POLY: A Polynomial Processing Language

Darrell Bethea
djb2118@columbia.edu

Michael G Dougherty
mgd2104@columbia.edu

Santosh Thammana
st2273@columbia.edu

Mohit Vazirani
mcv2107@columbia.edu [1]

December 20, 2005

[1]All four group members are first-year M.S. students in the Department of Computer Science

Contents

Chapter 1

Introduction

This report documents the Poly project, in which our group created a C-like language that includes polynomials as a primitive data type, allowing for programs which implement polynomial algorithms directly.

1.1 Background

Many languages which provide a seemingly broad array of mathematical tools fail to include automatic support for polynomial representation, causing users to have to improvise representations, which are inherently non-standardized and thus non-portable. We created the language Poly to fill this niche.

1.2 Related Work

There are solutions which will allow for the representation and manipulation of polynomials, but in general the support comes as part of inconveniently large applications. For example, programs such as Maple[1], Mathematica[2], and MatLAB[3] allow for manipulation of polynomials, but the applications themselves include so many other functions that they carry a great deal of overhead in the form of disk space and resident memory consumption. And even with the higher system requirements needed to run these applications, their implementations of polynomial math can leave something to be desired.

In Matlab, for example, one cannot add two polynomials of different order, without first explicitly increasing the order of the smaller-order polynomial by padding it with coefficients of 0.[4] While Poly does not suffer from this particular limitation, it is admittedly far less powerful than than Matlab, but it can accomplish many of the same polynomial operations as these other languages with far less overhead.

1.3 Goal

Our goal in designing Poly was to create a language that was a loose subset of C, with the power to do general arithmetic as well as polynomial arithmetic but without very much else. We wanted a concise, easily readable syntax which was reminiscent of C but which also introduced the various new polynomial operators in an intuitive way.

1.3.1 Portability

Because we have implemented our language interpreter using a combination of Java and ANTLR (which itself runs over Java), our language is as portable as the Java Virtual Machine itself, which is to say that Poly programs can run nearly anywhere.

1.3.2 Efficiency

Neither the Poly language itself nor its interpreter were written with efficiency as their primary objectives. Most programs in Poly won't be very long, since the the language makes polynomial processing very succinct. In that sense, then, Poly gains back some of what it loses in efficiency.

1.4 Main Language Features

In this section we document some main language features.

1.4.1 Data Types

The basic data types in our language are `int`, `float`, and `poly`. `poly` is the data type that we have created to represent polynomials. For the types `int` and `float`, the operations supported are addition, subtraction, multiplication, division, and (only between `int`s) modulus. Promotions from `int` to `float` happen in expressions as needed, but there are no conversions from `float` to `int`.

1.4.2 Basic Poly Operations

The `poly` type can be used in a limited way with the operations listed above. A `poly` can be added, subtracted, or multiplied with any `int` or `float`, on either side of the operator. Multiple `poly`s can be added or subtracted from each other. On the other hand, no `poly` can appear on the right side of a division, nor can two `poly`s be multiplied together using the built-in operator. Instead, the Poly language provides the tools needed to implement these algorithms, keeping the language itself simple and leaving the programmer free to write his or her own custom tools.

Along with the new `poly` data type comes new operations which act upon it. There are built-in operators in Poly that will return the order of a polynomial (the degree of its highest term of nonzero coefficient) or a specific coefficient of that polynomial. There is also the seemingly unnecessary ability to "concatenate" polynomials, which at first glance has no useful mathematical analog but which does, in fact, simplify most algorithms greatly.

1.4.3 Flow Control

Flow control in Poly is done using only three constructs: `if`, `if-else`, and `while`. We opted not to include for- and do-while-loops, realizing that each is simply a special instance of a basic `while` loop.

1.4.4 Internal Functions

The only internal function used by Poly is `print`, which will print out a string constant or the current value of an expression.

Chapter 2

Tutorial

2.1 Getting Started

2.1.1 Compiling the Environment

Programs written in POLY are parsed by ANTLR-generated Java programs and interpreted by a Java tree walker. These Java classes must all be built before any POLY programs can be run. To build these necessary files, one should run `make` from within the "Environment" directory. The makefile we have included with the distribution of the POLY interpreter will build all the necessary .java files from the ANTLR grammar and the .class files from all the .java files.

2.1.2 Running a POLY Program

To run a program written in POLY, first check the current directory to make sure it is the "Environment" directory of the POLY distribution. Then (assuming your test program is named "test.poly") run `java PolyMain test.poly` to execute the POLY program.

2.2 Anatomy of a POLY Program

A simple POLY program looks remarkably like a C program. Function prototypes appear first in the file, followed by function definitions, followed by the main body. In the main body, as well as in the bodies of functions, all variable declarations must occur before any other statements.

Let's take a look at a simple POLY program:

```
prototype int square (int);

function int square (int n) {
    return n*n;
}

int x = 5;
int y = square(x);

print x;
print y;
```

This program has one function, *square*, which returns the square of an integer. It then defines an integer variable, assigns its square to another variable using the function, and prints both variables. The output of the program will be

```
x = 5
y = 25
```

Now for a simple program that actually manipulates polynomials:

```
// a = 2x^2 + 3x + 1
poly a = [1,3,2];

// b = 4x + 8
poly b = [8,4];

poly c = a+b;

print c;
```

The output of this program will be

```
c = 2x^2 + 7x + 9
```

Notice that this program also uses comments (starting with // and lasting until the end of the line) and that it has no functions.

2.3 Interesting Examples

Now for some more practical examples of Poly programs that can do common polynomial computations.

2.3.1 Multiplying Polynomials

The following Poly program implements a function that takes two polynomials and returns their product.

```
//PolyMultiplication
prototype poly polyMult (poly, poly);

function poly polyMult (poly A, poly B) {
  poly C=[0];
  poly T=[0];
  int x=0;
  int s=0;

  while(x != (|A|+1)) {
    T = A[x] * B;
    s = x;
    while(s!=0) {
      T = [0]:T;
      s = s-1;
    }
    C = C+T;
    x = x+1;
  }

  return C;
}

print polyMult([5,2,7,2,3], [4,1,0,6,3,2.2]);
```

The function essentially iterates over the coefficients of the polynomial A, multiplying them each by the polynomial B and adding the resulting polynomial to C each time.

Here we see our first example of the concatenation operator at work. It is crucial to this algorithm, as the temporary polynomial T (which is product of B and the current coefficient of A) needs to also be increased in order before it can be added to C. The line `T = [0]:T;` is the POLY equivalent of multiplying a polynomial by x, increasing the order of every term.

2.3.2 Evaluation at a Point

The following POLY program implements a function which accepts a polynomial and a `float` and returns the value of that polynomial at that number. That is, if the `poly` is a polynomial in variable x, then it substitutes the `float` for x and evaluates.

```
//Substitution
prototype float substitute(poly, float);

function float substitute(poly A, float x) {
  int p = 1;
  int q = 0;
  float sum = 0.0;
  float pow = 1.0;

  while(p<|A|+1){
    q=0;
    pow = 1;
    while(q<p){
      pow = pow * x;
      q = q+1;
    }
    sum = sum+(pow*A[p]);
    p = p+1;
  }

  return sum+A[0];
}

print substitute([5,-2,0,0,7,2,3,4.23,0,6.4,2.5], 0.5);
```

This program iterates through the coefficients of A and evaluating each term at the given point and adding it to the running total, which it returns.

Chapter 3

Language Reference Manual

3.1 Lexical Conventions

3.1.1 Comments

The characters "//" denote a single-line comment.

3.1.2 Identifiers

An identifier consists of letters, digits, and underscores ("_"). However, the first character of an identifier must be either a letter of an underscore. Identifiers are case-sensitive.

3.1.3 Keywords

The following identifiers are reserved as keywords:

```
and  else       function  if     not
or   prototype  return    while
```

3.1.4 Numbers

A number can be either an integer or a float. An integer consists of one or more digits. A float consists of one or more digits followed by a decimal point ("`.`"), followed by one or more digits.

3.1.5 Polynomials

A constant polynomial (in the varaible x, for example) can be explicitly written in a program as a comma-separated list of coefficients enclosed within "`[`" and "`]`". The numbers in this list, from left to right, represent the coefficients of the $x^0, x^1, \ldots$ terms of the polynomial. Poly can only represent polynomials in a single variable.

3.1.6 Strings

A string is a sequence of characters enclosed by double quotes (`"`). A double quote inside the string is represented using two adjacent double quotes.

3.1.7 Other Tokens

The following symbols are also used in the language:

```
{   }   (   )   [   ]   ,
;   +   -   *   /   %   =
>   <   ==  !=  :
```

3.2 Types

Poly is a statically typed language. The following types are distinguished (at compile time):

- **int** : 32-bit integers
- **float** : 32-bit IEEE floating point format (but only to 3 decimal places)
- **poly** : represents a polynomial in a single variable with floating point and/or integer coefficients
- **string** : string of characters (constants only — these cannot be stored)
- **function** : user-defined functions

3.3 Expressions

3.3.1 Primary Expressions

Primary expressions are those expressions which are not arithmetic, logical, or relational. These include identifiers, constants, function calls, polynomial coefficients, polynomial order, concatenated polynomials, and any other expression surrounded by `(` and `)`. Primary expressions are right-value expressions, meaning they can appear on the right side of assignment statements.

Identifier

An identifier used by itself represents a variable and evaluates to the value stored in that variable. Identifiers are both left- and right-value expressions, meaning they can appear on either side of an assignment operator.

Constant

A constant is an explicit right-value term (such as a number, polynomial, or quote-enclosed string) that will evaluate to itself in a given context.

Function Calls

Syntax: *funcname*`(`*arg1*`,` *arg2*`,` ...`,` *argn*`)`

$funcname$ is the name of the function, and $arg1$, $arg2$, etc. are the arguments (if there are any) passed to the function.

Each argument must be an expression. Commas are used to separate adjacent arguments.

Coefficient Extraction

Syntax: *polyname*`[`*i*`]`

polyname is an expression which evaluates to a polynomial, and i is an integer.

This expression evaluates to the coefficient of the x^i term in the polynomial *polyname*.

Concatenation of Polynomials

Syntax: *polyA*`:`*polyB*
polyA and *polyB* are each expressions which evaluate to polynomials.
This concatenation evaluates to the polynomial formed by concatenating the coefficient list of *polyB* to the coefficient list of *polyA*. For example, `[4,3,5]:[7,6,2]` evaluates to `[4,3,5,7,6,2]`.

Order of a Polynomial

Syntax: `|`*polyname*`|`
polyname is an expression which evaluates to a polynomial.
This expression evaluates to the order of *polyname*, which is the value of the largest exponent for which *polyname* has a non-zero coefficient.

Parentheses

Syntax: `(` *expr* `)`
expr is any expression.
Enclosing *expr* in parentheses causes the expression to be considered an atomic element in the surrounding context, meaning that it will be fully evaluated before being used.

3.3.2 Arithmetic Expressions

Arithmetic expressions represent arithmetic done on some of the primary expressions. Primary expressions are taken as operands to the following rules.

Unary Operator "-"

Syntax: `-`*a*
a is an expression evaluating to `int`, `float`, or `poly`.
This returns the value of *a* multiplied by negative one.

Multiplicative Operators

Syntax:
a`*`*b* (multiplication)
a`/`*b* (division)
a`%`*b* (modulo)

The three operators share the same precedence level and are grouped left to right. Operands of type `int` can always be used on one or both sides of these operations. `float`s can be used freely in multiplication and division, but they cannot be used at all in modulo operations. If a `poly` is used in multiplication, it can be the first or second operand, but not both. For division, a `poly` can only be used as the first operand. The `poly` type can't be used in a modulo operation.

Additive Operators

Syntax:
a`+`*b* (addition)
a`-`*b* (subtraction)

a and *b* can be either an `int`, `float`, or `poly`. These have the same precedence level and are grouped left to right. They have lower precedence level than the multiplicative operators.

3.3.3 Relational Expressions

Syntax:

a`==`*b* (equal to)
a`!=`*b* (not equal to)
a`>`*b* (greater than)
a`<`*b* (less than)

a and *b* can each be an `int`, `float`, or `poly`.

In the case that the operands are some combination of `int`s and `float`s, a simple arithmetic comparison is made.

If an `int` or `float` is compared with a `poly`, then the `int` or `float` will be promoted to an order-zero polynomial so that the comparison can continue as it would with two polynomials.

If the operands are both polynomials, and the comparison is an equality operation (`==` or `!=`), the polynomials will be equal if and only if their orders are equal and each of their coefficients are equal — that is, for every exponent $0 \leq i \leq |a|$, we have $a[i] == b[i]$.

If the operands are both polynomials and the comparison is an inequality operation (`<` or `>`), the polynomials will be compared asymptotically. That is, the polynomial whose value is larger as x approaches infinity will be called greater than the other polynomial.

The values 1 or 0 will represent true and false, respectively. The appropriate number is returned after the evaluation of the relational operator. Relational operators require exactly two expressions.

3.3.4 Logical Expressions

Logical operators include `not`, `and`, and `or`. They are listed in relative order of precedence, from highest to lowest. As with relational expressions, a logical expression will evaluate to 1 or 0 for true or false, respectively. Numbers equal to zero and polynomials equal to `[0]` will evaluate to false. All other numbers and polynomials will evaluate to true.

`not` Operator

Syntax: `not(`*expr*`)`

expr is an expression.

If *expr* evaluates to true, `not(`*expr*`)` evaluates to false, otherwise it evaluates to true.

`and` Operator

Syntax: *expr1* `and` *expr2*

*expr*1 and *expr*2 are both expressions.

If *expr*1 and *expr*2 both evaluate to true, the above expression evaluates to true; otherwise, it evaluates to false.

`or` Operator

Syntax: *expr1* `or` *expr2*

*expr*1 and *expr*2 are both expressions.

If either *expr*1 or *expr*2 evaluate to true, the above expression evaluates to true; otherwise, it evaluates to false.

3.4 Statements

Statements are the basic elements of a program. A sequence of statements will be executed sequentially, unless the flow-control statements intervene. With the exception of `if`, `if-else`, and `while`, all statements end with a semicolon (`;`).

3.4.1 Assignments

Assignment statements will consist of an identifier followed by an equal sign (`=`) followed by an expression. The value of the expression will then be associated with the identifier.

3.4.2 Variable declarations

When an assignment statement is preceded by a variable type, the statement is interpreted as a variable declaration. Variables must be declared exactly once, and they cannot be used before they are declared. Variables declared within a function must be declared at the top of a function, and global variables must be declared at the top of the main area of a program.

3.4.3 Conditional statements

`if` Statement

Syntax:

```
if ( logExpr ) {
    statements
}
```

logExpr is a logical expression. *statements* represents a list of language statements.

In an `if` statement, if the logical expression *logExpr* evaluates to true, then the list of statements (*statements*) is executed. If *logExpr* evaluates to false, the list is not executed.

`if-else` Statement

Syntax:

```
if ( logExpr ) {
    statements1
} else {
    statements2
}
```

logExpr is a logical expression. *statements* represents a list of language statements.

In an `if-else` statement, if the logical expression *logExpr* evaluates to true, then the first list of statements (*statements*1) is executed and the second list (*statements*2) is ignored. If *logExpr* evaluates to false, the second list of statements (*statements*2) is executed and the first list (*statements*1) is ignored.

3.4.4 Iterative statements

Syntax:

```
while ( logExpr ) {
    statements
}
```

logExpr is a logical expression. *statements* represents a list of language statements.

