FINAL REPORT (COMS 4115)

Mathematical Expression Language (MEL)

Paresh Thatte (pat70@columbia.edu)

Manjiri Phadke (mp3212@columbia.edu)

16th August 2013

Table of Contents

1 Introduction

Mathematical Expression Language (MEL) is a language of symbols for representing patterns, restrictions and relationships of entities in the form of one or more equations of a set of variables. Equations are often represented in software systems. A small set of inputs can drive amortization schedule report, permission lists can trigger violation alerts on trade restrictions, and risk models can re-calculate pre-trading limits in real time. Manipulating these equations by expansion, factorization, solving (simultaneously) and other symbolic rearrangement using (high-school) algebra rules can reduce the complexity of a problem. MEL provides a way to program these algebra rules to solve, simplify and manipulate common mathematical equations.

2 Overview

MEL Language syntax allows expressing one or more equations using the standard algebraic notation. Equations of a valid form will be accepted and transformed by a programmer into a different form and other operations can be performed on them.

One transformation possible is to express the equation in terms of one variable, or solve for it given numbers. Other possible transformations are combining simultaneous equations, expanding a factorized expression or factorizing a simplified expression, collecting terms containing one of the variables in an expression.

In addition MEL supports basic programming language functionalities to allow users to implement algorithms. The functionalities are if-else statement blocks, for loops, functions and variables.

3 Language Tutorial

MEL supports basic programming language functionalities to allow users to implement algorithms. The functionalities are if-else statement blocks, for loops, functions and variables.

A programmer could solve algebraic equations like x + 3 = 5 and result will be 2. Equations can be expanded, for e.g. equation "**`(a+b)**2`**" *(** is the symbol used for to the power in MEL*) will be expanded as "**`a**2+2*a* b + **2`**". In addition operations like addTerm, removeTerm and find coefficients can be performed on a polynomial equation.

MEL programs must have a evaluate function (similar to main function in C) . The flow of the program is directed by the evaluate function. The program source files are compiled and translated into java source files which are built in-to executables using the java compiler.

3.1 Sample 1

Expand Equation: **(2x+4)**2**

```
func void evaluate:{
             Poly poly = "(2x+4)**2";
             Poly expandedPoly = "";
             [expand:poly, expandedPoly];
             print["Expand test case: "];
             print["Expansion for '(2x+4)**2' is "+expandedPoly];
      }

      func void expand:Poly poly, Poly expandedPoly
      {
             Term term;
             int exp;
             int curExp;
             int binCoeff;
             int localCoeff;
             int secondCoeff;
             int coeff;
             int zero;
             int one;
             zero = 0;
             one = 1;
             exp = poly.exp;
             if [exp != 1] {
                   for [i = 0; i <= exp; i = i + 1] {
                          curExp = exp - i;
                          binCoeff = [poly.binomialCoefficient:exp, i];
                          localCoeff = [poly.pow:poly.coeffs[one], curExp];
                          secondCoeff = [poly.pow:poly.coeffs[zero], i];
                          coeff = binCoeff * localCoeff * secondCoeff;
                          if [i != exp] {
                                [expandedPoly.addTerm:"" + coeff +
                          [poly.terms[one].syms.get:zero] + "**" + curExp];
                          } else {
                                [expandedPoly.addTerm:"" + coeff];
                          }
                   }
             }
      }
```

3.2 Sample 2

Solve a Polynomial Equation: `3x-11=x+1`

```
/**
 * @param args
 */
func void evaluate:{
       Sym s = "x";
       Eq e = "3x-11=x+1";
       [solve:s, e];
}

func void solve:Sym v, Eq e
{
       [move:v, e.rhs, e.lhs];
       [keep:v, e.lhs, e.rhs];
       [divide:e.lhs, e.rhs];
       print["Solve test case 3:"];
       print["Solution for '3x-11=x+1' is " + e.rhs];
}

func void divide:Poly from, Poly to
{
    int k;
    k = 1;
       for[i = 0; i<to.terms.length; i = i+1] {
              if [to.coeffs[i] == null] continue;
              to.coeffs[i] = to.coeffs[i]/from.coeffs[k];
       }
}

func void move:Sym v, Poly from, Poly to
{
       Term curTerm;
       for [i = 0; i < from.terms.length; i = i+1] {
              curTerm = from.terms[i];
              if [ curTerm!=null and [curTerm.hasSym:v] ]
              {
                     if [to.coeffs[i]==null]
                            to.coeffs[i]=0;
                     to.coeffs[i] = to.coeffs[i] - from.coeffs[i];
                     if [to.terms[i]==null] to.terms[i] = [curTerm.newTerm:];
                     [to.terms[i].addSym:v];
                     [from.terms[i].removeSym:v];
                     if [ [from.terms[i].syms.size:] == 0]
                     {
                            from.coeffs[i] = null;
                     }
              }
       }
}
```

```
      func void keep:Sym v, Poly from, Poly to
      {
            Term curTerm;
            Sym curSym = "";
            for [i = 0; i < from.terms.length; i = i+1]
            {
                  curTerm = from.terms[i];
                  if [curTerm==null] continue;
                  if [ [curTerm.syms.size:] > 1 or [curTerm.hasSym:v] != true] {
                        if [to.coeffs[i]==null] to.coeffs[i]=0;
                        to.coeffs[i] = to.coeffs[i] - from.coeffs[i];
                  }
                  for [j = 0; j < [curTerm.syms.size:]; j = j+1]
                  {
                        curSym = [curTerm.syms.get:j];
                        if [curSym.symbol != v.symbol]
                        {
                              if [to.terms[i]==null] to.terms[i] =
                              [curTerm.newTerm:];
                              [to.terms[i].addSym: curSym];
                              [from.terms[i].removeSym: curSym];
                        }
                  }
                  if [ [from.terms[i].syms.size:] == 0]
                  {
                        from.coeffs[i] = null;
                  }
            }
      }
```

3.3 Sample 3

Find Coefficient for term `x**2` in polynomial `x**3-3x**2+5x-1`

```
/**
       * @param args
       */
      func void evaluate:{
            Poly p = "x**3-3x**2+5x-1";
            print["Poly 'x**3-3x**2+5x-1' - Find Coeff test case: "];
            print["Find coefficient of term 'x**2'"];
            print[[p.findCoeff:"x**2"]];
      }
```

3.4 Sample 4

Add Term **4x** to polynomial **x**3-3x**2+5x-1**

```
/**
 * @param args
 */
func void evaluate:{
      Poly p = "x**3-3x**2+5x-1";
      [p.addTerm:"+4x"];
      print["Poly 'x**3-3x**2+5x-1' - AddTerm test case: "];
      print["Add term '4x' to poly"];
      print[p];
}
```

3.5 Sample 5

Remove Term **x**2** from polynomial equation **x**3-3x**2+5x-1**

```
/**
 * @param args
 */
func void evaluate:{
      Poly p = "x**3-3x**2+5x-1";
      [p.removeTerm:"x**2"];
      print["Poly 'x**3-3x**2+5x-1' - remove term test case: "];
      print["Remove 'x**2' term from poly"];
      print[p];
}
```

4 Language Reference Manual

4.1 Lexical Conventions

Tokens accepted:

- Punctuation
- Constants
- Identifiers
- Operators
- Keywords

Keywords accepted:

- Types
- Syntax blocks
- Built-in functions

4.1.1 Punctuation Tokens

Basic math symbols: +, -, *, /, =, <, >, exp symbol - **

These can be used in type declarations or as manipulation instructions.

