COMS 4115 Programming Languages and Translators
PolyWiz Language Final Report

Rose Chrin, Tamjeed Azad, Max Helman, Aditya Kankariya, Anthony Pitts
{ta2553, crc2194, mhh2148, ak4290, aep2195}

Spring 2021

Contents

1 Introduction to PolyWiz

1.1 A Mathematician's Dream

PolyWiz is truly a mathematician's dream. The primary goal is to support symbolic mathematics focused on polynomial functions. In addition, PolyWiz aims to provide unique TEX integration and plotting, allowing a user to not only perform numerical calculations but produce ready-to-show documents with plots in seconds.

1.2 General Language Features

PolyWiz is a strongly, statically typed and statically scoped language. It will be pass by reference behind the scenes to the programmer. This will enhance memory in the long run given all of the types in the language are immutable. In addition, PolyWiz will assume that all variables are constant. This imperative language will have syntax most similar to C, with some syntax having close similarity to Python as well. It will also assume all basic operations from C.

2 Language Tutorial

Writing a program in PolyWiz is a simple process that must follow a couple rules that are reminiscent of the C language, yet distinct in certain facets. First off, every program must include a **main** function that returns an **int**. All functions are declared using the **def** keyword, followed by the function's name and its parameters. All variables are strongly typed, meaning that one must specify the type of any parameter or variable they introduce. The variables in a function must be declared at the top of the function, before any other expressions or statements. Furthermore, all expressions are terminated with a semicolon **;**. Although there are many more paradigms in our language, that should be enough for a novice to write their first program. An example of a simple PolyWiz program is shown below:

```
/* This is a simple example program that defines
two functions and does simple arithmetic operations.*/
def int foo(int a)
{
  return a;
}

def int main()
{
  int a;
  a = 42;
  a = a + 5;
  printint(a);
  return 0;
}
```

Now that one has a basic program written in PolyWiz, how do they run it? Due to certain dependencies, it is highly recommended to perform the compilation process in the docker instance provided in the PolyWiz repository. One can install the required dependencies by entering the directory containing the Dockerfile and run the docker instance using the below two commands:

```
docker build . -t plt
docker run --rm -it -v `pwd`:/home/polywiz -w=/home/polywiz plt
/* Due to large installations, thee first command will take approx. 5 mins to run, so be patient :) */
/* Once the first command has been run, it doesn't have to be run again, and running the second command will
    suffice in future sessions. */
```

If the programmer wants to rebuild (or compile) the compiler, they can run "**make all**" in the root of the provided docker instance, which will output the **polywiz.native** file. Lets assume that the program written is called **example.wiz**. The compilation process consists of building the llvm code, compiling it, and then compiling the executable. Once those steps are complete, one can run their program by simplying calling the executable. An example of those exact commands are shown below:

```
/* builds & compiles the llvm code */
./polywiz.native example.wiz | llc -relocation-model=pic example.ll > example.s
cc -o example.exe example.s library_functions.o -lm /* builds the executable */
./example.exe /* runs the executable */
```

We also include a small compiling script that we call **enchant**, that runs the above compilation lines, and allows the user to specify the output file executable name. To run it, call

```
/* builds & compiles the llvm code */
./enchant example.wiz myexecutable
./myexecutable /* runs the executable */
/* to see help, run ./enchant -h */
```

Lastly, if someone wanted to do a bulk run of the regression test suite in PolyWiz' provided docker instance repository, they could simply run "**make**". This would do all the necessary compiling and execution of all test files to confirm that the compiler and language are working properly. The output of the test results can be found in the **testall.log** file.

3 Language Reference Manual

3.1 Basic Syntax

3.1.1 Comments

All comments are multiline, beginning with "/*" and ending with "*/".

```
    /* PolyWiz is an incredible
    programming language for
    symbolic mathematics and
    other fun things */

    /* OK let's write some code now */
    string my_str;
    string my_str = "I like PolyWiz"; /* But in reality, I love PolyWiz */
```

The grammar parser never sees comments that are made in the code, as the scanner ensures that their contents are never tokenized.

3.1.2 Identifiers

Identifiers in PolyWiz denote variable and functions. All identifiers consist of ASCII letters (non case-sensitive) and decimal digits; no special characters are permitted other than underscores. The first character must be an ASCII letter, not a number or an underscore. Identifiers cannot be reserved keywords. Here are some permitted identifiers:

```
    edwards_rocks, EdwardsRocks, plt_rocks, length_poly, cs_makes_me_1000000dollars
```

Here are examples of identifiers that are not allowed:

```
    4115_rocks, cs_makes_me_$1000000
```

In the grammar parser, these are denoted by the token ID.

3.1.3 Reserved Keywords

Keywords and built-in functions in PolyWiz are reserved identifiers that cannot be used for any other purpose. PolyWiz has the following keywords:

```
in, and, or, not, if, else, for, while, def, return, int, bool, float, string, poly, void, true, false
```

In the grammar parser, these are all denoted by capitalized letters of their words, for example NOT, IF, POLY.

3.1.4 Braces

PolyWiz uses braces to group statements and enforce static scoping. Whitespace (specifically empty lines, tabs, and extra spaces) is ignored. Control flow keywords must be followed by braces. Here is some sample code that demonstrates the proper use of braces:

```
int i;
/* Pretty code that works */
if (i > 27){
printstr("i is greater than 27");
}
```

Meanwhile, this code is a poor stylistic choice but is equivalent to the previous code:

```
int i;
/* Ugly code that works */
if        (i        > 27
)                 {
printstr(

"i is greater than 27");}
```

This code is not correct because it does not have the requisite braces after the if statement:

```
/* Pretty code that does not work */
int i;
if (i > 27)
   printstr("i is greater than 27");
```

In the parser, these are denoted by LBRACE and RBRACE.

3.1.5 Parentheses

Parentheses are used to override default precedence; anything inside parentheses automatically becomes the highest precedence.

```
int a;
int b;
/* Parentheses example */
int a = 1 + 2 * 3; /* a = 7 */
int b = (1 + 2) * 3; /* b = 9 */
```

3.1.6 Sequencing, ;

Return Type: $< T >$ where T is type of RHS expression's return value
Operand: Two expressions on each side of ; operator
Operation Logic: Evaluates the LHS expression, followed by the RHS expression.

```
/* ; operator example */
a = 5; 6 /* returns the value 6 after assigning a = 5 */
```

In the parser, these are denoted by `LPAREN` and `RPAREN`.

3.2 Data Types and Literals

3.2.1 Mutability

All data types in PolyWiz are inherently immutable; rather, variable assignment and reassignment are supported. PolyWiz is pass-by-reference.

3.2.2 Literals

The language supports literals of type int, boolean, float, string, and array. See data types section for details of these.

The **int literal** is represented as a sequence of digits from the set [0,9], and is representative of the int data type. Representative regular expression: `digits = [’0’-’9’]+`

The **string literal** is represented as a sequence of ASCII characters, not starting with a number but can include numbers. It is representative of the string data type. Representative regular expression: `(’"’[^’"’’\\’]*(’\\’_[^’"’’\\’]*)*’"’)`

The **float literal** is represented as a sequence of digits, with a single decimal point within the sequence body, with this sequence of digits possibly raised to some exponent. It is representative of the float data type. Representative regular expression: `digits ’.’  digit* ( [’e’ ’E’] [’+’ ’-’]? digits )?`

The **boolean literal** is represented by the keywords `true` and `false`; it represents the boolean data type.
Representative regular expression: `true | false`

The **array literal** is represented as a series of comma separated literals or variables that are enclosed in a left and right bracket, respectively. Representative pattern in parser: `LBRACK literal_values RBRACK` where literal_values are a comma separated series of expressions.

3.3 Primitive Data Types

The language supports primitive data types of int, float, and booleans, and additionally supports type string and array. Using floats and arrays as building blocks, the language fundamentally supports a new type named poly. Additional types will be defined as needed.

Primitive data types are all immutable in this language. When say, a variable is assigned one of these type values and it is changed, a completely new value is assigned and the old one is discarded.

3.3.1 Booleans

The boolean type can only have two values, true or false. This takes up 4 bytes of memory and supports the boolean operations and,or,<,>,<=,>=. Implementation could simply be an int of value 0 or 1 for false and true, respectively under the hood, but other implementations are also possible.

Examples:

```
bool x;
bool x = True;
if (not x) {
  printstr("x is not true");
}
```

3.3.2 Ints

The int type represents numerical integers and takes up 4 bytes of memory. Syntax and operations are mostly C-like, specifically supporting the operations `+,-,*,/,=`. Additionally, int supports the boolean operations `and,or,<,>,<=,>,=`.

Examples:

```
int x;
/* examples of ints: 4, 23, -5623 */
int x = 57;
```

```
x = x + 9; /* integer addition */
```

3.3.3 Floats

This float type represents floating point numbers, used to approximate non-integer real numbers. Using 8 bytes of space, implemented using IEEE 754-1985 double precision standard, it can precisely approximate real numbers in the range of $\pm 2.23 \times 10^{-308}$ to $\pm 1.80 \times 10^{308}$. It supports all operations that the int type supports.

The support for very large and very small numbers eliminates the need for types such as long and short. When these boundaries are exceeded, the language simply returns an overflow error.

These values use the same symbols for operations as the int type. Operations can occur between ints and floats, which would return a float. See operators section for more details.

Examples:

```
float x;
/* examples of float: -8.4, 71234.98, 1.234 * 10^59 */
float x = 4.0;
x = x * 3.5; /* multiplying floats */
```

3.3.4 Strings

The string type represents a concatenated, immutable block of either ASCII characters. It uses memory dynamically based on the size of string, and generally supports operations supported by Python's str type, including concatenation, indexing and reversal. They are declared and specified in the language using either double quotes or single quotes.

It is important to note that this language does not support the char type common to languages such as C and Java; all single chars are of type string with length 1.

```
string x;
string x = "stephen";
x = x + " " + "edwards" /* concatenation of strings */
```

3.4 Arrays

The array type is specified using `type[]`, and does not have mutable length and uses C-like syntax. Arrays can only consist of a single type and are immutable. Only 1D arrays are supported. In addition, indexing of arrays is supported for arrays of type float, integers, and booleans.

```
float[] x;
float y;
x = [ 4.5, 9.6, 3.2, 4.9 ];

/* array indexing */
y = x[2];
x[3] = 4.5;
```

3.5 The Poly Data Type

The poly data type is the centerpiece of the PolyWiz language. Much of Polywiz's standard library and operations, such as addition (+), multiplication (*), composition (:), plotting (`plot()`, `range_plot()`), are built around functionality with the type poly. The poly type specifies polynomials using an array of floats as variable coefficients and an array of ints for variable exponents, used in tandem to define polynomial functions. Polynomials can only be defined in terms of a single variable, say x, and they are instantiated in the following way:

3.5.1 Implementation

Create a new polynomial of the type `poly`. The first argument is a single list of coefficients of type `float` for each term. The second argument is a single list of exponents of type `int` for each respective term. The third argument is the length of both arrays of type `int`. Both lists should align and be of the same length, meaning that the `i`'th value in both `coefficients` and exponents correspond to the same term. This function is linked to the C standard library under the hood in order to create a variable of type poly, which is represented as an array.

```
poly poly1;
poly poly2;
/* Example of new_poly */
poly1 = new_poly([1.0, 2.0, 4.0], [3, 2, 1], 3); /* poly1 = x^3 + 2x^2 + 4x */
poly2 = new_poly([1.0], [1.0], 1); /* poly2 = x */
```

This instantiates the polynomials $1.0x^3 + 2.0x^2 + 4.0x^1$ and $1.0x^1$.

3.5.2 *order(p)*

Return an integer containing the order/degree of polynomial p of type poly.

```
/* Example of order */
poly poly1;
int poly1_order;
poly1 = new_poly([1.0, 2.0, 3.0], [2, 1, 0], 3); /* poly1 = x^2 + 2x + 3 */
int poly1_order = order(poly1); /* poly1_order = 2 */
```

3.6 Data Types in PolyWiz Grammar

Each of float, int, boolean, string, are all tokenized as literals of their type in the grammar parser (BLIT, FLIT, SLIT), and they each also have a separate token for their types, such as FLOAT, INT, BOOL. The poly type has a POLY token for its type, but its 'literal' is handled differently as instantiation is handled via the new_poly function, and is generalized under expr. All types of arrays have their own literal token, such as FLOAT_ARR_LIT, STRING_ARR_LIT and a token for their specific types, such as FLOAT_ARR.

3.7 Statements and Expressions

3.7.1 Statements

A PolyWiz program is made up of a combination of the following types of statements: Expressions, Variable assignments, Return statements, Function definitions, Function calls, If-else statements, For loops

3.7.2 If-Else Statements

If-else statements are used to make decisions based on the expression (condition) being evaluated. If a condition evaluates to true, the statements inside the if statement are evaluated, otherwise the program will either move on or evaluate an optional else statement as shown below. Any expression must evaluate to a valid boolean (true/false) in order to compile. The condition must be wrapped in parenthesis.

```
int x;
/* Example of if/else control flow */
int x = 5;
if (x > 100) { /* False, so program moves onto else statement */
   printstr("x is greater than 100");
}
else {
   printstr("x is less than or equal to 100"); /* This is what the program outputs */
}
```

3.7.3 For Loops

For loops in PolyWiz are incredibly similar to C: they are contained in parenthesis, and consist of a variable initialized to some initial variable followed by a semicolon, then a breaking condition followed by a semicolon, and finally an update rule for the variable. keyword to return control to the beginning of the loop for the next item in the sequence. You can modify all the variables in an array by iterating over the indices of the array.

```
/* To print out all the items in an array of integers until we see an integer greater than 10 */
int i;
int[] my_array;
int element;
my_array = [2,4,6,8]

for (int i = 0; i < 4; i++) {
    if(my_array[i] < 10) {
        printint(my_array[i]);
    }
    element = 0; /* This will not affect my_array */
}

/* To print out every even integer between 0 and 9 */
for (int i = 1; i < 10; i = i + 2) {
    printint(i);
}

/* To iterate over the indices of a sequence, you can use the length of the array*/
for (int i = 0; i < 4; i++) {
    printint(my_array[i]);
}
```

3.8 Expressions and Operators

Our language supports elementary unary and binary operators to accomplish a plethora of tasks.