All iterative statements are performed by using a `while` loop. The result of a `while` loop is that the list of statements (*statements*) will be executed until *logExpr* evaluates to false. If *logExpr* is initially false, the list of statements will not execute at all.

3.4.5 Return statements

Syntax:

```
return expr;
return;
```

expr an expression being returned by a function. The type to which the expression evaluates must match the function's return type. A return with no *expr* is used to exit a function that returns *void*.

3.5 Functions

3.5.1 Function Prototypes

Syntax: `prototype` *typename funcname* `(` *argtype1*`,` *argtype2*`, ...,` *argtypen*`)`

Before a function can be defined, it must first have a prototype. A function prototype begins with the keyword `prototype`, followed by the function's return type (*typename*). If the function does not return a value, then *typename* should be the keyword `void`. Next will be the function name (*funcname*), followed by a comma-separated list of types (*type*1...*typen*) which correspond to the types of the arguments in the function's argument list. This list could be empty if the function takes no arguments.

3.5.2 Function Declaration

Syntax:

```
function typename funcname ( type1 arg1, type2 arg2, ..., typen argn ) {
   statements
}
```

typename is the type of value being returned by the function (`int`, `float`, `poly`, or — if the function does not return a value — `void`). *funcname* is the name of the function itself. *type*1 . . . *typen* are the types of the arguments in the argument list (these should match those in the prototype for the function). *arg*1 . . . *argn* are the identifiers assigned to each argument in the argument list. References in the function body to the arguments will use these identifiers. All function arguments are passed by value. *statements* is a list of program statements which compromise the function body.

3.6 Internal Functions

3.6.1 `print` Function

Syntax: `print` *str-expr1*`,` *str-expr2*`, ...,` *str-exprn*

The `print` function takes one or more arguments ($str - expr1 \ldots str - exprn$) and prints them to the standard output one by one. The arguments can be either expressions or string constants. When the expression to be printed is an identifier, the `print` function will include the identifier's name in the output. For example, `print 5;` will simply print the number `5`, but (assuming the variable *b* is set to 5) `print b;` will print `b = 5`.

3.7 Program Layout

Programs written in Poly are organized strictly into the following sections, which are listed here in the order they should appear in a program:

3.7.1 Function Prototypes

If there are any functions in the program, then a prototype for each function must appear in this section before any other content.

3.7.2 Function Definitions

Immediately following the prototypes will be the definitions of each of the functions themselves. The layout of the body of a function will be the same as the main body of the program, with the exception that `return` statements are allowed within function bodies.

3.7.3 Main Body

Within the main body, all variable declarations must appear before any other statements.

Chapter 4

Project Plan

4.1 Processes

At the beginning of this project, sample programs were written which showed how the language would look. After discussion with the professor, the syntax was revised. From this syntax, a more formal understanding of the language was developed. Finally, the language reference manual was written. The task of writing the manual was split up among the entire group.

In the early stages of development, when the lexer and parser were being done, there was not as much of a need for a strict development process. At this point, we simply came up with an initial design of the lexer and parser, and made modifications to it as necessary.

However, as the project became more complex, division of labor was necessary. Michael's task was to create the interpreter, which interfaces with the walker. The walker was developed by Mohit and Santosh. Darrell did documentation and testing.

The walker was specified based on what had been done in the parser. Mohit and Santosh determined the possible subtree roots, their children, and the necessary actions to be taken at each node. They also determined where error checks were necessary. Development proceeded based on this analysis.

When designing the interpreter, Mike came up with the necessary Java classes. These classes were based on the various language constructs. Similar to the walker, error checks were ncessary to make sure certain actions aren't allowed. The run-time environment was then created.

During the design process, much communication was necessary between the walker people and Mike. Mike informed the walker people about which Java classes and functions the walker should call. In turn, Mohit and Santosh told Mike what Java functionality was necessary in order to walk the tree.

To perform testing, a few small test programs were initially created. These uncovered problems in development. As we needed more test programs, Darrell created a test suite. In this suite, programs ranged from simple to more complex. The suite tested the different language features. To make things easier, Darrell automated the running of all the tests. This automation compared program output to what was expected, and reported an error if there were differences. It's also possible to run just one test.

Documentation was written by Darrell in LaTeX. Other team members contributed where necessary, such as with the lessons learned. This documentation consists of the design, project plan, manual, code, and other required parts.

While tasks were split up, it was certainly necessary to collaborate with each other. If problems occurred, we would look at each others' code to see where the problem might be, and where it should be fixed. Various team members contributed to the test scripts. Also, decisions made along the way were done as a team.

4.2 Coding Practices

This section basically explains the standards and conventions that the developers adhered to while coding the various grammar and java files.

4.2.1 ANTLR Conventions

The Java code embedded in the grammar files in various places including the actions follows the ANSI style of formatting and indenting. Simple ANTLR rules without actions are written on a single line. More complex rules occupy more lines. For complex rules, the rule name and the "`:`" are tab-seperated, and the same holds for the "`:`" and the choice. If there are alternative choices, they are seperated by a "`|`" which is always in the same column as the "`:`". The same holds for the "`;`" that terminates the rule. For the tree parser, the semicolon is always at the first column.

4.2.2 Java Conventions

Indentation and Spacing

- Each level has an indentation of 4 white space characters.
- If the body of an `if` or `else` can be gracefully implemented in one line, it is, and braces are omitted.
- The left brace ({) is used immediately following a `while`, `for`, and multiline `if` and `else` statement, and immediately after a function definition.
- The right brace (}) takes a full line for itself, and is aligned in the same column as its matching right brace.
- No spaces between parentheses "`(`" and "`)`" and their contents, except in cases where it would severely inhibit readability.
- Add spaces between "`=`" and "`==`" operators and their operands, unless they are part of a for loop definition. All other operators take no spaces.
- Since we are using java, tabs are fine.
- The body of `if`, `else`, `for`, and `while` statements are indented.

Names

- Class names start with "Poly" followed by a reasonable 1-2 word description of the contents, with the first letter of each word capitolized.
- Variable names and method names are lower case. Multiple-word variables have the first letter of each words capitalized, except for the first word.

4.3 Team Responsibilities

Since the beginning of the semester, our group has met a minimum of once per week, as well as holding meetings nearly every week (after the proposal) with Professor Edwards. The responsibilities of each group member were as follows:

Darrell Bethea	Group leader; testing and documentation
Michael G Dougherty	Java back-end implementation
Santosh Thammana	ANTLR files and debugging
Mohit Vazirani	ANTLR files and debugging

4.4 Project Timeline

Our group imposed the following deadlines in an effort to keep the project progressing smoothly.

Sep 27	Project proposal
Oct 13	Finish lexer
Oct 20	LRM
Oct 27	Finish parser
Nov 1	Write portions of walker/interpreter to enable testing to begin
Nov 8	Get ”Hello World” to work
Nov 20	Get simple programs to work
Dec 15	Create test suite, Get more complicated programs to work
Dec 17	Complete coding/testing
Dec 20	Finish project report

4.5 Project Log

These are the dates of significant events and advances in the lifetime of the Poly languages.

Sep 27	Project proposal
Oct 4	Language revised
Oct 10	Lexer initially written
Oct 18	Initial LRM draft
Oct 20	LRM turned in
Oct 24	Parser initially written
Nov 1	Started interpreter and walker
Dec 3	ASTs display properly
Dec 13	Simple programs work
Dec 15	Test suite created
Dec 19	More complicated programs work
Dec 20	LRM, Documentation, Code, Test Suite all complete

4.6 Software Project Environment

4.6.1 Operating System

We did most of our testing in the CLIC labs, whose computers are running Redhat Enterprise Linux AS 4 with kernel 2.6.9-22.0.1.EL. However, since our interpreter runs over Java, our project itself is limited more by where the Java VM can run than by any specific operating system.

4.6.2 Java

We compiled our Java programs on systems running Java 1.5.0, but we compiled our programs using Java 1.4 to maintain backwards compatibility, since not all of the computers in the computer science network have been upgraded to Java 1.5.

4.6.3 ANTLR

The acronym “ANTLR” stands for “ANother Tool for Language Recognition.” According to the official website[5], ANTLR is “a language tool that provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions containing Java, C#, C++, or Python actions.” The grammar for Poly, including both the lexer and the parser, is written as ANTLR files, which are then compiled using

ANTLR into .java files which can be run with the rest of our Java code. We used ANTLR 2.7.5 for this project.

4.6.4 CVS

The CVS is a "Concurrent Version System" which developers use to control versioning of large groups of files being edited by multiple people over time. The version of CVS being used to host the Poly project during development was 1.11.17. The repository was located in Darrell's home directory.

4.6.5 GNU Make

For compiling our interpreter and running our test suite, we used GNU Make 3.80. In a way, using Make further limits our Poly interpreter to the systems on which Make itself runs, but since Make is only a tool to speed compilation and run program batches, it is ultimately only a convenience, and the Poly interpreter can be built and run without it.

Chapter 5

Architectural Design

The Poly compiler consists of a number of conceptual components, which in turn consist of one or more source files. The components and their roles are as follows: the lexer divides the input file into tokens, the parser builds an abstract syntax tree after analyzing the token syntax, the walker checks the semantics as it executes the code, and calls functions in both the type system, which contains methods and objects responsible for handling data, and the interpreter, which is primarily responsible for handling operations in the back-end. The symbol table keeps track of the state and scope (local and global) of variables, and the function control module essentially is responsible for the maintenance of the function call stack. Our own class of exceptions are accessed by a number of different modules, which serves to maintain consistency and accuracy.

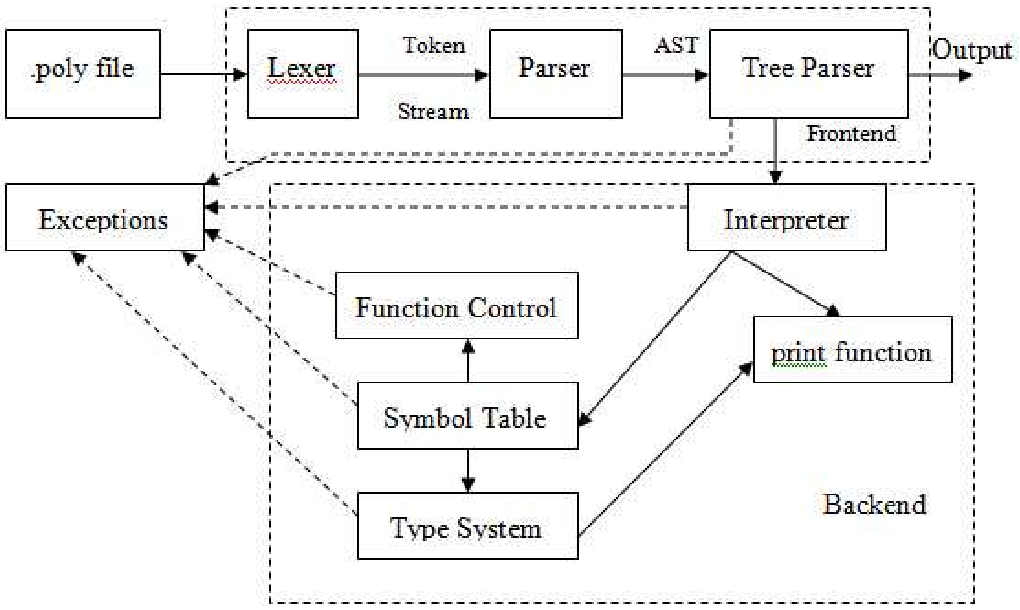

Figure 5.1: Architectural design of the Poly interpreter

The lexer and parser are implemented in the PolyGrammar.g file, and, through ANTLR, generate the necessary java code. The AST walker is implemented in PolyWalker.g, and also has its java code generated through ANTLR. PolyMain.java contains the main method which is run with a single command line argument of the file name containing the Poly function. It invokes the parser, lexer, and walker, and controls the instance of the PolyInterpreter class which is responsible for providing the interface between front-end and back-end.

The runtime environment, accessed by the walker through calls made to PolyInterpreter, contains java classes to handle each of the primary data types in Poly, including `int`s, `float`s, and `poly`s. The interpreter handles the printing of strings directly. In the case of PolyInt.java and PolyFloat.java, the classes are little more than wrapper classes over the base java data types, along with methods implemented to handle their interactions with each other, as well as with `poly`s. PolyPoly.java contains methods for operating on a java vector, which handle the basic functionality of a polynomial. The PolyFunction, found in PolyFunction.java, acts as a data type, in that it is inserted in the symbol table along with other variables, but it includes information about an additional AST, arguments, argument types, etc., and is static after declaration. PolyDataType.java contains information about the abstract class through which general functions that apply to all data types are accessed (such as print() and getName()). PolyException.java defines the class that expresses error messages, and PolySymbolTable.java implements symbol table-relevant methods.

PolyWalker.g and PolyGrammer.g were implemented by Santosh and Mohit, while PolyDataType.java, PolyException.java, PolyFloat.java, PolyInt.java, PolyPoly.java, PolyFuncProt.java, PolyFunction.java, PolyInterpreter.java, PolyMain.java, and PolySymbolTable.java were implemented by Michael.

Chapter 6

Test Plan

6.1 Testing Methods

For testing, we used a number of small tests to check the portion of the interpreter on which we were working, but we also used a regression test suite to make sure that new changes did not break any old functionality.

Once the development was done, we created some more elaborate programs to push the limits of our language, and we are pleased to say that by that point in our project, the only bugs we found were in our own POLY programs, not in the interpreter. Since these last programs are the most practical programs we have created — and since they were themselves added to the test suite — we include them here with the rest of the suite instead of by themselves. They are the files near the end of the suite which are named, rather than numbered.

The majority of the files in the test suite were created by Darrell, as were the files "testsuite" and "runtest" which actually power the suite.

6.2 Test Suite

The files in the test suite were crafted specifically to perform a systematic and incremental test of the language features as given in the Language Reference Manual.

The first two files make up the backbone of the test suite.

