Polynomial Calculator Programming Language (PCPL)

Donghui Xu (dx2116)
COMS 4115 Project
2010 Fall Semester
Professor: Stephen Edwards

1. Introduction

PCPL is a simple programming langue with built-in support for single variable polynomial functions. It supports operations, such as addition, subtraction, multiplication and division over polynomial expressions. It also provides tools that allow programmer to implement algorithms to solve various polynomial problems, for example to compute the derivative and integral of a given polynomial.

1.1. Overview of the language

The syntax of the PCPL is similar to those in Java and C++, and it is developed base on Microc.

The language will have five data types, namely, int, float, poly, string and boolean. A poly data type is represented by a list of floating point numbers representing the coefficients of the given polynomial . The first element in the floating point number list represents the constant term of the polynomial, the second element represents the degree 1 coefficient and higher degree coefficients follow in sequence. For example 1 + 3x^2 is represented as {1.0, 0.0, 3.0} in PCPL.

Operations (+, -, *) can be applied over two polynomials. Division (/) can be applied over a polynomial (as the dividend) and a floating point number or an integral value (as the divisor).

A small set of built-in functions and operators along with some basic control structures are implemented to allow programmer to solve more complex polynomial problems.

1.2. Polynomial operations:

- Poly initializer - {float-list}

  ```
  poly p ;
   p = {1.0,  0.0,  3.0};
  ```

  p is initialized to {1.0, 0.0, 3.0} which represents 1 + 3 x^2.

- Poly constructor – poly{int-literal}
  poly coefficient accessor – identifier[int-literal]

  ```
  poly p;
   p = poly{1};   /* 1 represents the degree of a polynomial*/
  p[0] = 2.0;
  p[1] = 3.0;
  ```

  poly{1} creates {0.0, 0.0}; it is then assigned appropriate coefficients via the poly coefficient accessors, p[0] and p[1].

- +, -, *, /
  Addition, subtraction and multiplication can be applied to polynomial as expected.
  Division can only be applied over a polynomial (as the dividend) and a floating point number or an integral value(as the divisor).

- << and >>
  PCPL implements a very convenient way to alter polynomials using shift-left and shift-right operators .The

following shows a polynomial before and after << and >>.

```
p = {1.0, 2.0, 3.0};        /* p : 1 + 2 x + 3 x^2 */
p1 = p >> 1;                /* p1 = {0.0, 1.0, 2.0, 3.0} :  x + 2 x^2 + 3 x^3 */
p2 = p << 1;                /* p2 =  {2.0, 3.0} : 2 + 3 x */
```

- deg(poly)

```
poly = p;
p = {1.0, 2.0};
println( deg(p));
```

deg function returns the degree of a polynomial. Print() and println() are implemented to facilitate the output.

2. Language Tutorial

2.1. Compiling and run PCPL programs

The PCPL language is a simple programming langue with built-in support for polynomial operations. The PCPL compiler, pcpl.exe, translates PCPL programs xxx.pcpl into Java source that can then be compiled and run along-with a Java implementation of polynomials. In order to compile and run PCPL programs, you will need to have both Java compiler javac and the Java Runtime Environment installed. The following command sequence shows how to compile and run a PCPL program Test.pcpl.

```
user> pcpl.exe -c Test.pcpl > Test.java
user> javac –d . Test.java PolynomialFunction.java
user> java –cp . poly.Test
```

The first command invokes the PCPL compiler on the input PCPL program, Test.pcpl. The PCPL compiler translates the program into Java source and wraps it in a class called poly.Test that is stored in Test.java file. The PCPL compiler assumes the source file to have .pcpl extension. The Test.java is then compiled along with the PolynomialFunction.java that contains the polynomial implementation. The resulting class poly.Test is then run using the java command.

The -c option supplied to pcpl.exe indicates that the program should be analyzed for any semantic errors and then compiled to Java sources. The pcpl.exe has the following command-line options.

- -a: print the source program and quit.
- -t: perform semantic analysis of the program and quit.
- -c: perform semantic analysis and translate to Java code.

2.2. To compile the PCPL compiler and run the test suite.

Open a window command prompt
cd to "pcpl" direcotry

(make sure the ocamlc, javac and java on your path)
to build the pcpl compiler: run make.bat

to run the tests: run tests.bat

(this will save the output for every test_XXX.pcpl into test_XXX.run file under tests directory)
to clean up: run clean.bat

Note:
If you have 64bit Java in your path, make sure use a syswow64 cmd window.

2.3. Structure of PCPL program and Example code

A PCPL program consists of primarily two components: an optional global variable declarations section at the beginning of the program and a function definition(s) section including a main function. As in Java the main function is required as the entry point of a program.

Each function includes local variable declarations and a list of statements. All local variables must be declared prior to any statements in the function.

Example 1:
The following code demonstrates the global function declaration, function definition and recursion in PCPL. It shows how to compute Fibonacci numbers using PCPL

```
/*global variable declaration*/
int g1;

/*function declaration*/
int main() {
    int num;
    num = 5;
    g1 = fib(5);

  println(g1);
  return 0;
}

int fib(int n) {
  if (n < 2) {
    return 1;
  }
  return fib(n-1) + fib(n-2);
}
```

Unlike in C PCPL does not require *fib* to be declared prior to its use in *main* function.

Example 2:
The following example shows how to create and manipulate polynomials.

```
int main(){
  int i;
  poly p1;
  poly p2;
  poly p3;
```

```
  p1 = {1.0,2.0,3.0};
  p2 = {2.0,1.0,-4.0};

  p3 = p1 + p2;
  p3 = p1 – p2;
  p3 = p3 * p1;
  p3 = p3 / 2.0;
  p1 = deepcopy(p3);

  p3 = p3 << 1;
  p3 = p3 >> 1;

  println(p1);
  println(p3);

  return 0;
}

poly deepcopy(poly p) {
  int i;
  poly result;

  result = poly{deg(p)};
  for (i=0; i <= deg(p); i=i+1) {
    result[i] = p[i];
  }

  return result;
}
```

The deepcopy function shows how to create a deep copy of a polynomial p. The built-in deg function gets the degree of p; A new polynomial is created via the polynomial constructor poly{int-expr} with degree equals to that of p and coefficients equals to 0. It then uses the polynomial coefficient accessor, p[int-expr], to populate the coefficients of the new polynomial with the same values as those of p.

Example 3

The following code shows how to compute the derivative of a polynomial with the given coefficients. .

test_poly_derivative.pcpl

```
int main(){

 poly result;
 poly p;
 p = {-1.0, 2.0, -6.0, 2.0};      /* -1.0 + 2.0x -6.0x^2 + 2.0x^3 */

 if (deg(p) == 0)
 {
   result = {0.0};
 }
 else
 {
   result = differentiate(p);
 }
```

```
 print("derivative of ");
 print(p);
 print(" = ");
 println(result);

 return 0;
}

/* precondition: deg(p) >=1

p = -1.0 + 2.0x -6.0x^2 + 2.0x^3
p' = 2.0 - 12.0x + 6.0x^2
*/

poly differentiate(poly p)
{
  int i;
  poly result;

  result = p;

  for (i = deg(p); i >=0 ; i=i-1) {
    result[i] = result[i] *i;
  }

  return result << 1;
}
```

Output

```
derivative of -1.0 + 2.0 x - 6.0 x^2 + 2.0 x^3 = 2.0 - 12.0 x + 6.0 x^2
```

To avoid the problem of assigning a coefficient to a wrong degree element, we applied << to the result, rather than manipulating the index in the for loop.

3. Language Reference Manual

3.1. Lexical convention

3.1.1. Tokens

There are five types of tokens in PCPL, namely identifiers, keywords, constants, operators and other separators.

3.1.2. Comments

A comment starts with "/*" and ends with "*/". The language does not support nested comments. Comments as well as white spaces including blanks, tabs and new lines are ignored.

3.1.3. Identifiers

An identifier consists of alphabetic characters and digits. An identifier must start with an alphabetic character and followed by zero or more alphabetic characters or digits. Identifiers are case sensitive. Variable names and function names are identifiers.

3.1.4. Keywords

Keywords are reserved identifiers. The following is the list of key words in this language:

if
else
for
while
return
int
float
poly
true
flase
boolean
string

3.1.5. Constants and datatypes

PCPL defines following types of constants, integers, floating point numbers, polynomials, string literals and booleans.

- Integers
  An integer is a sequence of one or more digits (0-9), e.g. 123.
- Floating point numbers
  since we rarely use the scientific notation, we decide that a floating point number will consist of an integer part, a decimal point (.) and a fraction part, e.g. 0.0 and 12.5.
- Polynomials
  A polynomial is represented by a list of floating point numbers. The list represents the coefficients of terms in the polynomial in increasing order of exponent from left to right.
  For example $2 - 3x + 5x^2$ is represented as {2.0, -3.0, 5.0} and 3x + 5x^3 is represented as {0.0, 3.0, 0.0, 5.0}.
- string literals
  A string literal is a sequence of characters enclosed in double quotes e.g. " hello" .
- boolean literals
  true and false are the only boolean literals.

3.2. Variable declarations

A variable needs to be declare before it can be assigned a value. Variables can be declared globally and locally. Global variables must be declared at the beginning of the program. Local variable must be declared prior to any statement in the function.
Variable declaration consists of a data type followed by a unique identifier. A few examples of variable declarations are given below.

```
int i;
float j;
poly p:
boolean b;
string s;
```

3.3. Expressions

PCPL supports the following kinds of expressions shown in decreasing order of precedence.

3.3.1. Primary expression

Primary expressions are identifiers, constants, parenthesized expressions and function calls grouped left-to-right. Primary expressions appear on the right hand side of the statement.

identifiers : p, deg
constants : 3, 1.2, true, "hi", {1.0, 2.0}
parenthesized expressions : (2+5)
function calls : max(2,3)

3.3.2. Unary operator

Unary operator (-) negates the resulting value of an expression group right-to-left. The expression could be an integer or a floating point number or a polynomial.

