAST();

            //System.out.println(tree.toStringTree());
            LLAntlrWalker walker = new LLAntlrWalker();
            walker.expr(tree);

        }
        catch(Exception e)
        {
                System.err.println(e.toString());
        }
  }
}
```

10. LLException.java

```
//Output an error.  Could potentially expand class to output line
//number of error (not very likely)
//Stephen Robinson
public class LLException extends RuntimeException
```

```
{

    LLException(String msg)
    {
        System.err.println("Error: " + msg);
    }

}
```

11. Listof.java

```
/*
 * List.java
 *
 * Created on April 18, 2007, 6:58 PM
 */

/**
 *
 * @author Joseanibal Colon R
 */
import java.util.ArrayList;

public class Listof {

    public String typename;
    public ArrayList content;
    //The typenames are: number, string, listof, fraction

    //        CONSTRUCTORS
   //---------------------------------------------------------------
    public Listof(String the_type, ArrayList stuff) {
        typename = the_type;
        content = (ArrayList) stuff.clone();
    }

    public Listof(String the_type){
        typename = the_type;
        content = new ArrayList();
    }

    public Listof(String the_type, Object stuff1, Object stuff2){
        Object copy1 = stuff1;
        Object copy2 = stuff2;
        typename = the_type;
        content = new ArrayList();
        content.add(copy1);
        content.add(copy2);
    }

    public Listof(String the_type, Listof stuff){
        typename = the_type;
        content = new ArrayList();
        content.add(stuff);
    }
    //-------------------------------------------------------
```

```
//      INFO FUNCTIONS
//--------------------------------------------------------
//size of first dimension
public int size(){
    return content.size();
}

//sizes of all dimensions:
public Listof realSize(){
  Listof copy = new Listof(typename, content);
  ArrayList result = new ArrayList();
  while(copy.typename.equals("listof")){
      result.add(new Integer(copy.content.size()));
      copy = (Listof) copy.content.get(0);
  }
  result.add(new Integer(copy.content.size()));
  return new Listof("number", result);
}

public int equals(Listof stuff){
    int result = 0;
    Listof comp = new Listof(typename, content);
    Listof size1 = comp.realSize();
    Listof stuff2 = stuff;
    Listof size2 = stuff2.realSize();
    if(size1.content.size() != size2.content.size()){
        return 0;
    }
    for(int k = 0; k < size1.content.size(); k++){
        int a = ((Integer) size1.content.get(k)).intValue();
        int b = ((Integer) size2.content.get(k)).intValue();
        if(a != b){
            return 0;
        }
    }
    if(typename.equals("fraction")){
        stuff2 = stuff2.toFracList();
        for(int i = 0; i < content.size(); i++){
            Fraction temp1 = (Fraction) content.get(i);
            Fraction temp2 = (Fraction) stuff2.content.get(i);
            if((temp1.eq(temp2)) == 1){
                result = 1;
            }
            else{
               result = 0;
               break;
            }
        }
    }
    if(typename.equals("number")){
        stuff2 = stuff2.toNumList();
        for(int i = 0; i < content.size(); i++){
            int temp1 = ((Integer) content.get(i)).intValue();
            int temp2 = ((Integer) stuff2.content.get(i)).intValue();
            if(temp1 == temp2){
                result = 1;
            }
            else{
               result = 0;
               break;
            }
        }
    }

    if(typename.equals("string")){
        if(stuff.typename.equals("string")){
            for(int i = 0; i < content.size(); i++){
                String temp1 = (String) content.get(i);
                String temp2 = (String) stuff2.content.get(i);
                if(temp1.equals(temp2)){
                    result = 1;
                }
                else{
                    result = 0;
                    break;
                }
            }
        }
    }

    if(typename.equals("listof")){
        for(int i = 0; i < content.size(); i++){
            Listof temp1 = (Listof) content.get(i);
            Listof temp2 = (Listof) stuff.content.get(i);
            result = temp1.equals(temp2);
            if(result == 0){
                break;
            }
        }
    }
    return result;
}


public Object get(int[] idx){
   int index = idx[0];
   if(typename.equals("listof")){
        Listof tempList;
        tempList = (Listof) content.get(index);
        //recursive call on successive lists:
        int[] new_idx = new int[idx.length - 1];
        for(int j = 1; j < idx.length ; j++){
            new_idx[j-1] = idx[j];
        }
        return tempList.get(new_idx);
   }
   return content.get(index);
}


//             SIMPLE MANIPULATIONS
//---------------------------------------------------------
public Listof appendPrimitive(Object stuff){
    ArrayList contntReturn = (ArrayList) content.clone();
```

```
        contntReturn.add(stuff);

        return new Listof(typename, contntReturn);
     }

    public Listof prependPrimitive(Object stuff){
        ArrayList contntReturn = (ArrayList) content.clone();
        contntReturn.add(0, stuff);

        return new Listof(typename, contntReturn);
    }

    public Listof appendList(Object stuff){
        ArrayList contntReturn = (ArrayList) content.clone();
        if(((Listof)stuff).typename == "listof")
        {
          for(int i = 0; i < ((Listof)stuff).size(); i++)
          {

  contntReturn.add(((Listof)this.content.get(i)).appendList(((Listof)stuff).conten
t.get(i)));
          }
        }
        else
        {
          for(int i = 0; i < ((Listof)stuff).size(); i++)
          {
              contntReturn.add( ((Listof)stuff).content.get(i) );
          }
        }

        return new Listof(typename, contntReturn);
     }

    public Listof prependList(Object stuff){
        ArrayList contntReturn = (ArrayList) content.clone();

        if(((Listof)stuff).typename == "listof")
        {
         for(int i = ((Listof)stuff).size() - 1; i >= 0; i--)
          {

  contntReturn.add(0,((Listof)this.content.get(i)).appendList(((Listof)stuff).cont
ent.get(i)));
          }
        }
        else
        {
         for(int i = ((Listof)stuff).size() - 1; i >= 0; i--)
          {
              contntReturn.add(0, ((Listof)stuff).content.get(i) );
          }
        }

        return new Listof(typename, contntReturn);
    }
```

```
public Listof remove(Object stuff){
    ArrayList contntReturn = (ArrayList) content.clone();
    contntReturn.remove(contntReturn.indexOf(stuff));

    return new Listof(typename, contntReturn);
}




//       MATH
//-----------------------------------------------------------

//  operator -> number
//------------------------------------------------------------
public Listof add(int num){
    ArrayList contntReturn = (ArrayList) content.clone();

    if(typename.equals("fraction")){
        for(int i = 0;  i < contntReturn.size(); i++){
            Fraction temp1, temp2;
            temp2 = new Fraction(num, 0);
            temp1 = (Fraction) contntReturn.get(i);
            temp1 = temp1.add(temp2);
            contntReturn.set(i, temp1);
        }
    }

    if(typename.equals("number")){
        for(int i = 0; i < contntReturn.size(); i++){
            int temp = ((Integer) contntReturn.get(i)).intValue();
            temp = temp + num;
            contntReturn.set(i, new Integer(temp));
        }
    }

    if(typename.equals("listof")){
        for(int i = 0; i < contntReturn.size(); i++){
            Listof tempList;
            tempList = (Listof) contntReturn.get(i);
            //recursive call on successive lists:
            tempList = tempList.add(num);
            contntReturn.set(i, tempList);
        }
    }
    return new Listof(typename, contntReturn);
}

public Listof minus(int num){
    ArrayList contntReturn = (ArrayList) content.clone();

    if(typename.equals("fraction")){
        for(int i = 0;  i < contntReturn.size(); i++){
            Fraction temp1, temp2;
            temp2 = new Fraction(num, 0);
            temp1 = (Fraction) contntReturn.get(i);
            temp1 = temp1.subtract(temp2);
```

```
                contntReturn.set(i, temp1);
            }
        }

        if(typename.equals("number")){
            for(int i = 0; i < contntReturn.size(); i++){
                int temp = ((Integer) contntReturn.get(i)).intValue();
                temp = temp - num;
                contntReturn.set(i, new Integer(temp));
            }
        }

        if(typename.equals("listof")){
            for(int i = 0; i < contntReturn.size(); i++){
                Listof tempList;
                tempList = (Listof) contntReturn.get(i);
                //recursive call on successive lists:
                tempList = tempList.minus(num);
                contntReturn.set(i, tempList);
            }
        }
        return new Listof(typename, contntReturn);
    }

    //SR
    public Listof fract(int num)
    {
        ArrayList contntReturn = (ArrayList)content.clone();

        if(typename == "listof")
        {
         for(int i = 0; i < content.size(); i++)
         {
              Listof tmp = ((Listof)content.get(i)).fract(num);
              contntReturn.set(i, tmp);
         }
         return new Listof("listof", contntReturn);