6.2.1 testsuite

When "make test" is called, this file is run with a single argument ("stop"). When "make testall" is called, it is called with the argument "all". The first case runs the tests in the suite in order until one fails, at which point it shows both the expected output and the experimental output, as well as a "diff" of the two and the code of the offending test program. The expected output was computed by hand and was stored in a file with a .expected extension.

```
tests='
00
01
02a
02b
02c
02
03a
03b
```

```
03c
03
04a
04b
04c
04
05a
05b
05c
05
06a
06b
06c
06
07
08
09
10
11
12
13
14
15
16
17
18
19
20
21a
21b
21c
22
23
24
25
26
PolyMult
Derivative
Integral
IntDeriv
DerivInt
Substitution
GreaterTest
UltimatePoly
CoeffAssign
'

for x in $tests ; do
if ! ../test/runtest $x $1 ; then
exit 1;
fi
```

```
done

exit

junk='
'
```

6.2.2 runtest

This script actually does the work of running a test program. It is called from the script above.

```
#!/bin/bash

test=$1
echo -n Trying test $test...
java PolyMain ../test/test$test.poly 2>&1 | grep -v ================\ program\ output\ ================
diff ../test/test$test.expected ../test/test$test.experimental > ../test/test$test.diff
LINES=`wc -l ../test/test$test.diff | cut -d\  -f1`
if [ $LINES != 0 ] ; then
    if [ $2 == "all" ] ; then
echo "    *** FAILED ***"
    else
echo
echo "!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!";
echo "Difference:";
cat ../test/test$test.diff
echo "----------------------------------------------------------------";
echo "Expected:";
cat ../test/test$test.expected
echo "----------------------------------------------------------------";
echo "Experimental:";
cat ../test/test$test.experimental
echo "----------------------------------------------------------------";
echo "Poly Code:";
cat ../test/test$test.poly
echo "!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!";
exit 1
    fi
else
    echo Success!
fi
```

6.2.3 Test case: 00

Program Source

```
// Comment test
```

Expected Output

6.2.4 Test case: 01

Program Source

```
// Test 01: Hello, World

print "Hello, World";
```

Expected Output

```
Hello, World
```

6.2.5 Test case: 02a

Program Source

```
// Test 02a: Variable declarations (int)

int a = 1;
print a;
```

Expected Output

```
a = 1
```

6.2.6 Test case: 02b

Program Source

```
// Test 02b: Variable declarations (float)

float b = 1.5;
print b;
```

Expected Output

```
b = 1.5
```

6.2.7 Test case: 02c

Program Source

```
// Test 02c: Variable declarations (poly)

poly c = [1,2,3];
print c;
```

Expected Output

```
c = 3x^2 + 2x + 1
```

6.2.8 Test case: 02

Program Source

```
// Test 02: Variable declarations

int a = 1;
float b = 1.5;
poly c = [1,2,3];

print a;
print b;
print c;
```

Expected Output

```
a = 1
b = 1.5
c = 3x^2 + 2x + 1
```

6.2.9 Test case: 03a

Program Source

```
// Test 03a: Assignment statements (int)

int a = 1;

print a;
a = 5;
print a;
```

Expected Output

```
a = 1
a = 5
```

6.2.10 Test case: 03b

Program Source

```
// Test 03b: Assignment statements (float)

float b = 1.5;

print b;
b = 3.5;
print b;
```

Expected Output

```
b = 1.5
b = 3.5
```

6.2.11 Test case: 03c

Program Source

```
// Test 03c: Assignment statements (poly)

poly c = [1,2,3];

print c;
c = [4,5,6];
print c;
```

Expected Output

```
c = 3x^2 + 2x + 1
c = 6x^2 + 5x + 4
```

6.2.12 Test case: 03

Program Source

```
// Test 03: Assignment statements

int a = 1;
float b = 1.5;
poly c = [1,2,3];

print a;
print b;
print c;

a = 5;
print a;

b = 3.5;
print b;

c = [4,5,6];
print c;
```

Expected Output

```
a = 1
b = 1.5
c = 3x^2 + 2x + 1
a = 5
b = 3.5
c = 6x^2 + 5x + 4
```

6.2.13 Test case: 04a

Program Source

```
// Test 04a: Addition (int)
```

```
int a = 1+4;
int d = 8-9;

print a;
print d;
```

Expected Output

```
a = 5
d = -1
```

6.2.14 Test case: 04b

Program Source

```
// Test 04b: Addition (float)

float b = 1.5+5.3;
float e = 4.6-3.6;

print b;
print e;
```

Expected Output

```
b = 6.8
e = 1.0
```

6.2.15 Test case: 04c

Program Source

```
// Test 04c: Addition (poly)

poly c = [1,2,3]+[3,5,2,4];
poly f = [9,8,7]-[4,2];

print c;
print f;
```

Expected Output

```
c = 4x^3 + 5x^2 + 7x + 4
f = 7x^2 + 6x + 5
```

6.2.16 Test case: 04

Program Source

```
// Test 04: Addition
```

```
int a = 1+4;
float b = 1.5+5.3;
poly c = [1,2,3]+[3,5,2,4];
int d = 8-9;
float e = 4.6-3.6;
poly f = [9,8,7]-[4,2];

print a;
print b;
print c;
print d;
print e;
print f;
```

Expected Output

```
a = 5
b = 6.8
c = 4x^3 + 5x^2 + 7x + 4
d = -1
e = 1.0
f = 7x^2 + 6x + 5
```

6.2.17 Test case: 05a

Program Source

```
// Test 05a: Multiplication (int)

int a = 1*4;
print a;
```

Expected Output

```
a = 4
```

6.2.18 Test case: 05b

Program Source

```
// Test 05b: Multiplication (float)

float b = 6*4;
float c = 5*5.3;
float d = 6.7*3;
float e = 2.9*6.3;

print b;
print c;
print d;
print e;
```

Expected Output

```
b = 24.0
c = 26.5
d = 20.1
e = 18.27
```

6.2.19 Test case: 05c

Program Source

```
// Test 05c: Multiplication (poly)

poly f = [1,2,3]*4;
poly g = 5*[6,7,8,9];
poly h = [2,3,4]*5.67;
poly i = 98.7*[6,5,43,2,1];

print f;
print g;
print h;
print i;
```

Expected Output

```
f = 12x^2 + 8x + 4
g = 45x^3 + 40x^2 + 35x + 30
h = 22.68x^2 + 17.01x + 11.34
i = 98.7x^4 + 197.4x^3 + 4244.1x^2 + 493.5x + 592.2
```

6.2.20 Test case: 05

Program Source

```
// Test 05: Multiplication

int a = 1*4;
float b = 6*4;
float c = 5*5.3;
float d = 6.7*3;
float e = 2.9*6.3;
poly f = [1,2,3]*4;
poly g = 5*[6,7,8,9];
poly h = [2,3,4]*5.67;
poly i = 98.7*[6,5,43,2,1];

print a;
print b;
print c;
print d;
print e;
print f;
print g;
print h;
print i;
```

Expected Output

```
a = 4
b = 24.0
c = 26.5
d = 20.1
e = 18.27
f = 12x^2 + 8x + 4
g = 45x^3 + 40x^2 + 35x + 30
h = 22.68x^2 + 17.01x + 11.34
i = 98.7x^4 + 197.4x^3 + 4244.1x^2 + 493.5x + 592.2
```

6.2.21 Test case: 06a

Program Source

```
// Test 06a: Division (int)

int a = 4/2;
int b = 13/4;

print a;
print b;
```

Expected Output

```
a = 2
b = 3
```

6.2.22 Test case: 06b

Program Source

```
// Test 06b: Division (float)

float c = 8/4;
float d = 4/5;
float e = 5/5.3;
float f = 6.7/3;
float g = 2.9/6.3;

print c;
print d;
print e;
print f;
print g;
```

Expected Output

```
c = 2.0
d = 0.0
```

```
e = 0.943
f = 2.233
g = 0.46
```

6.2.23 Test case: 06c

Program Source

```
// Test 06c: Division (poly)

poly h = [1,2,3]/4;
poly i = [2,3,4]/5.67;

print h;
print i;
```

Expected Output

```
h = 0
i = 0.705x^2 + 0.529x + 0.353
```

6.2.24 Test case: 06

Program Source

```
// Test 06: Division

int a = 4/2;
int b = 13/4;

float c = 8/4;
float d = 4/5;
float e = 5/5.3;
float f = 6.7/3;
float g = 2.9/6.3;

poly h = [1,2,3]/4;
poly i = [2,3,4]/5.67;

print a;
print b;
print c;
print d;
print e;
print f;
print g;
print h;
print i;
```

Expected Output

```
a = 2
b = 3
```

```
c = 2.0
d = 0.0
e = 0.943
f = 2.233
g = 0.46
h = 0
i = 0.705x^2 + 0.529x + 0.353
```

6.2.25 Test case: 07

Program Source

```
// Test 07: Modulus

int a = 4%2;
int b = 43%12;

print a;
print b;
```

Expected Output

```
a = 0
b = 7
```

6.2.26 Test case: 08

Program Source

```
// Test 08: Identifiers in Expressions

int a = 1;
int b = a;

float c = 4.0;
float d = c;

poly e = [1,2,3,0,5];
poly f = e;

print a;
print b;
print c;
print d;
print e;
print f;

a = b+a;
b = a-b;
print a;
print b;

a = a*b;
```

```
b = -a;
print a;
print b;

c = d+c;
d = c-d;
print c;
print d;

c = c*d;
d = c/d;
print c;
print d;

c = -d;
print c;

e = f+e;
f = e-f;
print e;
print f;

e = -[3,4,5];
f = -e;
print e;
print f;
```

Expected Output

```
a = 1
b = 1
c = 4.0
d = 4.0
e = 5x^4 + 3x^2 + 2x + 1
f = 5x^4 + 3x^2 + 2x + 1
a = 2
b = 1
a = 2
b = -2
c = 8.0
d = 4.0
c = 32.0
d = 8.0
c = -8.0
e = 10x^4 + 6x^2 + 4x + 2
f = 5x^4 + 3x^2 + 2x + 1
e = -5x^2 + -4x + -3
f = 5x^2 + 4x + 3
```

6.2.27 Test case: 09

Program Source

```
// Test 09: Poly Concatentation

poly e = [1,2,3,0,5];
poly f = [6,3,7.6,-8.442];
poly g = e:[1];
poly h = [2]:f;
poly i = g:h;
poly j = [1,2]:[3,4];

print e;
print f;
print g;
print h;
print i;
print j;

g = f:[1];
h = [2]:e;
i = h:g;
j = [6,9]:[12,15];

print g;
print h;
print i;
print j;
```

Expected Output

```
e = 5x^4 + 3x^2 + 2x + 1
f = -8.442x^3 + 7.6x^2 + 3x + 6
g = x^5 + 5x^4 + 3x^2 + 2x + 1
h = -8.442x^4 + 7.6x^3 + 3x^2 + 6x + 2
i = -8.442x^10 + 7.6x^9 + 3x^8 + 6x^7 + 2x^6 + x^5 + 5x^4 + 3x^2 + 2x + 1
j = 4x^3 + 3x^2 + 2x + 1
g = x^4 + -8.442x^3 + 7.6x^2 + 3x + 6
h = 5x^5 + 3x^3 + 2x^2 + x + 2
i = x^10 + -8.442x^9 + 7.6x^8 + 3x^7 + 6x^6 + 5x^5 + 3x^3 + 2x^2 + x + 2
j = 15x^3 + 12x^2 + 9x + 6
```

6.2.28 Test case: 10

Program Source

```
// Test 10: Poly Order

poly e = [1,2,3,0,5];
poly f = [6];
poly g = [3,5,4];
```

```
float a = |e|;
float b = |f|;
float c = |g|;

print a;
print |e|;
print b;
print |f|;
print c;
print |g|;
```

Expected Output

```
a = 4.0
4
b = 0.0
0
c = 2.0
2
```

6.2.29 Test case: 11

Program Source

```
// Test 11: Poly Coefficient Extraction

poly e = [1,2,3,0,5];
poly f = [6];
poly g = [3,5,4];
float a = e[2];
float b = f[0];
int d = 2;
float c = g[d];

print a;
print e[2];
print b;
print f[0];
print c;
print g[d];
```

Expected Output

```
a = 3.0
3
b = 6.0
6
c = 4.0
4
```

6.2.30 Test case: 12

Program Source

```
// Test 12: if, if-else

int a = 1;
float b = 2.4;
poly c = [a,b];

print a;
print b;
print c;

if (a == 1) {
  print "a is 1";
}

if (a > 0) {
  print "a > 0";
}

if (a < 2) {
  print "a < 2";
}

if (a != 1) {
  print "Never get here.";
}

if (b == 2.4) {
  print "b is 2.4";
} else {
  print "Never get here.";
}

if (b > 2) {
  print "b > 2";
}

if (b < 3) {
  print "b < 3";
}

if (b != 5) {
  print "b != 5";
}

if (c != [1,2.4]) {
  print "Never get here.";
} else {
  print "c is [1,2.4]";
}

if (c == [1,2.4]) {
  print "c is correct";
}

if (c > [100,2]) {
  print "c > x + 100";
}

if (c < [-100,3]) {
  print "c < 3x -100";
}
```

Expected Output

```
a = 1
b = 2.4
c = 2.4x + 1
a is 1
a > 0
a < 2
b is 2.4
b > 2
b < 3
b != 5
c is [1,2.4]
c is correct
c > x + 100
c < 3x -100
```

6.2.31 Test case: 13

Program Source

```
// Test 13: while

int a = 1;
float b = 2.4;
poly c = [a,b];

print a;
print b;
print c;

while (a < 4) {
  print a;
  a = a + 1;
}

while (b > 1) {
  print b;
  b = b - 1;
}

while (c > [1,1]) {
  print c;
  c = c - [0.5,0.5];
}
```

Expected Output

```
a = 1
b = 2.4
c = 2.4x + 1
a = 1
a = 2
a = 3
b = 2.4
b = 1.4
c = 2.4x + 1
c = 1.9x + 0.5
c = 1.4x
```

6.2.32 Test case: 14

Program Source

```
// Test 14: and, or, not

int a = 1;
float b = 2.4;
poly c = [a,b];

print a;
print b;
print c;

if (a == 1 and b == 2.4) {
  print "1 true";
}

if (a == 1 and b == 2.0) {
  print "2 true";
}

if (a == 0 and b == 2.4) {
  print "3 true";
}

if (a == 0 and b == 2.0) {
  print "4 true";
}
```

```
if (a == 1 or b == 2.4) {
  print "5 true";
}

if (a == 1 or b == 2.0) {
  print "6 true";
}

if (a == 0 or b == 2.4) {
  print "7 true";
}

if (a == 0 or b == 2.0) {
  print "8 true";
}

if (not a == 1) {
  print "9 true";
}

if (not a == 1 and b == 2.4) {
  print "10 true";
}

if (not a == 1 and b == 2.0) {
  print "11 true";
}

if (not a == 0 and b == 2.4) {
  print "12 true";
}

if (not a == 0 and b == 2.0) {
  print "13 true";
}

if (a == 1 or not b == 2.4) {
  print "14 true";
}

if (a == 1 or not b == 2.0) {
  print "15 true";
}

if (a == 0 or not b == 2.4) {
  print "16 true";
}

if (a == 0 or not b == 2.0) {
  print "17 true";
}
```

Expected Output

```
a = 1
b = 2.4
c = 2.4x + 1
1 true
5 true
6 true
7 true
12 true
14 true
15 true
17 true
```

6.2.33 Test case: 15

Program Source

```
// Test 15: void functions

prototype void hw ();

function void hw () {
  print "two";
}

print "one plus";
hw();
print "minus one equals";
hw();
```

Expected Output

```
one plus
two
minus one equals
two
```

6.2.34 Test case: 16

Program Source

```
// Test 16: int,float,poly functions

prototype int ifunc ();
prototype float ffunc ();
prototype poly pfunc ();

function int ifunc () {
  int out = 5;
  return out;
}

function float ffunc () {
  float out = 6.6;
  return out;
}

function poly pfunc () {
  poly out = [3,4.5];
  return out;
}

int a = ifunc();
float b = ffunc();
poly c = pfunc();

print a;
print b;
print c;

a = 6+ifunc();
b = 7.8-ffunc();
c = [4,2]+pfunc();

print a;
print b;
print c;

print |pfunc()|;
```

Expected Output

```
a = 5
b = 6.6
c = 4.5x + 3
a = 11
b = 1.2
c = 6.5x + 7
1
```

6.2.35 Test case: 17

Program Source

```
// Test 17: void functions with arguments - local vars

prototype void ifunc (int);
prototype void ffunc (float);
prototype void pfunc (poly);
prototype void iffunc (int, float);
prototype void fpfunc (float, poly);

function void ifunc (int a) {
  int aa = 45;
  print a;
  print aa;
}

function void ffunc (float b) {
  float bb = 4.5;
  print b;
  print bb;
}

function void pfunc (poly c) {
  poly cc = [2.5,1.0];
  print c;
  print cc;
}

function void iffunc (int d, float e) {
  int dd = 5+d;
  float ee = 1.9*e;
  print d;
  print dd;
  print e;
  print ee;
}

function void fpfunc (float e, poly f) {
  float ee = f[2];
  poly ff = f:[f[0]];
  print e;
  print ee;
  print f;
  print ff;
}

ifunc(3);
ffunc(2.3);
pfunc([34,5.0]);
iffunc(10,8.5);
fpfunc(4.5,[42,|[3,4,5]|,0.37]);
```