3.3.3. Binary operator

- Multiplicative operators
  The multiplicative operators are * and / grouped left-to-right.
  multiplication: expr * expr
  division: expr / expr

operator	Left operand	Right operand
*	int | float	int | float
	poly	poly
/	int | float	int | float
	Poly	int | float

Table shows legal applications of Multiplicative operators on operands of different types.

- Additive operators
  The additive operators are + and – group left to right.

  addition : expr + expr
  subtraction : expr – expr

  Both operations are grouped left- to- right. Additive operators have lower precedence than multiplicative operators.

operator	Left operand	Right operand
+/-	int | float	int | float
	poly	poly

Table shows legal applications of additive operators on operands of different types.

- PCPL implements a very convenient way to alter polynomials using shift-left and shift-right operators <<, >>. The following shows a polynomial before and after << and <<.

  p = {1.0,2.0,3.0};

```
/* p = 1 + 2 x + 3 x^2 */
p1 = p >> 1;
/* p1 :  x + 2 x^2 + 3 x^3 */
p2 = p << 1;
/* p2 = 2 + 3 x */
```

operator	Left operand	Right operand
<<, >>	poly	poly

Table shows legal applications of shift operators on operands of different types.

- Relational operators
  Relational operators are >, >=, <,<= group left-to-right.

  greater than : expr > expr
  greater or equal to: expr >= expr
  less than : expr < expr
  less than or equal to : expr <= expr

  The operation return false if specified relation is false, returns ture if the result is true, int and float can be one either side of the operator. The operations will be evaluated according to conventional mathematical formula. Relational operators have lower precedence than additive operators.

operator	Left operand	Right operand
>, >=, <, <=	int | float	int | float
	boolean	boolean

Table shows legal applications of Relational operators on operands of different types.

- Equality Operators
  Equality Operators are ==, ! =.

  equals to: expr == expr
  not equals to: expr != expr

  The operation return false if the specified relation is false, returns ture if the result is true. Equality operators have the same preference as relational operators.

operator	Left operand	Right operand
==, ! =.	int | float	int | float
	poly	poly
	boolean	boolean

Table shows legal applications of types Equality Operators on operands of different types

- Assignment operator
  Assignment operator is = group right-to-left.
  lvalue = expr
  lvalue is and identifier or a poly element access i.e. p[int_expr]. Assignment of polynormail is a refreence copy not a deep copy. In P1 = p2 , p1 and p2 point to the same polynomial.

In an assignment operation, the left operand is a lvalue and it gets assigned the value of the expression on the right hand side. Both operands need to be the same type. Assignment operator has lower precedence than relational operators and equality operators.

3.5 Statements

Unless indicated statements are executed in sequence.

3.5.1 Expression statements

Most statements are expression statements. They have the form
expression;
Most expression statements are assignments or function calls.

3.5.2 Conditional statement

Conditional statements are if or if-else statements. They have the forms

```
if (expression){
statement list 1}
```

```
if (expression){
  statement list1}
else{
statement list 2
}
```

The expression should be a boolean type. If expression is evaluated to true, the statement list 1 is executed; if false the statement list 2 is executed.

3.5.3 Iterative statement

Iterative statement has the form

```
while (expression)
{
  statement list
}
```

The expression should be a Boolean type.
If the expression is evaluated to true the statement list is executed. After the execution of the statement list the expression will be evaluated again and the statement list will be executed as long as the expression remains true.

3.5.4 For statement

The for statement has the form

```
for ( expr1; expr2; expr3)
{
 statement list
}
```

The for statement is equivalent to

```
expr1;
while (expr2)
{
statement list
```

```
expr3
}
```

3.6 Functions

A function contains a sequence of statements. Functions have the form

```
return_type  function_name ( parameter list)
{
  statement list;
}
```

The type of the expression in return statement must be the same as the return_type in the function declaration. All path returning from the function must return the same type as the declared return type.

3.7 Program

A program is a list of variable declarations followed by a list of function definitions one of which is a program entry point i.e. main. Within the main function other functions can be called. Nesting functions are not allowed. An example of a program could be

```
variable-declarations
 int main ()
{
  variable-declarations
   statements;
   return 0;
 }
function-declarations
```

4. Project Plan

4.1. Planning process

I planned to divide the deliverable (project proposal, LRM, final report) into smaller weekly or bi-weekly tasks. For example, for LRM I would divide tasks into designing AST, building the scanner, parser and writing a driver program to output the content of a simple program. The details of PCPL become clearer and more concrete over the time so I have to revise the plan and update the code as needed to reflect the change.

4.2 Programming style guide.

The following coding style was followed to improve readability of the code.

- The assignment operators are aligned when the lvalues have different lengths.

```
rtype    = $1;
fname    = $2;
formals  = $4;
locals   = List.rev $7;
body     = $8
```

- The actions in the parser are left-aligned when the productions have different lengths.

```
stmt:
    expr SEMI                { Expr($1) }
  | RETURN expr SEMI         { Return($2) }
```

```
| LBRACE stmt_list RBRACE    { Block($2) }
```

- Catch blocks are indented with two spaces relative to the corresponding try block.

```
let type_of_var global_map formal_map local_map s =
  try StringMap.find s global_map
  with Not_found ->
    try StringMap.find s formal_map
    with Not_found ->
      try StringMap.find s local_map
      with Not_found ->
        raise(Fail("undefined variable " ^ s ^ "."))
```

- The cases in match statements are aligned as shown below.

```
match dtype with
   Int -> "int"
|  Float -> "double"
|  Boolean -> "boolean"
|  Poly -> "PolynomialFunction"
|  String -> "String"
|  UnknownType -> "unknowntype"
```

- The block for each if statement is indented with two spaces.

```
if (t1==Poly && t2==Poly) then
  Poly
else if (t1==Int && t2==Int) then
  Int
else if (t1==Float && t2==Float) then
  Float
else if (t1==Float && t2==Int) then
  Float
else if (t1==Int && t2==Float) then
  Float
else
  raise(Fail("type mismatch"))
```

4.3 project timeline

5/25 – 6/8	• Research previous project • Revising the project proposal • Study algorithms on polynomial functions
6/9 – 6/28	Revising

	• LRM • Scanner • Parser
6/30 – 8/16	Work on • Java code to manipulate polynomial function • javacode.ml • typecheck.m • Testing suites • documentation

4.4 Development environment used (tools and languages)

4.4.1 Operating system

The programs are developed under windows7 operating system

4.4.2 OCaml

Ocaml programs are written in Textpad and compile using batch file under windows command shell.

4.4.3 Java

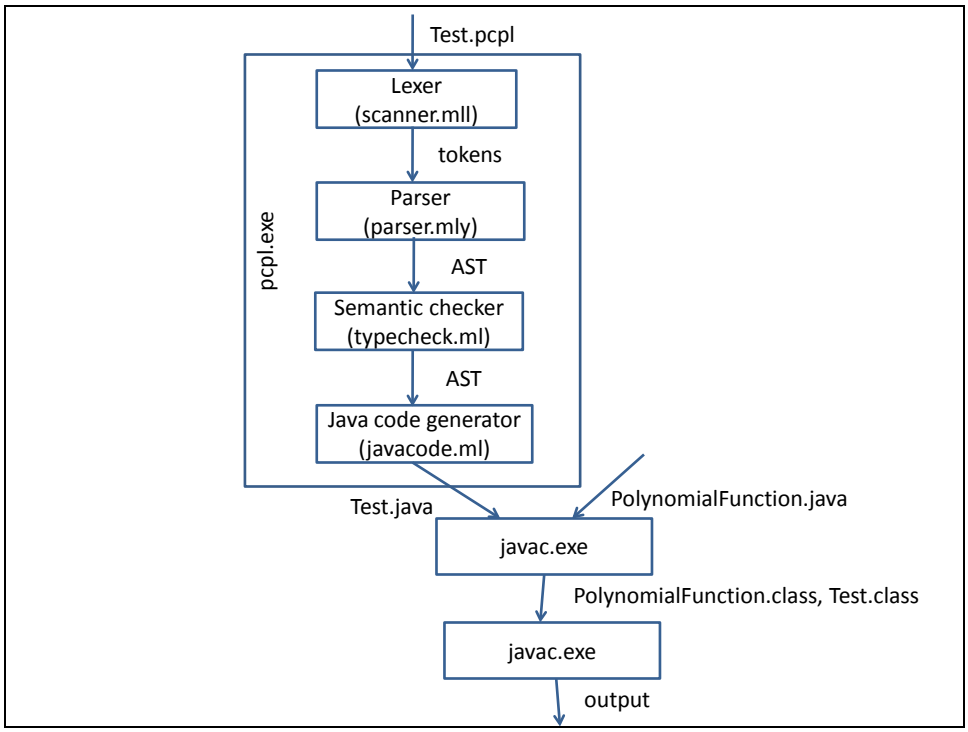
Java program, PolynormailFunction.java which manipulates polynomial operations is developed in Neatbean IDE.

The PolynormailFunction.java is modified from the Apache Software Foundation (ASF)'s Commons Math project. The Java document of the PolynormialFunction class can be found at * http://commons.apache.org/math/api-2.2/index.html and the original source code can be found at http://commons.apache.org/math/download_math.cgi

5. Architectural Design

The PCPL source code (test.pcpl) is fed to the PCPL compiler (pcpl.exe) that produces a java source file (test.java). Inside the PCPL compiler the lexser transforms the PCPL source code to tokens; the parser creates the AST from tokens; the semantic checker performs the type checking on the AST. If the AST passed the type checking, it is fed to the Java code generator that produces a java source file(test.java). Java compiler is then runs on test.java and PolynormailFunction.java to produce java bytecodes (PolynormialFunction.class and test.class) which is then interpreted by the java interpreter.