        }
        else //if(typename == "number")
        {
          for(int i = 0; i < content.size(); i++)
          {
              Fraction temp = new Fraction(((Integer)content.get(i)).intValue(),
num);
              contntReturn.set(i, temp);
          }
          return new Listof("fraction", contntReturn);
        }
    }

    public Listof fract(Listof numbers)
    {
        ArrayList contntReturn = (ArrayList)content.clone();

        if(typename == "listof")
        {
```

```
        for(int i = 0; i < content.size(); i++)
        {
              Listof tmp;
              if(numbers.typename == "number")
              {
                    tmp =
((Listof)content.get(i)).fract(((Integer)numbers.content.get(i)).intValue());
              }
              else
              {
                    tmp =
((Listof)content.get(i)).fract((Listof)numbers.content.get(i));
              }
              contntReturn.set(i, tmp);
        }
        return new Listof("listof", contntReturn);

        }
        else //if(typename == "number")
        {
          for(int i = 0; i < content.size(); i++)
          {
              Fraction temp = new Fraction(((Integer)content.get(i)).intValue(),
((Integer)numbers.content.get(i)).intValue());
              contntReturn.set(i, temp);
          }
          return new Listof("fraction", contntReturn);
        }
    }

    public Listof prefract(int num)
    {
        ArrayList contntReturn = (ArrayList)content.clone();

        if(typename == "listof")
        {
        for(int i = 0; i < content.size(); i++)
        {
              Listof tmp = ((Listof)content.get(i)).prefract(num);
              contntReturn.set(i, tmp);
        }
        return new Listof("listof", contntReturn);

        }
        else //if(typename == "number")
        {
          for(int i = 0; i < content.size(); i++)
          {
              Fraction temp = new Fraction(num,
((Integer)content.get(i)).intValue());
              contntReturn.set(i, temp);
          }
          return new Listof("fraction", contntReturn);
        }
    }
```

```
public Listof multiply(int num){
    ArrayList contntReturn = (ArrayList) content.clone();

    if(typename.equals("fraction")){
        for(int i = 0;  i < contntReturn.size(); i++){
            Fraction temp1, temp2;
            temp2 = new Fraction(num, 0);
            temp1 = (Fraction) contntReturn.get(i);
            temp1 = temp1.multiply(temp2);
            contntReturn.set(i, temp1);
        }
    }

    if(typename.equals("number")){
        for(int i = 0; i < contntReturn.size(); i++){
            int temp = ((Integer) contntReturn.get(i)).intValue();
            temp = temp * num;
            contntReturn.set(i, new Integer(temp));
        }
    }

    if(typename.equals("listof")){
        for(int i = 0; i < contntReturn.size(); i++){
            Listof tempList;
            tempList = (Listof) contntReturn.get(i);
            //recursive call on successive lists:
            tempList = tempList.multiply(num);
            contntReturn.set(i, tempList);
        }
    }
    return new Listof(typename, contntReturn);
}

public Listof divide(int num){
    ArrayList contntReturn = (ArrayList) content.clone();

    if(typename.equals("fraction")){
        for(int i = 0;  i < contntReturn.size(); i++){
            Fraction temp1, temp2;
            temp2 = new Fraction(num, 0);
            temp1 = (Fraction) contntReturn.get(i);
            temp1 = temp1.divide(temp2);
            contntReturn.set(i, temp1);
        }
    }

    if(typename.equals("number")){
        for(int i = 0; i < contntReturn.size(); i++){
            int temp = ((Integer) contntReturn.get(i)).intValue();
            temp = temp / num;
            contntReturn.set(i, new Integer(temp));
        }
    }

    if(typename.equals("listof")){
        for(int i = 0; i < contntReturn.size(); i++){
            Listof tempList;
```

```
                tempList = (Listof) contntReturn.get(i);
                //recursive call on successive lists:
                tempList = tempList.divide(num);
                contntReturn.set(i, tempList);
            }
        }
        return new Listof(typename, contntReturn);
    }


    //    operator -> fraction
    //-----------------------------------------------------------------

    public Listof add(Fraction num){
        ArrayList contntReturn = (ArrayList) content.clone();

        String typeReturn = typename;
        if(typename.equals("fraction")){
            for(int i = 0;  i < contntReturn.size(); i++){
                Fraction temp1;
                temp1 = (Fraction) contntReturn.get(i);
                temp1 = temp1.add(num);
                contntReturn.set(i, temp1);
            }
        }

        if(typename.equals("number")){
            for(int i = 0; i < contntReturn.size(); i++){
                int temp = ((Integer) contntReturn.get(i)).intValue();
                Fraction temp2 = new Fraction(temp, 0);
                temp2 = temp2.add(num);
                contntReturn.set(i, temp2);
            }
            typeReturn = "fraction";
        }

        if(typename.equals("listof")){
            for(int i = 0; i < contntReturn.size(); i++){
                Listof tempList;
                tempList = (Listof) contntReturn.get(i);
                //recursive call on successive lists:
                tempList = tempList.add(num);
                contntReturn.set(i, tempList);
            }
        }
        return new Listof(typeReturn, contntReturn);
    }

    public Listof minus(Fraction num){
        ArrayList contntReturn = (ArrayList) content.clone();

        String typeReturn = typename;
        if(typename.equals("fraction")){
            for(int i = 0;  i < contntReturn.size(); i++){
                Fraction temp1;
                temp1 = (Fraction) contntReturn.get(i);
                temp1 = temp1.subtract(num);
                contntReturn.set(i, temp1);
            }
        }

        if(typename.equals("number")){
            for(int i = 0; i < contntReturn.size(); i++){
                int temp = ((Integer) contntReturn.get(i)).intValue();
                Fraction temp2 = new Fraction(temp, 0);
                temp2 = temp2.subtract(num);
                contntReturn.set(i, temp2);
            }
            typeReturn = "fraction";
        }

        if(typename.equals("listof")){
            for(int i = 0; i < contntReturn.size(); i++){
                Listof tempList;
                tempList = (Listof) contntReturn.get(i);
                //recursive call on successive lists:
                tempList = tempList.minus(num);
                contntReturn.set(i, tempList);
            }
        }
        return new Listof(typeReturn, contntReturn);
    }

    public Listof multiply(Fraction num){
        ArrayList contntReturn = (ArrayList) content.clone();

        String typeReturn = typename;
        if(typename.equals("fraction")){
            for(int i = 0;  i < contntReturn.size(); i++){
                Fraction temp1;
                temp1 = (Fraction) contntReturn.get(i);
                temp1 = temp1.multiply(num);
                contntReturn.set(i, temp1);
            }
        }

        if(typename.equals("number")){
            for(int i = 0; i < contntReturn.size(); i++){
                int temp = ((Integer) contntReturn.get(i)).intValue();
                Fraction temp2 = new Fraction(temp, 0);
                temp2 = temp2.multiply(num);
                contntReturn.set(i, temp2);
            }
            typeReturn = "fraction";
        }

        if(typename.equals("listof")){
            for(int i = 0; i < contntReturn.size(); i++){
                Listof tempList;
                tempList = (Listof) contntReturn.get(i);
                //recursive call on successive lists:
                tempList = tempList.multiply(num);
                contntReturn.set(i, tempList);
            }
        }
        return new Listof(typeReturn, contntReturn);
    }

    public Listof divide(Fraction num){
        ArrayList contntReturn = (ArrayList) content.clone();

        String typeReturn = typename;
        if(typename.equals("fraction")){
            for(int i = 0;  i < contntReturn.size(); i++){
                Fraction temp1;
                temp1 = (Fraction) contntReturn.get(i);
                temp1 = temp1.divide(num);
                contntReturn.set(i, temp1);
            }
        }

        if(typename.equals("number")){
            for(int i = 0; i < contntReturn.size(); i++){
                int temp = ((Integer) contntReturn.get(i)).intValue();
                Fraction temp2 = new Fraction(temp, 0);
                temp2 = temp2.divide(num);
                contntReturn.set(i, temp2);
            }
            typeReturn = "fraction";
        }

        if(typename.equals("listof")){
            for(int i = 0; i < contntReturn.size(); i++){
                Listof tempList;
                tempList = (Listof) contntReturn.get(i);
                //recursive call on successive lists:
                tempList = tempList.divide(num);
                contntReturn.set(i, tempList);
            }
        }
        return new Listof(typeReturn, contntReturn);
    }