Expected Output

```
a = 3
aa = 45
b = 2.3
bb = 4.5
c = 5.0x + 34
cc = x + 2.5
```

```
d = 10
dd = 15
e = 8.5
ee = 16.15
e = 4.5
ee = 0.37
f = 0.37x^2 + 2x + 42
ff = 42x^3 + 0.37x^2 + 2x + 42
```

6.2.36 Test case: 18

Program Source

```
// Test 18: int functions with arguments - local vars

prototype int ifunc (int);
prototype int ffunc (float);
prototype int pfunc (poly);
prototype int iffunc (int, float);
prototype int fpfunc (float, poly);

function int ifunc (int a) {
  int aa = 45;
  print a;
  print aa;
  return aa;
}

function int ffunc (float b) {
  int bb = 4;
  print b;
  print bb;
  return bb;
}

function int pfunc (poly c) {
  poly cc = [2.5,1.0];
  print c;
  print cc;
  return |cc|;
}

function int iffunc (int d, float e) {
  int dd = 5+d;
  float ee = 1.9*e;
  print d;
  print dd;
  print e;
  print ee;
  return dd;
}
```

```
function int fpfunc (float e, poly f) {
  float ee = f[2];
  poly ff = f:[f[0]];
  print e;
  print ee;
  print f;
  print ff;
  return 0;
}

int a = ifunc(3);
int b = ffunc(2.3);
int c = 0;

print a;
print b;

c = pfunc([34,5.0]);
print c;
c = iffunc(10,8.5);
print c;
c = fpfunc(4.5,[42,|[3,4,5]|,0.37]);
print c;
```

Expected Output

```
a = 3
aa = 45
b = 2.3
bb = 4
a = 45
b = 4
c = 5.0x + 34
cc = x + 2.5
c = 1
d = 10
dd = 15
e = 8.5
ee = 16.15
c = 15
e = 4.5
ee = 0.37
f = 0.37x^2 + 2x + 42
ff = 42x^3 + 0.37x^2 + 2x + 42
c = 0
```

6.2.37 Test case: 19

Program Source

```
// Test 19: float functions with arguments - local vars
```

```
prototype float ifunc (int);
prototype float ffunc (float);
prototype float pfunc (poly);
prototype float iffunc (int, float);
prototype float fpfunc (float, poly);

function float ifunc (int a) {
  int aa = 45;
  print a;
  print aa;
  return aa;
}

function float ffunc (float b) {
  float bb = 4.5;
  print b;
  print bb;
  return bb;
}

function float pfunc (poly c) {
  poly cc = [2.5,1.0];
  print c;
  print cc;
  return |cc|;
}

function float iffunc (int d, float e) {
  int dd = 5+d;
  float ee = 1.9*e;
  print d;
  print dd;
  print e;
  print ee;
  return ee;
}

function float fpfunc (float e, poly f) {
  float ee = f[2];
  poly ff = f:[f[0]];
  print e;
  print ee;
  print f;
  print ff;
  return ee;
}

float a = ifunc(3);
float b = ffunc(2.3);
float c = 0;
```

```
print a;
print b;

c = pfunc([34,5.0]);
print c;
c = iffunc(10,8.5);
print c;
c = fpfunc(4.5,[42,|[3,4,5]|,0.37]);
print c;
```

Expected Output

```
a = 3
aa = 45
b = 2.3
bb = 4.5
a = 45.0
b = 4.5
c = 5.0x + 34
cc = x + 2.5
c = 1.0
d = 10
dd = 15
e = 8.5
ee = 16.15
c = 16.15
e = 4.5
ee = 0.37
f = 0.37x^2 + 2x + 42
ff = 42x^3 + 0.37x^2 + 2x + 42
c = 0.37
```

6.2.38 Test case: 20

Program Source

```
// Test 20: poly functions with arguments - local vars

prototype poly ifunc (int);
prototype poly ffunc (float);
prototype poly pfunc (poly);
prototype poly iffunc (int, float);
prototype poly fpfunc (float, poly);

function poly ifunc (int a) {
  int aa = 45;
  print a;
  print aa;
  return [a,aa];
}

function poly ffunc (float b) {
  float bb = 4.5;
  print b;
  print bb;
  return [b,bb];
}

function poly pfunc (poly c) {
  poly cc = [2.5,1.0];
  print c;
  print cc;
  return c:[|cc|]:cc;
}

function poly iffunc (int d, float e) {
  int dd = 5+d;
  float ee = 1.9*e;
  print d;
  print dd;
  print e;
  print ee;
  return [d,dd,e,ee];
}

function poly fpfunc (float e, poly f) {
  float ee = f[2];
  poly ff = f:[f[0]];
  print e;
  print ee;
  print f;
  print ff;
  return ff;
}

poly a = ifunc(3);
poly b = ffunc(2.3);
poly c = [0];

print a;
print b;

c = pfunc([34,5.0]);
print c;
c = iffunc(10,8.5);
print c;
c = fpfunc(4.5,[42,|[3,4,5]|,0.37]);
print c;
```

Expected Output

```
a = 3
aa = 45
b = 2.3
```

```
bb = 4.5
a = 45x + 3
b = 4.5x + 2.3
c = 5.0x + 34
cc = x + 2.5
c = x^4 + 2.5x^3 + x^2 + 5.0x + 34
d = 10
dd = 15
e = 8.5
ee = 16.15
c = 16.15x^3 + 8.5x^2 + 15x + 10
e = 4.5
ee = 0.37
f = 0.37x^2 + 2x + 42
ff = 42x^3 + 0.37x^2 + 2x + 42
c = 42x^3 + 0.37x^2 + 2x + 42
```

6.2.39 Test case: 21a

Program Source

```
// Test 21a: Deep function calls (int)

prototype int afunc (int);
prototype int bfunc (int);
prototype int cfunc (int);
prototype int dfunc (int);
prototype int efunc (int);
prototype int ffunc (int);

function int afunc (int a) {
  return a+4;
}

function int bfunc (int a) {
  return a*3;
}

function int cfunc (int a) {
  return a%10;
}

function int dfunc (int a) {
  return a*a;
}

function int efunc (int a) {
  return -a;
}

function int ffunc (int a) {
  return a-1;
}

print afunc(bfunc(cfunc(dfunc(efunc(ffunc(-18))))));
```

Expected Output

```
7
```

6.2.40 Test case: 21b

Program Source

```
// Test 21b: Deep function calls (float)

prototype float afunc (float);
prototype float bfunc (float);
prototype float cfunc (float);
prototype float dfunc (float);
prototype float efunc (float);
prototype float ffunc (float);

function float afunc (float a) {
  return a+3.8;
}

function float bfunc (float a) {
  return a*0.2;
}

function float cfunc (float a) {
  float b = 0.93;
  print a;
  print b;
  if(a != b) {
    return b;
  }
}

function float dfunc (float a) {
  return a/0.5;
}

function float efunc (float a) {
  return -a;
}

function float ffunc (float a) {
  return a-(-42.6);
}

print afunc(bfunc(cfunc(dfunc(efunc(ffunc(-7.2))))));
```

Expected Output

```
a = -70.8
b = 0.93
3.986
```

6.2.41 Test case: 21c

Program Source

```
// Test 21c: Deep function calls (poly)

prototype float afunc (poly);
prototype poly bfunc (poly);
prototype poly cfunc (poly);
prototype poly dfunc (poly);
prototype poly efunc (poly);
prototype poly ffunc (poly);

function float afunc (poly a) {
  print a;
  return a[3];
}

function poly bfunc (poly a) {
  print a;
  return a*3;
}

function poly cfunc (poly a) {
  print a;
  a = a*1.0;
  return a/10;
}

function poly dfunc (poly a) {
  print a;
  return a:[3,7];
}

function poly efunc (poly a) {
  print a;
  return -a;
}

function poly ffunc (poly a) {
  print a;
  return a+4;
}

print afunc(bfunc(cfunc(dfunc(efunc(ffunc([1,2,5]))))));
```

Expected Output

```
a = 5x^2 + 2x + 1
a = 5x^2 + 2x + 5
a = -5x^2 + -2x + -5
a = 7x^4 + 3x^3 + -5x^2 + -2x + -5
a = 0.7x^4 + 0.3x^3 + -0.5x^2 + -0.2x + -0.5
a = 2.1x^4 + 0.9x^3 + -1.5x^2 + -0.6x + -1.5
0.9
```

6.2.42 Test case: 22

Program Source

```
// Test 22: Recursive functions (if, if-else, and while within function bodies

prototype int afunc (int);
prototype int bfunc (int);
prototype int cfunc (int);

function int afunc (int a) {
  if (a > 0) {
    return bfunc(a-1);
  } else {
    return a;
  }
}

function int bfunc (int a) {
  int c = 0;
  print a;
  while (c < a) {
    print c;
    c = c+1;
  }
  return afunc(a);
}

function int fact (int a) {
  if (a == 1) {
    return 1;
  }
  return a*fact(a-1);
}

print afunc(5);
print fact(4);
```

Expected Output

```
a = 4
c = 0
c = 1
```

```
c = 2
c = 3
a = 3
c = 0
c = 1
c = 2
a = 2
c = 0
c = 1
a = 1
c = 0
a = 0
0
24
```

6.2.43 Test case: 23

Program Source

```
prototype float nestedWhileIf();

function float nestedWhileIf()
{
   int a = 3;
   int b = 2;
   int c = 1;

   while (a < 6)
   {
      while (b < a)
      {
         while (c < b)
         {
            if (a+b+c == 9 )
            {
               return a+b+c;
            }
            c = c + 1;
         }
         b = b + 1;
      }
      a = a + 1;
      //b = b-1;
   }
   return a;
}

print nestedWhileIf();
```

Expected Output

```
9.0
```

6.2.44 Test case: 24

Program Source

```
int b = 7;
float f = 2.0;
poly p = [2,8,10];
poly resultP = p/(f+(-f-f)*f/f);
print resultP;

print resultP + b%|resultP|;
```

Expected Output

```
resultP = -5.0x^2 + -4.0x + -1
-5.0x^2 + -4.0x
```

6.2.45 Test case: 25

Program Source

```
print """Double quoted string""";
print "Many quotes: """""""" ";
```

Expected Output

```
"Double quoted string"
Many quotes: """"
```

6.2.46 Test case: 26

Program Source

```
prototype float global();

function float global()
{
   return a;
}

int a = 4;
print global();
```

Expected Output

```
4.0
```

6.2.47 Test case: PolyMult

Program Source

```
//PolyMultiplication
prototype poly polyMult (poly, poly);

function poly polyMult (poly A, poly B) {
  poly C=[0];
  poly T=[0];
  int x=0;
  int s=0;

  while(x != (|A|+1)) {
    T = A[x] * B;
    s = x;
    while(s!=0) {
      T = [0]:T;
      s = s-1;
    }
    C = C+T;
    x = x+1;
  }

  return C;
}

print polyMult([5,2,7,2,3], [4,1,0,6,3,2.2]);
```

Expected Output

```
6.6x^9 + 13.4x^8 + 39.4x^7 + 37.4x^6 + 62.0x^5 + 41x^4 + 45x^3 + 30x^2 + 13x + 20
```

6.2.48 Test case: Derivative

Program Source

```
//Derivitives
prototype poly derivative(poly);

function poly derivative(poly A) {
  int x=0;
  float y=0.0;
  poly B=[0];
  int z=0;
  poly C=[0];
  if(|A|==0) {
    return [0];
  }
  if(|A|!=0) {
    B=[A[1]];
    x=1;
    while(x!=|A|) {
      x=x+1;
      y=A[x]*x;
      if(y==0) {
        z=z+1;
      } else {
        C = [y];
        while(z>0) {
```

```
        C = 0:C;
        z=z-1;
      }
      B=B:C;
    }
  }
}
return B;
}

print derivative([5,2,0,0,7,2,3,4.23,0,6.4,2.5]);
```

Expected Output

```
25.0x^9 + 57.6x^8 + 29.61x^6 + 18.0x^5 + 10.0x^4 + 28.0x^3 + 2
```

6.2.49 Test case: Integral

Program Source

```
//integral
prototype poly integral(poly);

function poly integral(poly A) {
  poly B=[0];
  int x=1;
  float y=0;
  float z=0.0;
  int p=0;
  int startyet = 0;
  poly C=[0];
  poly D=[0];
  while(|A|+1!=x) {
    y=x+1;
    z=A[x]/y;
    if(z==0) {
      p=p+1;
    } else {
      C = [z];
      while(p>0) {
C=0:C;
p=p-1;
      }
      if((x==1) or (startyet==0)) {
D=A[0]:C;
startyet = 1;
B=0:D;
      } else {
B=B:C;
      }
    }
    x=x+1;
  }

  return B;
}

//print integral([0,4,-4]);
//print integral([5,2]);
print integral([5,2,0,0,7,2,3,4.23,0,6.4,2.5]);

//2.5x^10 + 6.4x^9 + 4.23x^7 + 3x^6 + 2x^5 + 7x^4 + 2x + 5
```

Expected Output

```
0.227x^11 + 0.64x^10 + 0.529x^8 + 0.429x^7 + 0.333x^6 + 1.4x^5 + x^2 + 5x
```

6.2.50 Test case: IntDeriv

Program Source

```
prototype poly integral(poly);
prototype poly derivative(poly);

function poly integral(poly A) {
  poly B=[0];
  int x=1;
  float y=0;
  float z=0.0;
  int p=0;
  int startyet = 0;
  poly C=[0];
  poly D=[0];
  while(|A|+1!=x) {
    y=x+1;
    z=A[x]/y;
    if(z==0) {
      p=p+1;
    } else {
      C = [z];
      while(p>0) {
C=0:C;
p=p-1;
      }
      if((x==1) or (startyet==0)) {
D=A[0]:C;
startyet = 1;
B=0:D;
      } else {
B=B:C;
      }
    }
    x=x+1;
  }
  return B;
}

function poly derivative(poly A) {
  int x=0;
  float y=0.0;
  poly B=[0];
  int z=0;
  poly C=[0];
  if(|A|==0) {
    return [0];
  }
  if(|A|!=0) {
    B=[A[1]];
    x=1;
    while(x!=|A|) {
      x=x+1;
      y=A[x]*x;
      if(y==0) {
z=z+1;
      } else {
        C = [y];
while(z>0) {
  C = 0:C;
          z=z-1;
        }
        B=B:C;
      }
    }
  }
  return B;
}

//print derivative([5,2,0,0,7,2,3,4.23,0,6.4,2.5]);
//print integral([5,2,0,0,7,2,3,4.23,0,6.4,2.5]);
//print integral(derivative([5,2,0,0,7,2,3,4.23,0,6.4,2.5]));
print "original = ",[5,2,0,0,7,2,3,4.23,0,6.4,2.5];
print "new = ",integral(derivative([5,2,0,0,7,2,3,4.23,0,6.4,2.5]));
```

Expected Output

```
original =
2.5x^10 + 6.4x^9 + 4.23x^7 + 3x^6 + 2x^5 + 7x^4 + 2x + 5
new =
2.5x^10 + 6.4x^9 + 4.23x^7 + 3.0x^6 + 2.0x^5 + 7.0x^4 + 2x
```