3.8.1 Unary Operators

In the parser, this is represented by: `Unop operation * expr`.
There is only one operation whose operation is to the left of the expression, and that is the logical not operation, denoted by the keyword 'not'.

3.8.2 Binary Operators

In the parser, this is generally represented by: `Binop expr * operation * expr`

There are several operators that evaluate via these scheme. Binary arithmetic operators are `+,-,*,/,:`. These all directly return a new expression literal based on expression type, usually ints or floats; a notable exception is the string type, which uses '+' for concatenation.

We have logical operators that can be nested to represent complex boolean expressions; these logical operators are the logical $\wedge$ and $\vee$ operations, and and or.

We have comparison operations (essential for things like control flow) that always return booleans. These are
`==, !=, >, <, >=, and <=`.

For assignment, the parser specifically represents this by `Assign var * operation * expr`
Binary assignment operators are used to assign values to variables.

3.9 Operations

3.9.1 Addition, +

Return Type: $< T >$ where T is the type of both operands
Operands: Two variables of type $< T >$ on both sides of the addition operator (+)
Operation Logic: Returns the sum, which could be the polynomial sum, of the two operands.
Grammar:

```
%left PLUS
expr:
```

expr PLUS expr { Binop($1, Add, $3) }

```
/* + operator example */
poly poly1;
poly poly2;
poly poly_sum;

poly1 = new_poly([2.0, 4.0, 2.0], [2, 1, 0], 3); /* poly1 = 2x^2 + 4x + 2 */
poly2 = new_poly([1.0, 2.0, 3.0], [2, 1, 0], 3); /* poly2 = x^2 + 2x + 3 */
poly_sum = poly1 + poly2; /* poly_sum = 3x^2 + 6x + 5 */
```

3.9.2 Subtraction, -

Return Type: $< T >$ where T is the type of all operands
Operands: One or Two variables of type $< T >$, with at least one variable on the RHS of the subtraction operator (-)
Operation Logic: Returns the difference, which could be of polynomials, of the two operands.
If only one operand is supplied, it returns -1.0 * operand.
Grammar:

%left MINUS
expr:
expr MINUS expr { Binop($1, Sub, $3) }

```
poly poly1;
poly poly2;
poly poly_difference;
/* - operator example */
poly1 = new_poly([2.0, 4.0, 2.0], [2, 1, 0], 3); /* poly1 = 2x^2 + 4x + 2 */
poly2 = new_poly([1.0, 2.0, 3.0], [2, 1, 0], 3); /* poly2 = x^2 + 2x + 3 */
poly_difference = poly1 - poly2; /* poly_difference = x^2 + 2x - 1 */
```

3.9.3 Multiplication, *

Return Type: $< T >$ where T is the type of both operands
Operands: Two variables of type $< T >$ on both sides of the multiplication operator (*)
Operation Logic: Returns the two operands' product, which could be polynomial multiplication.
Grammar:

%left TIMES
expr:
expr TIMES expr { Binop($1, Mult, $3) }

```
poly poly1;
poly poly2;
poly poly_product;
/* * operator example */
poly1 = new_poly([1.0], [1], 1); /* poly1 = x */
poly2 = new_poly([1.0, 2.0, 3.0], [2, 1, 0], 3); /* poly2 = x^2 + 2x + 3 */
poly_product = poly1 * poly2; /* poly_product = x^3 + 2x^2 + 3x */
```

3.9.4 Division, /

Return Type: $< T >$ where T is the type of the first operand
Operands: Two variables of type $< T >$ on both sides or a Poly type on the LHS and a float on the RHS
the division operator (/). Polys can only be divided by a float, not another poly.
Operation Logic: Returns the first operand divided by the second.
Grammar:

%left DIVIDE
expr:

expr DIVIDE expr { Binop($1, Div, $3) }

```
 poly poly1;
 float denominator;
 poly poly_div;

 /* / operator example */
poly1 = new_poly([1.0], [2], 1); /* poly1 = x^2 */
 denominator = 2.0; /* poly2 = x */
poly_div = poly1 / denominator; /* poly_div = .5x^2 */
```

3.9.5 Constants Retriever,

Return Type: float [ ]
Operand: Poly variable on left side of the operator (#)
Operation Logic: Returns an array of the polynomial constants, from highest to lowest order.
Grammar:

%left CONST_RETRIEVER
expr:
    expr CONST_RETRIEEVER { Unop(Const_retriever, $1) }

```
 poly poly1;
 float[] poly1_consts;

 /* # operator example */
 poly1 = new_poly([1.0, 2.0, 3.0], [2, 1, 0], 3); // poly1 = x^2 + 2x + 3
 poly1_consts = poly1# ; // poly1_consts = [[1.0, 2.0, 3.0],[2, 1, 0]]
```

3.9.6 Polynomial Composition, :

Return Type: poly
Operands: Two poly variables on the left and right side of the composition operator (:)
Operation Logic: Returns the polynomial that forms by composing the polynomial on the left hand side of the : operator with the second polynomial, on the right hand side.
Grammar:

%left COMP_POLY
expr:
    expr COMP_POLY expr { Binop($1, Comp_poly, $3) }

```
 poly poly1;
 poly poly2;
 poly poly_composed;
 /* : operator example, composing a poly with another poly */
 poly1 = new_poly([1.0], [2], 1); /* poly1 = x^2 */
 poly2 = new_poly([1.0], [2], 1); /* poly2 = x^2 */
 poly_composed = poly1 : poly2; /* poly_composed = (x^2)^2 = x^4 */
```

3.9.7 Polynomial Evaluation, @

Return Type: float
Operands: One Poly and one float/int variable on the left and right side, respectively, of the @ operator
Operation Logic: Returns the value of the polynomial at the float/int location specified.
Grammar:

%left EVAL_POLY
expr:
    expr EVAL_POLY expr { Binop($1, Eval_poly, $3) }

```
poly poly1;
float poly1_value;
/* @ operator example, evaluating a poly at a specified independent variable location */
poly1 = new_poly([1.0], [2], 1); /* poly1 = x^2 */
float poly1_value = poly1 @ 2; /* poly1_val = 4.0 */
```

3.9.8 Convert Polynomial to String, to_str

Return Type: string
Operand: Poly variable
Operation Logic: Returns a string representation of the polynomial.

```
/* # operator example */
poly poly1;
string poly1_str;

poly1 = new_poly([1.0, 2.0, 3.0], [2, 1, 0], 3); /* poly1 = x^2 + 2x + 3 */
poly1_str = to_str(poly1); /* poly1_str = "x^2+2x+3" */
```

3.9.9 Power, ˆ

Return Type: float/int
Operand: Two floats/ints on each side of the power operator (ˆ)
Operation Logic: Returns the left hand side float/int raised to the right hand side float/int.
Grammar:

```
%left EXP
expr:
    expr EXP expr { Binop($1, Exp, $3) }
```

```
float power_result;
/* ^ operator example */
power_result = 2.0 ^ 3.0; /* power_result = 8.0 */
```

3.9.10 Absolute Value, ||

Return Type: float/int
Operand: A float/int inside the absolute value operator bars (||)
Operation Logic: Returns the absolute value of the float/int inside the absolute value bars.
Grammar:

```
%left ABS_VALUE
expr:
    ABS_VALUE expr ABS_VALUE { Unop(Abs_value, $1) }
```

```
float abs_value_result;
/* | | operator example */
abs_value_result = |-2.0|; /* abs_value_result = 2.0
```

3.9.11 Assignment, =

Return Type: RHS Value
Operand: An identifier on the LHS and a type $< T >$ on the RHS of the = operator
Operation Logic: If the RHS is a primitive, it stores the RHS' value into a variable, named the LHS string value. Otherwise, it stores the RHS' pointer location into a variable, named the LHS string value
Grammar:

```
%right ASSIGN
```

expr:
ID ASSIGN expr { Assign($1, $3) }

```
int a;
poly poly1;
poly poly2;

/* = operator example */
a = 5; /* Stores value 5 in variable "a" */

poly1 = new_poly([1.0, 2.0, 3.0], [2, 1, 0], 3);
poly2 = poly1; /* poly2 holds a pointer to poly1 */
```

3.9.12 Boolean Negation, not

Return Type: boolean
Operand: A boolean on the RHS of the not operator
Operation Logic: Returns the opposite boolean value as the operand.
Grammar:

%right NOT
expr:
NOT expr { Unop(Not, $1) }

```
bool a;
/* not operator example */
a = not 1==1 ; /* a = false */
```

3.9.13 Equality Comparison, ==

Return Type: boolean
Operand: Two values of type $< T >$ on each side of the == operator
Operation Logic: Returns True if both operands are of equal value.
Grammar:

%left EQ
expr:
expr EQ expr { Binop($1, Equal, $3) }

```
poly poly1;
poly poly2;
bool a;
/* == operator example */
poly1 = new_poly([1.0, 2.0, 3.0], [2, 1, 0], 3);
poly2 = poly1;
a = poly1 == poly2; /* a = true */
```

3.9.14 Less than comparison, <

Return Type: boolean
Operand: Two values of type $< T >$ on each side of the < operator
Operation Logic: Returns True if LHS is strictly less than RHS.
Grammar:

%left LT
expr:
expr LT expr { Binop($1, Less, $3) }

```
bool a;
```

```
/* < operator example */
a = 1 < 1; /* a = false */
```

3.9.15 Less than or equal to comparison, $<=$

Return Type: boolean
Operand: Two values of type $< T >$ on each side of the $<=$ operator
Operation Logic: Returns True if LHS is less than or equal to RHS.
Grammar:

```
%left LEQ
expr:
    expr LEQ expr { Binop($1, Leq, $3) }
```

```
bool a;
/* <= operator example */
a = 1 <= 1; /* a = true */
```

3.9.16 Greater than comparison, $>$

Return Type: boolean
Operand: Two values of type $< T >$ on each side of the $>$ operator
Operation Logic: Returns True if LHS is strictly greater than RHS.
Grammar:

```
%left GT
expr:
    expr GT expr { Binop($1, Greater, $3) }
```

```
bool a;
/* > operator example */
a = 1 > 1; /* a = false */
```

3.9.17 Greater than or equal to comparison, $>=$

Return Type: boolean
Operand: Two values of type $< T >$ on each side of the $>=$ operator
Operation Logic: Returns True if LHS is greater than or equal to RHS.
Grammar:

```
%left GEQ
expr:
    expr GEQ expr { Binop($1, Geq, $3) }
```

```
bool a;
/* >= operator example */
a = 1 >= 1; /* a = true */
```

3.9.18 Boolean or, *or*

Return Type: boolean
Operand: Two boolean values on each side of the or operator
Operation Logic: Returns True if LHS or RHS is true.
Grammar:

```
%left OR
expr:
    expr OR expr { Binop($1, Or, $3) }
```

```
bool a;
/* or operator example */
a = true or false; /* a = true */
```

3.9.19 Boolean and, *and*

Return Type: boolean
Operand: Two boolean values on each side of the or operator
Operation Logic: Returns True if both the LHS and RHS is true.
Grammar:

%left AND
expr:
expr AND expr { Binop($1, And, $3) }

```
bool a;
/* and operator example */
a = true and false; /* a = false */
```

3.9.20 Membership, in

Return Type: boolean
Operand: A value on the left-hand side and an array or string on the right-hand side. Works on int, float, string, and poly arrays.
Operation Logic: Returns true if the specified value is a member of the array or a substring of the string; false otherwise
Grammar:

%left IN
expr:
expr IN expr { Binop($1, InArray, $3) }

```
int[] arr;
string s;
/* in example */
arr = [5,6,7,8,9];
printb(4 in arr); /* prints false */
printb(5 in arr); /* prints true */

s = "edwards";
printb("ed" in s); /* prints true */
printb("eddy" in s); /* prints false */
```

3.10 Operator Precedence

This operator precedence table specifies, in increasing order, the compiler's priority and associativity for each operator.

Operator	Meaning	Associativity
;	Sequencing	Left to Right
=	Assignment	Right to Left
not	Boolean Negation	Right to Left
==, >, <, >=, <=	Comparisons	Left to Right
or	Or	Left to Right
and	And	Left to Right
ˆ	Power	Left to Right
||	Absolute Value	Non-associative
to_str	Convert poly to string	Left to Right
+, -	Addition, Subtraction	Left to Right
*, /	Multiplication, Division	Left to Right
-	Unary Subtraction	Non-associative
#	Constants Retriever	Left to Right
@	Evaluation	Left to Right
:	Composition	Left to Right

3.11 Functions

In the language a function is a statement that will take a list of arguments and return a single value or nothing. The list of arguments that it takes in will require type specification. The function definition will start with the keyword def and then the return type. If it returns nothing, keyword **void** is used instead. Its identifier will follow the return type.

Grammar:

fdecl:
DEF typ ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE

```
/*function definition example */
def string tex_string(float a, int b) { /* { begins body */
   poly poly1;
   string nice_n_tex;

   poly1 = new_poly([a, 2.0, 3.0], [2, b, 0], 3);
   nice_n_tex = print_tex(print1);
   return nice_n_tex; /*return statement with nice_n_tex type string
} /* closes body */

/*returns nothing */
def void add_poly(float a, int b) {
   poly poly1;
   poly poly2;
   poly poly3;

   poly1 = new_poly([a, 2.0, 3.0], [2, b, 0], 3);
   poly2 = new_poly([a, 2.0, 3.0], [2, b, 0], 3);
   poly3 = poly1 + poly2;
}
```

3.12 Function Calls

To call a function, the identifier along with its arguments in parentheses will be used. If a function is called using improper types or without sufficient arguments, an error will be raised during compilation, depending on why the arguments failed. In the grammar, a function call is an expression (expr), so it can be assigned to a variable or stand on its own.

```
def string tex_string(float a, int b) { /* { begins body */
   poly poly1;
```

```
    string nice_n_tex;

    poly1 = new_poly([a, 2.0, 3.0], [2, b, 0], 3);
    nice_n_tex = print_tex(print1);
    return nice_n_tex; /*return statement with nice_n_tex type string
}

/*function call examples */
printstr(tex_string(2.0, 2)); /*outputs a string */
printstr(tex_string(5.0, 2)); /*outputs a string */
```

3.12.1 Variable Assignment from Functions

Variables can be assigned to the return value of a function assuming the return type of the function and the type of the variable are the same. If they are not, this will raise a TypeError at runtime. In the grammar, this is done as an expression (expr) and is given as expression `EQ` expression.

```
string poly_tex_one;
int poly_tex_one;
/*variable assignment examples */
poly_tex_one = tex_string(2.0, 2.0); /*outputs a string */
poly_tex_one = tex_string(2.0, 2.0); /*would raise a TypeError at runtime */
```

3.13 Standard Library

3.13.1 Printing

PolyWiz supports print functions to display a string representation of any built-in type in standard output. To promote type saftey, we have different printing functions for each type: printstr() for strings, printint() for ints, and printb() for booleans. cannot take in a concatenation of two different types, it will raise a TypeError.