In type declarations:

=, <, > are allowed to separate the Left and Right side of an equation.

+, - are allowed to separate terms in a polynomial

*, ** are allowed within a term. The exponential token is only allowed with symbols.

As instructions:

+, -, *, / are allowed with integers, and = is allowed with identifiers

Separator: Period - . , Comma - ,

Commas are allowed to separate parameters in a function declaration or call.

Periods are allowed to reference parts of algebraic types.

Line Terminator: ;

These are allowed as termination tokens for statements

Parameter Grouping: quotes – “,”, parentheses - [,]

These are allowed in type or function declaration, and in array access instructions.

Block delimiters – {, }

These are allowed as function block or logical block delimiters

Block Comments: /*, */

These are allowed for entering a block of comments. All tokens within the block are ignored.

4.1.2 Identifiers

String literals: Upper or lowercase character strings

Identifiers are allowed in assignments to Types, constants and functions. All string literals except keywords can be used as identifiers.

4.1.3 Constants

Integer Literals: 0 - 9

Integer literals are allowed as constants such as a coefficient of a term, or as values of identifiers such as a loop counter etc.

Boolean: true, false

Boolean constants are allowed in conditional statements, as function return values, as identifiers for either purposes etc.

4.1.4 Operators

Basic math: +, -, *, /

Basic math operations are allowed with integer literals or identifiers referring to integer literals.

Comparison: ==, !=, <, <=, >, >=, !, and, or

Logical comparison operations can be performed on Boolean constants, identifiers referring to Boolean constants or the results of instructions that produce a Boolean constant. These are allowed in conditional statements.

Assignment: =

Assignment tokens are allowed with identifiers.

4.2 Keywords

4.2.1 Types

Collections - Array

Arrays can be constructed of any Types or constants, and can be assigned to identifiers.

Algebraic Types - Eq, Poly, Term, Sym

These are custom structures that are defined by the language for simplifying access to the individual parts of the algebraic notation being manipulated.

4.2.2 Syntax blocks

Function declaration – func

All MEL programs are organized within functions. The function keyword is followed by the return type. The identifier for the function name, and the parameter arguments list in parenthesis, followed by the statements block enclosed in the block delimiters.

Conditional/Loop statements – if/else, for

The looping construct consists of the keyword followed by - the loop identifier assignment, termination condition, and stepping instruction - in parenthesis, followed by a block of statements enclosed in the block delimiters.

The conditional keywords are followed by the comparison operator in parenthesis, followed by a block of statements within the block delimiters.

Control keywords – return, continue, break

The return keyword breaks the function block and returns immediately, while the continue keyword breaks the current instance of the looping construct.

Algebra keywords – lhs, rhs, terms, coeffs, syms

The two parts of an equation can be accessed using the lhs (left-hand side) and rhs (right-hand side) keywords. Each part is of type Poly.

The three types of parts in a Poly can be accessed using the keywords coeffs, and terms. Each returns an array indexed by the highest power symbol in that position. The coeffs returns an integer literal array, while the other two in turn return arrays – the symbols within each term, and their respective exponents.

4.2.3 Built-in functions

Term – hasSym, addSym, removeSym

The symbol manipulation functions support adding or removing a symbol from a given term. The default exponent is the current array index.

Poly – addTerm, removeTerm, findCoeff

These symbol manipulation functions support adding or removing a term from a given polynomial. The findCoeff function takes a pattern and returns its coefficient.

Print

The print function can be used with any constants or types associated with an identifier.

4.3 Precedence And Associativity

An Algebraic Type's part's access has the highest precedence.

Math operations follow the normal precedence and associativity rules:

*, /
+, -

Comparison operators have higher precedence than logical combination operators in conditional blocks:
!
==, !=, <, <=, >, >=
and, or

Assignment operators have lower precedence than type declarations and keywords.

4.4 Variable Scope

An MEL program is organized within functions. There are no global variables, all variables are local to the function they are declared in.

The function parameters are considered local variables, and have the same scope. The scope of Loop identifiers is within the declaring loop only.

4.5 Statements

Statements are present within function bodies and are terminated by the semi-colon token.

Basic statements

These are operations such as Type declarations, Identifier assignment, Type modification, Math operations, function calls and flow control instructions. These have to end in a termination token.

Statement Flow

Multiple statements can be listed within a function and they must be separated with semicolons. All statements will be evaluated in the order listed and will be affected by the instructions declared in the previous statements.

Syntax blocks

Function declarations, conditional blocks and loop constructs can be listed followed by block-delimiter tokens.

5 Project Plan

5.1 Process used for planning, specification, development and testing

Iterative Software development model was used to implement this project . An iterative life cycle model does not attempt to start with a full specification of requirements. Instead, development begins by specifying and implementing just part of the software, which is then reviewed in order to identify further requirements. This process is then repeated, producing a new version of the software at the end of each iteration of the model.

Iterative approach was used in all phases for this project :

- Planning
- Specification
- Development
- Testing

Planning :

Project planning started with brainstorming various ideas for implementing a compiler . The initial proposal was to create code for computing values for various mathematical formulas – like interest calculations and calculating amortization schedules . At this time we were not quite sure of components in Compiler Design , and this seemed more of a command line interface rather than a compiler. Using recommendations from Professor Edwards , we came up with a proposal for MEL.

Specification :

Once the proposal was approved , we started working on details for creating the compiler . Solving the calculator problem for HW1 was helpful in understanding the relation and functionality of scanner, parser, ast and code generator .

Specification phase consisted of creating Language Reference Manual for MEL.

Development and Testing :

Micro C compiler was taken as a base and we started implementing the MEL compiler .Each functionality was added phase wise , adding test cases in each phase.

The entire code base was regression tested after each new phase to make sure none of the existing functionality has broken, with addition of new functionalities .

5.2 Programming style guide used by the team

Since this compiler was written using OCAML , OCAML programming specifications and styles were followed.

We tried to follow the guidelines specified for writing OCAML code :
http://caml.inria.fr/resources/doc/guides/guidelines.en.html#programming

Some of the common guidelines we followed were :

- Write simple and clear programs.
- Subdivide the programs into small functions.
- Factor out snippets of repeated code by defining them in separate functions.
- Never copy-paste code when programming.
- Don't hesitate to comment when there's a difficulty.
- Prefer one comment at the beginning of the function.
- Always give the same name to function arguments which have the same meaning.
- Parentheses are meaningful: they indicate the necessity of using an unusual precedence. So they should be used wisely and not sprinkled randomly throughout programs. Usual mathematical precedence rules must apply.
- Use a version control system , to track changes. We used GIT .