   // only with 'number' operations
   //----------------------------------------------------------------

    public Listof exp(int num){
        ArrayList contntReturn = (ArrayList) content.clone();

        if(typename.equals("fraction")){
            for(int i = 0;  i < contntReturn.size(); i++){
                Fraction temp1 = (Fraction) contntReturn.get(i);
                temp1 = temp1.exp(num);
                contntReturn.set(i, temp1);
            }
        }

        if(typename.equals("number")){
            for(int i = 0; i < contntReturn.size(); i++){
                int temp = ((Integer) contntReturn.get(i)).intValue();
                temp = (int)Math.pow(temp, num);
                contntReturn.set(i, new Integer(temp));
            }
        }

        if(typename.equals("listof")){
            for(int i = 0; i < contntReturn.size(); i++){
                Listof tempList;
                tempList = (Listof) contntReturn.get(i);
                //recursive call on successive lists:
                tempList = tempList.exp(num);
                contntReturn.set(i, tempList);
            }
        }
        return new Listof(typename, contntReturn);
    }

    public Listof modulus(int num){
        ArrayList contntReturn = (ArrayList) content.clone();

        if(typename.equals("fraction")){
            for(int i = 0;  i < contntReturn.size(); i++){
                Fraction temp1 = (Fraction) contntReturn.get(i);
                temp1 = temp1.modulus(num);
                contntReturn.set(i, temp1);
            }
        }

        if(typename.equals("number")){
            for(int i = 0; i < contntReturn.size(); i++){
                int temp = ((Integer) contntReturn.get(i)).intValue();
                temp = temp%num;
                contntReturn.set(i, new Integer(temp));
            }
        }

        if(typename.equals("listof")){
            for(int i = 0; i < contntReturn.size(); i++){
                Listof tempList;
                tempList = (Listof) contntReturn.get(i);
                //recursive call on successive lists:
                tempList = tempList.modulus(num);
                contntReturn.set(i, tempList);
            }
        }
        return new Listof(typename, contntReturn);
    }


    //------------------PRE's--------------------------------
    //--------------Order sensitive operations only
    //----------number or fraction, operator -> list
    //  on numbers:
    //---------------------------------------------------------

    public Listof preminus(int num){
        ArrayList contntReturn = (ArrayList) content.clone();

        if(typename.equals("fraction")){
            for(int i = 0;  i < contntReturn.size(); i++){
                Fraction temp1, temp2;
                temp2 = new Fraction(num, 0);
                temp1 = (Fraction) contntReturn.get(i);
                temp1 = temp2.subtract(temp1);
                contntReturn.set(i, temp1);
            }
        }

        if(typename.equals("number")){
            for(int i = 0; i < contntReturn.size(); i++){
                int temp = ((Integer) contntReturn.get(i)).intValue();
                temp = num - temp;
                contntReturn.set(i, new Integer(temp));
            }
        }

        if(typename.equals("listof")){
            for(int i = 0; i < contntReturn.size(); i++){
                Listof tempList;
                tempList = (Listof) contntReturn.get(i);
                //recursive call on successive lists:
                tempList = tempList.preminus(num);
                contntReturn.set(i, tempList);
            }
        }
        return new Listof(typename, contntReturn);
    }

    public Listof predivide(int num){
        ArrayList contntReturn = (ArrayList) content.clone();

        if(typename.equals("fraction")){
            for(int i = 0;  i < contntReturn.size(); i++){
                Fraction temp1, temp2;
                temp2 = new Fraction(num, 0);
                temp1 = (Fraction) contntReturn.get(i);
                temp1 = temp2.divide(temp1);
                contntReturn.set(i, temp1);
            }
        }

        if(typename.equals("number")){
            for(int i = 0; i < contntReturn.size(); i++){
                int temp = ((Integer) contntReturn.get(i)).intValue();
                temp = num / temp;
                contntReturn.set(i, new Integer(temp));
            }
        }

        if(typename.equals("listof")){
            for(int i = 0; i < contntReturn.size(); i++){
                Listof tempList;
                tempList = (Listof) contntReturn.get(i);
                //recursive call on successive lists:
                tempList = tempList.predivide(num);
                contntReturn.set(i, tempList);
            }
        }
        return new Listof(typename, contntReturn);
    }


    //    operator -> fraction
    //----------------------------------------------------------------

    public Listof preminus(Fraction num){
        ArrayList contntReturn = (ArrayList) content.clone();

        String typeReturn = typename;
        if(typename.equals("fraction")){
            for(int i = 0;  i < contntReturn.size(); i++){
                Fraction temp1;
                temp1 = (Fraction) contntReturn.get(i);
                temp1 = num.subtract(temp1);
                contntReturn.set(i, temp1);
            }
        }

        if(typename.equals("number")){
            for(int i = 0; i < contntReturn.size(); i++){
                int temp = ((Integer) contntReturn.get(i)).intValue();
                Fraction temp2 = new Fraction(temp, 0);
                temp2 = num.subtract(temp2);
                contntReturn.set(i, temp2);
            }
            typeReturn = "fraction";
        }

        if(typename.equals("listof")){
            for(int i = 0; i < contntReturn.size(); i++){
                Listof tempList;
                tempList = (Listof) contntReturn.get(i);
                //recursive call on successive lists:
                tempList = tempList.preminus(num);
                contntReturn.set(i, tempList);
            }
        }
        return new Listof(typeReturn, contntReturn);
    }

    public Listof predivide(Fraction num){
        ArrayList contntReturn = (ArrayList) content.clone();

        String typeReturn = typename;
        if(typename.equals("fraction")){
            for(int i = 0;  i < contntReturn.size(); i++){
                Fraction temp1;
                temp1 = (Fraction) contntReturn.get(i);
                temp1 = num.divide(temp1);
                contntReturn.set(i, temp1);
            }
        }

        if(typename.equals("number")){
            for(int i = 0; i < contntReturn.size(); i++){
                int temp = ((Integer) contntReturn.get(i)).intValue();
                Fraction temp2 = new Fraction(temp, 0);
                temp2 = num.divide(temp2);
                contntReturn.set(i, temp2);
            }
            typeReturn = "fraction";
        }

        if(typename.equals("listof")){
            for(int i = 0; i < contntReturn.size(); i++){
                Listof tempList;
                tempList = (Listof) contntReturn.get(i);
                //recursive call on successive lists:
                tempList = tempList.predivide(num);
                contntReturn.set(i, tempList);
            }
        }
        return new Listof(typeReturn, contntReturn);
    }



   // only with 'number' operations
   //---------------------------------------------------------------

    public Listof preexp(int num){
        ArrayList contntReturn = (ArrayList) content.clone();

        if(typename.equals("number")){
            for(int i = 0; i < contntReturn.size(); i++){
                int temp = ((Integer) contntReturn.get(i)).intValue();
                temp = (int)Math.pow(num,temp);
                contntReturn.set(i, new Integer(temp));
            }
        }

        if(typename.equals("listof")){
            for(int i = 0; i < contntReturn.size(); i++){
                Listof tempList;
                tempList = (Listof) contntReturn.get(i);
                //recursive call on successive lists:
                tempList = tempList.preexp(num);
                contntReturn.set(i, tempList);
            }
        }
        return new Listof(typename, contntReturn);
    }

    //SR
    public Listof preexp(Fraction f){
        ArrayList contntReturn = (ArrayList) content.clone();
        if(typename.equals("number")){
            for(int i = 0; i < contntReturn.size(); i++){
              int t = ((Integer)contntReturn.get(i)).intValue();
              int n = f.getNumerator();
              int d = f.getDenominator();
                Fraction temp = new Fraction((int)Math.pow(n,t),
(int)Math.pow(d,t));
                contntReturn.set(i, temp);
            }
            return new Listof("fraction", contntReturn);
        }

        if(typename.equals("listof")){
            for(int i = 0; i < contntReturn.size(); i++){
                Listof tempList;
                tempList = (Listof) contntReturn.get(i);
                //recursive call on successive lists:
                tempList = tempList.preexp(f);
                contntReturn.set(i, tempList);
            }
            return new Listof("listof", contntReturn);
        }
        //should never be called
        return new Listof("fraction");

    }

    public Listof premodulus(int num){
        ArrayList contntReturn = (ArrayList) content.clone();

        if(typename.equals("number")){
            for(int i = 0; i < contntReturn.size(); i++){
                int temp = ((Integer) contntReturn.get(i)).intValue();
                temp = num%temp;
                contntReturn.set(i, new Integer(temp));
            }
        }

        if(typename.equals("listof")){
            for(int i = 0; i < contntReturn.size(); i++){
                Listof tempList;
                tempList = (Listof) contntReturn.get(i);
                //recursive call on successive lists:
                tempList = tempList.premodulus(num);
                contntReturn.set(i, tempList);
            }
        }
        return new Listof(typename, contntReturn);
    }