Methods: printint(), printstr(), printb()
Return Type: string
Parameter: Any expression or variable of a built-in type
Function Logic: Outputs a string to stdout representing function input.

```
string a;
bool b;
/*print example */
a = "Hello";
b = true;
printstr(a); /* Standard output will display: Hello */
printint(7); /* Standard output will display: 7 */
printb(b); /* Standard output will display: true */
```

3.13.2 Plotting

PolyWiz supports basic plotting of 2D polynomial functions, using two functions: `plot` and `range_plot`. The `plot` function takes in an array of polys and and a string representing a `.png` filepath; this will save a `.png` image of the polynomial plotted in the x-value range `-5.0, 5.0` in the desired filepath. The `range_plot` function takes in the same inputs, but also takes in a bottom x-range and a top x-range, and this will save a `.png` image of the polynomial plotted in the x-value range `range_bottom, range_top` in the desired filepath. Both functions return an integer; if that integer is `0`, the plot was successfully plotted and saved; if it is anything else, it did not plot successfully.

Method: `plot()`
Return Type: int
Parameters: an array of polynomials, filepath (string)
Function Logic: Returns integer 0 if plotted successfully; unsuccessful if it returns any other integer.

Method: `range_plot()`
Return Type: int
Parameters: an array of polynomials, range_bottom (float), range_top (float), filepath (string)
Function Logic: Returns integer 0 if plotted successfully; unsuccessful if it returns any other integer.

```
poly poly1;

/*graph example */
poly1 = new_poly([1.0, 2.0, 3.0], [2, 1, 0], 3); /* poly1 = x^2 + 2x + 3 */

plot([ poly1 ], "mystandardplot.png");

range_plot([ poly1 ], -4.0, 7.0, "myrangeplot.png");
```

3.13.3 TEX Integration

LATEX is the true mathematician's language, and as such, PolyWiz is designed to support seamless TEX integration. Every poly can be formatted in TEX , and entire documents including plots can be generated easily.

3.13.4 TEX Formatting

Method: print_poly()
Return Type: string
Operand: Poly variable on left side of the method (print_tex)
Function Logic: Returns a string representation of the typeset polynomial.

```
poly poly1;
string poly1_str;

/* print_poly() example */
poly1 = new_poly([1.0, 2.0, 3.0], [2, 1, 0], 3); /* poly1 = x^2 + 2x + 3 */
poly1_str = print_tex(poly1); /* poly1_str = "$$x^{2}+2x+3$$" */
```

3.13.5 Generating TEX Documents

Function: tex_document()
Return Type: string
Parameters: array a of strings containing text, typeset equations, and file paths to saved plots
array b of indices of plots in ascending order ([-1] indicates there are no images)
Function Logic: Returns TEX document containing equations and plots in the format of a string

```
poly poly1;
string intro;
string outro;
string poly1_str;
string doc1;

/* tex_document() example */
poly1 = new_poly([1.0, 2.0, 3.0], [2, 1, 0], 3); /* poly1 = x^2 + 2x + 3 */
string intro = "After much research, we present the Edwards polynomial:";
string outro = "This will revolutionize the field of compilers.";
poly1_str = print_tex(poly1) ; /* poly1_str = "$$x^{2}+2x+3$$" */
range_plot([ poly1 ], -10, 10, "mypc/coms4115/edwards.png");
doc1 = tex_document([intro, poly1_plt, poyl1_str, outro], [1]); /* generate document */
printstr(doc1); /* print document (to std out) */
```

This prints the following text:

```
\documentclass{article}
\begin{document}
\usepackage{graphicx}
After much research, we present the Edwards polynomial:
\begin{figure}[!h]
\centering
\includegraphics[width=3.5in]{mypc/coms4115/edwards.png}
\label{fig_sim}
\end{figure}
$$x^{2}+2x+3$$
This will revolutionize the field of compilers.
\end{document}
```

3.14 Sample Code

```
/*

Interesting program that proves the Mean Value Theorem's underlying property.

It does this through the following steps:

1) Creates a poly, called poly_original
2) Builds a function to get poly_original's derivative, called poly_derivative
3) Calculates the average slope between two endpoints of poly_original
4) Iterate over values on poly_derivative to find the point of the average slope
5) If this point is found, for any polynomial, then the property behind
      the Mean Value Theorem holds.

This program also plots poly_original and poly_derivative on the same graph
to help visualize the problem.

*/


/* user-defined function to take derivative of polynomial */
def float[] derivative(float[] consts_arr, int poly_order){
  float[] poly_derivative;
  int i;
  poly_derivative = initialize_floats(poly_order);

  /* use calculus techniques to get derivative of each term */
  for(i=0; i<=poly_order; i=i+1){
    if(i>0){
      poly_derivative[i-1] = consts_arr[i] * int_to_float(i);
    }
  }

  return poly_derivative;
}

/* user-defined function to calculate the average slope between two points */
def float slope(float x1, float y1, float x2, float y2){
  return (y2-y1)/(x2-x1);
}

/* user-defined function to get severeal values of polynomial */
def float[] poly_values(poly p, float x1, float x2, int num_of_points){
  float[] values;
  float current_x;
  float delta_x;
  float temp;
  int i;
```

```
  values = initialize_floats(num_of_points);

  /* guarentee x2 > x1 */
  if(x1 > x2){
    temp = x2;
    x2 = x1;
    x1 = temp;
  }

  current_x = x1;
  delta_x = (x2-x1) / int_to_float(num_of_points);
  for(i=0; i<num_of_points; i=i+1){
    values[i] = p @ current_x;
    current_x = current_x + delta_x;
  }

  return values;
}

/* user-defined function to find if value in arr by some err margin */
def bool approx_in(float value, float[] arr, int arr_len, float err){
  bool in_arr;
  int i;

  in_arr = false;

  /* force err to be a positive value */
  err = | err |;

  /* check if value is in arr by margin of err */
  for(i=0; i<arr_len; i=i+1){
    if( |arr[i]-value| <= err){
      in_arr = true;
    }
  }

  return in_arr;
}


def int main()
{
  /* instantiate variables */
  poly poly_original;
  poly poly_derivative;
  float[] consts_arr;
  int i;
  int poly_original_order;
  float[] poly_derivative_consts;
  int[] poly_derivative_exps;
  float average_slope;
  float x1;
  float x2;
  float err;
  int return_code;
  float[] derivative_values;

  /* LaTeX Stuff */
  string[] body;
  string text1;
  string text2;
  string text3;
  string text4;
  string text5;
  string poly_original_string;
```

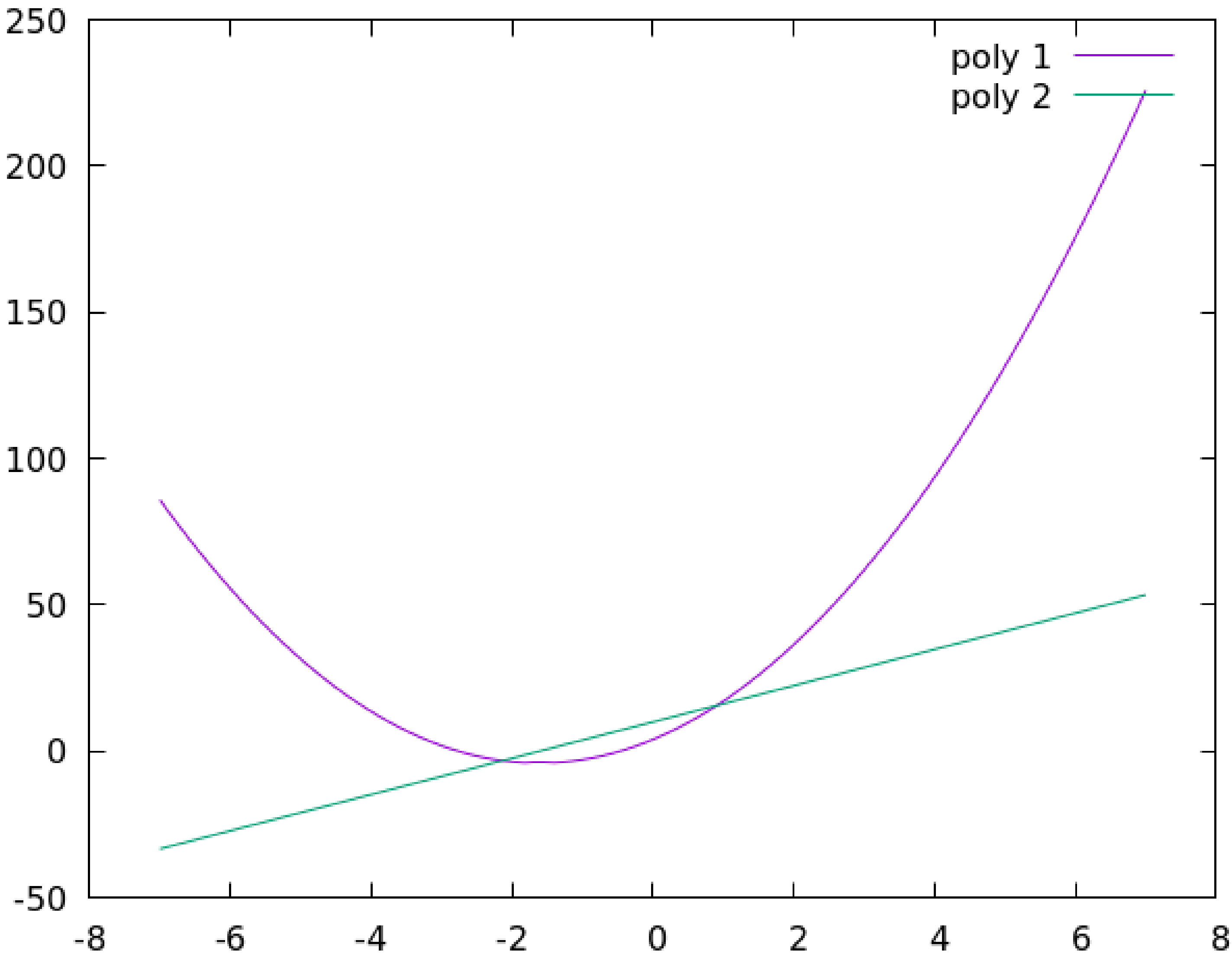
```
  string poly_derivative_string;
  string fp1;
  string fp2;
  string tdoc;

  poly_derivative_exps = initialize_ints(2);

  poly_original = new_poly([3.1, 10.0, 4.0], [2, 1, 0], 3);
  poly_original_order = order(poly_original);
  poly_original_string = print_tex(poly_original);

  consts_arr = poly_original#;

  poly_derivative_consts = derivative(consts_arr, poly_original_order);

  /* create the exponents array of derivative poly */
  for(i=0; i<poly_original_order; i=i+1){
    poly_derivative_exps[i] = i;
  }

  poly_derivative = new_poly(poly_derivative_consts, poly_derivative_exps, poly_original_order);
  poly_derivative_string = print_tex(poly_derivative);

  x1 = -7.0;
  x2 = 7.0;

  /* plot poly_original and poly_derivative together */

  return_code = range_plot([ poly_original, poly_derivative ], x1, x2, "plots/complexprogram_plot.png");

  /* calculate average slope of poly_original in range x1, x2 */
  average_slope = slope(x1, poly_original @ x1, x2, poly_original @ x2);

  /* collect several values on poly_derivative */
  derivative_values = poly_values(poly_derivative, x1, x2, 1000);

  err = 0.3;
  /* if slope is in poly_derivative's values by margin of err */
  if( approx_in(average_slope, derivative_values, 1000, err) ){

    text1 = "Every time I read a LaTeX document, I think, wow, this must be correct! - Prof. Christos
        Papadimitriou \\";
    text2 = "So, let's prove the MVT with Proof By LaTeX and PolyWiz. Consider the polynomial:";
    text3 = "Also, consider its derivative:";
    text4 = "Now, let's plot them both:";
    text5 = "And, as you can observe, the MVT holds! QED via LaTeX and PolyWiz";
    fp1 = "polywizard.png";
    fp2 = "plots/complexprogram_plot.png";

    body = [fp1, text1, text2, poly_original_string, text3, poly_derivative_string, text4, fp2, text5];

    tdoc = (tex_document(body, [0,7]));
    printstr(tdoc);
  }

  return 0;
}
```

This prints the following text:

```
\documentclass{article}
\usepackage{graphicx}
\begin{document}
\begin{figure}[h]
\centering
```

```
\includegraphics[width=2.5in]{polywizard.png}
\label{fig_sim}
\end{figure}
Every time I read a LaTeX document, I think, wow, this must be correct! - Prof. Christos Papadimitriou \\\\
So, let's prove the MVT with Proof By LaTeX and PolyWiz. Consider the polynomial:\\
$$3.100000x^{2}+10.000000x+4.000000$$\\
Also, consider its derivative:\\
$$+6.200000x+10.000000$$\\
Now, let's plot them both:\\
\begin{figure}[h]
\centering
\includegraphics[width=2.5in]{plots/complexprogram_plot.png}
\label{fig_sim}
\end{figure}
And, as you can observe, the MVT holds! QED via LaTeX and PolyWiz\\
\end{document}
```

And it saves the following plot at `plots/complexprogram_plot.png`:



4 Project Plan

4.1 Our Processes

Coordinating a large software project such as this was a task made even more difficult by the online semester where we were all living in different states. To combat the inherent disorder of the times, we put a very structured schedule into place. We had meetings over Zoom every Monday and Wednesday; we would also meet with our TA, Hans Montero, on Wednesdays before our own meetings. Zoom actually ended up working quite nicely, since we could share our screens with our code rather than all having to hunch over one laptop and breathe germs on each other during a pandemic. Additional meetings were called as needed. While we did establish a consistent cadence, Google Calendar was incredibly helpful for staying organized, and our group chat was always active.