5.3 Project Timeline

- **June 7th 2013** – Project Proposal submitted.
- **July 15th 2013** – Revised proposal and LRM submitted as per recommendations from Professor Edwards.
- **July 21st 2013** – First build of Scanner , Parser , Ast and Compiler.
- **August 3rd 2013** – Second build for Scanner , Parser, Ast and Compiler and also corresponding tests to test the functions.
- **August 4th 2013** – Completed implementing solve function.
- **August 7th 2013** – Started working on Final Project Report.
- **August 9th 2013** – Completed implementing expand function.
- **August 11th 2013** – Third build for Scanner , Parser, Ast and Compiler and also corresponding tests to test the functions.
- **August 12th 2013** – Added implementation for findCoefficients , AddTerm and RemoveTerm.
- **August 13th 2013** – Final build for Scanner , Parser, Ast and Compiler and also corresponding tests to test the functions.
- **August 14th 2013** – First revision of final project report completed.

- **August 16th 2013** – Final project completed and submitted.

5.4 Roles and Responsibilities of each team member

We worked on this project collectively for most part . We used GIT (for version control), and wrote the code for following modules. MicroC compiler was used as a reference for creating the below modules :

- Scanner
- Parser
- AST
- Code generator (compiler.ml)

We then merged the code at regular intervals and tested the functions for correctness . We also wrote individual test cases to test the above modules.

In addition Java code was written to incorporate MEL like functions in Java, so that after generating Java source code from MEL, the java code can be compiled to java.class file.

5.5 Software development environment used (tools and languages)

Code was written using Eclipse Ocaml plug in and cygwin shell was used to run programs.

- Operating System : Windows 8
- Programming Language : Ocaml 3.12.0 (including OCAMLYACC and OCAMLLEX)
- Text Editor: Eclipse OCAMLplug in , Notepad++
- Java : jdk 1.6.0
- Runtime Environment : Cygwin shell
- Other tools used : GIT for version control

6 Architectural Design

6.1 Architecture

The source file for the program written in MEL will have .mel extension. This file is the input to the scanner. This is the first phase of a compiler also known as Lexical Analysis or Scanning . We have used OCAMLLEX to generate scanner.ml. The next phase which is parsing uses the tokens produced by lexical analyzer to create the parse tree (a tree-like representation that depicts grammatical structure of the token stream). OCAMLYACC and parser.mly are used to parse, to generate an abstract syntax tree. The abstract syntax tree is defined in AST.mli.

The code generator takes the abstract syntax tree (intermediate representation of the source program) and maps it to the target language. The target language can be the machine code, but for this project we have selected the target language as Java. So the code generator, compile.ml creates a Java source code file. This .java file will be compiled using Java compiler to create the executable file.

Compiler can be classified into:
1) Front End
2) Back End

Defined below are the files which comprise front end and back end of a compiler:

Front End :

scanner.mll - This file provides token definitions, which are used by the parser, by mapping regular expressions to token names.

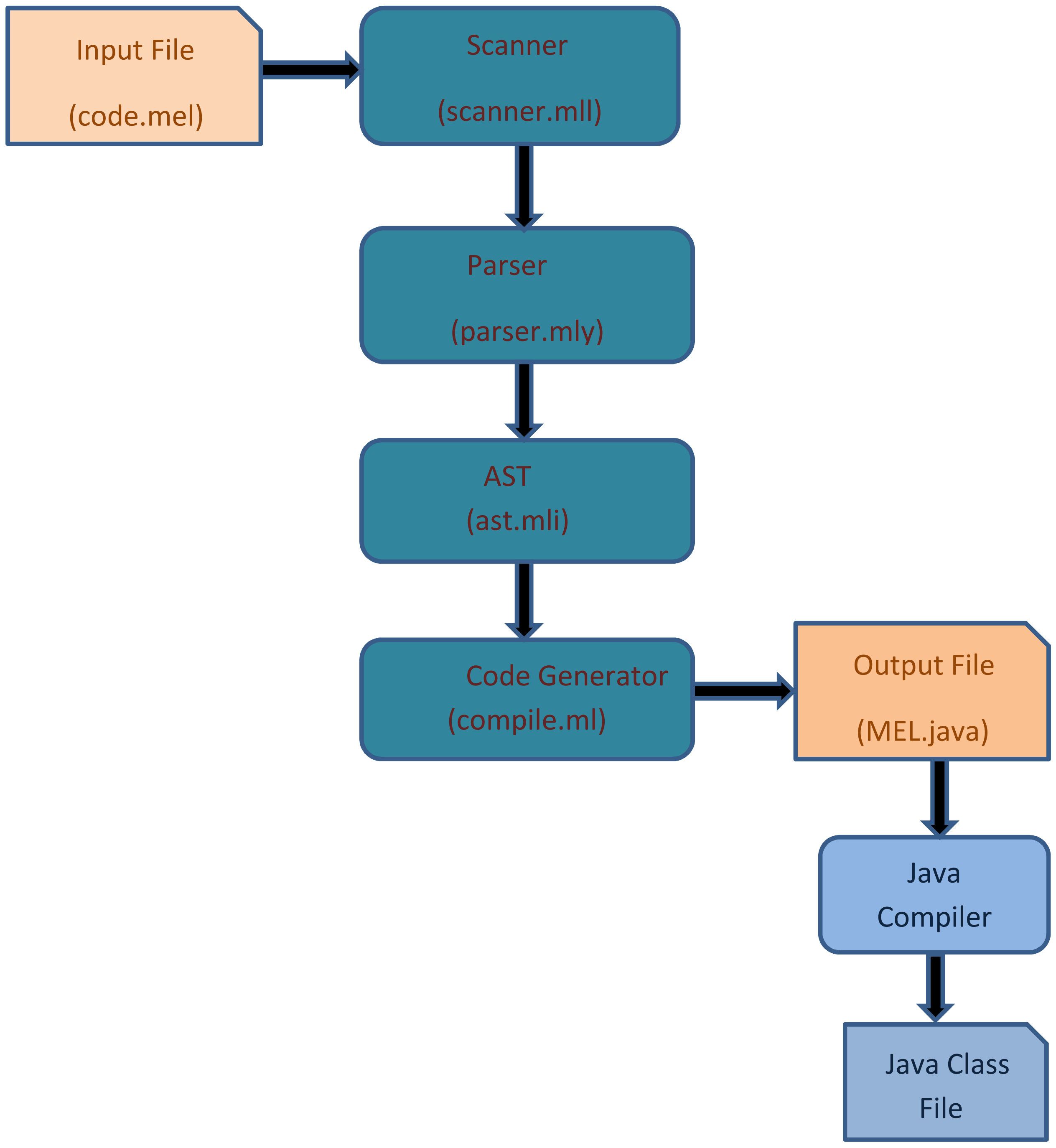
parser.mly - This file takes a stream of tokens and maps them to expressions. Additionally, associativity and precedence are defined to reduce conflicts associated with reduction and shifting.

ast.mli - This file provides the type definition and structure of expressions, which is used by the parser and by the backend.