    //--------on 2 Matrix Operands: ALL PAIRWISE, no Linear algebra
    //------------------------------------------------------------------------
    public Listof add(Listof stuff){
        ArrayList contntReturn = (ArrayList) content.clone();
        String typeReturn = typename;

        if(typename.equals("fraction")){
            for(int i = 0; i < contntReturn.size(); i++){
                Fraction temp1 = (Fraction) contntReturn.get(i);
                if(stuff.typename.equals("fraction")){
                    Fraction temp2 = (Fraction) stuff.content.get(i);
                    temp1 = temp1.add(temp2);
                    contntReturn.set(i, temp1);
                }
                else{
                    if(stuff.typename.equals("number")){
                        int temp2 = ((Integer) stuff.content.get(i)).intValue();
                        Fraction temp3 = new Fraction(temp2, 0);
                        temp1 = temp1.add(temp3);
                        contntReturn.set(i, temp1);
                    }
                }
            }
        }
        if(typename.equals("number")){
            for(int i = 0; i < contntReturn.size(); i++)
            {
                int temp1 = ((Integer)contntReturn.get(i)).intValue();
                if(stuff.typename.equals("fraction")){
                    Fraction temp2 = (Fraction) stuff.content.get(i);
                    Fraction temp3 = new Fraction(temp1, 0);
                    temp3 = temp3.add(temp2);
                    contntReturn.set(i, temp3);
                    typeReturn = "fraction";
                }
                else{
                    if(stuff.typename.equals("number"))
                    {
                        int temp2 = ((Integer) stuff.content.get(i)).intValue();
                        temp1 = temp1+temp2;
                        contntReturn.set(i, new Integer(temp1));
                    }
                }
            }
        }

        if(typename.equals("listof")){
            for(int i = 0; i < contntReturn.size(); i++){
                Listof temp1 = (Listof) contntReturn.get(i);
                Listof temp2 = (Listof) stuff.content.get(i);
                temp1 = temp1.add(temp2);
                contntReturn.set(i, temp1);
            }
        }
        return new Listof(typename, contntReturn);
    }

    public Listof minus(Listof stuff){
        ArrayList contntReturn = (ArrayList) content.clone();
        String typeReturn = typename;

        if(typename.equals("fraction")){
            for(int i = 0; i < contntReturn.size(); i++){
                Fraction temp1 = (Fraction) contntReturn.get(i);
                if(stuff.typename.equals("fraction")){
                    Fraction temp2 = (Fraction) stuff.content.get(i);
                    temp1 = temp1.subtract(temp2);
                    contntReturn.set(i, temp1);
                }
                else{
                    if(stuff.typename.equals("number")){
                        int temp2 = ((Integer) stuff.content.get(i)).intValue();
                        Fraction temp3 = new Fraction(temp2, 0);
                        temp1 = temp1.subtract(temp3);
                        contntReturn.set(i, temp1);
                    }
                }
            }
        }
        if(typename.equals("number")){
            for(int i = 0; i < contntReturn.size(); i++){
                int temp1 = ((Integer) contntReturn.get(i)).intValue();
                if(stuff.typename.equals("fraction")){
                    Fraction temp2 = (Fraction) stuff.content.get(i);
                    Fraction temp3 = new Fraction(temp1, 0);
                    temp3 = temp3.subtract(temp2);
                    contntReturn.set(i, temp3);
                    typeReturn = "fraction";
                }
                else{
                    if(stuff.typename.equals("number")){
                        int temp2 = ((Integer) stuff.content.get(i)).intValue();
                        temp1 = temp1-temp2;
                        contntReturn.set(i, new Integer(temp1));
                    }
                }
            }
        }

        if(typename.equals("listof")){
            for(int i = 0; i < contntReturn.size(); i++){
                Listof temp1 = (Listof) contntReturn.get(i);
                Listof temp2 = (Listof) stuff.content.get(i);
                temp1 = temp1.minus(temp2);
                contntReturn.set(i, temp1);
            }
        }
        return new Listof(typename, contntReturn);
    }

    public Listof multiply(Listof stuff){
        ArrayList contntReturn = (ArrayList) content.clone();
        String typeReturn = typename;

        if(typename.equals("fraction")){
            for(int i = 0; i < contntReturn.size(); i++){
                Fraction temp1 = (Fraction) contntReturn.get(i);
                if(stuff.typename.equals("fraction")){
                    Fraction temp2 = (Fraction) stuff.content.get(i);
                    temp1 = temp1.multiply(temp2);
                    contntReturn.set(i, temp1);
                }
                else{
                    if(stuff.typename.equals("number")){
                        int temp2 = ((Integer) stuff.content.get(i)).intValue();
                        Fraction temp3 = new Fraction(temp2, 0);
                        temp1 = temp1.multiply(temp3);
                        contntReturn.set(i, temp1);
                    }
                }
            }
        }
        if(typename.equals("number")){
            for(int i = 0; i < contntReturn.size(); i++){
                int temp1 = ((Integer) contntReturn.get(i)).intValue();
                if(stuff.typename.equals("fraction")){
                    Fraction temp2 = (Fraction) stuff.content.get(i);
                    Fraction temp3 = new Fraction(temp1, 0);
                    temp3 = temp3.multiply(temp2);
                    contntReturn.set(i, temp3);
                    typeReturn = "fraction";
                }
                else{
                    if(stuff.typename.equals("number")){
                        int temp2 = ((Integer) stuff.content.get(i)).intValue();
                        temp1 = temp1*temp2;
                        contntReturn.set(i, new Integer(temp1));
                    }
                }
            }
        }

        if(typename.equals("listof")){
            for(int i = 0; i < contntReturn.size(); i++){
                Listof temp1 = (Listof) contntReturn.get(i);
                Listof temp2 = (Listof) stuff.content.get(i);
                temp1 = temp1.multiply(temp2);
                contntReturn.set(i, temp1);
            }
        }
        return new Listof(typename, contntReturn);
    }

     public Listof divide(Listof stuff){
        ArrayList contntReturn = (ArrayList) content.clone();
        String typeReturn = typename;

        if(typename.equals("fraction")){
            for(int i = 0; i < contntReturn.size(); i++){
                Fraction temp1 = (Fraction) contntReturn.get(i);
                if(stuff.typename.equals("fraction")){
                    Fraction temp2 = (Fraction) stuff.content.get(i);
                    temp1 = temp1.divide(temp2);
                    contntReturn.set(i, temp1);
                }
                else{
                    if(stuff.typename.equals("number")){
                        int temp2 = ((Integer) stuff.content.get(i)).intValue();
                        Fraction temp3 = new Fraction(temp2, 0);
                        temp1 = temp1.divide(temp3);
                        contntReturn.set(i, temp1);
                    }
                }
            }
        }
        if(typename.equals("number")){
            for(int i = 0; i < contntReturn.size(); i++){
                int temp1 = ((Integer) contntReturn.get(i)).intValue();
                if(stuff.typename.equals("fraction")){
                    Fraction temp2 = (Fraction) stuff.content.get(i);
                    Fraction temp3 = new Fraction(temp1, 0);
                    temp3 = temp3.divide(temp2);
                    contntReturn.set(i, temp3);
                    typeReturn = "fraction";
                }
                else{
                    if(stuff.typename.equals("number")){
                        int temp2 = ((Integer) stuff.content.get(i)).intValue();
                        temp1 = temp1/temp2;
                        contntReturn.set(i, new Integer(temp1));
                    }
                }
            }
        }

        if(typename.equals("listof")){
            for(int i = 0; i < contntReturn.size(); i++){
                Listof temp1 = (Listof) contntReturn.get(i);
                Listof temp2 = (Listof) stuff.content.get(i);
                temp1 = temp1.divide(temp2);
                contntReturn.set(i, temp1);
            }
        }
        return new Listof(typename, contntReturn);
    }

    public Listof exp(Listof stuff){
        ArrayList contntReturn = (ArrayList) content.clone();
        String typeReturn = typename;

        if(typename.equals("fraction")){
            for(int i = 0; i < contntReturn.size(); i++){
                Fraction temp1 = (Fraction) contntReturn.get(i);
                if(stuff.typename.equals("number")){
                        int temp2 = ((Integer) stuff.content.get(i)).intValue();
                        temp1 = temp1.exp(temp2);
                        contntReturn.set(i, temp1);
                }
            }
        }
        if(typename.equals("number")){
            for(int i = 0; i < contntReturn.size(); i++){
                int temp1 = ((Integer) contntReturn.get(i)).intValue();
                if(stuff.typename.equals("number")){
                   int temp2 = ((Integer) stuff.content.get(i)).intValue();
```