We typically spent our Monday meetings outlining what needed to get done that week. Aggressive timelines were usually set, since we collectively preferred to be overworked but ahead than relaxed but behind. We would then individually start our own tasks on Tuesday, and discuss any roadblocks on Wednesday. This allowed us to attend Prof. Edwards's office hours later in the week for additional help if need be. Since we wrote a lot of code in small groups of two to three people, we would sometimes organize pair programming sessions over the weekend. Needless to say, we spent a lot of time together (including some very late nights) and were always aware of what was going on.

4.2 Programming Style Guide

Writing readable code is an incredibly important and often overlooked aspect of technical communication. To ensure that our code was as clean and as readable as possible, we tried our best to stick to the following rules for the OCaml portion of our code:

1. Use pattern matching wherever possible; avoid "if" and "then".
2. Make sure indentations align with each other.
3. Name all variables with underscores (snake case).

Similar rules were followed in the code that we wrote in C to ensure consistency throughout our entire stack.

4.3 Project Timeline

Date	Task
February 3	Language Proposal Submitted
February 24	Language Reference Manual Submitted
March 3	Scanner and Parser Complete
March 12	AST Complete
March 24	Hello World Submitted
April 25	Final Report and Presentation Complete
April 26	Final Presentation

4.4 Roles and Responsibilities

At the start of the project, our team member's were assigned the following roles:

Project Manager - Rose Chrin
Language Guru - Aditya Kankariya
System Architects - Max Helman and Tamjeed Azad
Tester - Anthony Pitts

However, as the project progressed, we realized that placing responsibility strictly inside the bounds of these titles would be a subpar approach to development. Instead, we followed the Agile methodology of iterative and incremental design of features. In other words, we split up the work by features, rather than feature domains. For example, if Rose Chrin were assigned to building polynomial addition, she would write all the required code for that functionality in the scanner, parser, AST, codegen, test suite, etc. At first, our team tried to break it up by having one person work in the scanner, one in the parser, one on testing... but soon realized that this was a detriment to the learning process and also made it nearly impossible to debug.

The amount of features that each team member built were approximately the same. Every team member contributed to all deliverables, such as the LRM and Final Report, equally.

4.5 Development Environment

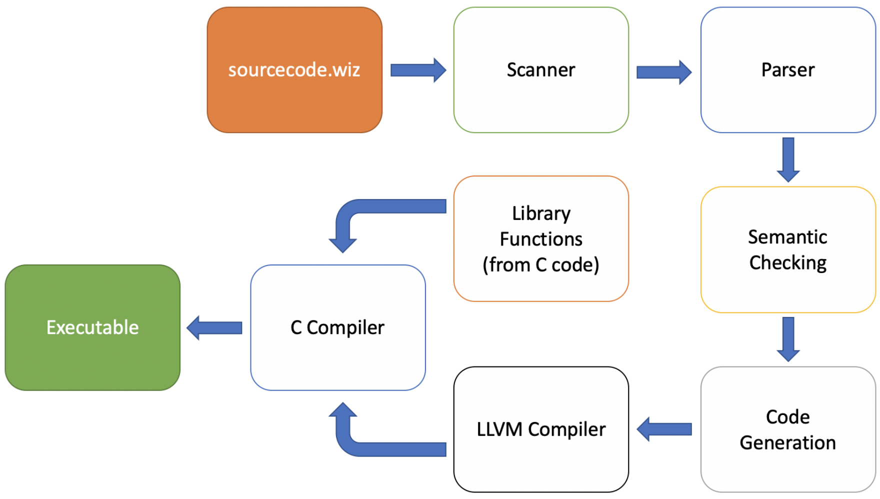
We used different coding environments, each suited to our preferences. A combination of VSCode, Sublime Text, Atom, Vim, and TMux were used as code editors; for maintaining dependencies and successful compilation and testing, Docker was used. Source code files were written mostly in OCaml, with a substantial amount of library functions written in C.

4.6 Project Log

Date	Task
February 3	Language Proposal Submitted
February 24	Language Reference Manual Submitted
March 3	Scanner and Parser Complete
March 12	AST Complete
March 24	Hello World Submitted
April 2	Strings Complete
April 6	Regression Tests Complete
April 10	Poly and Arrays Complete
April 15	Plotting and LaTeXComplete
April 17	Remaining Functionality Complete
April 25	Final Report and Presentation Complete
April 26	Final Presentation

5 Architectural Design

5.1 Diagram



Please note that due to the way we distributed work, we assigned ourselves features rather than levels of the stack. As such, we all worked on most or all parts of the compiler, even if the scope of our individual work in each level was narrow.

5.2 The Scanner

A PolyWiz program is used as input to the scanner (scanner.mll) which generates tokens accordingly. Tokens are generated for keywords, operations, literal values (such as string and float), and identifiers as described in our language reference manual.

5.3 The Parser

The tokens that the scanner generates are then fed into the parser, which converts them into an abstract syntax tree (AST). The AST must be consistent with the grammar we defined with production rules in polywizparse.mly and the datatypes we defined in ast.ml. Successful parsing indicates that a program is syntactically correct, although there may still be semantic errors. Our parser was written in OCamlYacc.

5.4 The Semantic Checker

The job of the semantic checker (semant.ml) is to create an extended abstract syntax tree (SAST) from the AST of objects. It does this through recursion to traverse the AST, while using an environment record to create a table. This table maps a string identifier to an object stack. In this stage, typing and scoping errors are caught such as proper assignment of values to variables based on their types. If a failure occurs, an error message is printed to the user.

5.5 The Code Generator

The code generator takes in the semantically checked AST, and translates it to output an LLVM intermediate representation of the source program. It first defines types in terms of native LLVM types, for example specifically defining ints, booleans, and floats as native LLVM `i32_type`, `i1_type`, and `double_type`. It defines the types of strings, polys, and arrays as pointers of their constituent types, such as `string_t = L.pointer_type i8_t`. After this, it describles SAST translation for instantiating arrays, accounting for global variables, and defining and filling functions, and lastly goes through a long list of patterns for building and evaluating expressions, often evaluating expressions using native LLVM operations but also using calls to the C function library where most applicable. It then describes translation for statements in the code, describes code for control flow, and describes translation for `while` and `for` loops. After all of this has been evaluated, the translated SAST is dumped into an LLVM module, which is then compiled using the LLVM compiler to generate assembly code; this assembly code is then used by the C compiler to link to the library functions written in C and generate the final executable file.

5.6 The C Library

The external C library named "library_functions.c" is linked to the llvm code that programs are compiled into. This allows for the Code Generator to use all the functionalities that it depends on. These functionalities provided in the C library center around Poly, plotting, and latex integration. The creation and allocation of poly-type variable are performed in this library, along with all poly operations. The library also has several functions to perform system calls to GNU plot, which is the plotting library that PolyWiz implements. Lastly, there are functions within the library that are dedicated to the formatting of latex string output.

6 Testing Plan

6.1 Test Suite and Example Tests

PolyWiz was built incrementally by adding functionality and the corresponding test to guarantee its successful build. The entire test suite can be found in the `./tests/` directory, at the root of the docker instance. Each test program designed to be successful is written in a `.wiz` file and the expected output is in a `.out` file, whose base filenames are identical; those that are designed to fail have the expected error saved in a `.err` file, once again with identical basenames. More detail about the automation of these tests and how to run them are discussed below. When these tests are run, their `.wiz` file output is compared to the expected output in the corresponding `.out` (or `.err`) file. The success or failure of the test, as compared to the anticipated output, is sent to the `testall.log` file. Two example tests and their correct output files are shown below:

test_poly_equal_comparison.wiz

```
def int main()
{
  poly poly1;
  poly poly2;

  poly1 = new_poly([0.0, 0.0, 4.0], [3, 1, 0], 3);
  poly2 = new_poly([4.0], [0], 1);
```

```
  printb(poly1 == poly2);

  return 0;
}
```

test_poly_equal_comparison.out

```
1
```

test_poly_not_equal_comparison.wiz

```
def int main()
{
  poly poly1;
  poly poly2;

  poly1 = new_poly([0.0, 0.0, 4.0], [3, 1, 0], 3);
  poly2 = new_poly([4.0], [0], 1);

  printb(poly1 != poly2);

  return 0;
}
```

test_poly_not_equal_comparison.out

```
0
```

fail-plot_wrong_type.wiz

```
def int main()
{
  poly poly1;
  poly poly2;
  poly poly_product;

  poly1 = new_poly([1.0, 2.0, 4.0], [2, 1, 0]);
  poly2 = new_poly([7.0, 4.0], [1, 0]);
  poly_product = poly1 * poly2;

  printint(plot([ 1.0, 3.0, 4.0 ], "plots/plot_single.png"));

  return 0;
}
```

fail-plot_wrong_type.err

```
Fatal error: exception Failure("illegal argument found float[] expected poly[] in [1.0, 3.0, 4.0]")
```

6.2 Automating Testing

As we had a large number of regression tests, automating testing was key to successful, smooth testing. A testing script is defined in `testall.sh`, and when this is run, every test code file in the `tests/` directory is sequentially compiled, executed, and compared against the desired output for that test file; for tests that are designed to fail, the returned error is checked against the expected error in that file's companion `<filename>.err` file. While each of these test are running, the dynamically created `.ll` and `.s` files for each test are removed upon successful test operation. In the event that a test fails, a `<filename>.diff` file is created, displaying the discrepancy between the desired output and the output that was produced.

Lastly, as mentioned in the Language Tutorial section, if someone wanted to do a bulk run of the regression test suite in PolyWiz' provided docker instance repository, they could simply run "**make**", as this will just execute `testall.sh`. This would do all the necessary compiling and execution of all test files to confirm that the compiler and language are working properly. The output of the test results can be found in the `testall.log` file.

6.3 Choosing and Writing Test Cases

Unit tests for a specific language feature were explicitly written by the author of that particular feature. This was done because the feature's author was assumed to be more intimately familiar with edge cases and the underlying implementation. We also assumed that the feature's author could come up with a use case that was perhaps most representative for the feature. That is not to say others did not have their eyes on what could go wrong; each group member individually wrote larger integration tests. This system ensured that everyone's features were subject to torture from each group member; essentially, we distributed the workload in a way that ensured features were tested rigorously by their original author, and then at least once again by someone else.

7 Lessons Learned

"There are no regrets in PLT, just lessons learned."

7.1 Tamjeed Azad

While staying home for the year due to the pandemic, the team project was one of my favorite aspects of the class; I found it really enjoyable to be able to consistently discuss difficult, interesting topics from the class, as well as put it into practice to make a real language while still being really far from campus. After putting countless hours of work in as a team throughout the semester, it feels really rewarding to see our final product come together at the end.

We had our stumbles, too; early on, we decided to implement large project features layer by layer instead of trying to implement small features end-to-end; this had disastrous consequences, as hours ended up going down the drain without any fruitful return, as fixing something in one layer, inevitable broke two or more things in another layer that another teammate had thought they had fixed. We quickly decided to shift towards the latter method to help ameliorate this. Additionally, we underestimated how much time, effort, and failure it would take to implement very basic features in our language, such as strings and arrays. Once we had gotten past these, though, the rest of the project was smooth sailing; people began getting tasks done and features implemented left and right, and our final language really came together in fruition over the final month.

Personally, I also appreciated how much I learned about a compiler's infrastructure, which I'd never really gotten a chance to appreciate before, especially being a student on the Intelligent Systems track. I spent a lot of time hashing out the plotting functionality using `gnuplot`, having to create a pipeline from the `codegen.ml` file, through the library functions implemented in C, to dynamically implemented system calls to `gnuplot`. I found it amazing when it finally worked to see firsthand how different layers of the code and the compiler worked with one another to successfully create our desired output.

My advice to future teams is to form groups with people who you know will put in the necessary work throughout the semester to get this project done. Even if the skill sets are not initially there, I personally think the honest willingness to learn and put in the time for the project is the most important trait in a teammate; a lot of the skill sets to develop your language you develop over the time of the class anyways. Good team chemistry is also a must, whether you are all friends or just peers; to do the project, you'll spend A LOT of time coding together, so without some level of chemistry and trust, and ability to both take and give constructive criticism and helpful advice, it will be very difficult to get things done in a timely fashion. Pick good teammates, trust them to put in the work, and give your teammates a reason to trust you too by consistently completing tasks from your end of things. Ensure these things, and you're bound to have a fulfilling project:)

7.2 Rose Chrin

This project gave me valuable lessons in both computer science but as well as working with others. Due to the nature of the online semester, our regular group meetings became a highlight for me. Often times, our meetings would run long because we were just discussing life with one another and checking-in to see how everyone was doing. As a result, I think we were able to form a much deeper bond as a group which, in turn, I believe was instrumental for our project and language. This bond allowed us to understand when someone was having a hard week and needed support or to ask the group for help if someone was struggling with a specific task. Contrasting this group to that of a different project I am in this semester, I learned the importance of group dynamic as well as communication when working on a team. In addition to our weekly or twice weekly meetings, we also had an active group chat to keep everyone updated with progress or problems.

From a computer science standpoint, this project taught we what it is like to build something from scratch. While in other classes I have had small projects, I had never had the opportunity to build something from start to finish. In doing so, I learned the importance of having a strong plan and flexibility at the same time. For example, we started off the project trying to do way too much at the same time. When trying to do "Hello World", we quickly learned that this approach was not successful. As a result, we adapted to a system that focused on implementing a single feature at the time across the stack which proved to be much more efficient. I also learned the importance of pair-programming and having someone to discuss ideas with. This made us much more efficient and helped us avoid mistakes.