6.2.51 Test case: DerivInt

Program Source

```
prototype poly integral(poly);
prototype poly derivative(poly);

function poly integral(poly A) {
  poly B=[0];
  int x=1;
  float y=0;
  float z=0.0;
  int p=0;
  int startyet = 0;
  poly C=[0];
  poly D=[0];
  while(|A|+1!=x) {
    y=x+1;
    z=A[x]/y;
    if(z==0) {
      p=p+1;
    } else {
      C = [z];
      while(p>0) {
C=0:C;
p=p-1;
      }
      if((x==1) or (startyet==0)) {
D=A[0]:C;
startyet = 1;
B=0:D;
      } else {
B=B:C;
      }
    }
    x=x+1;
  }

  return B;
}

function poly derivative(poly A) {
  int x=0;
  float y=0.0;
  poly B=[0];
  int z=0;
  poly C=[0];
  if(|A|==0) {
    return [0];
  }
  if(|A|!=0) {
    B=[A[1]];
    x=1;
    while(x!=|A|) {
      x=x+1;
      y=A[x]*x;
      if(y==0) {
z=z+1;
      } else {
        C = [y];
while(z>0) {
  C = 0:C;
          z=z-1;
        }
        B=B:C;
      }
    }
  }
  return B;
}

print "original = ",[5,2,0,0,7,2,3,4.23,0,6.4,2.5];
print "new = ",derivative(integral([5,2,0,0,7,2,3,4.23,0,6.4,2.5]));
```

Expected Output

```
original =
2.5x^10 + 6.4x^9 + 4.23x^7 + 3x^6 + 2x^5 + 7x^4 + 2x + 5
new =
2.5x^10 + 6.4x^9 + 4.23x^7 + 3.0x^6 + 2.0x^5 + 7.0x^4 + 2.0x + 5
```

6.2.52 Test case: Substitution

Program Source

```
//Substitution
prototype float substitute(poly, float);

function float substitute(poly A, float x) {
  int p = 1;
  int q = 0;
  float sum = 0.0;
  float pow = 1.0;

  while(p<|A|+1){
    q=0;
    pow = 1;
    while(q<p){
      pow = pow * x;
      q = q+1;
    }
    sum = sum+(pow*A[p]);
    p = p+1;
  }

  return sum+A[0];
}
```

```
print substitute([5,-2,0,0,7,2,3,4.23,0,6.4,2.5], 0.5);
```

Expected Output

```
4.595
```

6.2.53 Test case: GreaterTest

Program Source

```
poly A=[1,2,3,4,5,6,7,8];
poly B=[1,9,2,8,3,7,4,6,5];
poly C=[1,2,3,4,5];
poly D=[1,2,4,3,5];
poly E=[1.2,2.3,3.4,3.5,5.6];
poly F=[1.2,2.3,4.5,3.4,5.6];
poly G=[1.2,2.3,3.4,3.4,5.6];
poly H=[1.2,2.3,4.5,3.4,5.6];

print A;
print B;

if(A>B) {
  print "a is greater";
} else {
  print "b is greater";
}

B=B:(-1);

print A;
print B;

if(A>B) {
  print "a is greater";
} else {
  print "b is greater";
}

print C;
print D;

if(C>D) {
  print "c is greater";
} else {
  print "d is greater";
}

print E;
print F;

if(E>F) {
  print "e is greater";
} else {
  print "f is greater";
}

print G;
print H;

if(G>H) {
  print "g is greater";
} else {
  print "h is greater";
}

G=G:1;

print G;
print H;

if(G>H) {
  print "g is greater";
} else {
  print "h is greater";
}

G=G:(-1);

print G;
print H;

if(G>H) {
  print "g is greater";
} else {
  print "h is greater";
}
```

Expected Output

```
A = 8x^7 + 7x^6 + 6x^5 + 5x^4 + 4x^3 + 3x^2 + 2x + 1
B = 5x^8 + 6x^7 + 4x^6 + 7x^5 + 3x^4 + 8x^3 + 2x^2 + 9x + 1
b is greater
A = 8x^7 + 7x^6 + 6x^5 + 5x^4 + 4x^3 + 3x^2 + 2x + 1
B = -x^9 + 5x^8 + 6x^7 + 4x^6 + 7x^5 + 3x^4 + 8x^3 + 2x^2 + 9x + 1
a is greater
C = 5x^4 + 4x^3 + 3x^2 + 2x + 1
D = 5x^4 + 3x^3 + 4x^2 + 2x + 1
c is greater
E = 5.6x^4 + 3.5x^3 + 3.4x^2 + 2.3x + 1.2
F = 5.6x^4 + 3.4x^3 + 4.5x^2 + 2.3x + 1.2
e is greater
G = 5.6x^4 + 3.4x^3 + 3.4x^2 + 2.3x + 1.2
H = 5.6x^4 + 3.4x^3 + 4.5x^2 + 2.3x + 1.2
```

```
h is greater
G = x^5 + 5.6x^4 + 3.4x^3 + 3.4x^2 + 2.3x + 1.2
H = 5.6x^4 + 3.4x^3 + 4.5x^2 + 2.3x + 1.2
g is greater
G = -x^6 + x^5 + 5.6x^4 + 3.4x^3 + 3.4x^2 + 2.3x + 1.2
H = 5.6x^4 + 3.4x^3 + 4.5x^2 + 2.3x + 1.2
h is greater
```

6.2.54 Test case: UltimatePoly

Program Source

```
prototype float substitute(poly, float);
prototype poly integral(poly);
prototype float defIntegral(poly, float, float);
prototype poly derivative(poly);

function float substitute(poly A, float x) {
  int p = 1;
  int q = 0;
  float sum = 0.0;
  float pow = 1.0;

  while(p<|A|+1){
    q=0;
    pow = 1;
    while(q<p){
      pow = pow * x;
      q = q+1;
    }
    sum = sum+(pow*A[p]);
    p = p+1;
  }

  return sum+A[0];
}

function poly integral(poly A) {
  poly B=[0];
  int x=1;
  float y=0;
  float z=0.0;
  int p=0;
  int startyet = 0;
  poly C=[0];
  poly D=[0];
  while(|A|+1!=x) {
    y=x+1;
    z=A[x]/y;
    if(z==0) {
      p=p+1;
    } else {
      C = [z];
      while(p>0) {
C=0:C;
p=p-1;
      }
      if((x==1) or (startyet==0)) {
D=A[0]:C;
startyet = 1;
B=0:D;
      } else {
B=B:C;
      }
    }
    x=x+1;
  }

  return B;
}

function float defIntegral(poly A, float x, float y) {
  poly B = integral(A);
  return substitute(B, y) - substitute(B, x);
}

function poly derivative(poly A) {
  int x=0;
  float y=0.0;
  poly B=[0];
  int z=0;
  poly C=[0];
  if(|A|==0) {
    return [0];
  }
  if(|A|!=0) {
    B=[A[1]];
    x=1;
    while(x!=|A|) {
      x=x+1;
      y=A[x]*x;
      if(y==0) {
z=z+1;
      } else {
        C = [y];
while(z>0) {
  C = 0:C;
          z=z-1;
        }
        B=B:C;
      }
    }
  }
  return B;
}

poly A = [0,4,-4];

print substitute([0,0,2,-1.333],4.0);
print integral(A);
print substitute(integral(A), 4.0);
print substitute(integral(A), 0.0);
print defIntegral(A, 0.0, 4.0);
```

Expected Output

```
-53.312
-1.333x^3 + 2.0x^2
-53.333
0.0
-53.333
```

6.2.55 Test case: CoeffAssign

Program Source

```
//testCoeffAssign

prototype poly coeffAssign(poly, int, float);

function poly coeffAssign(poly A, int x, float p) {
  int y = x+1;
  int z = |A|;
  poly B = [0];

  while(not (z<y)){
    B = A[z]:B;
    z=z-1;
  }

  B = p:B;
  z = z-1;

  while(not (z<0)) {
    B = A[z]:B;
    z=z-1;
  }

  return B;

}

poly A = [0,1,2,0,3,0,4,5,0];
float x = -99.0;
```

```
print A;
print "replacing coefficient:", A[4];
print "with: ", x;
print coeffAssign(A, 4, x);
```

Expected Output

```
A = 5x^7 + 4x^6 + 3x^4 + 2x^2 + x
replacing coefficient:
3
with:
x = -99.0
5x^7 + 4x^6 + -99.0x^4 + 2x^2 + x
```

Chapter 7

Lessons Learned

We had a great time working on Poly and are quite proud of the result. We reached this point, however, only after learning much about how to go about creating a new programming language, and there are a number of things we wish we could have gone back in time to advise ourselves as we started.

- Keep it simple, Stupid. This is perhaps one of the defining principles of our project. We feel that what we lack, if anything, in terms of glitz we make up for in reliability, elegance, and functionality.
- Save the easy stuff for home. During our meetings, we'd often come up with ideas which were simple to understand and not difficult, albeit time-consuming, to implement. Rather than spend our meeting time working on those things, it would have been smarter to make such a list for each member, for them to complete by the time of the next meeting.
- Build incrementally. At first it seemed as if everything had to be working in order for any of it to work. Looking back, we realize that if we had immediately worked toward being able to run the simplest program we could think of, we could have spent less time debugging.
- In the end, the concept of a compiler can translate into a deceptively large amount of coding. If we had a better idea of the amount of sheer coding our project would entail, we probably would have spent more non-meeting time working on it.
- Be sure to brush up on the languages being used before you start. Too much of our time was wasted doing nothing other than trying to understand ANTLR syntax. By the time we had it able to recognize "Hello world", we could have had many other elements working.
- We did not realize until after we had done significant coding that adhering to a strict coding style was both necessary and more efficient.

Bibliography

[1] Maple, http://www.maplesoft.com

[2] Mathematica, http://www.wolfram.com/products/mathematica

[3] Matlab, http://www.mathworks.com

[4] http://www.rit.edu/~pnveme/pigf/Polynomials/poly_mat_convol.html

[5] ANTLR, http://www.antlr.org

Appendix A

Interpreter Code

A.1 ANTLR Code

A.1.1 PolyGrammar.g

```
/* Contains PolyLexer and PolyParser classes.
 *
 * Santosh Thammana - st2273@columbia.edu
 *
 */
class PolyLexer extends Lexer;

options
{
    testLiterals = false;
    charVocabulary = '\3'..'\377';
    exportVocab = Poly;
    k = 2;
}

tokens
{
    FLOAT;
    INTEGER;
}

{
    int nr_error = 0;

    public void reportError( String s )
    {
        super.reportError( s );
        nr_error++;
    }

    public void reportError( RecognitionException e )
    {
        super.reportError( e );
        nr_error++;
    }
}

PLUS     : '+' ;
MINUS    : '-' ;
TIMES    : '*' ;
MOD      : '%' ;
DIV      : '/' ;
ASSIGN   : '=' ;
EQ       : "==";
LT       : '<' ;
GT       : '>' ;
NEQ      : "!=";
COMMA    : ',' ;
DEGREE   : '|' ;
CONCAT   : ':' ;
LBRACKET : '[' ;
RBRACKET : ']' ;
LBRACE   : '{' ;
RBRACE   : '}' ;
LPAREN   : '(' ;
RPAREN   : ')' ;
SEMI     : ';' ;
DEC      : '.' ;

protected LETTER : ('a'..'z' | 'A'..'Z') ;
protected DIGIT  : '0'..'9' ;

ID  options
    {
        testLiterals = true;
    }
    : LETTER (LETTER | DIGIT | '_')* ;

NUMBER  :   (DIGIT)+
            (   '.' (DIGIT)+
                {
                    $setType(FLOAT);
                }
            |
                {
                    $setType(INTEGER);
                }
            )
        ;

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

WS  :   ( ' ' | '\t' | '\n' { newline(); } | '\r' )
        {
            $setType(Token.SKIP);
        }
    ;

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

class PolyParser extends Parser;

options
{
    buildAST = true;
    k = 2;
    exportVocab = Poly;
}

tokens
{
    FILE;
    ARG;
    ARGS;
    TYPELIST;
    FCARGS;
    STMTLIST;
    BLOCK;
    FUNCCALL;
    POLYCONST;
}

{
    int nr_error = 0;

    public void reportError( String s )
    {
        super.reportError( s );
        nr_error++;
    }

    public void reportError( RecognitionException e )
    {
        super.reportError( e );
        nr_error++;
    }
}

file    :   (proto)* (func)* stmtList EOF!
            {
                #file = #([FILE,"file"], file);
            }
        ;

protected type : "int" | "poly" | "float" ;
protected returnType : type | "void" ;

proto : "prototype"^ returnType ID typeList SEMI! ;

protected typeList  :   LPAREN! (type (COMMA! type)*)? RPAREN!
                        {
                            #typeList = #([TYPELIST,"typeList"], typeList);
                        }
                    ;

func    :   "function"^ returnType ID args
            LBRACE!
            stmtList
            RBRACE!
        ;

protected arg   :   type ID
                    {
                        #arg = #([ARG,"arg"], arg);
                    }
                ;

protected args  :   LPAREN! (arg (COMMA! arg)*)? RPAREN!
                    {
                        #args = #([ARGS,"args"], args);
                    }
                ;

protected fcargs    :   LPAREN! (expression (COMMA! expression)*)? RPAREN!
                        {
                            #fcargs = #([FCARGS,"fcargs"], fcargs);
                        }
                    ;

stmt : blockStmt | (lineStmt SEMI!) ;

protected stmtList  :   (varDecStmt SEMI!)*
                        (stmt)*
                        {
                            #stmtList = #([STMTLIST,"stmtlist"], stmtList);
                        }
                    ;
```

```
blockStmt : ifStmt | whileStmt ;

varDecStmt : type ID ASSIGN^ expression ;

protected block :   LBRACE! (stmt)* RBRACE!
                    {
                        #block = #([BLOCK,"block"], block);
                    }
                ;

ifStmt : "if"^ LPAREN! expression RPAREN! block ("else"! block)? ;

whileStmt : "while"^ LPAREN! expression RPAREN! block ;

lineStmt    :   "print"^ printExpr (COMMA! printExpr)*
            |   "return"^ (expression)?
            |   ID ASSIGN^ expression
            |   funcCall
            ;

printExpr : (expression | STRING) ;

funcCall    :   ID fcargs
                {
                    #funcCall = #([FUNCCALL,"funcCall"], funcCall);
                }
            ;

polyConst   :   LBRACKET! expression (COMMA! expression)* RBRACKET!
                {
                    #polyConst = #([POLYCONST,"polyConst"], polyConst);
                }
            ;

expression  : expr2 ( "or"^ expr2 )* ;
expr2       : expr3 ( "and"^ expr3 )* ;
expr3       : ( "not"^ expr3 ) | expr4 ;
expr4       : expr5 ( (EQ^ | NEQ^ | LT^ | GT^) expr5)? ;
expr5       : expr6 ( (PLUS^ | MINUS^) expr6 )* ;
expr6       : expr7 ( (TIMES^ | MOD^ | DIV^) expr7 )* ;
expr7       : expr8 ( CONCAT^ expr8 )* ;
expr8       : MINUS^ expr9 | expr9 ;
expr9       : funcCall
            | ID (LBRACKET^ expression RBRACKET!)?
            | DEGREE^ expression DEGREE!
            | LPAREN! expression RPAREN!
            | INTEGER
            | FLOAT
            | polyConst (LBRACKET^ expression RBRACKET!)?
            ;
```