```
                temp1 = (int) Math.pow(temp1,temp2);
                contntReturn.set(i, new Integer(temp1));
             }
         }
     }

     if(typename.equals("listof")){
         for(int i = 0; i < contntReturn.size(); i++){
             Listof temp1 = (Listof) contntReturn.get(i);
             Listof temp2 = (Listof) stuff.content.get(i);
             temp1 = temp1.exp(temp2);
             contntReturn.set(i, temp1);
         }
     }
     return new Listof(typename, contntReturn);
 }

 public Listof modulus(Listof stuff){
     ArrayList contntReturn = (ArrayList) content.clone();
     String typeReturn = typename;

     if(typename.equals("fraction")){
         for(int i = 0; i < contntReturn.size(); i++){
             Fraction temp1 = (Fraction) contntReturn.get(i);
             if(stuff.typename.equals("number")){
                     int temp2 = ((Integer) stuff.content.get(i)).intValue();
                     temp1 = temp1.modulus(temp2);
                     contntReturn.set(i, temp1);
             }
         }
     }
     if(typename.equals("number")){
         for(int i = 0; i < contntReturn.size(); i++){
             int temp1 = ((Integer) contntReturn.get(i)).intValue();
             if(stuff.typename.equals("number")){
                int temp2 = ((Integer) stuff.content.get(i)).intValue();
                temp1 = temp1%temp2;
                contntReturn.set(i, new Integer(temp1));
             }
         }
     }

     if(typename.equals("listof")){
         for(int i = 0; i < contntReturn.size(); i++){
             Listof temp1 = (Listof) contntReturn.get(i);
             Listof temp2 = (Listof) stuff.content.get(i);
             temp1 = temp1.modulus(temp2);
             contntReturn.set(i, temp1);
         }
     }
     return new Listof(typename, contntReturn);
 }


 //        Conversions
 //--------------------------------------------------------
```

```
public Listof toFracList(){
    Listof temp = new Listof(typename, content);
    temp = temp.multiply(new Fraction(1, 1));
    return temp;
}


public Listof toNumList(){
    ArrayList contntReturn = (ArrayList) content.clone();
    String typeReturn = typename;
    if(typeReturn.equals("listof")){
        for(int i = 0; i < contntReturn.size(); i++){
            Listof temp = (Listof) contntReturn.get(i);
            temp = temp.toNumList();
            contntReturn.set(i, temp);
        }
    }
    if(typeReturn.equals("fraction")){
        for(int j = 0; j < contntReturn.size(); j++){
            Fraction temp1 = (Fraction) contntReturn.get(j);
            int result = temp1.FracToNum();
            contntReturn.set(j, new Integer(result));
        }
        typeReturn = "number";
    }
    return new Listof(typeReturn, contntReturn);
}


// LIST of STRINGS: concatenation
//-----------------------------------------------------------------

public Listof add(String stuff){
    ArrayList contntReturn = (ArrayList) content.clone();

    if(typename.equals("string")){
        for(int i = 0;  i < contntReturn.size(); i++){
            String temp1;
            temp1 = (String) contntReturn.get(i);
            temp1 = temp1.concat(stuff);
            contntReturn.set(i, temp1);
        }
    }

    if(typename.equals("listof")){
        for(int i = 0; i < contntReturn.size(); i++){
            Listof tempList;
            tempList = (Listof) contntReturn.get(i);
            //recursive call on successive lists:
            tempList.add(stuff);
            contntReturn.set(i, tempList);
        }
    }
    return new Listof(typename, contntReturn);
}
```

```
public Listof preadd(String stuff){
    ArrayList contntReturn = (ArrayList) content.clone();

    if(typename.equals("string")){
        for(int i = 0;  i < contntReturn.size(); i++){
            String temp1;
            temp1 = (String) contntReturn.get(i);
            temp1 = stuff + temp1;
            contntReturn.set(i, temp1);
        }
    }

    if(typename.equals("listof")){
        for(int i = 0; i < contntReturn.size(); i++){
            Listof tempList;
            tempList = (Listof) contntReturn.get(i);
            //recursive call on successive lists:
            tempList.preadd(stuff);
            contntReturn.set(i, tempList);
        }
    }
    return new Listof(typename, contntReturn);
}

//    SET SET SET SET SET SET SET SET SET SET
//     Dynamic Matrix
//-------------------------------------------------------

public void set(Object stuff){
   if(typename.equals("listof")){
       for(int i = 0; i < content.size(); i++){
          Listof tempList;
          tempList = (Listof) content.get(i);
          //recursive call on successive lists:
          tempList.set(stuff);
          content.set(i, tempList);
        }
    }
   else{
       if(content.size() == 0){
           content.add(stuff);
       }
       else{
           for(int i = 0;  i < content.size(); i++){
                content.set(i, stuff);
            }
       }
   }
}

private void fixn(int[] idx){
   if(typename.equals("listof")){
     int[] new_idx = new int[idx.length - 1];
     for(int j = 1; j < idx.length ; j++){
             new_idx[j-1] = idx[j];
     }
     for(int b = 0; b < content.size(); b++){
           Listof templist = (Listof) content.get(b);
           Listof templistcopy = templist;
               if(templist.size()<=new_idx[0]){
                   templist.set(new_idx, 0);
               }
               else{
                   if(new_idx.length > 1){
                      int end = templist.size();
                      for(int h = 0; h < end; h++){
                            templistcopy = (Listof) templist.content.get(h);
                            if(templistcopy.size()<=new_idx[1]){
                               int[] new_idx2 = new int[new_idx.length - 1];
                               for(int j = 1; j < new_idx.length ; j++){
                                   new_idx2[j-1] = new_idx[j];
                               }
                               templistcopy.set(new_idx2, 0);
                               templist.content.set(h, templistcopy);
                            }
                      }
                   }
               }
               content.set(b, templist);
        }
     }
}


private void set(int[] idx, int stuff){
   int index = idx[0];
   int original = content.size();
   if(index > (content.size()-1)){
       for(int i = content.size(); i <= index; i++){
           if(idx.length >= 3){
              content.add(new Listof("listof"));
           }
           if(idx.length == 2){
               content.add(new Listof("number"));
           }
           if(idx.length == 1){
               content.add(new Integer(0));
           }
       }

       if(original>0){
               int fill = 0;
               int[] new_idx = new int[idx.length-1];
               if(typename.equals("listof")){
                   Listof crap = (Listof) content.get(0);
                   fill = crap.content.size()-1;

                   for(int j = 1; j < idx.length ; j++){
                        new_idx[j-1] = idx[j];
                    }
                   if(fill>new_idx[0]){
                        new_idx[0] = fill;
                   }
                   for(int j = 0; j <= index; j++ ){
                    Listof tempList;
                    tempList = (Listof) content.get(j);
                    if(tempList.content.size()<=new_idx[0]){
                           //recursive call on successive lists:
                           tempList.set(new_idx, 0);
                           content.set(j, tempList);
                    }
                   }
        }
       }
       else{
       if(typename.equals("listof")){
          for(int m = 0; m < content.size(); m++){
              Listof tempList;
              tempList = (Listof) content.get(m);
              //recursive call on successive lists:
              int[] new_idx = new int[idx.length - 1];
              for(int j = 1; j < idx.length ; j++){
                new_idx[j-1] = idx[j];
              }
              tempList.set(new_idx, 0);
              content.set(m, tempList);
           }
       }
      }
   }

   if(typename.equals("listof")){
       Listof tempList;
       tempList = (Listof) content.get(index);
       //recursive call on successive lists:
       int[] new_idx = new int[idx.length - 1];
       for(int j = 1; j < idx.length ; j++){
           new_idx[j-1] = idx[j];
       }
       tempList.set(new_idx, stuff);
       content.set(index, tempList);
   }
   else{
       content.set(index, new Integer(stuff));
   }

}


public void realset(int[] idx, int stuff){
    Listof toset = new Listof(typename, content);
    toset.set(idx, stuff);
    toset.fixn(idx);
    typename = toset.typename;
    content = toset.content;
}

private void fixf(int[] idx){
   if(typename.equals("listof")){
     int[] new_idx = new int[idx.length - 1];
     for(int j = 1; j < idx.length ; j++){
```

```
                new_idx[j-1] = idx[j];
            }
            for(int b = 0; b < content.size(); b++){
                  Listof templist = (Listof) content.get(b);
                  Listof templistcopy = templist;
                      if(templist.size()<=new_idx[0]){
                          templist.set(new_idx, new Fraction(0, 0));
                      }
                      else{
                          if(new_idx.length > 1){
                             int end = templist.size();
                             for(int h = 0; h < end; h++){
                                   templistcopy = (Listof) templist.content.get(h);
                                   if(templistcopy.size()<=new_idx[1]){
                                      int[] new_idx2 = new int[new_idx.length - 1];
                                      for(int j = 1; j < new_idx.length ; j++){
                                          new_idx2[j-1] = new_idx[j];
                                      }
                                      templistcopy.set(new_idx2, new Fraction(0, 0));
                                      templist.content.set(h, templistcopy);
                                   }
                             }
                          }
                      }
                      content.set(b, templist);
               }
            }
        }


        private void set(int[] idx, Fraction stuff){
           int index = idx[0];
           int original = content.size();
           if(index > (content.size()-1)){
               for(int i = content.size(); i <= index; i++){
                   if(idx.length >= 3){
                      content.add(new Listof("listof"));
                   }
                   if(idx.length == 2){
                       content.add(new Listof("fraction"));
                   }
                   if(idx.length == 1){
                       content.add(new Fraction(0, 0));
                   }
               }

               if(original>0){
                       int fill = 0;
                       int[] new_idx = new int[idx.length-1];
                       if(typename.equals("listof")){
                           Listof crap = (Listof) content.get(0);
                           fill = crap.content.size()-1;