My advice to future teams is to pick your team wisely focusing on the group dynamic and over communication. Choosing a strong group can be very difficult when friends are involved or if you don't know anyone in the class but is critical. At the same time, however, have a strong chemistry does not necessarily mean that you all have to know each other before. If this is case, I would advise to focus on team bonding in the early stages of the project. Whether its eating dinner together one night, doing homework, or just getting to know one another better, these activities help build trust and comfort amongst group members. This is important because it contributes to building a space where team members feel comfortable stating their true opinions, debating with other members if they don't agree, holding people accountable for their assigned work, and asking for help if needed. In addition, I think communication is really important in this project. When working on a language that has many inter-connected components, it is imperative to communicate to the rest of the group. Scheduling weekly calls (at a minimum), maintaining a group chat, good commit messages on Github, and being willing to be honest with the group all contribute to a successful team. Lastly, I recommend working on this project in small steps, focusing on implementing a single feature at a time. As discussed earlier, this saves time in the long run and is super helpful for debugging.

7.3 Max Helman

We originally tried dividing the project layer by layer; we soon realized that one person should oversee a feature in its entirety from top to bottom, with at least a second set of eyes on it at all times. Pair programming proved to be invaluable, even if it required more overhead in terms of planning. It was also important to know how everyone's skillsets fit together; I had previous functional programming experience, so I would volunteer to help with OCaml-specific questions. When I needed help with something like memory management in C, I also knew exactly who to ask.

I cannot emphasize the importance of having a second set of eyes on things. Whether it was for a purely technical question or just to see what was going on, working in this way helped me understand the majority of the program, specifically the parts I did not write. It was also incredibly helpful when I would be working with someone and one of us would encounter a bug that the other one had seen before. We all had some brilliant ideas; we also all had some terrible ones. Having someone else there exposed me to their brilliant ideas and kept my terrible ones in check. I think all of our contributions go well beyond the lines GitHub says we added.

I'm very proud that we were able to pull this off in an online semester. We are located in five different states across the country, but thanks to regular Zoom meetings, Google Calendar, and a group chat, we stayed very organized and on top of things.

My advice to future groups follows from above. Pick a well-rounded team, and ideally one where everyone knows each other's skillsets. You will be spending a lot of time together, so make sure you know they are accountable– I think in many ways, previous friends are more accountable since if they are reasonable people, they will not want to ruin a friendship over this project. Do not be afraid to speak up, but also listen to everything your teammates have to say; democracy rules in situations such as these. Assign each team member a language feature to be responsible for (this has the added benefit of everyone becoming intimately familiar with all layers of the stack). Perhaps most importantly, get good at giving and receiving criticism. Constructive feedback is much more helpful than simply stating everything looks good when, in fact, it does not.

7.4 Aditya Kankariya

I would highly recommend this class to anyone who's interested in improving their software engineering skills and gaining invaluable project experience with a team. Over the course of this semester, I learned countless things while working in a team environment to build a compiler. The lesson that stood out to me the most is having an understanding towards other team members. With a project of this size and scope, there were not just one or two people that envisioned PolyWiz. From the very beginning, everybody contributed unique, brilliant ideas that were all major aspects of the language. The whole team carried everyone.

I always thought of myself as someone that wrote code on my own, but after this experience, I realized that having a talented and hardworking team of engineers working towards a common goal enabled us to efficiently build such a robust language. What impressed me the most was that we accomplished this working remotely from different sides of the country thanks to our solid communication.

My advice to future groups is to pick members that will over communicate and be there for one another as much as possible. Make sure that there is always a roadmap for what is to be done in the current week, the next week, and long-term. Also, do not be afraid to speak up, be upfront, and hold each other accountable. I think that strong communication is the biggest key to working effectively within a group, especially when we are working remotely. If there is one thing we could do better from the start, we would have implemented one feature at a time end-end rather than adding all the features to one file at once and then moving onto the next file down the stack.

I can confidently say that no other class in my Columbia experience has better prepared me for whatever I will go on to do after college. I learned the importance of working together and playing to everyone's strengths to optimize completion. I love how we learned to better trust each other this semester and take criticism in a constructive manner. Most importantly, I have an amazing new set of friends from this class.

7.5 Anthony Pitts

Among the countless things that I learned throughout this project's duration, the two most crucial were functional programming and regression testing. I came into this class with hardly any functional programming experience and was able to slowly understand the many paradigms in functional programming. Specific to OCaml, I learned the power of pattern matching and not relying on classic control flow loops like for and while. In terms of regression testing, I learned about how to incrementally add smaller pieces of code and the tests that check its accuracy. Our team initially added many features to the language in the compiler all at once, and then there were too many errors to manage. However, we then choose a more incremental approach and were able to complete tasks much more consistently and efficiently.

I would also be sure to mention the importance of meeting all deadlines. Luckily, I worked with a group who was very good at meeting these deadlines. Some of the ways that we managed this was by having a group chat where we could instantaneously update each other about out progress or any issues we were having. This quick communication kept us informed about not only technical details but also administrative details around project deadlines. I learned that it is always the best choice to communicate more rather than less. There were several nights where our group sent hundreds of messages to each other. As long as team members are able to mute that chat when necessary, frequent communication helped us greatly in meeting all our deadlines.

Advice: My advice to any future teams taking on this project would be to learn about the power of incremental testing before they start coding the compiler. As stated above, our team made the mistake of adding all the code at once and that ended up being a waste of our time because it was impossible to debug. Instead, work on the project feature-by-feature to guarantee each step was successfully completed. Our team used git to manage our version control and I wuld certainly recommend this to everyone. By creating distinct branches for features we kept our repo very organized and simple to navigate.

8 Appendix

8.1 Dockerfile

```
# Based on 20.04 LTS
FROM ubuntu:focal

ENV TZ=America/New_York
RUN apt-get -yq update
RUN DEBIAN_FRONTEND="noninteractive" apt-get -y install tzdata
RUN apt-get -y upgrade && \
    apt-get -yq --no-install-suggests --no-install-recommends install \
    ocaml \
    menhir \
    llvm-10 \
    llvm-10-dev \
    m4 \
    git \
    aspcud \
    ca-certificates \
    python2.7 \
    pkg-config \
    cmake \
    opam \
    gnuplot

```

```
RUN ln -s /usr/bin/lli-10 /usr/bin/lli
RUN ln -s /usr/bin/llc-10 /usr/bin/llc

RUN opam init --disable-sandboxing
RUN opam install -y \
    llvm.10.0.0 \
    ocamlfind \
    ocamlbuild

WORKDIR /root

ENTRYPOINT ["opam", "config", "exec", "--"]

CMD ["bash"]
```

8.2 Makefile

```
# "make test" Compiles everything and runs the regression tests

.PHONY : test
test : all testall.sh
   ./testall.sh

# "make all" builds the executable as well as the "library_functions" library designed
# to test linking external code

.PHONY : all
all : polywiz.native library_functions

# "make polywiz.native" compiles the compiler
#
# The _tags file controls the operation of ocamlbuild, e.g., by including
# packages, enabling warnings
#
# See https://github.com/ocaml/ocamlbuild/blob/master/manual/manual.adoc

polywiz.native :
   opam config exec -- \
   ocamlbuild -use-ocamlfind polywiz.native

# "make clean" removes all generated files

.PHONY : clean
clean :
   ocamlbuild -clean
   rm -rf testall.log library_functions ocamlllvm *.diff *.o

# Testing the "library_functions" example
library_functions : library_functions.o
   gcc -o library_functions library_functions.c -DBUILD_TEST -lm

# Building the tarball

TESTS = \
  add1 arith1 arith2 arith3 fib float1 float2 float3 for1 for2 func1 \
  func2 func3 func4 func5 func6 func7 func8 func9 gcd2 gcd global1 \
  global2 global3 hello if1 if2 if3 if4 if5 if6 local1 local2 ops1 \
  ops2 library_functions var1 var2 while1 while2

FAILS = \
  assign1 assign2 assign3 dead1 dead2 expr1 expr2 expr3 float1 float2 \
  for1 for2 for3 for4 for5 func1 func2 func3 func4 func5 func6 func7 \
  func8 func9 global1 global2 if1 if2 if3 nomain library_functions printb print \
```

```
  return1 return2 while1 while2

TESTFILES = $(TESTS:%=test-%.pwiz) $(TESTS:%=test-%.out) \
      $(FAILS:%=fail-%.pwiz) $(FAILS:%=fail-%.err)

TARFILES = ast.ml sast.ml codegen.ml Makefile _tags polywiz.ml polywizparse.mly \
   README scanner.mll semant.ml testall.sh enchant \
   library_functions.c \
   Dockerfile \
   $(TESTFILES:%=tests/%)

polywiz.tar.gz : $(TARFILES)
   cd .. && tar czf polywiz/polywiz.tar.gz \
      $(TARFILES:%=polywiz/%)
```

8.3 enchant (compiling script)

```
#!/bin/sh

# Swift compilation script for PolyWiz

# Path to the LLVM interpreter
LLI="lli"
#LLI="/usr/local/opt/llvm/bin/lli"

# Path to the LLVM compiler
LLC="llc"

# Path to the C compiler
CC="cc"

# Path to the polywiz compiler. Usually "./polywiz.native"
# Try "_build/polywiz.native" if ocamlbuild was unable to create a symbolic link.
POLYWIZ="./polywiz.native"
#POLYWIZ="_build/polywiz.native"

# Set time limit for all operations
ulimit -t 30

inputfile=${1}
outputfile=${2}

Run() {
    echo $* 1>&2
    eval $* || {
    SignalError "$1 failed on $*"
    return 1
    }
}

Usage() {
    echo "Usage: ./enchant [source_code.wiz] [output filename]"
    echo "-h   Print help"
    exit 1
}

while getopts kdpsh c; do
    case $c in
    h) # Help
        Usage
        ;;
    esac
done
```

```
shift `expr $OPTIND - 1`

basename=`echo $1 | sed 's/.*\\///
                        s/.wiz//'`

generatedfiles="${basename}.ll ${basename}.s"

Run "$POLYWIZ" "$inputfile" ">" "${basename}.ll" &&
Run "$LLC" "-relocation-model=pic" "${basename}.ll" ">" "${basename}.s" &&
Run "$CC" "-o" "$outputfile" "${basename}.s" "library_functions.o -lm"
Run "rm -f $generatedfiles"

exit $globalerror
```

8.4 scanner.mll (scanner)

```
(* Ocamllex scanner for PolyWiz *)

{ open Polywizparse }

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

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"     { comment lexbuf }          (* Comments *)
| '('      { LPAREN }
| ')'      { RPAREN }
| '['      { LBRACK }
| ']'      { RBRACK }
| '{'      { LBRACE }
| '}'      { RBRACE }
| ';'      { SEMI }
| ':'      { COMPO }
| '@'      { EVAL }
| '#'      { CONST_RET }
| ','      { COMMA }
| '+'      { PLUS }
| '-'      { MINUS }
| '*'      { TIMES }
| '/'      { DIVIDE }
| '^'      { EXP }
| "|"      { ABS }
| '='      { ASSIGN }
| "=="     { EQ }
| "!="     { NEQ }
| '<'      { LT }
| "<="     { LEQ }
| "in"     { IN }
| ">"      { GT }
| ">="     { GEQ }
| "and"     { AND }
| "or"     { OR }
| "not"      { NOT }
| "if"     { IF }
| "else"   { ELSE }
| "for"    { FOR }
| "while"  { WHILE }
| "return" { RETURN }
| "int"    { INT }
| "bool"   { BOOL }
| "float"  { FLOAT }
```

```
| "string" { STRING }
| "poly" { POLY }
| "void"  { VOID }
| "true"  { BLIT(true) }
| "false" { BLIT(false) }
| "def"   { DEF }
| digits as lxm { LITERAL(int_of_string lxm) }
| digits '.' digit* ( ['e' 'E'] ['+' '-']? digits )? as lxm { FLIT(lxm) }
| ('"'[^'"''\\']*('\\'_[^'"''\\']*)*'"') as lxm { SLIT(lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

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

8.5 polywizparse.mly (parser)

```
/* Ocamlyacc parser for PolyWiz */

%{
open Ast
%}

%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA PLUS MINUS TIMES DIVIDE ASSIGN
%token NOT EQ NEQ LT LEQ GT GEQ AND OR IN
%token LBRACK RBRACK
%token RETURN IF ELSE FOR WHILE INT BOOL FLOAT VOID DEF STRING POLY
%token EXP ABS COMPO EVAL CONST_RET
%token <int> LITERAL
%token <bool> BLIT
%token <string> ID FLIT SLIT
%token EOF

%start program
%type <Ast.program> program

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left OR
%left AND
%left ABS
%left EQ NEQ
%left LT GT LEQ GEQ
%left IN
%left PLUS MINUS
%left TIMES DIVIDE
%left EXP
%left CONST_RET
%left EVAL
%left COMPO
%right NOT
%nonassoc LBRACK
%nonassoc RBRACK
%nonassoc LBRACE
%nonassoc RBRACE
%nonassoc LPAREN
%nonassoc RPAREN

%%

```

```
program:
  decls EOF { $1 }

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

fdecl:
   DEF typ ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE
     { { typ = $2;
    fname = $3;
    formals = List.rev $5;
   locals = List.rev $8;
    body = List.rev $9 } }

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

formal_list:
    typ ID                 { [($1,$2)]    }
  | formal_list COMMA typ ID { ($3,$4) :: $1 }

typ:
   typ LBRACK RBRACK { Array($1) }
  |  INT   { Int   }
  | BOOL { Bool }
  | FLOAT { Float }
  | VOID { Void }
  | STRING { String }
  | POLY { Poly }