Backend :

compiler.ml - This file provides entry point into the program as well as generating java code using the abstract syntax tree created from the front end.

6.2 Block Diagram of Major Components



7 Test Plan

Test programs were written for all the main features provided in the language.

For each feature provided in the MEL language :

1) Test source code is written as a separate .mel file.
2) A shell script is written which :
   - compiles this .mel file to .java code
   - generates java executable from .java code
   - executes the java executable

7.1 Test Programs

Example 1

Expand Function : To test expand function following files are written

1) test_expand.mel (Code in MEL language to expand a polynomial equation)
   Code is written to expand following polynomial equations:
   `(2x+4)**2`
2) test_expand.sh (Shell script to compile and execute the solve function)
3) MEL.java (java output for source test_expand.mel , generated after compiling test_expand.mel)

Input file – test_expand.mel

```
	func void evaluate:{
		Poly poly = "(2x+4)**2";
		Poly expandedPoly = "";
		[expand:poly, expandedPoly];
		print["Expand test case: "];
		print["Expansion for '(2x+4)**2' is "+expandedPoly];
	}

	func void expand:Poly poly, Poly expandedPoly
	{
		Term term;
		int exp;
		int curExp;
		int binCoeff;
		int localCoeff;
		int secondCoeff;
		int coeff;
		int zero;
		int one;
		zero = 0;
		one = 1;
		exp = poly.exp;
```

```
                if [exp != 1] {
                        for [i = 0; i <= exp; i = i + 1] {
                                curExp = exp - i;
                                binCoeff = [poly.binomialCoefficient:exp, i];
                                localCoeff = [poly.pow:poly.coeffs[one], curExp];
                                secondCoeff = [poly.pow:poly.coeffs[zero], i];
                                coeff = binCoeff * localCoeff * secondCoeff;
                                if [i != exp] {
                                        [expandedPoly.addTerm:"" + coeff +
                                [poly.terms[one].syms.get:zero] + "**" + curExp];
                                } else {
                                        [expandedPoly.addTerm:"" + coeff];
                                }
                        }
                }
        }
```

Output file – MEL.java

```
public class MEL {
        public static void expand(Poly poly, Poly expandedPoly) {
                Term term;
                int exp;
                int curExp;
                int binCoeff;
                int localCoeff;
                int secondCoeff;
                int coeff;
                int zero;
                int one;
                zero = 0;
                one = 1;
                exp = poly.exp;
                if (exp != 1) {
                        for (int i = 0; i <= exp; i = i + 1) {
                                curExp = exp - i;
                                binCoeff = poly.binomialCoefficient(exp, i);
                                localCoeff = poly.pow(poly.coeffs[one], curExp);
                                secondCoeff = poly.pow(poly.coeffs[zero], i);
                                coeff = binCoeff * localCoeff * secondCoeff;
                                if (i != exp) {
                                        expandedPoly.addTerm("" + coeff
                                                        + poly.terms[one].syms.get(zero) + "**"
+ curExp);
                                } else {
                                        expandedPoly.addTerm("" + coeff);
                                }
                        }
                }
        }

        public static void evaluate() {
                Poly poly = new Poly("(2x+4)**2");
```

```
            Poly expandedPoly = new Poly("");
            expand(poly, expandedPoly);
            System.out.println("Expand test case: ");
            System.out.println("Expansion for '(2x+4)**2' is " + expandedPoly);
      }

      public static void main(String args[]) {
            MEL.evaluate();
      }
}
```

Test Shell Script – test_expand.sh

```
../_build/mel < ../source/test_expand.mel > ../lib/MEL.java
javac -classpath "../lib/MELMath.jar;../lib/MELLib.jar" ../lib/MEL.java
java -classpath "../lib/;../lib/MELMath.jar;../lib/MELLib.jar" MEL
```

Output after running test_solve.sh:

```
$ ./test_expand.sh
Expand test case:
Expansion for '(2x+4)**2' is 4x**2+16x+16
```

Example 2:

Solve Function : To test solve function following files are written :

1) test_solve.mel (Code in MEL language to solve a polynomial equation)
   Code is written to solve following polynomial equations:

   ```
   x+3=4
   x-4=0
   3x-11=x+1
   3x=9
   7-y=5
   2y=10
   ```

2) test_solve.sh (Shell script to compile and execute the solve function)
3) MEL.java (java output for source test_solve.mel , generated after compiling test_solve.mel)

Input file – test_solve.mel

```
      /**
       * @param args
       */
      func void evaluate:{
            Sym s = "x";
            Eq e = "x+3=4";
            [solve:s, e];
            e = [e.newEq: "x-4=0"];
```

```
		[solve:s, e];
		e = [e.newEq: "3x-11=x+1"];
		[solve:s, e];
		e = [e.newEq: "3x=9"];
		[solve:s, e];
		e = [e.newEq: "7-y=5"];
		[solve:s, e];
		e = [e.newEq: "2y=10"];
		[solve:s, e];
	}

	func void solve:Sym v, Eq e
	{
		[move:v, e.rhs, e.lhs];
		[keep:v, e.lhs, e.rhs];
		[divide:e.lhs, e.rhs];
		print[e.rhs];
	}

	func void divide:Poly from, Poly to
	{
	    int k;
	    k = 1;
		for[i = 0; i<to.terms.length; i = i+1] {
			if [to.coeffs[i] == null] continue;
			to.coeffs[i] = to.coeffs[i]/from.coeffs[k];
		}
	}

	func void move:Sym v, Poly from, Poly to
	{
		Term curTerm;
		for [i = 0; i < from.terms.length; i = i+1] {
			curTerm = from.terms[i];
			if [ curTerm!=null and [curTerm.hasSym:v] ]
			{
				if [to.coeffs[i]==null]
					to.coeffs[i]=0;
				to.coeffs[i] = to.coeffs[i] - from.coeffs[i];
				if [to.terms[i]==null] to.terms[i] = [curTerm.newTerm:];
				[to.terms[i].addSym:v];
				[from.terms[i].removeSym:v];
				if [ [from.terms[i].syms.size:] == 0]
				{
					from.coeffs[i] = null;
				}
			}
		}
	}

	func void keep:Sym v, Poly from, Poly to
	{
		Term curTerm;
```

```
            Sym curSym = "";
            for [i = 0; i < from.terms.length; i = i+1]
            {
                  curTerm = from.terms[i];
                  if [curTerm==null] continue;
                  if [ [curTerm.syms.size:] > 1 or [curTerm.hasSym:v] != true] {
                        if [to.coeffs[i]==null] to.coeffs[i]=0;
                        to.coeffs[i] = to.coeffs[i] - from.coeffs[i];
                  }
                  for [j = 0; j < [curTerm.syms.size:]; j = j+1]
                  {
                        curSym = [curTerm.syms.get:j];
                        if [curSym.symbol != v.symbol]
                        {
                              if [to.terms[i]==null] to.terms[i] =
[curTerm.newTerm:];
                              [to.terms[i].addSym: curSym];
                              [from.terms[i].removeSym: curSym];
                        }
                  }
                  if [ [from.terms[i].syms.size:] == 0]
                  {
                        from.coeffs[i] = null;
                  }
            }
      }
```