The following figure shows the overall architectural design.



6. Test Plan

6.1. Two examples on representative source language programs and the target language program

Source lanaguage 1: test_global1.pcpl

```
/*test global variable*/
int a;
int b;
int incab()
{
    a = a + 1;
    b = b + 1;
    return 0;
}
int main()
{
   a = 42;
   b = 21;
   println(a);
   println(b);
   incab();
   println(a);
   println(b);
   return 0;
}
```

Target language: test_global1.java

```
package poly;
public class test_global1
{
public static void main(String[] args) throws Exception
{
polyMain();
}

public static int b;
public static int a;

static int polyMain() throws Exception{
a = 42;
b = 21;
System.out.println(a);
System.out.println(b);
incab();
System.out.println(a);
System.out.println(b);
return 0;
}
static int incab() throws Exception
{
a = (a + 1);
b = (b + 1);
return 0;
}
}
```

Source porgam 2: test_poly_integral.pcpl

```
/*function : Returns the integral of the polynomial with the given
coefficients.*/
int main(){
   poly result;
   poly p;
   p = {-1.0, 2.0, -6.0, 2.0};    /* -1.0 + 2.0x -6.0x^2 + 2.0x^3 */
   result = integrate(p);
   print("integral of ");
   print(p);
   println(" = ");
   println(result);
   return 0;
}

/*
p = -1.0 + 2.0x -6.0x^2 + 2.0x^3
integral of p  = -x + x^2 - 2.0x^3 + 0.5x^4
*/
poly integrate(poly p)
{
  int i;
  poly result;
  result = copy(p);
  for (i = deg(p); i >=0 ; i=i-1) {
   result[i] = result[i] / (i+1);
   }
   return result >> 1;
}

poly copy(poly a)
{
  int i;
  poly b;
  b = poly{deg(a)};
  for (i=0; i <= deg(a); i=i+1) {
    b[i] = a[i];
  }
  return b;
}
```

Target code: test_poly_integral.java

```
package poly;
public class test_poly_integral
{
public static void main(String[] args) throws Exception
{
polyMain();
}

static PolynomialFunction copy(PolynomialFunction a) throws Exception
{
int i;
PolynomialFunction b;
b = new PolynomialFunction(a.degree());
for (i = 0 ; (i <= a.degree()) ; i = (i + 1)) {
b.coefficients[i] = a.coefficients[i];
}
return b;
}
static PolynomialFunction integrate(PolynomialFunction p) throws Exception
{
int i;
PolynomialFunction result;
result = copy(p);
for (i = p.degree() ; (i >= 0) ; i = (i - 1)) {
result.coefficients[i] = (result.coefficients[i] / (i + 1));
}
return result.shiftRight(1);
}

static int polyMain() throws Exception
{
PolynomialFunction result;
PolynomialFunction p;
p = new PolynomialFunction(new double[]{ -1., 2., -6., 2. });
result = integrate(p);
System.out.print("integral of ");
System.out.print(p);
System.out.println(" = ");
System.out.println(result);
return 0;
}
}
```

6.2. test suite

the test plan is to test all the components mentioned in the LRM.

6.2.1 test_hello.pcpl

```
/*Test print()*/

int main()
{
      print("Hello, World!");
      print(71);
      print(1.0);
        print({1.0, 2.0});

      return 0;
}
```

Output

```
Hello, World!711.01.0 + 2.0 x
```

6.2.2 test_global1.pcpl

```
/*test global variable*/

int a;
int b;

int incab()
{
  a = a + 1;

 b = b + 1;

  return 0;
}

int main()
{

 a = 42;
 b = 21;


 println(a);
 println(b);

 incab();


 println(a);
 println(b);

 return 0;
}
```

Output

```
42
21
43
22
```

6.2.3 Test_var_decl_assign.pcpl

```
/*Test global and local variable declaration and assignment*/

int i;
float f;
boolean b;
string s;
poly p;

int main()
{

      int j;
      float g;
      boolean d;
      string t;
      poly q;

      i= 1;
      println(i);
      j = i;
      println(j);


      f = 2.5;
      println(f);
      g = f;
      println(g);


      b = true;
      println(b);
      d = b;
      println(d);


      s = "Hello!";
      println(s);
      t = s;
      println(t);

      p = {1.0, 2.0, 3.0};
      println(p);
      q = p;
      println(q);
```

```
      return 0;
}
```

Output

```
1
1
2.5
2.5
true
true
Hello!
Hello!
1.0 + 2.0 x + 3.0 x^2
1.0 + 2.0 x + 3.0 x^2
```

6.2.4 test_assign.pcpl

```
/*test basic arithmetic functions +,-,*,/ over int and float*/

int main()
{
   int i;
   int j;
   float f;
   float g;

   i = 2;
   j = 10;
   println("integer arithmetic");
   print("2+10 = ");
   println(i+j);
   print("2 - 10 = ");
   println(i-j);
   print("2 * 10 = ");
   println(i*j);
   print("2 / 10 = ");
   println(i/j);

   f = 2.5;
   g = 10.0;

   println("float arithmetic:");
   print("2.5 + 10.0 = ");
   println(f+g);
   print("2.5 - 10.0 = ");
   println(f-g);
   print("2.5 * 10.0 = ");
   println(f*g);
   print("2.5 / 10.0 = " );
   println(f/g);  /*should be 0.25*/

   return 0;
}
```

Output

```
1
2
1.0
2.5
true
false
Hello!
Hello, World!
1.0 + 2.0 x + 3.0 x^2
4.0 + 5.0 x + 6.0 x^2
```

6.2.5 test_arith.pcpl

```
/*test basic arithmetic functions +,-,*,/ over int and float*/

int main()
{
   int i;
   int j;
   float f;
   float g;

   i = 2;
   j = 10;
   println("integer arithmetic");
   print("2+10 = ");
   println(i+j);
   print("2 - 10 = ");
   println(i-j);
   print("2 * 10 = ");
   println(i*j);
   print("2 / 10 = ");
   println(i/j);

   f = 2.5;
   g = 10.0;

   println("float arithmetic:");
   print("2.5 + 10.0 = ");
   println(f+g);
   print("2.5 - 10.0 = ");
   println(f-g);
   print("2.5 * 10.0 = ");
   println(f*g);
   print("2.5 / 10.0 = " );
   println(f/g);  /*should be 0.25*/

   return 0;
}
```

Output

```
integer arithmetic
2+10 = 12
2 - 10 = -8
2 * 10 = 20
2 / 10 = 0
float arithmetic:
2.5 + 10.0 = 12.5
2.5 - 10.0 = -7.5
2.5 * 10.0 = 25.0
2.5 / 10.0 = 0.25
```

6.2.6 test_arith_poly.pcpl

```
/*test arithmetic operations  + - * / << >> and deg() over polynomials*/

int main()
{
   poly p;
   poly q;

   p = {1.0, 2.0, 3.0};
   q = {1.0, 0.0, 3.0};

   println(p);
   println(q);

   println({2.0} + p);
   println( p + {2.0});
   println({2.0} - p);
   println( p - {2.0});
   println({2.0} * p);
   println(p *{2.0});
   println(p/2.0);

   println(p + q);
   println( p - q);
   println(p*q);

   print("deg(p) = ");
   println(deg(p));

   print(p);
   print(" >> 1 = ");
   println(p >> 1);
   print(p);
   print(" << 1 = ");
   println(p << 1);

   return 0;
}
```

Output

```
1.0 + 2.0 x + 3.0 x^2
1.0 + 3.0 x^2
3.0 + 2.0 x + 3.0 x^2
3.0 + 2.0 x + 3.0 x^2
1.0 - 2.0 x - 3.0 x^2
-1.0 + 2.0 x + 3.0 x^2
2.0 + 4.0 x + 6.0 x^2
2.0 + 4.0 x + 6.0 x^2
0.5 + x + 1.5 x^2
2.0 + 2.0 x + 6.0 x^2
2.0 x
1.0 + 2.0 x + 6.0 x^2 + 6.0 x^3 + 9.0 x^4
deg(p) = 2
1.0 + 2.0 x + 3.0 x^2 >> 1 = x + 2.0 x^2 + 3.0 x^3
1.0 + 2.0 x + 3.0 x^2 << 1 = 2.0 + 3.0 x
```

6.2.7 test_ops1.pcpl

```
/* test operators */

int main()
{

  println(1 + 2);
  println(1 - 2);
  println(1 * 2);
  println(100 / 2);
  println(99);

  println(1 == 2);
  println(1 == 1);
  println(99);

  println(1 != 2);
  println(1 != 1);
  println(99);

  println(1 < 2);
  println(2 < 1);
  println(99);

  println(1 <= 2);
  println(1 <= 1);
  println(2 <= 1);
  println(99);

  println(1 > 2);
  println(2 > 1);
  println(99);

  println(1 >= 2);
  println(1 >= 1);
  println(2 >= 1);
  println(99);
```

```
  println(1.5 + 2.0);
  println(1.5 - 2.0);
  println(1.5 * 2.0);
  println(100.0 / 2.0);
  println(99);

  println(1.5 == 2.0);
  println(1.5 == 1.5);
  println(99);

  println(1.5 != 2.0);
  println(1.5 != 1.5);
  println(99);

  println(1.5 < 2.0);
  println(2.0 < 1.5);
  println(99);

  println(1.5 <= 2.0);
  println(1.5 <= 1.5);
  println(2.0 <= 1.5);
  println(99);

  println(1.5 > 2.0);
  println(2.0 > 1.5);
  println(99);

  println(1.5 >= 2.0);
  println(1.5 >= 1.5);
  println(2.0 >= 1.5);

  return 0;
}
```

Output

```
3
-1
2
50
99
false
true
99
true
false
99
true
false
99
true
true
false
```

```
99
false
true
99
false
true
true
99
3.5
-0.5
3.0
50.0
99
false
true
99
true
false
99
true
false
99
true
true
false
99
false
true
99
false
true
true
```