A.1.2 PolyWalker.g

```
/* Contains the PolyWalker class that is the tree parser, which calls routines in PolyInterpreter.java.
 *
 * Mohit Vazirani - mcv2107@columbia.edu
 *
 */
{
    import java.util.*;
    import java.io.*;
}

class PolyWalker extends TreeParser;
options
{
    importVocab = Poly;
}

{
    PolyInterpreter ipt = new PolyInterpreter();
}

varType returns [String s]
{
    s = null;
}
    :   "int"
        {
            s = "int";
        }
    |   "poly"
        {
            s = "poly";
        }
    |   "float"
        {
            s = "float";
        }
    ;

retType returns [String s]
{
    String a;
    s = null;
}
    :   a = varType
        {
            s = a;
        }
    |   "void"
        {
            s = "void";
        }
;

pExpr returns [Object x]
{
    PolyDataType a;
    x = null;
}
    :   s:STRING
        {
            x = s.getText();
        }
    |   a = expr
        {
            x = a;
        }
;

expr returns [PolyDataType r]
{
    PolyDataType a = null, b = null;
    String s = null;
    String[] c;
    PolyDataType[] d;
    Vector v = new Vector();
    Object obj;
    r = null;
}
    :   #(FILE (r=expr)*)
    |   #("or" a = expr b = expr)
        {
            r = a.or(b);
        }
    |   #("and" a = expr b = expr)
        {
            r = a.and(b);
        }
    |   #("not" a = expr)
        {
            r = a.not();
        }
    |   #(EQ a = expr b = expr)
        {
            r = a.equals(b);
        }
    |   #(NEQ a = expr b = expr)
        {
            r = a.neq(b);
        }
    |   #(LT a = expr b = expr)
        {
```

```
        r = a.less(b);
    }
|   #(GT a = expr b = expr)
    {
        r = a.greater(b);
    }
|   #(PLUS a = expr b = expr)
    {
        r = a.add(b);
    }
|   (#(MINUS expr expr)) => #(MINUS a = expr b = expr)
    {
        r = a.minus(b);
    }
|   #(MINUS a = expr)
    {
        r = a.uminus();
    }
|   #(TIMES a = expr b = expr)
    {
        if((a instanceof PolyPoly) && (b instanceof PolyPoly))
        {
            a.error("Cannot multiply two instances of type poly");
        }
        else
        {
            r = a.multiply(b);
        }
    }
|   #(MOD a = expr b = expr)
    {
        if((a instanceof PolyInt)&&(b instanceof PolyInt))
        {
            r = a.mod(b);
        }
        else
        {
            a.error("Only integers can be in a modulus expression");
        }
    }
|   #(DIV a = expr b = expr)
    {
        if(b instanceof PolyPoly)
        {
            a.error("Cannot divide by an instance of poly");
        }
        else
        {
            r = a.divide(b);
        }
    }
```

```
|   #(CONCAT a = expr b = expr)
    {
        r = a.concat(b);
    }
|   #(FUNCCALL id:ID d = mexpr)
    {
        a = new PolyInt(0);
        if(!ipt.st.containsVar(id.getText(),true))
        {
            a.error("Function definition for " + id.getText() + " does not exist.");
        }
        else
        {
            PolyFunction pf = (PolyFunction)ipt.getVariable(id.getText());
            r = ipt.funcInvoke(this,pf,d);

            if(!(ipt.checkReturn()) && (!(pf.getRettype().equals("void"))))
            {
                a.error("No value returned by function " + id.getText());
            }

            ipt.resetFlowControl();
        }
    }
|   #(LBRACKET a = expr b = expr)
    {
        if(!((a instanceof PolyPoly) && (b instanceof PolyInt)))
            a.error("Error computing coefficient - type mismatch.");
        else
            r = a.coeff(b);
    }
|   #(DEGREE a = expr)
    {
        if(!(a instanceof PolyPoly))
        {
            a.error("Order only operates on poly instances.");
        }
        else
        {
            r = a.degree();
        }
    }
|   i:INTEGER
    {
        r = new PolyInt(Integer.parseInt(i.getText()));
    }
|   f:FLOAT
    {
        r = new PolyFloat(Float.parseFloat(f.getText()));
    }
|   idAtom:ID
```

```
    {
        if(!ipt.st.containsVar(idAtom.getText(),true))
        {
            a = new PolyInt(0);
            a.error(idAtom.getText() + " not defined.");
        }
        else
        {
            r = ipt.getVariable(idAtom.getText());
        }
    }
|   #(POLYCONST
        (a = expr
        {
            if(a instanceof PolyPoly)
            {
                a.error("A poly variable cannot consist of poly coefficients.");
            }
            else
            {
                v.add(a);
            }
        }
        )+
        {
            r = new PolyPoly(v);
        }
    )

|   #("prototype" s = retType idProt:ID c = vlist
    {
        ipt.addProto(s,idProt.getText(),c);
    }
    )

|   #("function" s = retType funcName:ID v = argList body:.
    {
        String[] types = (String[])v.elementAt(0);
        String[] names=(String[])v.elementAt(1);
        ipt.funcRegister(s,funcName.getText(),types,names,#body);
    }
    )
|   #(STMTLIST
        (stmt:.
        {
            if(ipt.canProceed())
            {
                r = expr(#stmt);
            }
        }
        )*
    )
|   (#(ASSIGN varType ID expr)) => #(ASSIGN s = varType idNew:ID a = expr)
    {
        if(ipt.st.containsVar(idNew.getText(),false))
        {
            a = new PolyInt(0);
            a.error("Variable " + idNew.getText() + " has already been declared.");
        }

        if(s.equals("float") && ((a.typename()).equals("int")))
        {
            a = new PolyFloat(PolyInt.intValue(a));
        }
        if(s.equals(a.typename()))
        {
            if(s.equals("int"))
            {
                b = new PolyInt(0);
            }
            else
            {
                if(s.equals("float"))
                {
                    b = new PolyFloat(0);
                }
                else
                {
                    b = new PolyPoly(new Vector());
                }
            }
            b.setName(idNew.getText());

            r = ipt.assign(b,a);
        }
        else
        {
            if(a == null)
            {
                a = new PolyInt(0);
            }
            a.error("Type mismatch during declaration of variable " + idNew.getText());
        }
    }
|   #(ASSIGN idAssgn:ID a = expr)
    {
        if(ipt.st.containsVar(idAssgn.getText(),true))
        {
            b = ipt.getVariable(idAssgn.getText());

            if((b.typename()).equals("float") && ((a.typename()).equals("int")))
            a = new PolyFloat(PolyInt.intValue(a));
```

```
                if((a.typename()).equals(b.typename()))
                {
                    r = ipt.assign(b,a);
                }
                else
                {
                    a.error("Type mismatch during assignment of variable " + idAssgn.getText());
                }
            }
            else
            {
                a.error(idAssgn.getText() + " has not been declared.");
            }
        }
    |   #(BLOCK
            (stmtB:.
            {
                if(ipt.canProceed())
                {
                    r = expr(#stmtB);
                }
            }
            )*
        )
    |   #("if" a = expr thenp:. (elsep:.)?)
        {
            if(a.boolVal())
            {
                r = expr(#thenp);
            }
            else
            {
                if(null != elsep)
                {
                    r = expr(#elsep);
                }
            }
        }
    |   #("while" cond:. whilebody:.)
        {
            a = expr(#cond);
            while(a.boolVal() && ipt.canProceed())
            {
                r = expr(#whilebody);
                a = expr(#cond);
            }
        }
    |   #("print"
            (obj = pExpr
            {
```

```
            PrintWriter writer = new PrintWriter(System.out,true);

            if(obj instanceof String)
            {
                System.out.println(obj);
            }
            else
            {
                if(obj instanceof PolyDataType)
                {
                    ((PolyDataType)obj).print(writer);
                }
                else
                {
                    a = new PolyInt(0);
                    a.error("Invalid print argument.");
                }
            }
        }
        )+
    )
|   #("return"
    {
        a = null;
        b = new PolyInt(0);
    }
        (a = expr)?
        {
            if(ipt.inGlobalScope())
            {
                b.error("Cannot return from global scope.");
            }

            PolyFunction pf = (PolyFunction)ipt.getVariable(ipt.getActiveFuncName());
            ipt.setReturn();

            if(pf.getRettype().equals("float") && ((a.typename()).equals("int")))
            {
                a = new PolyFloat(PolyInt.intValue(a));
            }

            if(a != null)
            {
                if(!(pf.getRettype().equals(a.typename())))
                {
                    a.error("Value returned doesn't match return type of function " + pf.getName())
                }
                else
                {
                    a.setName("");
                    r = ipt.rvalue(a);
                    }
                }
                else
                {
                    if(!(pf.getRettype().equals("void")))
                    {
                        b.error("Function " + pf.getName() + " did not return a value.");
                    }
                }
            }
        )
;


mexpr returns [PolyDataType[] p]
{
    Vector v = new Vector();
    PolyDataType a;
    p = null;
}
:   #(FCARGS
        (
            a = expr
            {
                v.add(a);
            }

        )*
    )
    {
        p = ipt.convertExprList(v);
    }
;

argList returns [Vector v]
{
    v = new Vector();
    Vector types = new Vector();
    Vector names = new Vector();
    String t;
}
:   #(ARGS
        (
            #(ARG t = varType n:ID
            {
                types.add(t);
                names.add(n.getText());
            }
            )
        )*
    )
    {
        v.add(ipt.convertVarList(types));
        v.add(ipt.convertVarList(names));
    }
;

vlist returns [String[] sv]
{
    Vector v;
    String t;
    sv = null;
    v = new Vector();
}
:   #(TYPELIST
        (
            t = varType{v.add(t);}
        )*
    )
    {
        sv = ipt.convertVarList(v);
    }
;
```

A.2 Java Code

A.2.1 PolyDataType.java

```
import java.io.PrintWriter;

/**
 * The base data type class
 *
 * Michael Dougherty - mgdoug2104@columbia.edu
 *
 */
public class PolyDataType
{
    String name;
    int variableID;
    int uid;

    public PolyDataType() {
        name = null;
    }

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

    public String getName() {
        return name;
    }

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

    public PolyDataType copy() {
        return new PolyDataType();
    }

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

    public PolyDataType error( String msg ) {
        throw new PolyException( "illegal operation: " + msg);
    }

    public PolyDataType error( PolyDataType b, String msg ) {
        if ( null == b )
            return error( msg );
        throw new PolyException(
    "illegal operation: " + msg
    + "( <" + typename() + "> "
    + ( name != null ? name : "<?>" )
    + " and "
    + "<" + b.typename() + "> "
    + ( b.getName() != null ? b.getName() : "<?>" )
    + " )" );
    }


    public void print( PrintWriter w) {
        if ( name != null )
            w.print( name + " = " );
        w.println( "<undefined>" );
    }

    public void printsh( PrintWriter w ) {
        if ( name != null )
            w.print( name + " = " );
        w.print( "<undefined>" );
    }

    public void println() {
        print( new PrintWriter( System.out, true) );
    }

    public void print() {
        printsh( new PrintWriter( System.out, true ) );
    }
```

```
public PolyDataType uminus() {
    return error( "-" );
}

public PolyDataType add( PolyDataType b ) {
    return error( b, "+" );
}

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

public PolyDataType multiply( PolyDataType b ) {
    return error( b, "*" );
}

public PolyDataType divide( PolyDataType b ) {
    return error( b, "/" );
}

public PolyDataType greater( PolyDataType b ) {
    return error( b, ">" );
}

public PolyDataType less( PolyDataType b ) {
    return error( b, "<" );
}

public PolyDataType equals( PolyDataType b ) {
    return error( b, "==" );
}

public PolyDataType neq( PolyDataType b ) {
    return error( b, "!=" );
}

public PolyDataType and( PolyDataType b ) {
    return error( b, "&&" );
}

public PolyDataType or( PolyDataType b ) {
    return error( b, "||" );
}

public PolyDataType mod( PolyDataType b) {
    return error( b, "%" );
}

public PolyDataType degree() {
    return error( "degree");
    }

    public PolyDataType coeff(PolyDataType b) {
        return error( b, "coefficient");
    }

    public PolyDataType concat(PolyDataType b) {
        return error(b, ":");
    }

    public PolyDataType not(  ) {
        return error( "!" );
    }

    public PolyDataType tf() {
        return error( "tf" );
    }

    public boolean boolVal() {
        return false;
    }
}
```

A.2.2 PolyException.java

```
/**
 * Exception class: messages are generated in various classes
 *
 * Michael Dougherty
 */
class PolyException extends RuntimeException {
    PolyException( String msg ) {
        System.err.println( "Error: " + msg );
    }
}
```

A.2.3 PolyFloat.java

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

/**
 * The wrapper class for float
 *
 * Michael Dougherty - mgd2104@columbia.edu
 */
class PolyFloat extends PolyDataType {
    float var;

    public PolyFloat( float x ) {
        var = x;
    }
```

```
public String typename() {
    return "float";
}

public PolyDataType copy() {
    return new PolyFloat( var );
}

public static float floatValue( PolyDataType b ) {
    if ( b instanceof PolyFloat )
        return ((PolyFloat)b).var;
    if ( b instanceof PolyInt )
        return (float) ((PolyInt)b).var;
    b.error( "cast to float" );
    return 0;
}

public float getval() {
    return var;
}

public void print( PrintWriter w ) {
    if ( name != null )
        w.print( name + " = " );
    if(var>=0)
        w.println( (double)((int)((var*1000.0)+.5))/1000  );
    else if(var < 0)
        w.println( (double)((int)((var*1000.0)-.5))/1000 );
    w.flush();
}

public void printsh( PrintWriter w ) {
    if(var>=0)
        w.print( (double)((int)((var*1000.0)+.5))/1000 );
    else if(var < 0)
        w.print( (double)((int)((var*1000.0)-.5))/1000 );
    w.flush();
}

public PolyDataType uminus() {
    return new PolyFloat( -var );
}

public PolyDataType minus( PolyDataType b) {
    if( b instanceof PolyPoly )
        return b.uminus().add( this );
    return new PolyFloat( var - floatValue(b) );
}

public PolyDataType add( PolyDataType b ) {
    if( b instanceof PolyPoly )
        return b.add( this );
    return new PolyFloat( var + floatValue(b) );
}

public PolyDataType multiply( PolyDataType b ) {
    if ( b instanceof PolyPoly )
        return b.multiply( this );
    return new PolyFloat( var * floatValue(b) );
}

public PolyDataType divide( PolyDataType b ) {
    if ( b instanceof PolyPoly )
        return error(b, "/");
    return new PolyFloat( var / floatValue(b) );
}

public PolyDataType greater( PolyDataType b ) {
    if ( b instanceof PolyPoly )
        return b.less(this);
    return new PolyInt( var > floatValue(b) ? 1 : 0);
}

public PolyDataType less( PolyDataType b ) {
    if ( b instanceof PolyPoly )
        return b.greater( this );
    return new PolyInt( var < floatValue(b) ? 1 : 0);
}

public PolyDataType equals( PolyDataType b ) {
    if ( b instanceof PolyPoly )
        return b.equals( this );
    return new PolyInt( var == floatValue(b) ? 1 : 0);
}

public PolyDataType neq( PolyDataType b ) {
    if ( b instanceof PolyPoly )
        return b.neq( this );
    return new PolyInt( var != floatValue(b) ? 1 : 0 );
}

public PolyDataType and( PolyDataType b ) {
    if ( b instanceof PolyPoly){
        b.and(this);
    }
    return new PolyInt( (((floatValue(b) != 0) && (var != 0)) ? 1 : 0));
}

public PolyDataType or( PolyDataType b ) {
    if ( b instanceof PolyPoly ){
        b.or(this);
        }

        return new PolyInt( (((floatValue(b) != 0) || (var != 0)) ? 1 : 0));
    }

    public PolyDataType not() {
        return new PolyInt( ((var == 0) ? 1 : 0) );
    }

    public PolyDataType tf() {
        return new PolyInt( ((var == 0) ? 0 : 1) );
    }

    public boolean boolVal() {
        return var == 0 ? false : true;
    }

    public PolyDataType concat(PolyDataType b) {
        Vector v1 = new Vector();
        v1.addElement(this);
        if(b instanceof PolyPoly) {
            ((PolyPoly)b).addme(v1);
        }
        else
            v1.addElement(b);
        return new PolyPoly(v1);
    }
}
```

A.2.4 PolyFuncProt.java