Output file – MEL.java

```
public class MEL {
      public static void keep(Sym v, Poly from, Poly to) {
            Term curTerm;
            Sym curSym = new Sym("");
            for (int i = 0; i < from.terms.length; i = i + 1) {
                  curTerm = from.terms[i];
                  if (curTerm == null)
                        continue;
                  if (curTerm.syms.size() > 1 || curTerm.hasSym(v) != true) {
                        if (to.coeffs[i] == null)
                              to.coeffs[i] = 0;
                        to.coeffs[i] = to.coeffs[i] - from.coeffs[i];
                  }
                  for (int j = 0; j < curTerm.syms.size(); j = j + 1) {
                        curSym = curTerm.syms.get(j);
                        if (curSym.symbol != v.symbol) {
                              if (to.terms[i] == null)
                                    to.terms[i] = curTerm.newTerm();
                              to.terms[i].addSym(curSym);
                              from.terms[i].removeSym(curSym);
                        }
                  }
                  if (from.terms[i].syms.size() == 0) {
                        from.coeffs[i] = null;
```

```
                }
            }
    }

    public static void move(Sym v, Poly from, Poly to) {
            Term curTerm;
            for (int i = 0; i < from.terms.length; i = i + 1) {
                    curTerm = from.terms[i];
                    if (curTerm != null && curTerm.hasSym(v)) {
                            if (to.coeffs[i] == null)
                                    to.coeffs[i] = 0;
                            to.coeffs[i] = to.coeffs[i] - from.coeffs[i];
                            if (to.terms[i] == null)
                                    to.terms[i] = curTerm.newTerm();
                            to.terms[i].addSym(v);
                            from.terms[i].removeSym(v);
                            if (from.terms[i].syms.size() == 0) {
                                    from.coeffs[i] = null;
                            }
                    }
            }
    }

    public static void divide(Poly from, Poly to) {
            int k;
            k = 1;
            for (int i = 0; i < to.terms.length; i = i + 1) {
                    if (to.coeffs[i] == null)
                            continue;
                    to.coeffs[i] = to.coeffs[i] / from.coeffs[k];
            }
    }

    public static void solve(Sym v, Eq e) {
            move(v, e.rhs, e.lhs);
            keep(v, e.lhs, e.rhs);
            divide(e.lhs, e.rhs);
            System.out.println(e.rhs);
    }

    public static void evaluate() {
            Sym s = new Sym("x");
            Eq e = new Eq("x+3=4");
            solve(s, e);
            e = e.newEq("x-4=0");
            solve(s, e);
            e = e.newEq("3x-11=x+1");
            solve(s, e);
            e = e.newEq("3x=9");
            solve(s, e);
            e = e.newEq("7-y=5");
            solve(s, e);
            e = e.newEq("2y=10");
```

```
              solve(s, e);
       }

       public static void main(String args[]) {
              MEL.evaluate();
       }
}
```

Test Shell Script – test_solve.sh

```
../_build/mel < ../source/test_solve.mel > ../lib/MEL.java
javac -classpath "../lib/MELMath.jar;../lib/MELLib.jar" ../lib/MEL.java
java -classpath "../lib/;../lib/MELMath.jar;../lib/MELLib.jar" MEL
```

Output after running test_solve.sh:

```
$ ./test_solve.sh
1
4
6
3
2
5
```

7.2 Test Suites

Test cases were written for all the main features provided in MEL. Test cases were written for the following features :

1) Solve a polynomial equation.
2) Expand a polynomial equation.
3) FindCoefficients for a term in the polynomial equation.
4) AddTerm and RemoveTerm for a polynomial equation.
5) MoveSymbol and KeepSymbol for a polynomial equation.
6) Test cases were written for testing basic features in the language like – if statement , for loop, print and evaluate functions.

Test Case (Source Files) Listing

```
/MEL/source/test_divide.mel
/MEL/source/test_evaluate.mel
/MEL/source/test_expand1.mel
/MEL/source/test_expand2.mel
/MEL/source/test_expand3.mel
/MEL/source/test_expand4.mel
/MEL/source/test_forloop.mel
/MEL/source/test_if.mel
/MEL/source/test_keep.mel
```

```
/MEL/source/test_move.mel
/MEL/source/test_poly_addterm.mel
/MEL/source/test_poly_findcoeff.mel
/MEL/source/test_poly_parsing.mel
/MEL/source/test_poly_removeTerm.mel
/MEL/source/test_print.mel
/MEL/source/test_solve.mel
/MEL/source/test_solve1.mel
/MEL/source/test_solve2.mel
/MEL/source/test_solve3.mel
/MEL/source/test_solve4.mel
/MEL/source/test_solve5.mel
/MEL/source/test_solve6.mel
```

Test Scripts:

Listing of Test Scripts for each of the above Test programs :

```
/MEL/test/test_divide.sh
/MEL/test/test_evaluate.sh
/MEL/test/test_expand1.sh
/MEL/test/test_expand2.sh
/MEL/test/test_expand3.sh
/MEL/test/test_expand4.sh
/MEL/test/test_forloop.sh
/MEL/test/test_if.sh
/MEL/test/test_keep.sh
/MEL/test/test_move.sh
/MEL/test/test_poly_addterm.sh
/MEL/test/test_poly_findcoeff.sh
/MEL/test/test_poly_parsing.sh
/MEL/test/test_poly_removeterm.sh
/MEL/test/test_print.sh
/MEL/test/test_solve1.sh
/MEL/test/test_solve2.sh
/MEL/test/test_solve3.sh
/MEL/test/test_solve4.sh
/MEL/test/test_solve5.sh
/MEL/test/test_solve6.sh
```

A shell script which will execute all the above tests was written:

./test_all.sh

```
./test_expand1.sh
echo ""
./test_expand2.sh
echo ""
./test_expand3.sh
echo ""
```

```
./test_expand4.sh
echo ""
./test_poly_parsing.sh
echo ""
./test_poly_addterm.sh
echo ""
./test_poly_findcoeff.sh
echo ""
./test_poly_removeterm.sh
echo ""
./test_divide.sh
echo ""
./test_evaluate.sh
echo ""
./test_forloop.sh
echo ""
./test_if.sh
echo ""
./test_keep.sh
echo ""
./test_move.sh
echo ""
./test_print.sh
echo ""
./test_solve1.sh
echo ""
./test_solve2.sh
echo ""
./test_solve3.sh
echo ""
./test_solve4.sh
echo ""
./test_solve5.sh
echo ""
./test_solve6.sh
```