6.2.7 test_poly_ops.pcpl

```
/* test polynomial operators */

int main()
{

  poly p;
  poly q;

  p = {1.0, 1.0};
  q = {2.0, 0.0, 2.0};

  println(p);
  println(p + q);
  println(p - q);
  println(p * q);
  println(p /2);

  println(p == q);
  println(p == p);
```

```
  println(p != q);
  println(p != p);

  println(deg(p));
  println(p >> 1);
  println(p << 1);

  println(-p);
  println(p[0]);
  println(p[1]);

  return 0;
}
```

Output

```
1.0 + x
3.0 + x + 2.0 x^2
-1.0 + x - 2.0 x^2
2.0 + 2.0 x + 2.0 x^2 + 2.0 x^3
0.5 + 0.5 x
false
true
true
false
1
x + x^2
1.0
-1.0 - x
1.0
1.0
```

6.2.8 test_if1.pcpl

```
/*test if statement*/

int main()
{
  if (true)
  {

    println(42);
  }

  println(17);
  return 0;
}
```

Output

```
42
17
```

6.2.9 test_if2.pcpl

```
/*test if-else statement*/

int main()

{

 if (true)
 {
   println(42);
 }
 else
 {
  println(8);

 }
 println(17);

 return 0;

}
```

Output

```
42
17
```

6.2.10 test_if3.pcpl

```
/* test if statement*/

int main()
{
  if (false)
  {
     println(42);
  }

  println(17);
  return 0;

}
```

Output

```
17
```

6.2.11 test_if4.pcpl

```
/*test if-else statement*/
```

```
int main()
{
 if (false)
 {
    println(42);
 }
 else
 {
  println(8);
 }

  println(17);
  return 0;
}
```

Output

```
8
17
```

6.2.12 test_for1.pcpl

```
/*test for loop*/

int main()
{
  int i;

  for (i = 0 ; i < 5 ; i = i + 1) {
      println(i);
  }

  println(42);
  return 0;
}
```

Output

```
0
1
2
3
4
42
```

6.2.13 test_while1.pcpl

```
/* test while loop*/

int main()
{
  int i;
  i = 5;
```

```
  while (i > 0) {
    println(i);
    i = i - 1;
  }
  println(42);

  return 0;
}
```

Output

```
5
4
3
2
1
42
```

6.2.14 test_func1.pcpl

```
/*test function*/

float add(float a, float b)
{
   return a + b;
}

int main()
{
  float a;
  a = add(39.0, 3.0);
  println(a);
  return 0;
}
```

Output

```
42.0
```

6.2.15 test_func2.pcpl

```
/*test function*/

int fun(int x, int y)
{
  println(x+y);
  return 0;
}

int main()
{
  int i;
  i = 1;
```

```
  fun(i = 2, i = i+1);
  println(i); /* should be 3 */
  return 0;
}
```

Output

```
5
3
```

6.2.16 test_gcd.pcpl

```
/*test gcd*/

int gcd(int a, int b)
{
  while (a != b) {
    if (a > b)
    {
       a = a - b;
    }
    else
    {

       b = b - a;
    }
  }

 return a;
}

int main()
{
   println(gcd(2,14));
   println(gcd(3,15));
   println(gcd(99,121));

   return 0;
}
```

Output

```
2
3
11
```

6.2.17 test_fib.pcpl

```
/*test recursion*/

int main()
{
```

```
  println(fib(0));
  println(fib(1));
  println(fib(2));

  println(fib(3));
  println(fib(4));
  println(fib(5));

  return 0;
}

int fib(int x)
{
  if (x < 2)
  {
     return 1;
  }
  else
  {
     return fib(x-1) + fib(x-2);
  }
}
```

Output

```
1
1
2
3
5
8
```

6.2.18 test_poly_eval.pcpl

```
/*test poly operations
function : Compute the value of the function for the given argument.*/

int main(){

 float result;
 poly p;

 p = {-1.0, 2.0, -6.0, 2.0}; /* -1.0 + 2.0x -6.0x^2 + 2.0x^3 */

 if (deg(p) > 0)
 {
    result = evaluate(p, 3.0);

    print("p = ");
    println(p);
    println("x = 3");
    println(result);
 }
```

```
 return 0;
}

/* using Horner's Method to evaluate the polynomial
@ precondition: deg(p) >=0
p = -1.0 + 2.0x -6.0x^2 + 2.0x^3
x = 3
result = 5

 x ¦   x³    x²    x¹    x°
 3 ¦   2    -6     2    -1
   ¦         6     0     6
   +------------------------
       2     0     2     5
*/

float evaluate(poly p, float x)
{
  float result;
  int i;
  int j;

  i = deg(p);

  result = p[i];
  for (j = i-1 ; j >=0; j=j-1) {
      result = result*x + p[j];
  }
  return result;

}
```

Output

```
p = -1.0 + 2.0 x - 6.0 x^2 + 2.0 x^3
x = 3
5.0
```

6.2.19 test_poly_derivative.pcpl

```
/*test poly operations.
function : Returns the derivative of the polynomial with the given
coefficients.*/

int main(){

 poly result;
 poly p;
 p = {-1.0, 2.0, -6.0, 2.0};      /* -1.0 + 2.0x -6.0x^2 + 2.0x^3 */

 if (deg(p) == 0)
 {
   result = {0.0};
```

```
 }
 else
 {
   result = differentiate(p);
 }

 print("derivative of ");
 print(p);
 print(" = ");
 println(result);

 return 0;
}

/* precondition: deg(p) >=1

p = -1.0 + 2.0x -6.0x^2 + 2.0x^3
p' = 2.0 - 12.0x + 6.0x^2
*/

poly differentiate(poly p)
{
  int i;
  poly result;

  result = p;

  for (i = deg(p); i >=0 ; i=i-1) {
    result[i] = result[i] *i;
  }

  return result << 1;
}
```

Output

```
derivative of -1.0 + 2.0 x - 6.0 x^2 + 2.0 x^3 = 2.0 - 12.0 x + 6.0 x^2
```

6.2.20 test_integral.pcpl

```
/*function : Returns the integral of the polynomial with the given
coefficients.*/
/*demonstrate a function that makes a deep copy of a poly function */

int main(){

 poly result;
 poly p;
 p = {-1.0, 2.0, -6.0, 2.0};      /* -1.0 + 2.0x -6.0x^2 + 2.0x^3 */

 result = integrate(p);

 print("integral of ");
 print(p);
```

```
 println(" = ");
 println(result);

 return 0;
}

/*
p = -1.0 + 2.0x -6.0x^2 + 2.0x^3
integral of p  = -x + x^2 - 2.0x^3 + 0.5x^4
*/

poly integrate(poly p)
{
  int i;
  poly result;

  result = copy(p);

  for (i = deg(p); i >=0 ; i=i-1) {
    result[i] = result[i] / (i+1);
  }

  return result >> 1;

}

poly copy(poly a)
{
  int i;
  poly b;

  b = poly{deg(a)};
  for (i=0; i <= deg(a); i=i+1) {
    b[i] = a[i];
  }

  return b;
}
```

Output

```
integral of -1.0 + 2.0 x - 6.0 x^2 + 2.0 x^3 =
-x + x^2 - 2.0 x^3 + 0.5 x^4
```

6.3. Automation in testing

Bat file runtests.bat is used to run tests. It invokes pcpl.exe on each test file, producing a java file; it then invokes javac.exe on PolynormialFunction.java and test_xxxx.java to produce java class files. Finally it invokes java.exe on the class files and captures the output in test_xxxx .run files.

7. Lessons Learned

- Start early.
- Make small and incremental changes, one at a time. I started with the microc source code. Adding one new data type or new component at a time to the project. It is easy to identify a bug, if the change is small and I also felt less overwhelmed by the tasks by focusing on small tasks.

- During the course of this project, I realized that designing a programming language or writing a software program are not much different than writing in a natural langue. In both cases we try to convey something to the users, either an idea or a way to solve a problem, etc. So when I write a program I should keep in my mind what problem I try to solve, how I should present the idea\solution to the reder in a concise and easy to understand way.
- And one of the best ways to learn a new language is to read a lot code. I benefit greatly from reading other students' project and leaned a lot from them.

8. Appendix

PCPL compiler souce code:

- scanner.mll
- parser.mly
- ast.ml
- typecheck.ml
- javacode.ml
- pcpl.ml

PolynomialFunction.java

Bat files:

- clean.bat
- make.bat
- runtests.bat

scanner.mll

```
{
  open Parser
  exception Eof
}

let digit = ['0'-'9']

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf }       (* Whitespace *)
| "/*"       { comment lexbuf }               (* Comments *)
| '('        { LPAREN }
| ')'        { RPAREN }
| '['        { LBRACKET }
| ']'        { RBRACKET }
| '{'        { LBRACE }
| '}'        { RBRACE }
| ';'        { SEMI }
| ','        { COMMA }
| ':'        { COLON }
```

```
| '+'        { PLUS }
| '-'        { MINUS }
| '*'        { TIMES }
| '/'        { DIVIDE }
| '^'        { CARET }
| "<<" { LSHIFT }
| ">>" { RSHIFT }
| '='        { ASSIGN }
| "=="       { EQ }
| "!="       { NEQ }
| '<'        { LT }
| "<="       { LEQ }
| '>'        { GT }
| ">="       { GEQ }
| "&&"       { AND }
| "||"       { OR }
| '!'        { NOT }
| "if"       { IF }
| "else"     { ELSE }
| "for"      { FOR }
| "while"    { WHILE }
| "return"   { RETURN }
| "true"     { TRUE }
| "false"    { FALSE }
| "int"      { INT }
| "float"    { FLOAT }
| "string"   { STRING }
| "boolean"  { BOOLEAN }
| "poly"     { POLY }
| digit+ as lxm { INTLITERAL(int_of_string lxm) }
| digit+ ('.' digit+)+ as lxm { FLOATLITERAL(float_of_string lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
| '"' ([^ '"' '\n' '\r' '\\'] | "\\\\" | "\\\"" | "\\n" | "\\t")* '"' as lxm { STRINGLITERAL(lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

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

Note: | '"' ([^ '"' '\n' '\r' '\\'] | "\\\\" | "\\\"" | "\\n" | "\\t")* '"' is a direct copy of regular expression from previous class project.

---

parser.mly