```
/**
 * Poly Function Prototype class
 *
 * Michael Dougherty - mgd2104@columbia.edu
 */


class PolyFuncProt {
    String name;
    String rettype;
    String types[];

    public PolyFuncProt(String rettype, String name, String[] types ) {
        this.name = name;
        this.rettype = rettype;
        this.types = types;
    }

    public String getName() {
        return name;
    }
    public String getRettype() {
        return rettype;
    }

    public String[] getTypes() {
        return types;
    }
}
```

A.2.5 PolyFunction.java

```
import java.io.PrintWriter;
import antlr.collections.AST;

/**
 * The function data type (including internal functions)
 *
 * Michael Dougherty - mgd2104@columbia.edu
 */
class PolyFunction extends PolyDataType {
    // we need a reference to the AST for the function entry
    String[] args;
    String[] varnames;
    String rettype;
    AST body;
    PolySymbolTable pst;   // the symbol table of the parent

    public PolyFunction( String name,
                         String rettype,
                         String[] args,
                         String[] varnames,
                         AST body,
                         PolySymbolTable pst) {
        super( name );
        this.args = args;
        this.varnames = varnames;
        this.rettype = rettype;
        this.body = body;
        this.pst = pst;
    }

    public String typename() {
        return "function";
    }

    public PolyDataType copy() {
        return new PolyFunction( name, rettype, args, varnames, body, pst );
    }

    public void print( PrintWriter w ) {
        if ( name != null )
```

```
            w.print( name + " = " );
        w.print( "<function>(" );
        for ( int i=0; ; i++ ) {
            if ( i >= args.length - 1 )
                break;
            w.print( args[i] );
            w.print( "," );
        }
        w.println( ")" );
    }

    public String[] getArgs() {
        return args;
    }

    public String[] getVarnames() {
        return varnames;
    }

    public String getRettype() {
        return rettype;
    }

    public PolySymbolTable getParentSymbolTable() {
        return pst;
    }

    public AST getBody() {
        return body;
    }
}
```

A.2.6 PolyInterpreter.java

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

/** Interpreter routines that are called directly from the tree walker.
 *
 * Michael Dougherty - mgd2104@columbia.edu
 *
 */
class PolyInterpreter {
    PolySymbolTable st;
    PolySymbolTable main_st;
    LinkedList protos;
```

```
final static int fc_none = 0;
final static int fc_return = 1;

private int control = fc_none;

public PolyInterpreter() {
    main_st = null;
    st = new PolySymbolTable(main_st, st, "main");
    main_st = st;
    protos = new LinkedList();
}

public PolyDataType[] convertExprList( Vector v ) {
    /* Note: expr list can be empty */
    PolyDataType[] x = new PolyDataType[v.size()];
    for ( int i=0; i<x.length; i++ )
        x[i] = (PolyDataType) v.elementAt( i );
    return x;
}

public static String[] convertVarList( Vector v ) {
    /* Note: var list can be empty */
    String[] sv = new String[ v.size() ];
    for ( int i=0; i<sv.length; i++ )
        sv[i] = (String) v.elementAt( i );
    return sv;
}

public static PolyDataType getNumber( String s ) {
    if ( s.indexOf( '.' ) >= 0 )
        return new PolyFloat( Float.parseFloat( s ) );
    return new PolyInt( Integer.parseInt( s ) );
}

public PolyDataType getVariable( String s ) {
    // default static scoping
    PolyDataType x = st.getVar(s);
    if ( x == null )
        throw new PolyException("Variable not defined");
    return x;
}

public PolyDataType rvalue( PolyDataType a ) {
    if ( a.name == null )
        return a;
    return a.copy();
}

public PolyDataType deepRvalue( PolyDataType a ) {
    if ( a.name == null )
        return a;
    if ( a instanceof PolyPoly )
        return ((PolyPoly)a).deepCopy();
    return a.copy();
}

public PolyDataType assign( PolyDataType a, PolyDataType b ) {
    if ( a.name != null ) {
        PolyDataType x = deepRvalue( b );
        x.setName( a.name );
        st.setVar( x.name, x );
        return x;
    }

    return a.error( b, "=" );
}

public void st_dump() {
    if(st != null)
        st.st_dump();
}

public void addProto(String rettype, String name, String[] args){
    PolyFuncProt prot = new PolyFuncProt(rettype, name, args);

    if(protos.size() == 0) {
        protos.add(prot);
    }
    else {
        ListIterator i = protos.listIterator(0);
        PolyFuncProt item;
        while(i.hasNext()){
            item = (PolyFuncProt)i.next();
            if(name.equals(item.getName()))
                throw new PolyException("Function name already exists.");
        }
        protos.add(prot);
    }
}

public PolyDataType funcInvoke(PolyWalker walker,
                               PolyDataType func,
                               PolyDataType[] params ) throws antlr.RecognitionException {

    // func must be an existing function
    if ( !( func instanceof PolyFunction ) )
        return func.error( "not a function" );

    String[] args = ((PolyFunction)func).getArgs();
    String[] varnames = ((PolyFunction)func).getVarnames();
    if ( args.length != params.length )
        return func.error( "unmatched length of parameters" );
    for(int j=0;j<args.length;j++){
        if(args[j].equals("float") && params[j].equals("int")) {
        }
        else if(!args[j].equals(params[j].typename())) {
            return func.error( "unmatched parameter types: " + args[j] + " and " + params[j] + ".")
        }
    }

    // create a new symbol table
    st = new PolySymbolTable( main_st, st, func.getName());

    // assign actual parameters to formal arguments
    for ( int i=0; i<varnames.length; i++ ) {
        PolyDataType d;
        if(args[i].equals("float") && ((params[i].typename()).equals("int"))) {
            d = new PolyFloat(PolyInt.intValue(params[i]));
            d.setName( varnames[i] );
            st.setVar( varnames[i], d);
            continue;
        }
        d = deepRvalue( params[i] );
        d.setName( varnames[i] );
        st.setVar( varnames[i], d);
    }

    // call the function body
    PolyDataType r = walker.expr( ((PolyFunction)func).getBody() );

    // remove this symbol table and return
    st = st.getParent();

    return r;
}


public void funcRegister( String rettype,
                          String name,
                          String[] args,
                          String[] varnames,
                          AST body ) throws PolyException {
    if(protos.size() == 0)
        throw new PolyException("Function declaration may not precede prototype definition");

    ListIterator i = protos.listIterator(0);
    PolyFuncProt item;
    String[] protoargs;
    String protoret;

    while(i.hasNext()){
        item = (PolyFuncProt)i.next();
        if(item.getName().equals(name)){
            protoargs = item.getTypes();
            protoret = item.getRettype();
            if(args.length != protoargs.length)
                throw new PolyException("Number of args in function declaration does not match numbe
            else
            {
                for(int j=0;j<args.length;j++){
                    if(!args[j].equals(protoargs[j]))
                        throw new PolyException("Function declaration argument type " + args[j] + "
                }//end for
            }//end else
            if(!rettype.equals(protoret))
                throw new PolyException("Function return type does not match prototype return type.'
            break;
        }//end if
    }//end while

    st.add( new PolyFunction( name, rettype, args, varnames, body, st ) );
}

public String getActiveFuncName(){
    return st.getFuncName();
}

public void setReturn( ) {
    control = fc_return;
}

public void resetFlowControl() {
    control = fc_none;
}

public boolean checkReturn () {
    return control == fc_return;
}

public boolean canProceed() {
    return control == fc_none;
}

public boolean inGlobalScope() {
    return st == main_st;
}
}
```

A.2.7 PolyInt.java

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

/**
```

```
 * the wrapper class of int
 *
 * Michael Dougherty - mgd2104@columbia.edu
 */
class PolyInt extends PolyDataType {
    int var;

    public PolyInt( int x ) {
        var = x;
    }

    public String typename() {
        return "int";
    }

    public PolyDataType copy() {
        return new PolyInt( var );
    }

    public static int intValue( PolyDataType b ) {
        if ( b instanceof PolyFloat )
            return (int) ((PolyFloat)b).var;
        if ( b instanceof PolyInt )
            return ((PolyInt)b).var;
        b.error( "cast to int" );
        return 0;
    }

    public int getval() {
        return var;
    }

    public void printsh( PrintWriter w ) {
        w.print( var );
        w.flush();
    }

    public void print( PrintWriter w ) {
        if ( name != null )
            w.print( name + " = " );
        w.println( Integer.toString( var ) );
        w.flush();
    }

    public PolyDataType uminus() {
        return new PolyInt( -var );
    }

    public PolyDataType minus( PolyDataType b ) {
        if ( b instanceof PolyPoly )
            return b.add(uminus());
        if ( b instanceof PolyInt )
            return new PolyInt( var - intValue(b) );
        return new PolyFloat( var - PolyFloat.floatValue(b) );
    }

    public PolyDataType add( PolyDataType b ) {
        if ( b instanceof PolyPoly )
            return b.add(this);
        if ( b instanceof PolyInt )
            return new PolyInt( var + intValue(b) );
        return new PolyFloat( var + PolyFloat.floatValue(b) );
    }

    public PolyDataType multiply( PolyDataType b ) {
        if ( b instanceof PolyPoly )
            return b.multiply(this);
        if ( b instanceof PolyInt )
            return new PolyInt( var * intValue(b) );
        return new PolyFloat( var * PolyFloat.floatValue(b) );
    }

    public PolyDataType divide( PolyDataType b ) {
        if ( b instanceof PolyPoly )
            return b.multiply(this);
        if ( b instanceof PolyInt )
            return new PolyInt( (int)(var / intValue(b)) );
        return new PolyFloat( (float)(var / PolyFloat.floatValue(b)));
    }

    public PolyDataType greater( PolyDataType b ) {
        if ( b instanceof PolyInt )
            return new PolyInt( ((var > intValue(b)) ? 1 : 0) );
        return b.less( this );
    }

    public PolyDataType less( PolyDataType b ) {
        if ( b instanceof PolyInt )
            return new PolyInt( ((var < intValue(b)) ? 1 : 0) );
        return b.greater( this );
    }

    public PolyDataType equals( PolyDataType b ) {
        if ( b instanceof PolyInt )
            return new PolyInt( ((var == intValue(b)) ? 1 : 0) );
        return b.equals( this );
    }

    public PolyDataType neq( PolyDataType b ) {
        if ( b instanceof PolyInt )
            return new PolyInt( ((var != intValue(b)) ? 1 : 0) );
        return b.neq( this );
    }

    public PolyDataType and( PolyDataType b ) {
        if ( b instanceof PolyInt )
            return new PolyInt( (((intValue(b)!=0) && (var!=0)) ? 1 : 0));
        return b.and( this );
    }

    public PolyDataType or( PolyDataType b ) {
        if ( b instanceof PolyInt )
            return new PolyInt( (((intValue(b)!=0) || (var!=0)) ? 1 : 0));
        return b.or( this );
    }

    public PolyDataType mod( PolyDataType b) {
        if ( b instanceof PolyInt ) {
            return new PolyInt(var % intValue(b));
        }
        return error(b, "%");
    }

    public PolyDataType not() {
        return new PolyInt( ((var == 0) ? 1 : 0) );
    }

    public PolyDataType tf() {
        return new PolyInt( ((var == 0) ? 0 : 1) );
    }

    public boolean boolVal() {
        return  var == 0 ? false : true;
    }

    public PolyDataType concat(PolyDataType b) {
        Vector v1 = new Vector();
        v1.addElement(this);
        if(b instanceof PolyPoly) {
            ((PolyPoly)b).addme(v1);
        }
        else
            v1.addElement(b);
        return new PolyPoly(v1);
    }
}
```

A.2.8 PolyMain.java

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

/**
 * The main class
 *
 * Michael Dougherty - mgd2104@columbia.edu
 */
class PolyMain {
    //    static boolean verbose = false;

    public static void execFile( String filename ) {
        try {
            InputStream input = ( null != filename ) ?
                (InputStream) new FileInputStream( filename ) :
                (InputStream) System.in;

            PolyLexer lexer = new PolyLexer( input );

            PolyParser parser = new PolyParser( lexer );

            parser.file();
             // Parse the input program
             //at this point, I assume I have a properly parsed file.


             if ( lexer.nr_error > 0 || parser.nr_error > 0 ) {
                 System.err.println( "Parsing errors found. Stop." );
                 return;
             }

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

             PolyWalker walker = new PolyWalker();
             // Traverse the tree created by the parser

             System.out.println( "================ program output ======================" );

             PolyDataType r = walker.expr( tree );

        } catch( IOException e ) {
            System.err.println( "Error: I/O: " + e );
        } catch( RecognitionException e ) {
            System.err.println( "Error: Recognition: " + e );
        } catch( TokenStreamException e ) {
            System.err.println( "Error: Token stream: " + e );
        } catch( Exception e ) {
            System.err.println( "Error: " + e );
        }
    }

    public static void main( String[] args ) {
        if ( args.length >= 1 && args[args.length-1].charAt(0) != '-' )
            execFile( args[args.length-1] );
        else
            System.out.println("Usage: java PolyMain exec.file\n");

        System.exit( 0 );
    }
}
```