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

vdecl:
   typ ID SEMI { ($1, $2) }

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

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

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

element:
  {[]}
  | elements_list {List.rev $1}

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

```
expr:
   LBRACK element RBRACK { ArrayLit($2)   }
  |  LITERAL        { Literal($1)          }
  | FLIT           { Fliteral($1)        }
  | BLIT            { BoolLit($1)          }
  | SLIT            { Sliteral($1)         }
  | ID              { Id($1)               }
  | expr PLUS  expr { Binop($1, Add, $3)  }
  | expr MINUS expr { Binop($1, Sub, $3)  }
  | expr TIMES expr { Binop($1, Mult, $3) }
  | expr DIVIDE expr { Binop($1, Div, $3) }
  | expr EXP   expr { Binop($1, Exp, $3)  }
  | expr COMPO expr { Binop($1, Compo, $3) }
  | expr CONST_RET { Unop(Const_ret, $1)  }
  | expr EVAL  expr { Binop($1, Eval, $3) }
  | expr EQ    expr { Binop($1, Equal, $3) }
  | expr NEQ   expr { Binop($1, Neq, $3)  }
  | expr LT    expr { Binop($1, Less, $3) }
  | expr LEQ   expr { Binop($1, Leq, $3)  }
  | expr GT    expr { Binop($1, Greater, $3) }
  | expr GEQ   expr { Binop($1, Geq, $3)  }
  | expr IN    expr { Binop($1, In, $3)   }
  | expr AND   expr { Binop($1, And, $3)  }
  | expr OR    expr { Binop($1, Or,  $3)  }
  | expr LBRACK expr RBRACK { Binop($1, Ele_at_ind, $3) }
  | ABS expr ABS    { Unop(Abs, $2)        }
  | MINUS expr %prec NOT { Unop(Neg, $2)  }
  | NOT expr        { Unop(Not, $2)        }
  | expr LBRACK expr RBRACK ASSIGN expr { ArrAssignInd($1, $3, $6) }
  | ID ASSIGN expr { Assign($1, $3)       }
  | ID LPAREN args_opt RPAREN { Call($1, $3) }
  | LPAREN expr RPAREN { $2                }

args_opt:
    /* nothing */ { [] }
  | args_list { List.rev $1 }

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

8.6 ast.ml (AST)

```
(* Abstract Syntax Tree and functions for printing it *)

type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater | Geq |
          In | And | Or | Exp | Compo | Eval | Ele_at_ind

type uop = Neg | Not | Abs | Const_ret

type typ = Int | Bool | Float | Void | String | Array of typ | Poly

type bind = typ * string

type expr =
    Literal of int
  | Fliteral of string
  | Sliteral of string
  | BoolLit of bool
  | ArrayLit of expr list
  | Id of string
  | Binop of expr * op * expr
```

```
  | Unop of uop * expr
  | ArrAssignInd of expr * expr * expr
  | Assign of string * expr
  | Call of string * expr list
  | Noexpr

type stmt =
    Block of stmt list
  | Expr of expr
  | Return of expr
  | If of expr * stmt * stmt
  | For of expr * expr * expr * stmt
  | While of expr * stmt

type var_decl =
  Vardecl of typ * expr

type func_decl = {
    typ : typ;
    fname : string;
    formals : bind list;
    locals : bind list;
    body : stmt list;
  }

type program = bind list * func_decl list


(* Pretty-printing functions *)

let string_of_op = function
    Add -> "+"
  | Sub -> "-"
  | Mult -> "*"
  | Div -> "/"
  | Exp -> "^"
  | Equal -> "=="
  | Neq -> "!="
  | Less -> "<"
  | Leq -> "<="
  | Greater -> ">"
  | Geq -> ">="
  | In -> "in"
  | And -> "and"
  | Or -> "or"
  | Compo -> ":"
  | Eval -> "@"
  | Ele_at_ind -> "[]"


  let string_of_uop = function
    Neg -> "-"
  | Not -> "not "
  | Abs -> "| "
  | Const_ret -> "# "

  let string_of_2nd_uop = function
    Neg -> ""
  | Not -> ""
  | Const_ret -> ""
  | Abs -> " |"

let rec string_of_expr = function
    Literal(l) -> string_of_int l
  | Fliteral(l) -> l
```

```
  | Sliteral(l) -> l
  | BoolLit(true) -> "true"
  | BoolLit(false) -> "false"
  | ArrayLit (e) -> "[" ^ String.concat ", " (List.map string_of_expr e) ^ "]"
  | Id(s) -> s
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
  | Unop(o, e) -> string_of_uop o ^ string_of_expr e ^ string_of_2nd_uop o
  | Assign(v, e) -> v ^ " = " ^ string_of_expr e
  | ArrAssignInd(arr, ind, e) -> string_of_expr arr ^ string_of_expr ind ^ " = " ^ string_of_expr e
  | Call(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | Noexpr -> ""

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

let rec string_of_typ = function
    Int -> "int"
  | Bool -> "bool"
  | Float -> "float"
  | Void -> "void"
  | String -> "string"
  | Poly -> "poly"
  | Array(t) -> string_of_typ t ^ "[]"

let string_of_vdecl (t, id) = string_of_typ t ^ " " ^ id ^ ";\n"

let string_of_fdecl fdecl =
  "def " ^ string_of_typ fdecl.typ ^ " " ^
  fdecl.fname ^ "(" ^ String.concat ", " (List.map snd fdecl.formals) ^
  ")\n{\n" ^
  String.concat "" (List.map string_of_vdecl fdecl.locals) ^
  String.concat "" (List.map string_of_stmt fdecl.body) ^
  "}\n"

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

8.7 sast.ml (SAST)

```
(* Semantically-checked Abstract Syntax Tree and functions for printing it *)

open Ast

type sexpr = typ * sx
and sx =
    SLiteral of int
  | SFliteral of string
  | SBoolLit of bool
  | SSliteral of string
  | SArrayLit of sexpr list
```

```
  | SId of string
  | SBinop of sexpr * op * sexpr
  | SUnop of uop * sexpr
  | SArrAssignInd of sexpr * sexpr * sexpr
  | SAssign of string * sexpr
  | SCall of string * sexpr list
  | SNoexpr

type sstmt =
    SBlock of sstmt list
  | SExpr of sexpr
  | SReturn of sexpr
  | SIf of sexpr * sstmt * sstmt
  | SFor of sexpr * sexpr * sexpr * sstmt
  | SWhile of sexpr * sstmt

type sfunc_decl = {
    styp : typ;
    sfname : string;
    sformals : bind list;
    slocals : bind list;
    sbody : sstmt list;
  }

type sprogram = bind list * sfunc_decl list

(* Pretty-printing functions *)

let rec string_of_sexpr (t, e) =
  "(" ^ string_of_typ t ^ " : " ^ (match e with
    SLiteral(l) -> string_of_int l
  | SBoolLit(true) -> "true"
  | SBoolLit(false) -> "false"
  | SFliteral(l) -> l
  | SSliteral(l) -> l
  | SArrayLit (l) -> "[" ^ String.concat ", " (List.map string_of_sexpr l) ^ "]"
  | SId(s) -> s
  | SBinop(e1, o, e2) ->
      string_of_sexpr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_sexpr e2
  | SUnop(o, e) -> string_of_uop o ^ string_of_sexpr e
  | SAssign(v, e) -> v ^ " = " ^ string_of_sexpr e
  | SArrAssignInd(arr, ind, e) -> string_of_sexpr arr ^ string_of_sexpr ind ^ " = " ^ string_of_sexpr e
  | SCall(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_sexpr el) ^ ")"
  | SNoexpr -> ""
            ) ^ ")"

let rec string_of_sstmt = function
    SBlock(stmts) ->
      "{\n" ^ String.concat "" (List.map string_of_sstmt stmts) ^ "}\n"
  | SExpr(expr) -> string_of_sexpr expr ^ ";\n";
  | SReturn(expr) -> "return " ^ string_of_sexpr expr ^ ";\n";
  | SIf(e, s, SBlock([])) ->
      "if (" ^ string_of_sexpr e ^ ")\n" ^ string_of_sstmt s
  | SIf(e, s1, s2) -> "if (" ^ string_of_sexpr e ^ ")\n" ^
      string_of_sstmt s1 ^ "else\n" ^ string_of_sstmt s2
  | SFor(e1, e2, e3, s) ->
      "for (" ^ string_of_sexpr e1 ^ " ; " ^ string_of_sexpr e2 ^ " ; " ^
      string_of_sexpr e3 ^ ") " ^ string_of_sstmt s
  | SWhile(e, s) -> "while (" ^ string_of_sexpr e ^ ") " ^ string_of_sstmt s

let string_of_sfdecl fdecl =
  "def " ^ string_of_typ fdecl.styp ^ " " ^
  fdecl.sfname ^ "(" ^ String.concat ", " (List.map snd fdecl.sformals) ^
  ")\n{\n" ^
```

```
  String.concat "" (List.map string_of_vdecl fdecl.slocals) ^
  String.concat "" (List.map string_of_sstmt fdecl.sbody) ^
  "}\n"

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

8.8 semant.ml (Semantic Checker)

```
(* Semantic checking for the PolyWiz compiler *)

open Ast
open Sast

module StringMap = Map.Make(String)

(* Semantic checking of the AST. Returns an SAST if successful,
   throws an exception if something is wrong.

   Check each global variable, then check each function *)

let check (globals, functions) =

  (* Verify a list of bindings has no void types or duplicate names *)
  let check_binds (kind : string) (binds : bind list) =
    List.iter (function
   (Void, b) -> raise (Failure ("illegal void " ^ kind ^ " " ^ b))
      | _ -> ()) binds;
    let rec dups = function
        [] -> ()
      |  ((_,n1) :: (_,n2) :: _) when n1 = n2 ->
     raise (Failure ("duplicate " ^ kind ^ " " ^ n1))
      | _ :: t -> dups t
    in dups (List.sort (fun (_,a) (_,b) -> compare a b) binds)
  in

  (**** Check global variables ****)

  check_binds "global" globals;

  (**** Check functions ****)

  (* Collect function declarations for built-in functions: no bodies *)
  let built_in_decls =
    let add_bind map (name, ty) = StringMap.add name {
      typ = if name="new_poly" then Poly
            else if name="poly_at_ind" then Float
            else if name="to_str" then String
            else if name="tex_document" then String
            else if name="print_tex" then String
            else if name="order" then Int
            else if name="plot" then Int
            else if name="range_plot" then Int
            else if name="plot_many" then Int
            else if name="range_plot_many" then Int
            else if name="initialize_floats" then Array(Float)
            else if name="initialize_ints" then Array(Int)
            else if name="int_to_float" then Float
            else Void;
      fname = name;
      formals = if name="new_poly" then [(Array(Float), "x"); (Array(Int), "y"); (Int, "z")]
                else if name="poly_at_ind" then [(Poly, "x");(Int, "y")]
```

```
            else if name="to_str" then [(Poly, "x")]
            else if name="tex_document" then [(Array(String), "x"); (Array(Int), "y")]
            else if name="print_tex" then [(Poly, "x")]
            else if name="order" then [(Poly, "x")]
            else if name="plot_many" then [(Array(Poly), "x"); (String, "y")]
            else if name="range_plot_many" then [(Array(Poly), "x");(Float, "y");(Float, "z");(String, "y")]
            else if name="initialize_floats" then [(Int, "x")]
            else if name="initialize_ints" then [(Int, "x")]
            else if name="int_to_float" then [(Int, "x")]
            else if name="plot" then [(Array(Poly), "x"); (String, "y")]
            else if name="range_plot" then [(Array(Poly), "x");(Float, "y");(Float, "z");(String, "y")]
            else  [(ty, "x")];
    locals = []; body = [] } map
  in List.fold_left add_bind StringMap.empty [ ("printint", Int);
                             ("printb", Bool);
                             ("printf", Float);
               ("printstr", String);
          ("new_poly", Bool);
          ("to_str", Bool);
          ("tex_document", String);
          ("print_tex", String);
          ("order", Bool);
          ("plot", Bool);
          ("range_plot", Bool);
          ("plot_many", Bool);
          ("range_plot_many", Bool);
          ("initialize_floats", Bool);
          ("initialize_ints", Bool);
          ("int_to_float", Bool);
          ("poly_at_ind", Bool) ]
in

(* Add function name to symbol table *)
let add_func map fd =
  let built_in_err = "function " ^ fd.fname ^ " may not be defined"
  and dup_err = "duplicate function " ^ fd.fname
  and make_err er = raise (Failure er)
  and n = fd.fname (* Name of the function *)
  in match fd with (* No duplicate functions or redefinitions of built-ins *)
       _ when StringMap.mem n built_in_decls -> make_err built_in_err
     | _ when StringMap.mem n map -> make_err dup_err
     | _ -> StringMap.add n fd map
in

(* Collect all function names into one symbol table *)
let function_decls = List.fold_left add_func built_in_decls functions
in

(* Return a function from our symbol table *)
let find_func s =
  try StringMap.find s function_decls
  with Not_found -> raise (Failure ("unrecognized function " ^ s))
in

let _ = find_func "main" in (* Ensure "main" is defined *)

let check_function func =
  (* Make sure no formals or locals are void or duplicates *)
  check_binds "formal" func.formals;
  check_binds "local" func.locals;

  (* Raise an exception if the given rvalue type cannot be assigned to
     the given lvalue type *)
  let check_assign lvaluet rvaluet err =
     if lvaluet = rvaluet then lvaluet else raise (Failure err)
```

```
    in

    (* Build local symbol table of variables for this function *)
    let symbols = List.fold_left (fun m (ty, name) -> StringMap.add name ty m)
                  StringMap.empty (globals @ func.formals @ func.locals )
    in

    (* Return a variable from our local symbol table *)
    let type_of_identifier s =
      try StringMap.find s symbols
      with Not_found -> raise (Failure ("undeclared identifier " ^ s))
    in