```
%{
  open Ast
  let parse_error s = Printf.ksprintf failwith "ERROR: %s" s
%}

%token SEMI LPAREN RPAREN LBRACKET RBRACKET LBRACE RBRACE COMMA COLON
%token PLUS MINUS TIMES DIVIDE CARET LSHIFT RSHIFT ASSIGN
%token EQ NEQ LT LEQ GT GEQ
%token AND OR NOT
%token RETURN IF ELSE FOR WHILE
%token INT FLOAT BOOLEAN STRING POLY
%token TRUE FALSE
%token <int> INTLITERAL
%token <float> FLOATLITERAL
```

```
%token <string> ID
%token <string> STRINGLITERAL
%token EOF

%nonassoc NOCOMMA
%nonassoc COMMA

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE

%start program
%type <Ast.program> program

%%

program:
   /* nothing */ { [], [] }
 | program vdecl { ($2 :: fst $1), snd $1 }
 | program fdecl { fst $1, ($2 :: snd $1) }

fdecl:
   type_decl ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE
     { { rtype   = $1;
         fname   = $2;
        formals = $4;
        locals  = List.rev $7;
        body    = $8
       }
     }

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

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

formal:
    type_decl ID
    { { vtype = $1;
        vname = $2;
    } }

vdecl_list:
    /* nothing */    { [] }
  | vdecl_list vdecl { $2 :: $1 }

type_decl:
    INT     { Int }
  | FLOAT   { Float }
  | BOOLEAN { Boolean }
  | POLY    { Poly }
  | STRING  { String }

vdecl:
    type_decl ID SEMI
    { { vtype = $1;
```

```
        vname = $2;
      }
    }

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

stmt:
    expr SEMI                                                    { Expr($1) }
  | RETURN expr SEMI                                             { Return($2) }
  | LBRACE stmt_list RBRACE                                      { Block($2) }
  | IF LPAREN expr RPAREN stmt %prec NOELSE                      { If($3, $5, Block([])) }
  | IF LPAREN expr RPAREN stmt ELSE stmt                         { If($3, $5, $7) }
  | FOR LPAREN expr_opt SEMI expr_opt SEMI expr_opt RPAREN stmt { For($3, $5, $7, $9) }
  | WHILE LPAREN expr RPAREN stmt                                { While($3, $5) }

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

/*
float_list:
    FLOATLITERAL %prec NOCOMMA    { [$1] }
  | FLOATLITERAL COMMA float_list { $1 :: $3 }
*/

lvalue:
    ID                              { Id($1) }
  | ID LBRACKET expr RBRACKET       { PolyElmt($1,$3)}

expr:
    INTLITERAL                      { IntLiteral($1) }
  | FLOATLITERAL                    { FloatLiteral($1) }
  | STRINGLITERAL                   { StringLiteral($1) }
  | LBRACE actuals_list RBRACE      { PolyLiteral(List.rev $2) }
  | POLY LBRACE expr RBRACE         { PolyInit($3) }
  | lvalue                          { Lvalue($1) }
  | TRUE                            { BooleanLiteral(true) }
  | FALSE                           { BooleanLiteral(false) }
  | 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 LSHIFT expr                { Binop($1, Lshift, $3) }
  | expr RSHIFT expr                { Binop($1, Rshift, $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) }
  | MINUS expr                      { Negate($2) }
  | lvalue ASSIGN expr              { Assign($1, $3) }
  | ID LPAREN actuals_opt RPAREN    { Call($1, $3) }
  | LPAREN expr RPAREN              { $2 }

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

actuals_list:
    expr                    { [$1] }
```

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

---

ast.ml

```
type op = Add | Sub | Mult | Div | Lshift | Rshift| Equal | Neq | Less | Leq | Greater | Geq
type datatype = Int | Float | Boolean | Poly | String | UnknownType

type lvalue =
    Id of string
  | PolyElmt of string * expr

and expr =
    IntLiteral of int
  | FloatLiteral of float
  | BooleanLiteral of bool
  | StringLiteral of string
  | PolyLiteral of expr list
  | PolyInit of expr
  | Lvalue of lvalue
  | Binop of expr * op * expr
  | Negate of expr
  | Assign of lvalue * expr
  | Call of string * expr list
  | Noexpr



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

type var_decl = {
    vtype: datatype;
    vname: string;
}

type func_decl = {
    rtype : datatype;
    fname : string;
    formals : var_decl list;
    locals : var_decl list;
    body : stmt list;
  }

type program = var_decl list * func_decl list

let string_of_datatype dtype =
  match dtype with
    Int -> "int"
  | Float -> "float"
  | Boolean -> "boolean"
  | Poly -> "poly"
  | String -> "string"
  | UnknownType -> "unknownype"

let string_of_binop o =
  match o with
    Add -> "+"
```

```
  | Sub -> "-"
  | Mult -> "*"
  | Div -> "/"
  | Lshift -> "<<"
  | Rshift -> ">>"
  | Equal -> "=="
  | Neq -> "!="
  | Less -> "<"
  | Leq -> "<="
  | Greater -> ">"
  | Geq -> ">="

let rec string_of_expr = function
    IntLiteral(l) -> string_of_int l
  | FloatLiteral(l) -> string_of_float l
  | BooleanLiteral(l) -> string_of_bool l
  | StringLiteral(l) -> l
  | PolyLiteral(l) -> "{ "^ String.concat ", " (List.map string_of_expr l)^" }"
  | PolyInit(l) -> "poly{ "^ string_of_expr l^" }"
  | Lvalue(lv) -> string_of_lvalue lv
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^
      string_of_binop o ^ " " ^
      string_of_expr e2
  | Negate(e) -> "-" ^ string_of_expr e
  | Assign(v, e) -> string_of_lvalue v ^ " = " ^ string_of_expr e
  | Call(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | Noexpr -> ""

and string_of_lvalue = function
    Id(s) -> s
  | PolyElmt(s, e) -> s ^ "[ " ^ string_of_expr e  ^ " ]"

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 (" ^ 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

let string_of_vdecl vdecl = (string_of_datatype vdecl.vtype) ^ " " ^ vdecl.vname ^ ";\n"

let string_of_formal formal =
  string_of_datatype formal.vtype ^ " " ^ formal.vname

let string_of_fdecl fdecl =
  string_of_datatype fdecl.rtype ^ " " ^ fdecl.fname ^ "(" ^ String.concat ", " (List.map
string_of_formal fdecl.formals) ^ ")\n{\n" ^
  String.concat "" (List.map string_of_vdecl fdecl.locals) ^
  String.concat "" (List.map string_of_stmt fdecl.body) ^
  "}\n"

let string_of_program (vars, funcs) =
  String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
  String.concat "\n" (List.map string_of_fdecl funcs)
```

typecheck.ml

```
(* generate code for java *)

(* TODO: Binop in expr, and the whole stmt *)

open Ast

module StringMap = Map.Make(String)

exception Fail of string

let type_of_var global_map formal_map local_map s =
  try StringMap.find s global_map
  with Not_found ->
     try StringMap.find s formal_map
     with Not_found ->
        try StringMap.find s local_map
        with Not_found ->
           raise(Fail("undefined variable " ^ s ^ "."))

let type_of_lvalue global_map formal_map local_map = function
    Id(s) -> type_of_var global_map formal_map local_map s
  | PolyElmt(s, e) -> Float

let rec typecheck_poly_literal = function
    [] -> Poly
  | hd::tl ->
      match hd with
        FloatLiteral(l) -> typecheck_poly_literal tl
      | Negate(e) ->
         (match e with
           FloatLiteral(l2) -> typecheck_poly_literal tl
         | _ -> raise(Fail("polynomial initializer can only use float literals"))
         )
      | _ -> raise(Fail("polynomial initializer can only use float literals"))

let rec type_of_expr func_map global_map formal_map local_map = function
    IntLiteral(l) -> Int
  | FloatLiteral(l) -> Float
  | BooleanLiteral(l) -> Boolean
  | StringLiteral(l) -> String
  | PolyLiteral(l) -> typecheck_poly_literal l
  | PolyInit(e) ->
      let t1 = type_of_expr func_map global_map formal_map local_map e
      in
      if t1 == Int then
         Poly
      else
         raise(Fail("PolyInit must take an expression of type int"))
  | Lvalue(lv) -> type_of_lvalue global_map formal_map local_map lv
  | Binop(e1, o, e2) ->
      let t1 = type_of_expr func_map global_map formal_map local_map e1
      in
      let t2 = type_of_expr func_map global_map formal_map local_map e2
      in
```