Output after running test_all.sh

```
$ ./test_all.sh
Expand test case:
Expansion for '(2x+4)**2' is 4x**2+16x+16

Expand test case:
Expansion for '(2x-3)**3' is 8x**3-36x**2+54x-27
8x**3-36x**2+54x-27

Expand test case:
Expansion for '(x+2)**2' is x**2+4x+4
x**2+4x+4

Expand test case:
Expansion for '(x-1)**2' is x**2-2x+1
```

```
x**2-2x+1

Parse Poly test case:
x**3-3x**2+5x-1

Poly 'x**3-3x**2+5x-1'
AddTerm test case:
Add term '4x' to poly
x**3-3x**2+9x-1

Poly 'x**3-3x**2+5x-1' - Find Coeff test case:
Find coefficient of term 'x**2'
-3

Poly 'x**3-3x**2+5x-1' - remove term test case:
Remove 'x**2' term from poly
x**3+5x-1

Divide test case:
Solution for '2x=4' is 2

Execute test case:
x

'For-loop' test case:
0
1
2

'If' test case:
true

Keep symbol on left-side:
(move non-symbol terms to right-side)
Equation '2x+3=4x+6'
Left-side: 2x
Right-side: 4x+3

Move symbol to left-side:
Equation '5x+3=2x+4'
Left-side: 3x+3
Right-side: 4

Print test case:
Hello

Solve test case 1:
Solution for 'x+3=4' is 1

Solve test case 2:
Solution for '0=x-4' is 4

Solve test case 3:
```

```
Solution for '3x-11=x+1' is 6

Solve test case 4:
Solution for '3x=9' is 3

Solve test case 5:
Solution for '7-y=5' is 2

Solve test case 6:
Solution for '2y=10' is 5
```

8 Lessons Learned

Writing a compiler was a new and very valuable experience as we had not done this in any of the earlier courses taken. This was our first experience with OCAML as well.

We learned about the components in a compiler and also got a chance to implement a small compiler end-to-end on our own. Resolving all the shift-reduce conflicts during the development of MEL compiler further enhanced the understanding and importance of a grammar, its regular expressions, building automata and having unambiguous grammar for a language . Understanding Micro C compiler is a good way to start on this compiler development project. Moreover HW1 assignment was very helpful in creating our basic understanding of compiler components and OCAML language.

We used GIT for source control and eclipse as IDE for development, which helped in individual development and merging the code at regular intervals.

We used Eclipse as our IDE which helped with the OCaml syntax and initial build. The Makefile was generated from the eclipse-generated files using ocamldep, and subsequent builds to native binary were during using a build script. Test scripts did a clean re-build and execution of all configured tests, which was a great way to verify that new changes had not broken existing functionality.

We have tried to incorporate all the features given in the proposal and LRM. Definitely it would have been better if we had started early, but we spend around a month in coming to a final proposal and LRM. So the key advice to future students for this course, is the same as many others have already stated, which is start early in the semester on the project.

9 Appendix

9.1 Compiler Components

ast.mli

```
type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater | Geq | And |
Or

type variable =
                IntVar of string
         | ArrayVar of string * int
    | SymAssign of string * string
         | EqAssign of string * string
         | PolyAssign of string * string
         | TermVar of string

type fparam =
                Int of string
         | Array of string
         | Sym of string
         | Eq of string
         | Poly of string
         | Term of string

type varcall  =
                IntVarCall of string
         | ArrayVarCall of string * expr
         | SymVarCall of string
         | EqVarCall of string
         | PolyVarCall of string
         | TermVarCall of string
    | DeRefId of varcall * string
    | DeRefArray of varcall * string
and expr =
      Literal of int
    | StringLiteral of string
    | Var of varcall
    | Binop of expr * op * expr
    | Assign of varcall * expr
    | ExprAssign of expr * expr
    | Call of string * expr list
    | LocalCall of expr * expr list
    | Noexpr

type functiontype =
        VoidRet
    | IntRet
    | ArrayRet

type stmt =
    Block of stmt list
```

```
  | Expr of expr
  | Return of expr
  | If of expr * stmt * stmt
  | For of expr * expr * expr * stmt
  | While of expr * stmt
      | Print of expr

type functiondecl = {
             ftype : functiontype;
    fname : string;
    formals : fparam list;
    locals : variable list;
    body : stmt list;
  }

type program = functiondecl list
```

parser.mly

```
%{ open Ast %}

%token SEMI LPAREN RPAREN LBRACE RBRACE LBRACKET RBRACKET COMMA DOT COLON
%token PLUS MINUS TIMES DIVIDE MOD POWER ASSIGN
%token EQ NEQ LT LEQ GT GEQ
%token AND OR
%token RETURN IF ELSE FOR WHILE INT VOID ARRAY SYM EQN POLY TERM FUNC PRINT
%token <int> LITERAL
%token <string> STRINGLITERAL
%token <string> ID
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left DOT
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE MOD POWER
%left SQRT SUM AVG ABS LOG

%start program
%type <Ast.program> program

%%

program:
   /* nothing */ { [] }
 | program functiondecl { $2 :: $1 }

functiondecl:
   functype ID COLON formals_opt LBRACE variabledecl_list stmt_list RBRACE
     { { ftype = $1;
```

```
                  fname = $2;
        formals = $4;
        locals = List.rev $6;
        body = List.rev $7 } }

functype:
             FUNC VOID  { VoidRet }
        | FUNC INT { IntRet }
        | FUNC ARRAY { ArrayRet }

formals_opt:
    /* nothing */ { [] }
  | formal_list   { List.rev $1 }

formaldecl:
             INT ID                { Int($2) }
        | ARRAY ID             { Array($2) }
        | SYM ID               { Sym($2) }
        | EQN ID               { Eq($2) }
        | POLY ID              { Poly($2) }
        | TERM ID              { Term($2) }

formal_list:
    formaldecl                         { [$1] }
  | formal_list COMMA formaldecl   { $3 :: $1 }

variabledecl_list:
    /* nothing */    { [] }
  | variabledecl_list variabledecl { $2 :: $1 }

variabledecl:
    INT ID SEMI { IntVar($2) }
  | ARRAY ID LPAREN LITERAL RPAREN SEMI { ArrayVar($2, $4) }
  | SYM ID ASSIGN STRINGLITERAL SEMI { SymAssign($2, $4) }
  | EQN ID ASSIGN STRINGLITERAL SEMI { EqAssign($2, $4) }
  | POLY ID ASSIGN STRINGLITERAL SEMI { PolyAssign($2, $4) }
  | TERM ID SEMI { TermVar($2) }

variablecall:
    ID                                 { IntVarCall($1) }
  | ID LPAREN expr RPAREN              { ArrayVarCall($1, $3) }
  | variablecall DOT ID                { DeRefId($1, $3) }
  | variablecall LBRACKET ID RBRACKET  { DeRefArray($1, $3) }

stmt_list:
    /* nothing */  { [] }
  | stmt_list stmt { $2 :: $1 }

stmt:
    expr SEMI { Expr($1) }
  | RETURN expr SEMI { Return($2) }
  | LBRACE stmt_list RBRACE { Block(List.rev $2) }
  | IF LBRACKET expr_if RBRACKET stmt %prec NOELSE { If($3, $5, Block([])) }
```

```
    | IF LBRACKET expr_if RBRACKET stmt ELSE stmt    { If($3, $5, $7) }
    | FOR LBRACKET expr_opt SEMI expr_opt SEMI expr_opt RBRACKET stmt
       { For($3, $5, $7, $9) }
    | WHILE LBRACKET expr RBRACKET stmt { While($3, $5) }
    | PRINT expr SEMI { Print($2) }

expr_opt:
    /* nothing */ { Noexpr }
    | expr          { $1 }

expr_if:
    expr          { $1 }
    | expr AND expr { Binop($1, And,   $3) }
    | expr OR  expr { Binop($1, Or,   $3) }

expr:
    LITERAL           { Literal($1) }
    | STRINGLITERAL     { StringLiteral($1) }
    | expr PLUS   expr { Binop($1, Add,   $3) }
    | expr MINUS  expr { Binop($1, Sub,   $3) }
    | expr TIMES  expr { Binop($1, Mult,  $3) }
    | expr DIVIDE expr { Binop($1, Div,   $3) }
    | expr EQ     expr { Binop($1, Equal, $3) }
    | expr NEQ    expr { Binop($1, Neq,   $3) }
    | expr LT     expr { Binop($1, Less,  $3) }
    | expr LEQ    expr { Binop($1, Leq,   $3) }
    | expr GT     expr { Binop($1, Greater,  $3) }
    | expr GEQ    expr { Binop($1, Geq,   $3) }
    | variablecall     { Var($1) }
    | expr ASSIGN expr { ExprAssign($1, $3) }
    | LBRACKET expr COLON actuals_opt RBRACKET { LocalCall($2, $4) }
    | LBRACKET expr RBRACKET { $2 }

actuals_opt:
    /* nothing */ { [] }
    | actuals_list  { List.rev $1 }

actuals_list:
    expr                       { [$1] }
    | actuals_list COMMA expr { $3 :: $1 }
```