A.2.9 PolyPoly.java

```
import java.io.PrintWriter;
import java.util.*;
import java.io.*;
import java.lang.Math;

/**
 * the wrapper class for PolyPoly
 *
 * Michael Dougherty - mgd2104@columbia.edu
 *
 */
class PolyPoly extends PolyDataType {
    Vector v;

    public PolyPoly( ) {
        v = new Vector();
    }

    public PolyPoly( Vector v ) {
        Vector w = simplify(v);
        this.v = (Vector)w.clone();
    }

    public String typename() {
        return "poly";
    }

    public PolyDataType copy() {
        return new PolyPoly( v );
    }

    public PolyDataType deepCopy() {
        return new PolyPoly( (Vector)v.clone());
    }

    public void print( PrintWriter w ) {
        int anythingyet = 0;
        if ( name != null )
            w.print( name + " = " );
        for(int i = v.size()-1;i>=0;i--) {
            if(PolyFloat.floatValue((PolyDataType)v.elementAt(i)) == 1) {
                if(i != v.size()-1)
                    w.print(" + ");
                //dont display the coefficient(1)
                if(i == 0) //element of the poly of order 0
                    w.print("1");
                else if(i == 1) {
                    w.print("x");
                    anythingyet = 1;
                }
                else {
                    anythingyet = 1;
                    w.print("x^" + i);
                }
            }
            else if(PolyFloat.floatValue((PolyDataType)v.elementAt(i)) == -1) {
                if(i != v.size()-1)
                    w.print(" + ");
                if(i == 0)
                    w.print("-1");
                else if(i==1){
                    w.print("-x");
                    anythingyet=1;
                }
                else {
                    anythingyet = 1;
                    w.print("-x^" + i);
                }
            }
            else if(PolyFloat.floatValue((PolyDataType)v.elementAt(i)) != 0) {
                if(i != v.size()-1)
                    w.print(" + ");
                //display the coefficient(not 1 or 0)
                if(i == 0)
                    ((PolyDataType)v.elementAt(i)).printsh(w);
                else if(i==1) {
                    anythingyet = 1;
                    ((PolyDataType)v.elementAt(i)).printsh(w);
                    w.print("x");
                }
                else {
                    anythingyet = 1;
                    ((PolyDataType)v.elementAt(i)).printsh(w);
                    w.print("x^" + i);
                }
            }
            else if((PolyFloat.floatValue((PolyDataType)v.elementAt(i)) == 0) && (i==0) && !(anythingye
                w.print(0);
        }
        w.println("");
    }
```

```
public PolyDataType uminus() {
    Vector v2 = new Vector();
    for(int i=0;i<v.size();i++)
        v2.addElement(((PolyDataType)v.elementAt(i)).uminus());
    return new PolyPoly(v2);
}

public PolyDataType minus( PolyDataType b ) {
    if( b instanceof PolyPoly ) {
        return add(b.uminus());
    }

    Vector v2 = (Vector)v.clone();

    v2.setElementAt(((PolyDataType)v2.elementAt(0)).add(b.uminus()), 0);
    return new PolyPoly(v2);
}


public PolyDataType add( PolyDataType b ) {
    Vector v2 = new Vector();
    if( b instanceof PolyPoly ) {
        for(int i=0;i<Math.max(v.size(), 1+((PolyInt)b.degree()).getval());i++) {
            if(i>=v.size()) {
                v2.add(b.coeff(new PolyInt(i)));
            }
            else if(i>((PolyInt)b.degree()).getval()) {
                v2.add(((PolyDataType)v.elementAt(i)));
            }
            else {
                v2.add(((PolyDataType)v.elementAt(i)).add( b.coeff(new PolyInt(i))));
            }
        }
        return new PolyPoly( v2 );
    }
    v2.add(((PolyDataType)v.elementAt(0)).add( b ));
    for(int i=1;i<v.size();i++) {
        v2.add((PolyDataType)v.elementAt(i));
    }
    return new PolyPoly( v2 );
}

public PolyDataType multiply( PolyDataType b) {
    if( b instanceof PolyPoly ) {
        return error( b, "*" );
    }
    Vector v2 = new Vector();
    for(int i=0;i<v.size();i++)
        v2.addElement(((PolyDataType)v.elementAt(i)).multiply( b ));
    simplify(v2);
    return new PolyPoly( v2 );
}

public PolyDataType divide( PolyDataType b) {
    if( b instanceof PolyPoly ) {
        return error( b, "/" );
    }
    Vector v2 = new Vector();
    for(int i=0;i<v.size();i++) {
        v2.addElement(((PolyDataType)v.elementAt(i)).divide( b ));
    }
    return new PolyPoly( v2 );
}

public PolyDataType tf() {
    for(int i=0;i<v.size();i++){
        if(0 != PolyInt.intValue((PolyDataType)v.elementAt(i)))
            return new PolyInt(1);
    }
    return new PolyInt(0);
}

public boolean boolVal() {
    for(int i=0;i<v.size();i++){
        if(0 != PolyInt.intValue((PolyDataType)v.elementAt(i)))
            return true;
    }
    return false;
}

public Vector simplify(Vector v) {
    for(int i=v.size()-1;i>0;i--) {
        if(v.elementAt(i) instanceof PolyInt) {
            if(((PolyInt)v.elementAt(i)).getval() == 0) {
                v.removeElementAt(i);
            }
            else
                return v;
        }
        else if(v.elementAt(i) instanceof PolyFloat) {
            if(((PolyFloat)v.elementAt(i)).getval() == 0) {
                v.removeElementAt(i);
            }
            else
                return v;
        }
    }
    return v;
}

public PolyDataType greater(PolyDataType b) {
    if(b instanceof PolyPoly) {
        if(v.size() > ((PolyInt)(b.degree())).getval()+1) {
            if(v.lastElement() instanceof PolyInt) {
                int a = (((PolyInt)((PolyDataType)v.lastElement())).getval());
                if(a>0)
                    return new PolyInt(1);
                else
                    return new PolyInt(0);
            }
            else {
                double a = (((PolyFloat)((PolyDataType)v.lastElement())).getval());
                if(a>0)
                    return new PolyInt(1);
                else
                    return new PolyInt(0);
            }
        }
        if(v.size() < ((PolyInt)(b.degree())).getval()+1) {
            if(b.coeff(new PolyInt(((PolyInt)b.degree()).getval())) instanceof PolyInt) {
                int a = (((PolyInt)b.coeff(new PolyInt(((PolyInt)b.degree()).getval()))).getval());
                if(a>0)
                    return new PolyInt(0);
                else
                    return new PolyInt(1);
            }
            else {
                double a = (((PolyFloat)b.coeff(new PolyInt(((PolyInt)b.degree()).getval()))).getval
                if(a>0)
                    return new PolyInt(0);
                else
                    return new PolyInt(1);
            }
        }
        for(int i = v.size()-1; i >= 0; i--) {
            if(((PolyInt)((PolyDataType)v.elementAt(i)).greater(b.coeff(new PolyInt(i)))).getval()
                return new PolyInt(1);
            if(((PolyInt)((PolyDataType)v.elementAt(i)).less(b.coeff(new PolyInt(i)))).getval() !=
                return new PolyInt(0);
        }
        return new PolyInt(0);
    }
    if( v.size() > 0)
        return new PolyInt(0);
    if( ((PolyInt)((PolyDataType)v.firstElement()).greater(b)).getval() != 0)
        return new PolyInt(1);
    return new PolyInt(0);
}

public PolyDataType less(PolyDataType b) {
    if(b instanceof PolyPoly) {
        if(v.size() < ((PolyInt)(b.degree())).getval()+1) {
            if(v.lastElement() instanceof PolyInt) {
```

```
                int a = (((PolyInt)((PolyDataType)v.lastElement())).getval());
                if(a>0)
                    return new PolyInt(1);
                else
                    return new PolyInt(0);
            }
            else {
                double a = (((PolyFloat)((PolyDataType)v.lastElement())).getval());
                if(a>0)
                    return new PolyInt(1);
                else
                    return new PolyInt(0);
            }
        }
        if(v.size() > ((PolyInt)(b.degree())).getval()+1) {
            if(b.coeff(new PolyInt(((PolyInt)b.degree()).getval())) instanceof PolyInt) {
                int a = (((PolyInt)b.coeff(new PolyInt(((PolyInt)b.degree()).getval()))).getval());
                if(a>0)
                    return new PolyInt(0);
                else
                    return new PolyInt(1);
            }
            else {
                double a = (((PolyFloat)b.coeff(new PolyInt(((PolyInt)b.degree()).getval()))).getva
                if(a>0)
             return new PolyInt(0);
                else
             return new PolyInt(1);
            }
}
        for(int i = v.size()-1; i >= 0; i--) {
            if(((PolyInt)((PolyDataType)v.elementAt(i)).greater(b.coeff(new PolyInt(i)))).getval()
                return new PolyInt(0);
            if(((PolyInt)((PolyDataType)v.elementAt(i)).less(b.coeff(new PolyInt(i)))).getval() !=
                return new PolyInt(1);
        }
        return new PolyInt(0);
    }
    if( v.size() > 0)
        return new PolyInt(0);
    if( ((PolyInt)((PolyDataType)v.firstElement()).less(b)).getval() != 0)
        return new PolyInt(1);
    return new PolyInt(0);
}

public PolyDataType equals(PolyDataType b) {
    if(b instanceof PolyPoly) {
        if(v.size() < ((PolyInt)(b.degree())).getval()+1)
            return new PolyInt(0);
        if(v.size() > ((PolyInt)(b.degree())).getval()+1)
            return new PolyInt(0);
        for(int i = v.size()-1; i >= 0; i--) {
            if(((PolyInt)((PolyDataType)v.elementAt(i)).greater(b.coeff(new PolyInt(i)))).getval()
                return new PolyInt(0);
            if(((PolyInt)((PolyDataType)v.elementAt(i)).less(b.coeff(new PolyInt(i)))).getval() !=
                return new PolyInt(0);
        }
        return new PolyInt(1);
    }
    if( v.size() > 0)
        return new PolyInt(0);
    if( ((PolyInt)((PolyDataType)v.firstElement()).equals(b)).getval() != 0)
        return new PolyInt(1);
    return new PolyInt(0);
}

public PolyDataType neq(PolyDataType b) {
    if(b instanceof PolyPoly) {
        if(v.size() < ((PolyInt)(b.degree())).getval()+1)
            return new PolyInt(1);
        if(v.size() > ((PolyInt)(b.degree())).getval()+1)
            return new PolyInt(1);
        for(int i = v.size()-1; i >= 0; i--) {
            if(((PolyInt)((PolyDataType)v.elementAt(i)).greater(b.coeff(new PolyInt(i)))).getval()
                return new PolyInt(1);
            if(((PolyInt)((PolyDataType)v.elementAt(i)).less(b.coeff(new PolyInt(i)))).getval() !=
                return new PolyInt(1);
        }
        return new PolyInt(0);
    }
    if( v.size() > 0)
        return new PolyInt(0);
    if( ((PolyInt)((PolyDataType)v.firstElement()).neq(b)).getval() != 0)
        return new PolyInt(1);
    return new PolyInt(0);
}

public PolyDataType and(PolyDataType b) {
    PolyDataType aret = tf();
    PolyDataType bret = b.tf();
    int aval = ((PolyInt)aret).intValue(aret);
    int bval = ((PolyInt)bret).intValue(bret);
    if(aval!=0 && bval!=0)
        return new PolyInt(1);
    return new PolyInt(0);
}

public PolyDataType or(PolyDataType b) {
    PolyDataType aret = tf();
    PolyDataType bret = b.tf();
    int aval = ((PolyInt)aret).intValue(aret);
    int bval = ((PolyInt)bret).intValue(bret);
    if(aval!=0 || bval!=0)
        return new PolyInt(1);
    return new PolyInt(0);
}

public PolyDataType not() {
    PolyDataType ret = tf();
    if(((PolyInt)ret).intValue(ret) == 0)
        return new PolyInt(1);
    return new PolyInt(0);
}

public PolyDataType degree() {
    return new PolyInt(v.size()-1);
}

public PolyDataType coeff(PolyDataType b) {
    if(b instanceof PolyInt) {
        int index = ((PolyInt)b).intValue(b);
        if( index > v.size()-1)
            return error(b, "coefficient");
        if( v.elementAt( index ) instanceof PolyInt) {
            int coef = ((PolyInt)v.elementAt(index)).intValue((PolyDataType)v.elementAt(index));
            return new PolyInt( coef );
        }
        if( v.elementAt( index ) instanceof PolyFloat) {
            float coef = ((PolyFloat)v.elementAt(index)).floatValue((PolyDataType)v.elementAt(index
            return new PolyFloat( coef );
        }
    }
    return error(b, "coefficient");
}

public PolyDataType concat(PolyDataType b) {
    Vector v1 = (Vector)v.clone();
    if(b instanceof PolyPoly) {
        ((PolyPoly)b).addme(v1);
    }
    else
        v1.addElement(b);
    return new PolyPoly(v1);
}

public Vector addme(Vector v1) {
    Vector v2 = (Vector)v.clone();
    v1.addAll(v2);
    return v1;
}
}
```

A.2.10 PolySymbolTable.java

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

/**
 * Symbol table class
 *
 * Michael Dougherty - mgd2104@columbia.edu
 */
class PolySymbolTable extends LinkedList {
    PolySymbolTable main_parent;
    PolySymbolTable parent;
    String funcname;

    public PolySymbolTable(){
        main_parent = null;
        parent = null;
    }

    public PolySymbolTable( PolySymbolTable main_parent, PolySymbolTable parent, String funcname) {
        this.main_parent = main_parent;
        this.parent = parent;
        this.funcname = funcname;
    }

    public PolySymbolTable getParent() {
        return parent;
    }

    public String getFuncName( ) {
        return funcname;
    }

    public boolean containsVar( String name, boolean enable_global ) {
        ListIterator i = listIterator(0);
        PolyDataType item;

        if(size() == 0)
            {}
        else {
            while(i.hasNext()){
                item = (PolyDataType)i.next();
                if(item.getName().equals(name))
                    return true;
            }
        }

        if(main_parent != null && enable_global) {
            return main_parent.containsVar(name, false);
        }
        return false;
    }

    public void st_dump() {
        ListIterator i = listIterator(0);
        PolyDataType item;

        if(size() == 0)
            {}
        else{
            while(i.hasNext()) {
                item = (PolyDataType)i.next();
                item.println();
            }
        }
        //System.out.println("In main parent: ");
        if(main_parent != null)
            main_parent.st_dump();

        return;
    }

    public PolyDataType getVar( String name) {
        ListIterator i = listIterator(0);
        PolyDataType item;

        while(i.hasNext()){
            item = (PolyDataType)i.next();
            if(item.getName().equals(name))
                return item;
        }

        if(main_parent != null) {
            return main_parent.getVar(name);
        }

        return null;
    }

    public void setVar( String name, PolyDataType data ) {
        ListIterator i = listIterator(0);
        PolyDataType item;

        while (i.hasNext()){
            item = (PolyDataType)i.next();
            if(item.getName().equals(name)) {
                //name was found in the list, so just replace that var
                i.remove();
                add(data);
                return;
            }
        }
        add(data);
    }
}
```

A.3 Misc

A.3.1 makefile

```
# Makefile for the Poly programming language

GRAMMAR = PolyTokenTypes.java PolyTokenTypes.txt \
          PolyLexer.java PolyParser.java

WALKER = PolyWalker.java \
         PolyWalkerTokenTypes.java PolyWalkerTokenTypes.txt

GENJAVA = $(GRAMMAR) $(WALKER)

CLASSES = PolyTokenTypes.class PolyLexer.class PolyParser.class \
          PolyWalkerTokenTypes.class PolyWalker.class \
          PolyDataType.class PolyInt.class PolyFloat.class \
          PolyPoly.class PolyFunction.class\
          PolyException.class PolySymbolTable.class PolyInterpreter.class \
          PolyMain.class

all : $(CLASSES)

$(CLASSES): %.class : %.java $(GENJAVA)
javac -source 1.4 -g $<

$(GRAMMAR): PolyGrammar.g makefile
rm -f $(GRAMMAR)
java antlr.Tool $<

$(WALKER): PolyWalker.g PolyTokenTypes.txt makefile
rm -f $(WALKER)
java antlr.Tool $<
javac -source 1.4 -g PolyWalker.java
javac -source 1.4 -g PolyWalkerTokenTypes.java

zip: all
zip -r polysrc‘date +%y%m%d‘.zip *.java *.class *.g makefile

clean:
rm -f Poly*.class *~
rm -f PolyLexer.java PolyParser.java
rm -f PolyTokenTypes.java PolyTokenTypes.txt PolyWalker.java
rm -f PolyWalkerTokenTypes.java PolyWalkerTokenTypes.txt
```

```
dist-clean: clean
rm -f PolyAntlr* PolyInternalFunction.java

test:
../test/testsuite stop

testall:
../test/testsuite all
```