```
      (match o with
        Add | Sub | Mult ->
          if (t1==Poly && t2==Poly) then
             Poly
          else if (t1==Int && t2==Int) then
             Int
          else if (t1==Float && t2==Float) then
             Float
          else if (t1==Float && t2==Int) then
             Float
          else if (t1==Int && t2==Float) then
             Float
          else
             raise(Fail("type mismatch in binop: " ^ string_of_expr e1 ^ " " ^
string_of_binop o ^ " " ^ string_of_expr e2))
      | Div ->
          if (t1==Poly && (t2==Int || t2 ==Float)) then
             Poly
          else if (t1==Int && t2==Int) then
             Int
          else if (t1==Float && t2==Float) then
             Float
          else if (t1==Float && t2==Int) then
             Float
          else if (t1==Int && t2==Float) then
             Float
          else
             raise(Fail("type mismatch in binop: " ^ string_of_expr e1 ^ " " ^
string_of_binop o ^ " " ^ string_of_expr e2))
      | Lshift | Rshift ->
          if t1==Poly && t2==Int then
             Poly
          else
             raise(Fail("type mismatch in binop: " ^ string_of_expr e1 ^ " " ^
string_of_binop o ^ " " ^ string_of_expr e2))
      | Equal | Neq ->
          if t1==t2 then
             Boolean
          else if ((t1==Int || t1==Float) && (t2==Float || t2==Int)) then
             Boolean
          else
                raise(Fail("type mismatch in binop: " ^ string_of_expr e1 ^ " " ^
string_of_binop o ^ " " ^ string_of_expr e2))
      | Less | Greater | Leq | Geq ->
          if ((t1==Int || t1==Float) && (t2==Float || t2==Int)) then
             Boolean
          else
             raise(Fail("type mismatch in binop: " ^ string_of_expr e1 ^ " " ^
string_of_binop o ^ " " ^ string_of_expr e2))
      )
  | Negate(e) ->
      let t = type_of_expr func_map global_map formal_map local_map e
      in
      (match t with
         Poly -> Poly
       | Int -> Int
       | Float -> Float
```

```
       | _ -> raise(Fail("Cannot negate expression: " ^ string_of_expr e))
      )
  | Assign(v, e) ->
      let t1 = type_of_lvalue global_map formal_map local_map v
      in
      let t2 = type_of_expr func_map global_map formal_map local_map e
      in
      if t1 != t2 then
         raise(Fail("type mismatch in Assignment: " ^ string_of_lvalue v ^ " = " ^
string_of_expr e))
      else
         t1
  | Call(f, el) ->
      (try
         let fdecl = StringMap.find f func_map
         in
         if List.length fdecl.formals != List.length el then
            raise(Fail("Argument/Formals count mismatch in function call: " ^ f))
         else
            let formal_type_list = List.map (fun v -> v.vtype) fdecl.formals
            in
            let actual_type_list = List.map (fun e -> type_of_expr func_map global_map
formal_map local_map e) el
            in
            List.fold_left2
                 (fun rtype t1 t2 ->
                    if t1 != t2 then
                      raise(Fail("types of formal and actual parameters don't match at
call to function: " ^ f))
                    else
                      rtype) fdecl.rtype formal_type_list actual_type_list
      with Not_found ->
         match f with
            "print" ->
               (if List.length el > 1 then
                  raise(Fail("print cannot take multiple arguments"))
               else
                  UnknownType
               )
          | "println" ->
               (if List.length el > 1 then
                  raise(Fail("print cannot take multiple arguments"))
               else
                  UnknownType
               )
          | "deg" ->
               (if List.length el > 1 then
                  raise(Fail("print cannot take multiple arguments"))
                else
                  if type_of_expr func_map global_map formal_map local_map (List.hd el)
!= Poly then
                     raise(Fail("deg function can only take an argument of type poly"))
                  else
                     Int
               )
          | _ -> raise(Fail("call to undefined function: " ^ f ^ ".")))
  | Noexpr -> raise(Fail("Noexpre."))
```

```
let rec typecheck_stmts func_map fdecl global_map formal_map local_map = function
    [] -> ()
  | hd::tl ->
      typecheck_stmt func_map fdecl global_map formal_map local_map hd;
      typecheck_stmts func_map fdecl global_map formal_map local_map tl

and typecheck_stmt func_map fdecl global_map formal_map local_map = function
    Block(stmts) ->
      typecheck_stmts func_map fdecl global_map formal_map local_map stmts
(*      List.fold_left (fun noth f_map f_decl g_map fv_map lv_map stmt -> let t =
typecheck_stmt f_map f_decl g_map fv_map lv_map stmt in noth) () func_map fdecl
global_map formal_map local_map stmts
*)
  | Expr(expr) ->
      let t = type_of_expr func_map global_map formal_map local_map expr
      in
      do_nothing t
  | Return(expr) ->
      let t = type_of_expr func_map global_map formal_map local_map expr
      in
      if t != fdecl.rtype then
         raise(Fail("declared return type and returned type mismatch in function " ^
fdecl.fname))
  | If(e, s, Block([])) ->
      let etype = type_of_expr func_map global_map formal_map local_map e
      in
      if etype != Boolean then
         raise(Fail("if condition " ^ string_of_expr e ^ " does not have boolean type."))
      else
         typecheck_stmt func_map fdecl global_map formal_map local_map s
  | If(e, s1, s2) ->
      let etype = type_of_expr func_map global_map formal_map local_map e
      in
      if etype != Boolean then
         raise(Fail("if condition " ^ string_of_expr e ^ " does not have boolean type."))
      else
         let noth = typecheck_stmt func_map fdecl global_map formal_map local_map s1
         in
         typecheck_stmt func_map fdecl global_map formal_map local_map s2
  | For(e1, e2, e3, s) ->
      let t1 = type_of_expr func_map global_map formal_map local_map e1
      in
      let t2 = type_of_expr func_map global_map formal_map local_map e2
      in
      if t2 != Boolean then
         raise(Fail("for condition " ^ string_of_expr e2 ^ " does not have boolean
type."))
      else
         let t3 = type_of_expr func_map global_map formal_map local_map e3
         in
         typecheck_stmt func_map fdecl global_map formal_map local_map s
  | While(e, s) ->
      let etype = type_of_expr func_map global_map formal_map local_map e
      in
      if etype != Boolean then
```

```
         raise(Fail("while condition " ^ string_of_expr e ^ " does not have boolean
type."))
      else
         typecheck_stmt func_map fdecl global_map formal_map local_map s

and do_nothing e =
  ()

let typecheck_fdecl global_map func_map fdecl =
  let formal_map = List.fold_left
              (fun formal_map fv_decl ->
                if StringMap.mem fv_decl.vname formal_map then
                    raise (Fail ("formal parameter: '" ^ fv_decl.vname ^
                        "' in function " ^ fdecl.fname ^ " has already been defined."))
                else
                    if StringMap.mem fv_decl.vname global_map then
                        raise (Fail ("formal parameter: '" ^ fv_decl.vname ^
                            "' in function " ^ fdecl.fname ^ " shadows a global
variable."))
                    else
                        StringMap.add fv_decl.vname fv_decl.vtype formal_map)
StringMap.empty fdecl.formals
  in
  let local_map = List.fold_left
              (fun local_map lv_decl ->
                if StringMap.mem lv_decl.vname local_map then
                    raise (Fail ("local variable: '" ^ lv_decl.vname ^
                        "' in function " ^ fdecl.fname ^ " has already been defined."))
                else
                    if StringMap.mem lv_decl.vname formal_map then
                        raise (Fail ("local variable: '" ^ lv_decl.vname ^
                            "' in function " ^ fdecl.fname ^ " shadows a formal
parameter."))
                    else
                        if StringMap.mem lv_decl.vname formal_map then
                            raise (Fail ("local variable: '" ^ lv_decl.vname ^
                                "' in function " ^ fdecl.fname ^ " shadows a formal
parameter."))
                        else
                            StringMap.add lv_decl.vname lv_decl.vtype local_map)
StringMap.empty fdecl.locals
  in
  typecheck_stmts func_map fdecl global_map formal_map local_map fdecl.body

let rec typecheck_fdecls global_map func_map = function
    [] -> ()
  | hd::tl ->
     typecheck_fdecl global_map func_map hd;
     typecheck_fdecls global_map func_map tl

let typecheck_program (vars, funcs) =
  let global_map = List.fold_left
            (fun g_map gv_decl ->
              if StringMap.mem gv_decl.vname g_map then
                  raise (Fail ("global variable: '" ^ gv_decl.vname ^
                     "' has already been defined."))
              else
```

```
                  StringMap.add gv_decl.vname gv_decl.vtype g_map) StringMap.empty vars
  in
  let func_map = List.fold_left
              (fun f_map func_decl ->
                if StringMap.mem func_decl.fname f_map then
                    raise (Fail ("function: '" ^ func_decl.fname ^
                        "' has already been defined."))
                else
                    StringMap.add func_decl.fname func_decl f_map) StringMap.empty funcs
  in
  typecheck_fdecls global_map func_map funcs
```

---

javacode.ml

```
(* generate code for java *)

open Ast

let package_del = "package poly;"

let import_decl = ""

let main_fdecl = "public static void main(String[] args) throws
Exception\n{\npolyMain();\n}\n"

let type_of_expr global_vars local_vars exp =
  match exp with
    Lvalue(lv) ->
      (match lv with
        Id(s) ->
            (try let var = List.find (fun a -> if a.vname = s then true else false)
(global_vars @ local_vars)
                in
                var.vtype
            with Not_found -> UnknownType
            )
      | PolyElmt(s, e) -> Float
      )
  | PolyLiteral(l) -> Poly
  | _ -> UnknownType


let jstring_of_datatype dtype =
  match dtype with
    Int -> "int"
  | Float -> "double"
  | Boolean -> "boolean"
  | Poly -> "PolynomialFunction"
  | String -> "String"
  | UnknownType -> "unknowntype"


let rec jstring_of_expr global_vars local_vars = function
    IntLiteral(l) -> string_of_int l
  | FloatLiteral(l) -> string_of_float l
```