    (* Return a semantically-checked expression, i.e., with a type *)
    let rec expr = function
        Literal l -> (Int, SLiteral l)
      | Fliteral l -> (Float, SFliteral l)
      | BoolLit l -> (Bool, SBoolLit l)
      | Sliteral l -> (String, SSliteral l)
      | ArrayLit l ->
      if List.length l > 0 then
          let l' = List.map expr l in
          let array_type = (List.nth l' 0) in
          match array_type with
            (Int,_) -> (Array(Int), SArrayLit l')
          | (Float,_) -> (Array(Float), SArrayLit l')
          | (Bool,_) -> (Array(Bool), SArrayLit l')
          | (String,_) -> (Array(String), SArrayLit l')
          | (Poly,_) -> (Array(Poly), SArrayLit l')
          |  _ -> raise (Failure ("not a valid array type"))
      else (Void, SArrayLit([]))
      | Noexpr     -> (Void, SNoexpr)
      | Id s       -> (type_of_identifier s, SId s)
      | ArrAssignInd(e1, ind, e) as ex ->
          let (lt_arr, _) = expr e1 in
          let lt = (match lt_arr with
             Array(Int) -> Int
            |Array(Float) -> Float
            |Array(Bool) -> Bool
            |_ -> raise (Failure ("This array type does not index assignment."))
          ) in
          let (rt, e') = expr e in
          let err = "illegal assignment " ^ string_of_typ lt ^ " = " ^
            string_of_typ rt ^ " in " ^ string_of_expr ex in
          (check_assign lt rt err, SArrAssignInd(expr e1, expr ind, (rt, e')))
      | Assign(var, e) as ex ->
          let lt = type_of_identifier var
          and (rt, e') = expr e in
          let err = "illegal assignment " ^ string_of_typ lt ^ " = " ^
            string_of_typ rt ^ " in " ^ string_of_expr ex
          in (check_assign lt rt err, SAssign(var, (rt, e')))
      | Unop(op, e) as ex ->
          let (t, e') = expr e in
          let ty = match op with
            Neg when t = Int || t = Float -> t
          | Not when t = Bool -> Bool
          | Abs when t = Int || t = Float -> t
          | Const_ret when t = Poly -> Array(Float)
          | _ -> raise (Failure ("illegal unary operator " ^
                                 string_of_uop op ^ string_of_typ t ^
                                 " in " ^ string_of_expr ex))
          in (ty, SUnop(op, (t, e')))
      | Binop(e1, op, e2) as e ->
          let (t1, e1') = expr e1
          and (t2, e2') = expr e2 in
```

```
     (* All binary operators require operands of the same type *)
     let same = t1 = t2 in
     (* Determine expression type based on operator and operand types *)
     let ty = match op with
       Add | Sub | Mult | Div | Exp when same && t1 = Int -> Int
     | Add | Sub | Mult | Div | Exp when same && t1 = Float -> Float
     | Add | Sub | Mult | Compo when same && t1 = Poly -> Poly
     | Div when t1=Poly && t2=Float -> Poly
     | Eval when t1=Poly && t2=Float -> Float
     | Exp when false==same-> Float
     | Equal | Neq            when same              -> Bool
     | Less | Leq | Greater | Geq
              when same && (t1 = Int || t1 = Float) -> Bool
     | In when t2=Array(t1) -> Bool
     | And | Or when same && t1 = Bool -> Bool
     | Ele_at_ind -> (match t1 with
           Array(Int) -> Int
         | Array(Float) -> Float
         | Array(Bool) -> Bool
         | _ -> raise (Failure ("This array type does not support indexing."))
       )
     | _ -> raise (
    Failure ("illegal binary operator " ^
                 string_of_typ t1 ^ " " ^ string_of_op op ^ " " ^
                 string_of_typ t2 ^ " in " ^ string_of_expr e))
     in (ty, SBinop((t1, e1'), op, (t2, e2')))
  | Call(fname, args) as call ->
     let fd = find_func fname in
     let param_length = List.length fd.formals in
     if List.length args != param_length then
       raise (Failure ("expecting " ^ string_of_int param_length ^
                       " arguments in " ^ string_of_expr call))
     else let check_call (ft, _) e =
       let (et, e') = expr e in
       let err = "illegal argument found " ^ string_of_typ et ^
         " expected " ^ string_of_typ ft ^ " in " ^ string_of_expr e
       in (check_assign ft et err, e')
     in
     let args' = List.map2 check_call fd.formals args
     in (fd.typ, SCall(fname, args'))
in

let check_bool_expr e =
  let (t', e') = expr e
  and err = "expected Boolean expression in " ^ string_of_expr e
  in if t' != Bool then raise (Failure err) else (t', e')
in

(* Return a semantically-checked statement i.e. containing sexprs *)
let rec check_stmt = function
    Expr e -> SExpr (expr e)
  | If(p, b1, b2) -> SIf(check_bool_expr p, check_stmt b1, check_stmt b2)
  | For(e1, e2, e3, st) ->
 SFor(expr e1, check_bool_expr e2, expr e3, check_stmt st)
  | While(p, s) -> SWhile(check_bool_expr p, check_stmt s)
  | Return e -> let (t, e') = expr e in
    if t = func.typ then SReturn (t, e')
    else raise (
 Failure ("return gives " ^ string_of_typ t ^ " expected " ^
     string_of_typ func.typ ^ " in " ^ string_of_expr e))

   (* A block is correct if each statement is correct and nothing
     follows any Return statement. Nested blocks are flattened. *)
  | Block sl ->
     let rec check_stmt_list = function
```

```
            [Return _ as s] -> [check_stmt s]
          | Return _ :: _ -> raise (Failure "nothing may follow a return")
          | Block sl :: ss -> check_stmt_list (sl @ ss) (* Flatten blocks *)
          | s :: ss        -> check_stmt s :: check_stmt_list ss
          | []             -> []
        in SBlock(check_stmt_list sl)

    in (* body of check_function *)
    { styp = func.typ;
      sfname = func.fname;
      sformals = func.formals;
      slocals = func.locals;
      sbody = match check_stmt (Block func.body) with
   SBlock(sl) -> sl
      | _ -> raise (Failure ("internal error: block didn't become a block?"))
    }
  in (globals, List.map check_function functions)
```

8.9 codegen.ml (Code Generation)

```
(* Code generation: translate takes a semantically checked AST and
produces LLVM IR

LLVM tutorial: Make sure to read the OCaml version of the tutorial

http://llvm.org/docs/tutorial/index.html

Detailed documentation on the OCaml LLVM library:

http://llvm.moe/
http://llvm.moe/ocaml/

*)

module L = Llvm
module A = Ast
open Sast

module StringMap = Map.Make(String)

(* translate : Sast.program -> Llvm.module *)
let translate (globals, functions) =
  let context    = L.global_context () in

  (* Create the LLVM compilation module into which
     we will generate code *)
  let the_module = L.create_module context "PolyWiz" in

  (* Get types from the context *)
  let i32_t      = L.i32_type    context
  and i8_t       = L.i8_type     context
  and i1_t       = L.i1_type     context
  and float_t    = L.double_type context
  and void_t     = L.void_type  context in

  (* String type *)
  let string_t = L.pointer_type i8_t in

  (* Poly type *)
  let poly_t = L.pointer_type float_t in

  (* array types *)
  let float_arr_t = L.pointer_type float_t in
```

```
  let int_arr_t = L.pointer_type i32_t in
  let string_arr_t = L.pointer_type string_t in
  let bool_arr_t = L.pointer_type i1_t in
  let poly_arr_t = L.pointer_type poly_t in

(* Return the LLVM type for a PolyWiz type *)
  let rec ltype_of_typ = function
      A.Int  -> i32_t
    | A.Bool -> i1_t
    | A.Float -> float_t
    | A.Void -> void_t
    | A.String -> string_t
    | A.Poly -> poly_t
    | A.Array(t) -> L.pointer_type (ltype_of_typ t)
  in

  (* Array creation, initialization, indexing
     Note: Beets (2017) was very helpful here since their arrays are similar to ours
     So we'd like to cite their code for this part*)
  let ci = L.const_int i32_t in
  let new_arr t len builder =
    let s_tot = L.build_add (L.build_mul (L.build_intcast
     (L.size_of (ltype_of_typ t)) i32_t "tmp" builder)
     len "tmp" builder) (ci 1) "tmp" builder in
    let arr = L.build_pointercast (L.build_array_malloc (ltype_of_typ t) s_tot "tmp" builder)
     (L.pointer_type (ltype_of_typ t)) "tmp" builder in
    arr in

  let instantiate_arr t elems builder =
    let arr = new_arr t
     (ci (List.length elems)) builder in
    let _ =
      let assign_value i =
        let ind = L.build_add (ci i)
         (ci 0) "tmp" builder in
        L.build_store (List.nth elems i)
         (L.build_gep arr [| ind |] "tmp" builder) builder in
      let n = ((List.length elems)-1) in
        let rec rec_count cnt =
          if cnt = n then ignore (assign_value cnt)
          else (ignore (assign_value cnt); rec_count (cnt+1)) in
        rec_count 0 in

    arr in

  let list_length e =
    (match e with
      (_, SArrayLit(l)) -> List.length l
      | _ -> 0
    ) in

  (* Create a map of global variables after creating each *)
  let global_vars : L.llvalue StringMap.t =
    let global_var m (t, n) =
      let init = match t with
          A.Float -> L.const_float (ltype_of_typ t) 0.0
        | _ -> L.const_int (ltype_of_typ t) 0
      in StringMap.add n (L.define_global n init the_module) m in
    List.fold_left global_var StringMap.empty globals in

  let printf_t : L.lltype =
      L.var_arg_function_type i32_t [| L.pointer_type i8_t |] in
  let printf_func : L.llvalue =
      L.declare_function "printf" printf_t the_module in

```

```
(* Define each function (arguments and return type) so we can
   call it even before we've created its body *)
let function_decls : (L.llvalue * sfunc_decl) StringMap.t =
  let function_decl m fdecl =
    let name = fdecl.sfname
    and formal_types =
 Array.of_list (List.map (fun (t,_) -> ltype_of_typ t) fdecl.sformals)
    in let ftype = L.function_type (ltype_of_typ fdecl.styp) formal_types in
    StringMap.add name (L.define_function name ftype the_module, fdecl) m in
  List.fold_left function_decl StringMap.empty functions in

(* Fill in the body of the given function *)
let build_function_body fdecl =
  let (the_function, _) = StringMap.find fdecl.sfname function_decls in
  let builder = L.builder_at_end context (L.entry_block the_function) in

  let int_format_str = L.build_global_stringptr "%d\n" "fmt" builder
  and float_format_str = L.build_global_stringptr "%g\n" "fmt" builder
  and str_format_str = L.build_global_stringptr "%s\n" "fmt" builder in

  (* Construct the function's "locals": formal arguments and locally
     declared variables. Allocate each on the stack, initialize their
     value, if appropriate, and remember their values in the "locals" map *)
  let local_vars =
    let add_formal m (t, n) p =
      L.set_value_name n p;
 let local = L.build_alloca (ltype_of_typ t) n builder in
      ignore (L.build_store p local builder);
 StringMap.add n local m

    (* Allocate space for any locally declared variables and add the
     * resulting registers to our map *)
    and add_local m (t, n) =
 let local_var = L.build_alloca (ltype_of_typ t) n builder
 in StringMap.add n local_var m
    in

    let formals = List.fold_left2 add_formal StringMap.empty fdecl.sformals
        (Array.to_list (L.params the_function)) in
    List.fold_left add_local formals fdecl.slocals
  in

  (* Return the value for a variable or formal argument.
     Check local names first, then global names *)
  let lookup n = try StringMap.find n local_vars
                 with Not_found -> StringMap.find n global_vars
  in

  (* Construct code for an expression; return its value *)
  let rec expr builder ((ast_typ, e) : sexpr) = match e with
 SLiteral i -> L.const_int i32_t i
    | SBoolLit b -> L.const_int i1_t (if b then 1 else 0)
    | SSliteral l -> L.build_global_stringptr ( String.sub l 1 ((String.length l) - 2)) "str" builder
    | SFliteral l -> L.const_float_of_string float_t l
    | SArrayLit l ->
      let l' = (List.map (expr builder) l) in
      let arr_element_type = function
          A.Array(A.Int) -> A.Int
        | A.Array(A.Float) -> A.Float
        | A.Array(A.Bool) -> A.Bool
        | A.Array(A.String) -> A.String
        | A.Array(A.Poly) -> A.Poly
        | _ -> raise (Failure ("Invalid array type")) in
      let array_type = arr_element_type ast_typ in
      instantiate_arr array_type l' builder
```

```
      | SNoexpr    -> L.const_int i32_t 0
      | SId s      -> L.build_load (lookup s) s builder
      | SArrAssignInd (e1, ind, e) ->
        let e' = expr builder e in
        let ind_llvm = expr builder ind in
        let arr = expr builder e1 in
        ignore (
          (match e1 with
            (A.Array(A.Float), SId(arrName)) ->
              let set_arr_at_ind_f_external_func = L.declare_function "set_arr_at_ind_f" (L.function_type float_arr_t [|float_arr_t; float_t; i32_t|]) the_module in
              let new_arr = L.build_call set_arr_at_ind_f_external_func [|arr; e'; ind_llvm|] "set_arr_at_ind_f_llvm" builder in
              L.build_store new_arr (lookup arrName) builder
            |(A.Array(A.Int), SId(arrName)) ->
              let set_arr_at_ind_i_external_func = L.declare_function "set_arr_at_ind_i" (L.function_type int_arr_t [|int_arr_t; i32_t; i32_t|]) the_module in
              let new_arr = L.build_call set_arr_at_ind_i_external_func [|arr; e'; ind_llvm|] "set_arr_at_ind_i_llvm" builder in
              L.build_store new_arr (lookup arrName) builder
            |(A.Array(A.Bool), SId(arrName)) ->
              let set_arr_at_ind_b_external_func = L.declare_function "set_arr_at_ind_b" (L.function_type bool_arr_t [|bool_arr_t; i1_t; i32_t|]) the_module in
              let new_arr = L.build_call set_arr_at_ind_b_external_func [|arr; e'; ind_llvm|] "set_arr_at_ind_b_llvm" builder in
              L.build_store new_arr (lookup arrName) builder
            | _ -> raise (Failure ("This array type does not support index assignment."))
          )
        ); e'
      | SAssign (s, e) -> let e' = expr builder e in
                      ignore(L.build_store e' (lookup s) builder); e'
      | SBinop (((A.Poly,_ ) as e1), op, ((A.Float,_ ) as e2)) -> (* Binary op where e1 (poly), e2 (float) *)
        let e1' = expr builder e1
        and e2' = expr builder e2 in
        (match op with
            A.Div -> let poly_division_external_func = L.declare_function "poly_division" (L.function_type poly_t [|poly_t; float_t|]) the_module in
                    L.build_call poly_division_external_func [| e1'; e2' |] "poly_division_llvm" builder
          | A.Eval -> let eval_poly_external_func = L.declare_function "eval_poly" (L.function_type float_t [|poly_t; float_t|]) the_module in
                    L.build_call eval_poly_external_func [| e1'; e2' |] "eval_poly_llvm" builder
          | _ -> raise (Failure "This operation is invalid for a poly and float operand.")
        )

      | SBinop (((A.Poly,_ ) as e1), op, ((A.Poly,_ ) as e2)) -> (* Binary op where e1 (poly), e2 (poly) *)
     let e1' = expr builder e1
     and e2' = expr builder e2 in
     (match op with
       A.Add    -> let poly_addition_external_func = L.declare_function "poly_addition" (L.function_type poly_t [|poly_t; poly_t|]) the_module in
                    L.build_call poly_addition_external_func [| e1'; e2' |] "poly_addition_llvm" builder
     | A.Sub    -> let poly_subtraction_external_func = L.declare_function "poly_subtraction" (L.function_type poly_t [|poly_t; poly_t|]) the_module in
                    L.build_call poly_subtraction_external_func [| e1'; e2' |] "poly_subtraction_llvm" builder
     | A.Mult   -> let poly_multiplication_external_func = L.declare_function "poly_multiplication" (L.function_type poly_t [|poly_t; poly_t|]) the_module in
                    L.build_call poly_multiplication_external_func [| e1'; e2' |] "poly_multiplication_llvm" builder
     | A.Compo   -> let poly_composition_external_func = L.declare_function "poly_composition" (L.function_type poly_t [|poly_t; poly_t|]) the_module in
                    L.build_call poly_composition_external_func [| e1'; e2' |] "poly_composition_llvm" builder
    | A.Exp    -> raise (Failure "internal error: semant should have rejected ^ on poly")
     | A.Equal  -> let equal_compare_poly_external_func = L.declare_function "equal_compare_poly" (L.function_type i1_t [|poly_t; poly_t|]) the_module in
                    L.build_call equal_compare_poly_external_func [| e1'; e2' |] "equal_compare_poly_llvm" builder
```