Scanner.mll

```
{ open Parser }

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"      { comment lexbuf }           (* Comments *)
| '+'       { PLUS }
| '-'       { MINUS }
| '*'       { TIMES }
| '/'       { DIVIDE }
| '('       { LPAREN }
| ')'      { RPAREN }
| '{'      { LBRACE }
| '}'      { RBRACE }
| '['      { LBRACKET }
| ']'      { RBRACKET }
| ';'      { SEMI }
| ':'      { COLON }
| ','      { COMMA }
| '.'      { DOT }
| '='      { ASSIGN }
| "=="     { EQ }
| "!="     { NEQ }
| '<'      { LT }
| "<="     { LEQ }
| ">"      { GT }
| ">="     { GEQ }
| "and"    { AND }
| "or"     { OR }
| "func"   { FUNC }
| "if"     { IF }
| "else"   { ELSE }
| "for"    { FOR }
| "while"  { WHILE }
| "return" { RETURN }
| "int"    { INT }
| "Sym"    { SYM }
| "Eq"     { EQN }
| "Poly"   { POLY }
| "Term"   { TERM }
| "void"   { VOID }
| "array"  { ARRAY }
| "print"  { PRINT }
| ['-']*['0'-'9']+ as lxm { LITERAL(int_of_string lxm) }
| '\"'[^'"']*'\"' as lxm
            { STRINGLITERAL(String.sub lxm 1 ((String.length lxm) - 2)) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

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

compile.ml

```
open Ast

let string_of_variable = function
             ArrayVar(s, i) -> "int[] " ^ s ^ "= new int[" ^ string_of_int i ^ "];\n"
        | IntVar(s) -> "int " ^ s ^ ";\n"
        | SymAssign(s, t) -> "Sym " ^ s ^ "= new Sym(\"" ^ t ^ "\");\n"
        | EqAssign(s, t) -> "Eq " ^ s ^ "= new Eq(\"" ^ t ^ "\");\n"
        | PolyAssign(s, t) -> "Poly " ^ s ^ "= new Poly(\"" ^ t ^"\");\n"
        | TermVar(s) -> "Term " ^ s ^ ";\n"

let rec string_of_variablecall = function
          IntVarCall(s) -> s
        | SymVarCall(s) -> s
        | EqVarCall(s) -> s
        | PolyVarCall(s) -> s
        | TermVarCall(s) -> s
        | ArrayVarCall(s, e) -> s ^ "[" ^ string_of_expr e ^ "]"
        | DeRefId(s, a) -> string_of_variablecall s ^ "." ^ a
        | DeRefArray(s, a) -> string_of_variablecall s ^ "[" ^ a ^ "]"
and string_of_expr = function
    Literal(l) -> string_of_int l
  | StringLiteral(l) -> "\"" ^ l ^ "\""
        | Var(s) -> string_of_variablecall s
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^
      (match o with
       Add -> "+" | Sub -> "-" | Mult -> "*" | Div -> "/"
      | Equal -> "==" | Neq -> "!=" | And -> "&&" | Or -> "||"
      | Less -> "<" | Leq -> "<=" | Greater -> ">" | Geq -> ">=") ^ " " ^
      string_of_expr e2
  | Assign(v, e) -> string_of_variablecall v ^ " = " ^ string_of_expr e
  | ExprAssign(e1, e2) -> string_of_expr e1 ^ " = " ^ string_of_expr e2
  | Call(f, el) ->
              "MEL." ^ f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^
")"
  | LocalCall(e, el) -> string_of_expr e ^ "(" ^ String.concat ", " (List.map
string_of_expr el) ^ ")"
  | Noexpr -> ""

let rec string_of_stmt = function
    Block(stmts) ->
      "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
  | Expr(expr) -> string_of_expr expr ^ ";\n";
  | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n";
  | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s
  | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^
      string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
  | For(e1, e2, e3, s) ->
      "for (int " ^ string_of_expr e1  ^ " ; " ^ string_of_expr e2 ^ " ; " ^
      string_of_expr e3  ^ ") " ^ string_of_stmt s
  | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt s
        | Print(e) -> "System.out.println(" ^ string_of_expr e ^ ");\n"

let string_of_fparam = function
    Int(id) -> "int " ^ id
  | Sym(id) -> "Sym " ^ id
  | Eq(id) -> "Eq " ^ id
  | Poly(id) -> "Poly " ^ id
  | Term(id) -> "Term " ^ id
  | Array(id) -> "int[] "^ id
```