```
  | BooleanLiteral(l) -> string_of_bool l
  | StringLiteral(l) -> l
  | PolyLiteral(l) -> "new PolynomialFunction(new double[]{ "^ String.concat ", "
(List.map (jstring_of_expr global_vars local_vars) l)^" })"
  | PolyInit(l) -> "new PolynomialFunction(" ^ jstring_of_expr global_vars local_vars l ^
")"
  | Lvalue(lv) -> jstring_of_lvalue global_vars local_vars lv
  | Binop(e1, o, e2) ->
      (match (type_of_expr global_vars local_vars e1) with
        Poly ->
          (jstring_of_expr global_vars local_vars e1 ^ "." ^
          (match o with
            Add -> "add"
          | Sub -> "subtract"
          | Mult -> "multiply"
          | Div -> "divide"
          | Lshift -> "shiftLeft"
          | Rshift -> "shiftRight"
          | Equal -> "equals"
          | Neq -> "notEqual"
          | _ -> "operator " ^ string_of_binop o ^ " not implemented for polynomials"
          ) ^ "(" ^ (jstring_of_expr global_vars local_vars e2) ^ ")"
          )
      | _ ->
          ( "(" ^ jstring_of_expr global_vars local_vars e1 ^ " " ^
            (
            match o with
            Add -> "+" | Sub -> "-" | Mult -> "*" | Div -> "/" | Lshift -> "<<" | Rshift
-> ">>"
            | Equal -> "==" | Neq -> "!="
            | Less -> "<" | Leq -> "<=" | Greater -> ">" | Geq -> ">="
            ) ^ " " ^ jstring_of_expr global_vars local_vars e2 ^ ")"
          )
      )
  | Negate(e) ->
      (match (type_of_expr global_vars local_vars e) with
        Poly ->
          "(" ^ jstring_of_expr global_vars local_vars e ^ ".negate()" ^ ")"
      | _ -> "-" ^ jstring_of_expr global_vars local_vars e
      )
  | Assign(v, e) -> jstring_of_lvalue global_vars local_vars v ^ " = " ^ jstring_of_expr
global_vars local_vars e
  | Call(f, el) ->
      (match f with
        "print" -> "System.out.print(" ^ (String.concat "" (List.map (jstring_of_expr
global_vars local_vars) el)) ^ ")"
      | "println" -> "System.out.println(" ^ (String.concat "" (List.map (jstring_of_expr
global_vars local_vars) el)) ^ ")"
      | "deg" -> (String.concat "" (List.map (jstring_of_expr global_vars local_vars)
el)) ^ ".degree()"
      | _ -> f ^ "(" ^ String.concat ", " (List.map (jstring_of_expr global_vars
local_vars) el) ^ ")"
      )
  | Noexpr -> ""

and jstring_of_lvalue global_vars local_vars = function
    Id(s) -> s
```

```
  | PolyElmt(s, e) -> s ^ ".coefficients[" ^ jstring_of_expr global_vars local_vars e  ^
"]"

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

let jstring_of_vdecl vdecl =
  (jstring_of_datatype vdecl.vtype) ^ " " ^ vdecl.vname ^ ";\n"

let jstring_of_gvdecl gvdecl =
  "public static " ^ jstring_of_vdecl gvdecl

let jstring_of_formal formal =
  jstring_of_datatype formal.vtype ^ " " ^ formal.vname

let jstring_of_fdecl global_vars fdecl =
  let local_vars = (List.map (fun a -> { vname = a.vname; vtype = a.vtype })
fdecl.formals)
                   @ (List.map (fun a -> { vname = a.vname; vtype = a.vtype })
fdecl.locals)
  in
  (match fdecl.fname with
    "main" -> "static " ^ jstring_of_datatype fdecl.rtype ^ " polyMain()"
    | _ -> "static " ^ jstring_of_datatype fdecl.rtype ^ " " ^ fdecl.fname ^
           "(" ^  String.concat ", " (List.map jstring_of_formal fdecl.formals) ^ ")"
   ) ^ " throws Exception" ^
  "\n{\n" ^
  String.concat "" (List.map jstring_of_vdecl fdecl.locals) ^
  String.concat "" (List.map (jstring_of_stmt global_vars local_vars) fdecl.body) ^
  "}\n"

let jstring_of_program (vars, funcs) file_name =
  let global_vars = List.map (fun a -> { vname = a.vname; vtype = a.vtype }) vars
  in
  package_del ^ "\n" ^ import_decl ^ "\n\n" ^
  "public class " ^ (String.sub file_name 0 ((String.length file_name) - 5)) ^
  "\n{\n" ^ main_fdecl ^
  String.concat "" (List.map jstring_of_gvdecl vars) ^ "\n" ^
  String.concat "\n" (List.map (jstring_of_fdecl global_vars) funcs) ^
  "\n}"
```

---

pcpl.ml

```
type action = Ast | Compile | Javacode | Typecheck | All


let _ =
  let action = if Array.length Sys.argv > 1 then
      List.assoc Sys.argv.(1) [ ("-a", Ast);
                                ("-j", Javacode);
                                ("-t", Typecheck);
                                ("-c", All)]
  else Compile in
  let input_file = ref Sys.argv.(2) in
  let input = open_in !input_file in
  let lexbuf = Lexing.from_channel input in
  let program = Parser.program Scanner.token lexbuf in
  match action with
    Ast -> let listing = Ast.string_of_program program
           in print_string listing
  | Typecheck -> Typecheck.typecheck_program program
  | Javacode ->
    let listing =
      Javacode.jstring_of_program program !input_file
    in print_endline listing
  | All ->
     ignore (Typecheck.typecheck_program program);
     let listing =
           Javacode.jstring_of_program program !input_file
     in print_endline listing
```

PolynomialFunction.java

```
/*
 * This Java program is modified from
 * the Apache Software Foundation (ASF)'s Commons Math project.
 * Java document of the PolynormialFunction class at
 * http://commons.apache.org/math/api-2.2/index.html
 * Original source code at
 * http://commons.apache.org/math/download_math.cgi
 */
package poly;

import java.util.Arrays;

/**
 * Immutable representation of a real polynomial function with real coefficients.
 * <p>
 * <a href="http://mathworld.wolfram.com/HornersMethod.html">Horner's Method</a>
 *  is used to evaluate the function.</p>
 *
 * @version $Revision: 1042376 $ $Date: 2010-12-05 16:54:55 +0100 (dim. 05 déc. 2010) $
 */
public class PolynomialFunction {

    /**
     * The coefficients of the polynomial, ordered by degree -- i.e.,
```

```
  * coefficients[0] is the constant term and coefficients[n] is the
  * coefficient of x^n where n is the degree of the polynomial.
  */
// private final double coefficients[];
   public double coefficients[];

 /**
  * Construct a polynomial with the given coefficients.  The first element
  * of the coefficients array is the constant term.  Higher degree
  * coefficients follow in sequence.  The degree of the resulting polynomial
  * is the index of the last non-null element of the array, or 0 if all elements
  * are null.
  * <p>
  * The constructor makes a copy of the input array and assigns the copy to
  * the coefficients property.</p>
  *
  * @param c polynomial coefficients
  * @throws NullPointerException if c is null
  * @throws NoDataException if c is empty
  */
 public PolynomialFunction(double c[]) {
     int n = c.length;
     // assume n > 0
     while ((n > 1) && (c[n - 1] == 0)) {
         --n;
     }
     this.coefficients = new double[n];
     System.arraycopy(c, 0, this.coefficients, 0, n);
 }

 // may not need this one!
 public PolynomialFunction(int degree) {
     int n = degree + 1;
     // assume n > 0
     this.coefficients = new double[n];
     for (int i=0; i < n; i++)
     {
         this.coefficients[i] = 0;
     }
 }
 /* shiftLeft corespond to the polynormail operator << in pcpl
    p = {1.0, 2.0,3.0}
    q = p.shiftLeft(1)
    q is {2.0, 3.0}
  */
 public PolynomialFunction shiftLeft(int i) throws Exception
 {
     if (i >= this.coefficients.length || i < 0)
     {
         throw new Exception("illegal shift left");
     }

     double[] nc = new double[this.coefficients.length - i];
     System.arraycopy(this.coefficients, i, nc, 0, nc.length);
     return new PolynomialFunction(nc);
 }
```

```
/* shiftRight corespond to the polynormail operator >> in pcpl
   p = {1.0, 2.0,3.0}
   q = p.shiftRight(1)
   q is {0.0, 1.0, 2.0, 3.0}
 */
public PolynomialFunction shiftRight(int i) throws Exception
{
    if (i < 0)
    {
        throw new Exception("illegal shift right");
    }

    double[] nc = new double[this.coefficients.length + i];
    System.arraycopy(this.coefficients, 0, nc, i, this.coefficients.length);
    return new PolynomialFunction(nc);
}

 /**
 *  Returns the degree of the polynomial
 *
 * @return the degree of the polynomial
 */
public int degree() {
    return coefficients.length - 1;
}

/**
 * Returns a copy of the coefficients array.
 * <p>
 * Changes made to the returned copy will not affect the coefficients of
 * the polynomial.</p>
 *
 * @return  a fresh copy of the coefficients array
 */
public double[] getCoefficients() {
    return coefficients.clone();
}

 /**
 * Add a polynomial to the instance.
 * @param p polynomial to add
 * @return a new polynomial which is the sum of the instance and p
 */
public PolynomialFunction add(final PolynomialFunction p) {

    // identify the lowest degree polynomial
    final int lowLength  = Math.min(coefficients.length, p.coefficients.length);
    final int highLength = Math.max(coefficients.length, p.coefficients.length);

    // build the coefficients array
    double[] newCoefficients = new double[highLength];
    for (int i = 0; i < lowLength; ++i) {
        newCoefficients[i] = coefficients[i] + p.coefficients[i];
    }
    System.arraycopy((coefficients.length < p.coefficients.length) ?
                     p.coefficients : coefficients,
                     lowLength,
```

```
                        newCoefficients, lowLength,
                        highLength - lowLength);

        return new PolynomialFunction(newCoefficients);

    }

    /**
     * Subtract a polynomial from the instance.
     * @param p polynomial to subtract
     * @return a new polynomial which is the difference the instance minus p
     */
    public PolynomialFunction subtract(final PolynomialFunction p) {