                           for(int j = 1; j < idx.length ; j++){
                                new_idx[j-1] = idx[j];
                            }
                           if(fill>new_idx[0]){
```

```
                          new_idx[0] = fill;
                     }
                     for(int j = 0; j <= index; j++ ){
                      Listof tempList;
                      tempList = (Listof) content.get(j);
                      if(tempList.content.size()<=new_idx[0]){
                             //recursive call on successive lists:
                             tempList.set(new_idx, new Fraction(0, 0));
                             content.set(j, tempList);
                      }
                     }
          }
         }
         else{
         if(typename.equals("listof")){
            for(int m = 0; m < content.size(); m++){
                Listof tempList;
                tempList = (Listof) content.get(m);
                //recursive call on successive lists:
                int[] new_idx = new int[idx.length - 1];
                for(int j = 1; j < idx.length ; j++){
                  new_idx[j-1] = idx[j];
                }
                tempList.set(new_idx, new Fraction(0, 0));
                content.set(m, tempList);
             }
         }
        }
     }

     if(typename.equals("listof")){
         Listof tempList;
         tempList = (Listof) content.get(index);
         //recursive call on successive lists:
         int[] new_idx = new int[idx.length - 1];
         for(int j = 1; j < idx.length ; j++){
             new_idx[j-1] = idx[j];
         }
         tempList.set(new_idx, stuff);
         content.set(index, tempList);
     }
     else{
         content.set(index, stuff);
     }

  }


  public void realset(int[] idx, Fraction stuff){
      Listof toset = new Listof(typename, content);
      toset.set(idx, stuff);
      toset.fixf(idx);
      typename = toset.typename;
      content = toset.content;
  }
```

```java
public Listof resize(int[] size){
    Listof copy = new Listof(typename, content);
    int siz = size[0];
    int[] idx = new int[size.length];
    for(int i = 0; i < size.length; i++){
        idx[i] = size[i] - 1;
    }
    if(siz > (copy.content.size())){
        copy.realset(idx, 0);
    }
    else{
        Listof newCopy = new Listof(copy.typename);
        if(newCopy.typename.equals("number")){
            for(int i = 0; i < siz; i++){
                int stuff = ((Integer)copy.content.get(i)).intValue();
                newCopy = newCopy.appendPrimitive(new Integer(stuff));
            }
        }
        if(newCopy.typename.equals("fraction")){
            for(int i = 0; i < siz; i++){
                Fraction stuff = (Fraction) copy.content.get(i);
                newCopy = newCopy.appendPrimitive(stuff);
            }
        }
        if(newCopy.typename.equals("string")){
            for(int i = 0; i < siz; i++){
                String stuff = (String) copy.content.get(i);
                newCopy = newCopy.appendPrimitive(stuff);
            }
        }
        if(newCopy.typename.equals("listof")){
            for(int i = 0; i < siz; i++){
                Listof stuff = (Listof) copy.content.get(i);
                newCopy = newCopy.appendPrimitive(stuff);
            }
        }
        copy = newCopy;
    }
    if((size.length - 1) > 0){
        int[] new_sz = new int[size.length - 1];
        for(int j = 1; j < size.length ; j++){
             new_sz[j-1] = size[j];
        }
        for(int i = 0; i < copy.content.size(); i++){
             Listof newCopy = (Listof) copy.content.get(i);
             newCopy = newCopy.resize(new_sz);
             copy.content.set(i, newCopy);
        }
    }
    return copy;
}


//      USEFUL functions
//---------------------------------------------------

public String toString(){
        return content.toString();
    }

    public Object clone(){
        Listof copy = new Listof(typename, (ArrayList) content.clone());
        return copy;
    }
}
```

12. Fraction.java

```
/*
 * Fraction.java
 *
 * Created on April 18, 2007, 6:58 PM
 *
 */

/**
 *
 * @author Joseanibal Colon R
 */
public class Fraction {

    private int numerator, denominator;

    /** Creates a new instance of Fraction */
    public Fraction(int numer, int denom) {
        //ensure non-zero denominator
        if (denom == 0)
            denom = 1;
        //make numerator store the sign
        if (denom < 0){
            numer = numer * -1;
            denom = denom * -1;
        }

        numerator = numer;
        denominator = denom;

        reduce();
    }

    public int getNumerator(){
        return numerator;
    }

    public int getDenominator(){
        return denominator;
    }

    public Fraction reciprocal(){
        return new Fraction(denominator, numerator);
    }

    public Fraction add(Fraction op2){
        int commonDenominator = denominator * op2.getDenominator();
        int numerator1 = numerator * op2.getDenominator();
        int numerator2 = op2.getNumerator() * denominator;
        int sum = numerator1 + numerator2;

        return new Fraction (sum, commonDenominator);
    }

    public Fraction subtract(Fraction op2){
        int commonDenominator = denominator * op2.getDenominator();
        int numerator1 = numerator * op2.getDenominator();
        int numerator2 = op2.getNumerator() * denominator;
        int difference = numerator1 - numerator2;

        return new Fraction (difference, commonDenominator);
    }

    public Fraction multiply(Fraction op2){
        int numer = numerator * op2.getNumerator();
        int denom = denominator * op2.getDenominator();

        return new Fraction(numer, denom);
    }

    public Fraction divide(Fraction op2){

        return multiply(op2.reciprocal());
    }

    public Fraction add(int op2){
        Fraction conv = new Fraction(op2, 0);
        Fraction curr = new Fraction(numerator, denominator);
        Fraction result = curr.add(conv);

        return result;
    }

    public Fraction subtract(int op2){
        Fraction conv = new Fraction(op2, 0);
        Fraction curr = new Fraction(numerator, denominator);
        Fraction result = curr.subtract(conv);

        return result;
    }

    public Fraction multiply(int op2){
        Fraction conv = new Fraction(op2, 0);
        Fraction curr = new Fraction(numerator, denominator);
        Fraction result = curr.multiply(conv);

        return result;
    }

    public Fraction divide(int op2){
        Fraction conv = new Fraction(op2, 0);
        Fraction curr = new Fraction(numerator, denominator);
        Fraction result = curr.divide(conv);
        return result;
    }

    //SR
    public Fraction exp(int exponent){
        int numer = (int)Math.pow(numerator,exponent);
        int denom = (int)Math.pow(denominator,exponent);

        return new Fraction(numer, denom);
    }

    public Fraction modulus(int mod){
        double floor = Math.floor((double)numerator/((double)denominator *
(double)mod));
        int result = mod*(int)floor;
        Fraction frac = new Fraction(numerator, denominator);
        return frac.subtract(new Fraction (result, 0));
    }

    public Fraction uminus(){
        Fraction zero = new Fraction(0, 0);
        Fraction curr = new Fraction(numerator, denominator);
        return zero.subtract(curr);
    }


    public int FracToNum(){
        return numerator/denominator;
    }

    public int eq(Fraction op2){

        return( numerator == op2.getNumerator() &&
                denominator == op2.getDenominator() )?1:0;
    }

    public int ne(Fraction op2){

        return( numerator != op2.getNumerator() ||
                denominator != op2.getDenominator() )?1:0;
    }

    public int gt(Fraction op2){
        float d1 = (float)numerator / (float)denominator;
        float d2 = (float)op2.getNumerator() / (float)op2.getDenominator();

        return d1>d2?1:0;
    }

    public int ge(Fraction op2){
        float d1 = (float)numerator / (float)denominator;
        float d2 = (float)op2.getNumerator() / (float)op2.getDenominator();

        return d1>=d2?1:0;
    }
    public int lt(Fraction op2){
        float d1 = (float)numerator / (float)denominator;
        float d2 = (float)op2.getNumerator() / (float)op2.getDenominator();

        return d1<d2?1:0;
    }

    public int le(Fraction op2){
        float d1 = (float)numerator / (float)denominator;
        float d2 = (float)op2.getNumerator() / (float)op2.getDenominator();

        return d1<=d2?1:0;
    }

    public String toString(){
        String result;

        if(numerator == 0)
            result = "0";
        else
            if(denominator == 1)
                result = numerator + "";
            else
                result = numerator + "/" + denominator;

        return result;
    }

    private void reduce(){
        if (numerator != 0){
            int common = gcd(Math.abs(numerator), denominator);

            numerator = numerator / common;
            denominator = denominator / common;
        }
    }

    private int gcd(int num1, int num2){
        while (num1 != num2)
            if (num1 > num2)
                num1 = num1 - num2;
            else
                num2 = num2 - num1;

        return num1;
    }
}
```

13. GUI.java

```
/*
 * GUI.java
 *
 * Created on May 6, 2007, 3:32 AM
 */
```

```
/**
 * @author Joseanibal Colon R
 */
import java.awt.*;
import java.beans.PropertyChangeListener;
import java.util.ArrayList;
import java.util.Random;
import javax.swing.*;
import java.awt.event.*;

public class GUI {

    private int width = 600;
    private int height = 650;
    private int rid;

    private JFrame frame;
    private JPanel panel, subpanel1, subpanel2;
    private ArrayList printouts = new ArrayList();
    private JTextField[] texts = new JTextField[18];
    private JTextField input;
    private JLabel inputLabel, outputLabel;
    private JButton refresh;