```
let string_of_functiontype = function
             VoidRet -> "void"
        | IntRet -> "int"
        | ArrayRet -> "int[]"

let string_of_fdecl fdecl =
  "public static " ^ string_of_functiontype fdecl.ftype ^ " " ^
       fdecl.fname ^ "(" ^ String.concat ", " (List.map string_of_fparam
fdecl.formals) ^ ")\n{\n" ^
  String.concat "" (List.map string_of_variable fdecl.locals) ^
  String.concat "" (List.map string_of_stmt fdecl.body) ^
  "}\n"

let string_of_program funcs =
       "public class MEL { \n" ^
  String.concat "\n" (List.map string_of_fdecl funcs) ^ "\n" ^
       "public static void main(String args[]){"^
       "MEL.evaluate();"^
       "}"^
       "} \n"
```

mel.ml

```
let _ =
  let lexbuf = Lexing.from_channel stdin in
  let program = Parser.program Scanner.token lexbuf in
       let listing = Compile.string_of_program program in print_string listing
```

9.2 MELLib (Java Library)

Eq.java

```
public class Eq {

       public Poly lhs = null;
       public Poly rhs = null;

       public Eq(String equation) {
              String[] polys = equation.split("=");
              lhs = new Poly(polys[0]);
              rhs = new Poly(polys[1]);
       }

       public static Eq newEq(String x) {
              return new Eq(x);
       }

}
```

Poly.java

```
import java.util.regex.Pattern;


import org.apache.commons.math3.util.ArithmeticUtils;

public class Poly {

	private static Pattern exponentPattern = Pattern.compile("\\(.*\\).*");
	//single variable polynomial of max 10 terms,
	//optionally an exponent for the entire (linear) polynomial
	public Term[] terms = new Term[10];
	public Integer[] coeffs = new Integer[10];
	public int exp = 1;

	public Poly() {
	}

	public Poly(String poly) {
		this();
		if(exponentPattern.matcher(poly).matches()) {
			String[] polyParts = poly.split("\\*\\*");
			exp = Integer.valueOf(polyParts[1]);
			poly = polyParts[0].replaceAll("\\(", "").replaceAll("\\)", "");
		}
		String[] curTerms = poly.replaceAll("\\-", "\\+\\-").split("\\+");
		for(String curTerm : curTerms) {
			addTerm(curTerm);
		}
	}

	public static Poly newPoly() {
		return new Poly();
	}

	public static Poly newPoly(String poly) {
		return new Poly(poly);
	}

	public void addTerm(String pattern) {
		if(pattern == null || pattern.isEmpty())
			return;
		if(pattern.matches(".*[a-zA-Z].*")) {
			int expIdx = pattern.indexOf("**"), coeff = 1;
			try {
				String sCoeff = pattern.split("[a-zA-Z]")[0];
				if(sCoeff.contains("+") || sCoeff.contains("-"))
					sCoeff = sCoeff.replaceAll("\\-", "\\+\\-
").split("\\+")[1];
				coeff = "-".equals(sCoeff)?-1:Integer.valueOf(sCoeff);
			} catch(Exception e){/*swallow*/}
```

```
				int exp = 1;
				if(expIdx != -1){
					exp =
Integer.valueOf(pattern.substring(pattern.indexOf("**")+2));
				}

				if (coeffs[exp]==null) coeffs[exp] = 0;
				coeffs[exp] = coeffs[exp]+coeff;
				terms[exp] = coeff!=0?new Term(new Sym("x")):null;
			} else {
				if (coeffs[0]==null) coeffs[0] = 0;
				String sCoeff = pattern.split("[a-zA-Z]")[0];
				if(sCoeff.contains("+") || sCoeff.contains("-"))
					sCoeff = sCoeff.replaceAll("\\-", "\\+\\-
").split("\\+")[1];
				coeffs[0] = coeffs[0]+("-".equals(sCoeff)?-
1:Integer.valueOf(sCoeff));
				terms[0] = new Term();
			}
		}

		public void removeTerm(String pattern) {
			if(pattern == null || pattern.isEmpty() || !pattern.matches(".*[a-zA-
Z].*"))
				return;
			int expIdx = pattern.indexOf("**");
			int exp = 1;
			if(expIdx != -1) {
				exp =
Integer.valueOf(pattern.substring(pattern.indexOf("**")+2));
			}
			terms[exp] = null;
			coeffs[exp] = null;
		}

		public int findCoeff(String pattern) {
			Poly tempPoly = new Poly(pattern);
			for(int i=0; i<tempPoly.terms.length; i++) {
				if(tempPoly.terms[i] != null) {
					if (terms[i].hasSym(tempPoly.terms[i].syms.get(0)))
						return coeffs[i];
				}
			}
			return 0;
		}

		public int binomialCoefficient(int n, int k){
			return (int)ArithmeticUtils.binomialCoefficient(n, k);
		}

		public int pow(int k, int e) {
			return ArithmeticUtils.pow(k, e);
		}
```

```
	public String toString() {
		StringBuilder polyString = new StringBuilder();
		for(int i=coeffs.length-1; i>=0; i--) {
			if(coeffs[i]==null) continue;
			polyString.append(coeffs[i]>0&&polyString.length()>0?"+":"");

	polyString.append(coeffs[i]!=1||terms[i].syms.size()==0?coeffs[i]:"");

	polyString.append(terms[i].syms.size()>0?terms[i].syms.get(0).symbol:"");
			polyString.append(i>1?"**"+i:"");
		}
		if(exp != 1) {
			StringBuilder fullPolyString = new
StringBuilder("(").append(polyString).append(")").append("**").append(exp);
			return fullPolyString.toString();
		}
		return polyString.toString();
	}

	public static void main(String[] args) {
		Poly p = new Poly("");
		System.out.println(p);
		p = new Poly("x**3-3x**2+5x-1");
		System.out.println(p);
		p.addTerm("+4x");
		System.out.println(p);
		System.out.println(p.findCoeff("x**2"));
		p.removeTerm("x**2");
		System.out.println(p);
	}

}
```

Sym.java

```
public class Sym {

	String symbol = null;

	public Sym(String x) {
		symbol = x;
	}

	@Override
	public boolean equals(Object s) {
		if(s == this || s instanceof Sym)
			return ((Sym)s).symbol.equals(symbol);
		return false;
	}

	@Override
	public int hashCode(){
		return symbol!=null?symbol.hashCode():-1;
	}

}
```

Term.java

```
import java.util.ArrayList;
import java.util.List;

public class Term {

	//Single variable equation only for now
	//(all terms use the same variable)
	public List<Sym> syms = new ArrayList<Sym>();

	public Term() {
		this.syms = new ArrayList<Sym>();
	}

	public static Term newTerm() {
		return new Term();
	}

	public Term(String symbol) {
		this(new Sym(symbol));
	}

	public Term(Sym sym) {
		this();
		syms.add(sym);
	}

	public boolean hasSym(Sym x) {
		return syms.contains(x);
	}

	public void addSym(Sym x) {
		if (syms == null)
			syms = new ArrayList<Sym>();
		if(!syms.contains(x))
			syms.add(x);
	}

	public void removeSym(Sym x) {
		if (syms == null)
			syms = new ArrayList<Sym>();
		else
			syms.remove(x);
	}

}
```