        // identify the lowest degree polynomial
        int lowLength  = Math.min(coefficients.length, p.coefficients.length);
        int highLength = Math.max(coefficients.length, p.coefficients.length);

        // build the coefficients array
        double[] newCoefficients = new double[highLength];
        for (int i = 0; i < lowLength; ++i) {
            newCoefficients[i] = coefficients[i] - p.coefficients[i];
        }
        if (coefficients.length < p.coefficients.length) {
            for (int i = lowLength; i < highLength; ++i) {
                newCoefficients[i] = -p.coefficients[i];
            }
        } else {
            System.arraycopy(coefficients, lowLength, newCoefficients, lowLength,
                             highLength - lowLength);
        }

        return new PolynomialFunction(newCoefficients);

    }

    /**
     * Negate the instance.
     * @return a new polynomial
     */
    public PolynomialFunction negate() {
        double[] newCoefficients = new double[coefficients.length];
        for (int i = 0; i < coefficients.length; ++i) {
            newCoefficients[i] = -coefficients[i];
        }
        return new PolynomialFunction(newCoefficients);
    }

    /**
     * Multiply the instance by a polynomial.
     * @param p polynomial to multiply by
     * @return a new polynomial
     */
    public PolynomialFunction multiply(final PolynomialFunction p) {

        double[] newCoefficients = new double[coefficients.length + p.coefficients.length
- 1];
```

```
    for (int i = 0; i < newCoefficients.length; ++i) {
        newCoefficients[i] = 0.0;
        for (int j = Math.max(0, i + 1 - p.coefficients.length);
             j < Math.min(coefficients.length, i + 1);
             ++j) {
            newCoefficients[i] += coefficients[j] * p.coefficients[i-j];
        }
    }

    return new PolynomialFunction(newCoefficients);

}

/*divide the instance by a double.*/
public PolynomialFunction divide(double x) throws Exception {
    if (x == 0)
    {
        throw new Exception("division by zero");
    }

    double[] newCoefficients = new double[coefficients.length];

    for (int i = 0; i < newCoefficients.length; ++i) {
        newCoefficients[i] = this.coefficients[i]/x;
    }

    return new PolynomialFunction(newCoefficients);
}

/* p1 != p2 */
public boolean notEqual(final PolynomialFunction other) {
    return !this.equals(other);
}

 /** Returns a string representation of the polynomial.
 * <p>The representation is user oriented. Terms are displayed lowest
 * degrees first. The multiplications signs, coefficients equals to
 * one and null terms are not displayed (except if the polynomial is 0,
 * in which case the 0 constant term is displayed). Addition of terms
 * with negative coefficients are replaced by subtraction of terms
 * with positive coefficients except for the first displayed term
 * (i.e. we display <code>-3</code> for a constant negative polynomial,
 * but <code>1 - 3 x + x^2</code> if the negative coefficient is not
 * the first one displayed).</p>

 * @return a string representation of the polynomial

 */
@Override
 public String toString() {

   StringBuilder s = new StringBuilder();
   if (coefficients[0] == 0.0) {
     if (coefficients.length == 1) {
       return "0";
     }
```

```
   } else {
     s.append(Double.toString(coefficients[0]));
   }

   for (int i = 1; i < coefficients.length; ++i) {

     if (coefficients[i] != 0) {

       if (s.length() > 0) {
         if (coefficients[i] < 0) {
           s.append(" - ");
         } else {
           s.append(" + ");
         }
       } else {
         if (coefficients[i] < 0) {
           s.append("-");
         }
       }

       double absAi = Math.abs(coefficients[i]);
       if ((absAi - 1) != 0) {
         s.append(Double.toString(absAi));
         s.append(' ');
       }

       s.append("x");
       if (i > 1) {
         s.append('^');
         s.append(Integer.toString(i));
       }
     }

   }

   return s.toString();

 }

/** {@inheritDoc} */
@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + Arrays.hashCode(coefficients);
    return result;
}

/** {@inheritDoc} */
@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (!(obj instanceof PolynomialFunction))
        return false;
    PolynomialFunction other = (PolynomialFunction) obj;
    if (!Arrays.equals(coefficients, other.coefficients))
```

```
            return false;
        return true;
    }
}
```

---

make.bat

```
ocamllex scanner.mll
ocamlyacc parser.mly
ocamlc -c ast.ml
ocamlc -c parser.mli
ocamlc -c scanner.ml
ocamlc -c parser.ml
ocamlc -c typecheck.ml
ocamlc -c Javacode.ml
ocamlc -c pcpl.ml
ocamlc -o pcpl.exe ast.cmo typecheck.cmo Javacode.cmo parser.cmo scanner.cmo pcpl.cmo
```

clean.bat

```
del *.cmo
del *.cmi
del parser.ml
del parser.mli
del scanner.ml
del pcpl.exe
del poly\*.class
```

runtests.bat

```
cd tests
..\pcpl.exe -c test_arith.pcpl > ..\poly\test_arith.java
cd ..
javac poly\test_arith.java poly\PolynomialFunction.java
java -cp . poly.test_arith > tests\test_arith.run

cd tests
..\pcpl.exe -c test_arith_poly.pcpl > ..\poly\test_arith_poly.java
cd ..
javac poly\test_arith_poly.java poly\PolynomialFunction.java
java -cp . poly.test_arith_poly > tests\test_arith_poly.run

cd tests
..\pcpl.exe -c test_assign.pcpl > ..\poly\test_assign.java
cd ..
javac poly\test_assign.java poly\PolynomialFunction.java
java -cp . poly.test_assign > tests\test_assign.run

cd tests
..\pcpl.exe -c test_fib.pcpl > ..\poly\test_fib.java
cd ..
```

```
javac poly\test_fib.java poly\PolynomialFunction.java
java -cp . poly.test_fib > tests\test_fib.run

cd tests
..\pcpl.exe -c test_for1.pcpl > ..\poly\test_for1.java
cd ..
javac poly\test_for1.java poly\PolynomialFunction.java
java -cp . poly.test_for1 > tests\test_for1.run

cd tests
..\pcpl.exe -c test_func1.pcpl > ..\poly\test_func1.java
cd ..
javac poly\test_func1.java poly\PolynomialFunction.java
java -cp . poly.test_func1 > tests\test_func1.run

cd tests
..\pcpl.exe -c test_func2.pcpl > ..\poly\test_func2.java
cd ..
javac poly\test_func2.java poly\PolynomialFunction.java
java -cp . poly.test_func2 > tests\test_func2.run

cd tests
..\pcpl.exe -c test_gcd.pcpl > ..\poly\test_gcd.java
cd ..
javac poly\test_gcd.java poly\PolynomialFunction.java
java -cp . poly.test_gcd > tests\test_gcd.run

cd tests
..\pcpl.exe -c test_global1.pcpl > ..\poly\test_global1.java
cd ..
javac poly\test_global1.java poly\PolynomialFunction.java
java -cp . poly.test_global1 > tests\test_global1.run

cd tests
..\pcpl.exe -c test_hello.pcpl > ..\poly\test_hello.java
cd ..
javac poly\test_hello.java poly\PolynomialFunction.java
java -cp . poly.test_hello > tests\test_hello.run

cd tests
..\pcpl.exe -c test_if1.pcpl > ..\poly\test_if1.java
cd ..
javac poly\test_if1.java poly\PolynomialFunction.java
java -cp . poly.test_if1 > tests\test_if1.run

cd tests
..\pcpl.exe -c test_if2.pcpl > ..\poly\test_if2.java
cd ..
javac poly\test_if2.java poly\PolynomialFunction.java
java -cp . poly.test_if2 > tests\test_if2.run

cd tests
..\pcpl.exe -c test_if3.pcpl > ..\poly\test_if3.java
cd ..
```

```
javac poly\test_if3.java poly\PolynomialFunction.java
java -cp . poly.test_if3 > tests\test_if3.run

cd tests
..\pcpl.exe -c test_if4.pcpl > ..\poly\test_if4.java
cd ..
javac poly\test_if4.java poly\PolynomialFunction.java
java -cp . poly.test_if4 > tests\test_if4.run

cd tests
..\pcpl.exe -c test_ops1.pcpl > ..\poly\test_ops1.java
cd ..
javac poly\test_ops1.java poly\PolynomialFunction.java
java -cp . poly.test_ops1 > tests\test_ops1.run

cd tests
..\pcpl.exe -c test_poly_derivative.pcpl > ..\poly\test_poly_derivative.java
cd ..
javac poly\test_poly_derivative.java poly\PolynomialFunction.java
java -cp . poly.test_poly_derivative > tests\test_poly_derivative.run

cd tests
..\pcpl.exe -c test_poly_eval.pcpl > ..\poly\test_poly_eval.java
cd ..
javac poly\test_poly_eval.java poly\PolynomialFunction.java
java -cp . poly.test_poly_eval > tests\test_poly_eval.run

cd tests
..\pcpl.exe -c test_poly_integral.pcpl > ..\poly\test_poly_integral.java
cd ..
javac poly\test_poly_integral.java poly\PolynomialFunction.java
java -cp . poly.test_poly_integral > tests\test_poly_integral.run

cd tests
..\pcpl.exe -c test_poly_ops.pcpl > ..\poly\test_poly_ops.java
cd ..
javac poly\test_poly_ops.java poly\PolynomialFunction.java
java -cp . poly.test_poly_ops > tests\test_poly_ops.run

cd tests
..\pcpl.exe -c test_var_decl_assign.pcpl > ..\poly\test_var_decl_assign.java
cd ..
javac poly\test_var_decl_assign.java poly\PolynomialFunction.java
java -cp . poly.test_var_decl_assign > tests\test_var_decl_assign.run

cd tests
..\pcpl.exe -c test_while1.pcpl > ..\poly\test_while1.java
cd ..
javac poly\test_while1.java poly\PolynomialFunction.java
java -cp . poly.test_while1 > tests\test_while1.run
```