    /** Creates a new instance of GUI */
    public GUI() {
        rid = 0;
        String space = " ";
        String pointer = ">";
        frame = new JFrame ("The Leraning Language Window");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        for(int i = 0; i < 17; i++){
            printouts.add(space);
            texts[i] = new JTextField();
        }
        printouts.add(pointer);
        texts[17] = new JTextField();

        input = new JTextField();
        input.setBackground(Color.green);

        inputLabel = new JLabel("INPUT:");
        outputLabel = new JLabel("OUTPUT:");

        panel = new JPanel();
        panel.setPreferredSize(new Dimension(width, height));
        panel.setBackground(Color.green);
        panel.setLayout(new BoxLayout (panel, BoxLayout.Y_AXIS));

        subpanel1 = new JPanel();
        refresh = new JButton("Refresh");
        refresh.addActionListener(new refreshListener());
        subpanel1.setBackground(Color.green);
        subpanel1.setLayout(new BoxLayout (subpanel1, BoxLayout.X_AXIS));
        subpanel1.add(outputLabel);
        subpanel1.add(Box.createHorizontalGlue());
        subpanel1.add(refresh);
        panel.add(subpanel1);

        for(int i = 0; i < 18; i++){
            texts[i].setText( (String) printouts.get(i));
            panel.add(texts[i]);
        }

        subpanel2 = new JPanel();
        subpanel2.setBackground(Color.green);
        subpanel2.setLayout(new BoxLayout (subpanel2, BoxLayout.X_AXIS));
        subpanel2.add(inputLabel);
        subpanel2.add(Box.createHorizontalGlue());

        panel.add(subpanel2);
        panel.add(Box.createVerticalGlue());
        panel.add(input);
        frame.getContentPane().add(panel);
        display();
    }

    public void display(){
        frame.pack();
        frame.setVisible(true);
    }

    public void println(String stuff){
        print(stuff);
        String empty = "";
        for(int i = 0; i <= 16; i++){
            printouts.set(i, printouts.get(i+1));
        }
        printouts.set(17, empty);
        //printouts.set(17, stuff);
        //for(int i = 0; i < 18; i++){
        //    texts[i].setText( (String) printouts.get(i));
        //}

    }

    public void println(int number){
        print(number);
        String stuff = "";
        //String stuff = Integer.toString(number);
        for(int i = 0; i <= 16; i++){
            printouts.set(i, printouts.get(i+1));
        }
        printouts.set(17, stuff);
        //for(int i = 0; i < 18; i++){
        //    texts[i].setText( (String) printouts.get(i));
        //}

    }

    public void println(Fraction thing){
        print(thing);
        String stuff = "";
        //String stuff = thing.toString();
        for(int i = 0; i <= 16; i++){
            printouts.set(i, printouts.get(i+1));
        }
        printouts.set(17, stuff);
        //for(int i = 0; i < 18; i++){
        //    texts[i].setText( (String) printouts.get(i));
        //}

    }

    public void println(Listof thing){
        print(thing);
        String stuff = "";
        //String stuff = thing.toString();
        for(int i = 0; i <= 16; i++){
            printouts.set(i, printouts.get(i+1));
        }
        printouts.set(17, stuff);
        //for(int i = 0; i < 18; i++){
        //    texts[i].setText( (String) printouts.get(i));
        //}

    }


    public void print(String stuff){
        String copy = (String) printouts.get(17);
        stuff = copy.concat(stuff);
        printouts.set(17, stuff);
        for(int i = 0; i < 18; i++){
            texts[i].setText( (String) printouts.get(i));
        }

    }

    public void print(int number){
        String stuff = Integer.toString(number);
        String copy = (String) printouts.get(17);
        stuff = copy.concat(stuff);
        printouts.set(17, stuff);
        for(int i = 0; i < 18; i++){
            texts[i].setText( (String) printouts.get(i));
        }

    }

    public void print(Fraction thing){
        String stuff = thing.toString();
        String copy = (String) printouts.get(17);
        stuff = copy.concat(stuff);
        printouts.set(17, stuff);
        for(int i = 0; i < 18; i++){
            texts[i].setText( (String) printouts.get(i));
        }

    }
```

```
    public void print(Listof thing){
        String stuff = thing.toString();
        String copy = (String) printouts.get(17);
        stuff = copy.concat(stuff);
        printouts.set(17, stuff);
        for(int i = 0; i < 18; i++){
            texts[i].setText( (String) printouts.get(i));
        }

    }

    public String read(){
        Color color = input.getBackground();
        input.setBackground(Color.white);
        input.getInputMap().put(KeyStroke.getKeyStroke(KeyEvent.VK_ENTER,
0),"check");
        Action readStuff = new readAction();
        input.getActionMap().put("check", readStuff);
        while(rid == 0){};
        String stuff = input.getText();
        println(stuff);
        rid = 0;
        input.setBackground(color);
        return stuff;
    }


    public void reprint(){
        for(int i = 0; i < 18; i++){
            texts[i].setText( (String) printouts.get(i));
        }
        Random generator = new Random();
        int r = generator.nextInt(128)+128;
        int g = generator.nextInt(128)+128;
        int b = generator.nextInt(128)+128;
        Color color = new Color(r, g, b);
        panel.setBackground(color);
        subpanel1.setBackground(color);
        subpanel2.setBackground(color);
        input.setBackground(color);
    }


    private class refreshListener implements ActionListener{
        public void actionPerformed(ActionEvent e) {
            reprint();
        }
    }

    private class readAction extends AbstractAction{
        public void actionPerformed(ActionEvent e){
            rid = 1;
            input.resetKeyboardActions();
        }

    }
```

```
}
```

14. LLIO.java

```
import java.io.*;

//Used by "compiled" LL code for all I/O operations
//Stephen Robinson
//v1.0
//Updated to use the GUI as std.  Old code commented out in case
//we want to be able to generate code without the GUI
public class LLIO {

  BufferedReader in;
  BufferedWriter out;
  GUI std;
  //BufferedReader stdin;

  public LLIO()
  {
        std = new GUI();
        //stdin = new BufferedReader(new InputStreamReader(System.in));
  }

  public void openOutFile(String s)
  {
        File Out = new File(s);
        try {
              out = new BufferedWriter(new FileWriter(Out));
        }
        catch (Exception ex)
        {
              output(ex.toString());
        }
  }

  public void closeOutFile()
  {
        try {
              out.close();
        }
        catch (Exception ex)
        {
              output(ex.toString());
        }
  }

  public void openInfile(String s)
  {
        File In = new File(s);
        try {
              out = new BufferedWriter(new FileWriter(In));
        } catch (Exception ex)
        {
              output(ex.toString());
        }
```

```
}

public void closeInFile()
{
      try {
            out.close();
      }
      catch (Exception ex)
      {
            output(ex.toString());
      }
}

public String readLineFromFile()
{
      String s = "";
      try
      {
            s = in.readLine();
      }
      catch (IOException e)
      {
            System.err.println(e.toString());
      }
      return s;
}

public String readWordFromFile()
{
      String s = "";
      try
      {
            int c;
            c = in.read();
            while((c = in.read()) != ' ' && c != '\n' && c != '\r')
            {
                  s += c;
            }
      }
      catch (IOException e)
      {
            System.err.println(e.toString());
      }
      return s;
}

public String read()
{
      String s = std.read();
      return s;
}

public void outputToFile(String s)
{
      try{
            out.write(s);
      }
```

```
		catch (IOException e)
		{
			System.err.println(e.toString());
		}
	}

	public void outputLineToFile(String s)
	{
		try{
			out.write(s);
			out.newLine();
		}
		catch (IOException e)
		{
			System.err.println(e.toString());
		}
	}

	public void outputToFile(int n)
	{
		try{
			out.write(n);
		}
		catch (IOException e)
		{
			System.err.println(e.toString());
		}
	}

	public void outputLineToFile(int n)
	{
		try{
			out.write(n);
			out.newLine();
		}
		catch (IOException e)
		{
			System.err.println(e.toString());
		}
	}

	public void outputToFile(Fraction f)
	{
		try{
			out.write(f.getNumerator());
			if(f.getDenominator() != 1)
			{
				out.write("/");
				out.write(f.getDenominator());
			}
		}
		catch (IOException e)
		{
			System.err.println(e.toString());
		}
	}
```

```
public void outputLineToFile(Fraction f)
{
      try
      {
            out.write(f.getNumerator());
            if(f.getDenominator() != 1)
            {
                  out.write("/");
                  out.write(f.getDenominator());

            }
            out.newLine();
      }
      catch (IOException e)
      {
            System.err.println(e.toString());
      }

}

public void outputToFile(Listof l)
{
      if(l.typename == "number")
      {
            outputToFile("{ ");
            for(int i = 0; i < l.size() - 1; i++)
            {
                  outputToFile(((Integer)l.content.get(i)).toString());
                  outputToFile(" , ");
            }
            outputToFile(((Integer)l.content.get(l.size() - 1)).toString());
            outputToFile(" }");
      }
      else if(l.typename == "string")
      {
            outputToFile("{ ");
            for(int i = 0; i < l.size() - 1; i++)
            {
                  outputToFile((String)l.content.get(i));
                  outputToFile(" , ");
            }
            outputToFile((String)l.content.get(l.size() - 1));
            outputToFile(" }");
      }
      else if(l.typename == "fraction")
      {
            outputToFile("{ ");
            for(int i = 0; i < l.size() - 1; i++)
            {
                  outputToFile((Fraction)l.content.get(i));
                  outputToFile(" , ");
            }
            outputToFile((Fraction)l.content.get(l.size() - 1));
            outputToFile(" }");
      }
      else
      {
            outputToFile("{ ");
            for(int i = 0; i < l.size() - 1; i++)
            {
                  outputToFile((Listof)l.content.get(i));
                  outputToFile(" , ");
            }
            outputToFile((Listof)l.content.get(l.size() - 1));
            outputToFile(" }");
      }
}

public void outputLineToFile(Listof l)
{
      if(l.typename == "number")
      {
            outputToFile("{ ");
            for(int i = 0; i < l.size() - 1; i++)
            {
                  outputToFile(((Integer)l.content.get(i)).toString());
                  outputToFile(" , ");
            }
            outputToFile(((Integer)l.content.get(l.size() - 1)).toString());
            outputLineToFile(" }");
      }
      else if(l.typename == "string")
      {
            outputToFile("{ ");
            for(int i = 0; i < l.size() - 1; i++)
            {
                  outputToFile((String)l.content.get(i));
                  outputToFile(" , ");
            }
            outputToFile((String)l.content.get(l.size() - 1));
            outputLineToFile(" }");
      }
      else if(l.typename == "fraction")
      {
            outputToFile("{ ");
            for(int i = 0; i < l.size() - 1; i++)
            {
                  outputToFile((Fraction)l.content.get(i));
                  outputToFile(" , ");
            }
            outputToFile((Fraction)l.content.get(l.size() - 1));
            outputLineToFile(" }");
      }
      else
      {
            outputToFile("{ ");
            for(int i = 0; i < l.size() - 1; i++)
            {
                  outputToFile((Listof)l.content.get(i));
                  outputToFile(" , ");
            }
            outputToFile((Listof)l.content.get(l.size() - 1));
            outputToFile(" }");
      }
```

```java
}



public void output(String s)
{
      std.print(s);
      //System.out.print(s);
}

public void outputLine(String s)
{
      std.println(s);
      //System.out.println(s);
}

public void output(int n)
{
      std.print(n);
      //System.out.print(n);
}

public void outputLine(int n)
{
      std.println(n);
      //System.out.println(n);
}

public void output(Fraction f)
{
      std.print(f.getNumerator());
      //System.out.print(f.getNumerator());
      if(f.getDenominator() != 1)
      {
            std.print("/");
            std.print(f.getDenominator());
            //System.out.print("/");
            //System.out.print(f.getDenominator());
      }
}

public void outputLine(Fraction f)
{
      std.print(f.getNumerator());
      //System.out.print(f.getNumerator());
      if(f.getDenominator() != 1)
      {
            std.print("/");
            std.println(f.getDenominator());
            //System.out.print("/");
            //System.out.println(f.getDenominator());
      }
      else
      {
            std.println("");
            //System.out.println();
      }
```

```
}

public void output(Listof l)
{
      if(l.typename == "number")
      {
            output("{ ");
            for(int i = 0; i < l.size() - 1; i++)
            {
                  output(((Integer)l.content.get(i)).toString());
                  output(" , ");
            }
            output(((Integer)l.content.get(l.size() - 1)).toString());
            output(" }");
      }
      else if(l.typename == "string")
      {
            output("{ ");
            for(int i = 0; i < l.size() - 1; i++)
            {
                  output((String)l.content.get(i));
                  output(" , ");
            }
            output((String)l.content.get(l.size() - 1));
            output(" }");
      }
      else if(l.typename == "fraction")
      {
            output("{ ");
            for(int i = 0; i < l.size() - 1; i++)
            {
                  output((Fraction)l.content.get(i));
                  output(" , ");
            }
            output((Fraction)l.content.get(l.size() - 1));
            output(" }");
      }
      else
      {
            output("{ ");
            for(int i = 0; i < l.size() - 1; i++)
            {
                  output((Listof)l.content.get(i));
                  output(" , ");
            }
            output((Listof)l.content.get(l.size() - 1));
            output(" }");
      }
}

public void outputLine(Listof l)
{
      if(l.typename == "number")
      {
            output("{ ");
            for(int i = 0; i < l.size() - 1; i++)
            {
                    output(((Integer)l.content.get(i)).toString());
                    output(" , ");
                }
                output(((Integer)l.content.get(l.size() - 1)).toString());
                outputLine(" }");
            }
            else if(l.typename == "string")
            {
                output("{ ");
                for(int i = 0; i < l.size() - 1; i++)
                {
                    output((String)l.content.get(i));
                    output(" , ");
                }
                output((String)l.content.get(l.size() - 1));
                outputLine(" }");
            }
            else if(l.typename == "fraction")
            {
                output("{ ");
                for(int i = 0; i < l.size() - 1; i++)
                {
                    output((Fraction)l.content.get(i));
                    output(" , ");
                }
                output((Fraction)l.content.get(l.size() - 1));
                outputLine(" }");
            }
            else
            {
                output("{ ");
                for(int i = 0; i < l.size() - 1; i++)
                {
                    output((Listof)l.content.get(i));
                    output(" , ");
                }
                output((Listof)l.content.get(l.size() - 1));
                output(" }");
            }
    }
}
```

Test Files

1. sortlist.ll

```
# George Liao
# Learning Language Test Code
# May 7, 2007

#compiles and runs correctly as of 9pm May 7th.

listof number called theList
listof number called tempList
listof number called orderedList

theList <- 17.5.3.19.24.4.12
tempList <- theList

number called listlength <- |tempList|

# repeat until the list is emptied
number called tempMax
number called maxIndex
repeat until listlength = 0
  tempMax <- tempList[0]
  maxIndex <- 0

  # search for max remaining in list
  repeat listlength times
        if tempMax < tempList[nth] then
              tempMax <- tempList[nth]
              maxIndex <- nth
        endif
  endloop

      #add the new max to the sorted list
  orderedList <- tempmax. orderedList
  listof number called newTempList
  repeat listlength times
        if maxIndex != nth then
              newTempList <- newTempList . tempList[nth]
        endif
  endloop
  tempList <- newtemplist
  listlength <- |templist|

endloop

display "Unordered list: "
displayline theList
display "Ordered list: "
displayline orderedList
```

2. test2.ll

```
#list with non-list operator test
Listof num called lst <- 1.2.3.4.5.6.7.8.9.10

display "The list: "
displayline lst
display "The list squared: "
displayline lst ^ 2
display "2 raised to the list: "
displayline 2 ^ lst
display "The list times 2: "
displayline lst * 2
display "The list plus 2: "
displayline lst + 2
```

```
display "The list int div 2: "
displayline lst / 2
display "2 int div the list: "
displayline 2 / lst
display "The list divided by 2: "
displayline lst // 2
display "2 divided by the list: "
displayline 2 // lst
display "The list mod by 2: "
displayline lst % 2
display "2 mod by the list: "
displayline 2 % lst
display "The list minus 2: "
displayline lst - 2
display "2 minus the list: "
displayline 2 - lst
```