```
    | A.Neq  -> let nequal_compare_poly_external_func = L.declare_function "nequal_compare_poly" (L.function_type i1_t [|poly_t; poly_t|]) the_module in
                L.build_call nequal_compare_poly_external_func [| e1'; e2' |] "nequal_compare_poly_llvm" builder
    | A.Less   -> raise (Failure "internal error: semant should have rejected > on poly")
    | A.Leq    -> raise (Failure "internal error: semant should have rejected <= on poly")
    | A.Greater -> raise (Failure "internal error: semant should have rejected > on poly")
    | A.Geq    -> raise (Failure "internal error: semant should have rejected >= on poly")
    | A.And | A.Or -> raise (Failure "internal error: semant should have rejected and/or on poly")
   | _ -> raise (Failure "This operation is invalid for two poly operands."))

   | SBinop (((A.Array(arr_typ),_ ) as e1), op, ((A.Int,_ ) as e2)) -> (* Binary op where e1 (array), e2 (int) *)
    let e1' = expr builder e1
    and e2' = expr builder e2 in
    (match op with
      A.Ele_at_ind   -> (
      match arr_typ with
          A.Int ->
            let arr_at_ind_i_external_func = L.declare_function "arr_at_ind_i" (L.function_type i32_t [|int_arr_t;i32_t|]) the_module in
            L.build_call arr_at_ind_i_external_func [| e1'; e2' |] "arr_at_ind_i_llvm" builder
        | A.Float ->
            let arr_at_ind_f_external_func = L.declare_function "arr_at_ind_f" (L.function_type float_t [|float_arr_t;i32_t|]) the_module in
            L.build_call arr_at_ind_f_external_func [| e1'; e2' |] "arr_at_ind_f_llvm" builder
        | A.Bool ->
            let arr_at_ind_b_external_func = L.declare_function "arr_at_ind_b" (L.function_type i1_t [|bool_arr_t;i32_t|]) the_module in
            L.build_call arr_at_ind_b_external_func [| e1'; e2' |] "arr_at_ind_b_llvm" builder
        | _ -> raise (Failure ("This array type does not support indexing."))
     )
     | _ -> raise (Failure ("This operation is invalid for array and int operands."))
   )

     | SBinop (((A.Float,_ ) as e1), op, ((A.Float,_ ) as e2)) -> (* Binary op where e1 (float), e2 (float) *)
    let e1' = expr builder e1
    and e2' = expr builder e2 in
    (match op with
      A.Add    -> L.build_fadd e1' e2' "tmp" builder
    | A.Sub    -> L.build_fsub e1' e2' "tmp" builder
    | A.Mult   -> L.build_fmul e1' e2' "tmp" builder
    | A.Div    -> L.build_fdiv e1' e2' "tmp" builder
   | A.Exp    ->
     let pow_external_func_ff = L.declare_function "pow_operator_ff" (L.function_type float_t [|float_t;float_t|]) the_module in
      L.build_call pow_external_func_ff [| e1'; e2' |] "pow_operator_ff_llvm" builder
    | A.Equal  -> L.build_fcmp L.Fcmp.Oeq e1' e2' "tmp" builder
    | A.Neq    -> L.build_fcmp L.Fcmp.One e1' e2' "tmp" builder
    | A.Less   -> L.build_fcmp L.Fcmp.Olt e1' e2' "tmp" builder
    | A.Leq    -> L.build_fcmp L.Fcmp.Ole e1' e2' "tmp" builder
    | A.Greater -> L.build_fcmp L.Fcmp.Ogt e1' e2' "tmp" builder
    | A.Geq    -> L.build_fcmp L.Fcmp.Oge e1' e2' "tmp" builder
    | A.And | A.Or -> raise (Failure "internal error: semant should have rejected and/or on float")
   | _ -> raise (Failure "This operation is invalid for two float operands."))

      | SBinop (((A.Float,_ ) as e1), op, ((A.Int,_ ) as e2)) -> (* Binary op where e1 (float), e2 (int) *)
    let e1' = expr builder e1
    and e2' = expr builder e2 in
    (match op with
      A.Add    -> L.build_fadd e1' e2' "tmp" builder
    | A.Sub    -> L.build_fsub e1' e2' "tmp" builder
    | A.Mult   -> L.build_fmul e1' e2' "tmp" builder
    | A.Div    -> L.build_fdiv e1' e2' "tmp" builder
   | A.Exp    ->
     let pow_external_func_fi = L.declare_function "pow_operator_fi" (L.function_type float_t [|float_t;i32_t|]) the_module in
```

```
       L.build_call pow_external_func_fi [| e1'; e2' |] "pow_operator_fi_llvm" builder
    | A.Equal  -> L.build_fcmp L.Fcmp.Oeq e1' e2' "tmp" builder
    | A.Neq    -> L.build_fcmp L.Fcmp.One e1' e2' "tmp" builder
    | A.Less   -> L.build_fcmp L.Fcmp.Olt e1' e2' "tmp" builder
    | A.Leq    -> L.build_fcmp L.Fcmp.Ole e1' e2' "tmp" builder
    | A.Greater -> L.build_fcmp L.Fcmp.Ogt e1' e2' "tmp" builder
    | A.Geq    -> L.build_fcmp L.Fcmp.Oge e1' e2' "tmp" builder
    | A.And | A.Or ->
        raise (Failure "internal error: semant should have rejected and/or on float")
   | _ -> raise (Failure "This operation is invalid for these operands."))

       | SBinop (((A.Int,_ ) as e1), op, ((A.Float,_ ) as e2)) -> (* Binary op where e1 (int), e2 (float) *)
    let e1' = expr builder e1
    and e2' = expr builder e2 in
    (match op with
      A.Add    -> L.build_fadd e1' e2' "tmp" builder
    | A.Sub    -> L.build_fsub e1' e2' "tmp" builder
    | A.Mult   -> L.build_fmul e1' e2' "tmp" builder
    | A.Div    -> L.build_fdiv e1' e2' "tmp" builder
   | A.Exp    ->
     let pow_external_func_if = L.declare_function "pow_operator_if" (L.function_type float_t [|i32_t;float_t|]) the_module in
       L.build_call pow_external_func_if [| e1'; e2' |] "pow_operator_if_llvm" builder
    | A.Equal  -> L.build_fcmp L.Fcmp.Oeq e1' e2' "tmp" builder
    | A.Neq    -> L.build_fcmp L.Fcmp.One e1' e2' "tmp" builder
    | A.Less   -> L.build_fcmp L.Fcmp.Olt e1' e2' "tmp" builder
    | A.Leq    -> L.build_fcmp L.Fcmp.Ole e1' e2' "tmp" builder
    | A.Greater -> L.build_fcmp L.Fcmp.Ogt e1' e2' "tmp" builder
    | A.Geq    -> L.build_fcmp L.Fcmp.Oge e1' e2' "tmp" builder
    | A.And | A.Or ->
        raise (Failure "internal error: semant should have rejected and/or on float")
   | _ -> raise (Failure "This operation is invalid for these operands."))

       | SBinop (((t,_ ) as e1), A.In, ((A.Array(_),_ ) as e2)) -> (* Binary op where op is "in" *)
   let e1' = expr builder e1 in
   let e2' = expr builder e2 in
   (match t with
     A.Int    -> let int_arr_contains_func = L.declare_function "int_arr_contains" (L.function_type i1_t [|i32_t; int_arr_t |]) the_module in
       L.build_call int_arr_contains_func [| e1'; e2'|] "int_arr_contains_llvm" builder
   | A.Float  -> let float_arr_contains_func = L.declare_function "float_arr_contains" (L.function_type i1_t [|float_t; float_arr_t |]) the_module in
     L.build_call float_arr_contains_func [| e1'; e2' |] "float_arr_contains_llvm" builder
   | A.String -> let string_arr_contains_func = L.declare_function "string_arr_contains" (L.function_type i1_t [|string_t; string_arr_t |]) the_module in
     L.build_call string_arr_contains_func [| e1'; e2' |] "string_arr_contains_llvm" builder
   | A.Poly   -> let poly_arr_contains_func = L.declare_function "poly_arr_contains" (L.function_type i1_t [|poly_t; poly_arr_t |]) the_module in
     L.build_call poly_arr_contains_func [| e1'; e2' |] "poly_arr_contains_llvm" builder
   | _ -> raise (Failure "This operation is invalid for these operands."))

     | SBinop (e1, op, e2) -> (* Binary op where e1, e2 are both ints*)
    let e1' = expr builder e1
    and e2' = expr builder e2 in
    (match op with
      A.Add    -> L.build_add e1' e2' "tmp" builder
    | A.Sub    -> L.build_sub e1' e2' "tmp" builder
    | A.Mult   -> L.build_mul e1' e2' "tmp" builder
   | A.Div    -> L.build_sdiv e1' e2' "tmp" builder
   | A.Exp    ->
     let pow_external_func_ii = L.declare_function "pow_operator_ii" (L.function_type i32_t [|i32_t;i32_t|]) the_module in
       L.build_call pow_external_func_ii [| e1'; e2' |] "pow_operator_ii_llvm" builder
    | A.And    -> L.build_and e1' e2' "tmp" builder
    | A.Or     -> L.build_or e1' e2' "tmp" builder
```

```
      | A.Equal  -> L.build_icmp L.Icmp.Eq e1' e2' "tmp" builder
      | A.Neq    -> L.build_icmp L.Icmp.Ne e1' e2' "tmp" builder
      | A.Less   -> L.build_icmp L.Icmp.Slt e1' e2' "tmp" builder
      | A.Leq    -> L.build_icmp L.Icmp.Sle e1' e2' "tmp" builder
      | A.Greater -> L.build_icmp L.Icmp.Sgt e1' e2' "tmp" builder
      | A.Geq    -> L.build_icmp L.Icmp.Sge e1' e2' "tmp" builder
     | _ -> raise (Failure "This operation is invalid for these operands.")
      )
       | SUnop(op, ((t, _) as e)) ->
           let e' = expr builder e in
      (match op with
        A.Neg when t = A.Float -> L.build_fneg e' "tmp" builder
     | A.Neg                -> L.build_neg e' "tmp" builder
     | A.Not                -> L.build_not e' "tmp" builder
     | A.Const_ret ->
       let constants_retriever_external_func = L.declare_function "constants_retriever" (L.function_type float_arr_t [|poly_t|]) the_module in
       L.build_call constants_retriever_external_func [| e' |] "constants_retriever_llvm" builder
     | A.Abs when t = A.Float ->
       let abs_external_func_floats = L.declare_function "abs_operator_float" (L.function_type float_t [|float_t|]) the_module in
       L.build_call abs_external_func_floats [| e' |] "abs_operator_float_llvm" builder
     | A.Abs                -> 
       let abs_external_func_ints = L.declare_function "abs_operator_int" (L.function_type i32_t [|i32_t|]) the_module in
       L.build_call abs_external_func_ints [| e' |] "abs_operator_int_llvm" builder
     )
       | SCall ("printint", [e]) | SCall ("printb", [e]) ->
      L.build_call printf_func [| int_format_str ; (expr builder e) |]
        "printf" builder
       | SCall ("printf", [e]) ->
      L.build_call printf_func [| float_format_str ; (expr builder e) |]
        "printf" builder
       | SCall ("printstr", [e]) ->
      L.build_call printf_func [| str_format_str ; (expr builder e) |] "printf" builder
       | SCall ("new_poly", [e1;e2;e3]) ->
         let e1' = expr builder e1 in
         let e2' = expr builder e2 in
         let e3' = expr builder e3 in
         let new_poly_external_func = L.declare_function "new_poly" (L.function_type poly_t [|float_arr_t; int_arr_t; i32_t|]) the_module in
         L.build_call new_poly_external_func [| e1'; e2'; e3'|] "new_poly_llvm" builder
       | SCall ("poly_at_ind", [e1;e2]) ->
         let poly_at_ind_external_func = L.declare_function "poly_at_ind" (L.function_type float_t [|poly_t; i32_t|]) the_module in
         L.build_call poly_at_ind_external_func [| expr builder e1; expr builder e2 |] "poly_at_ind_llvm" builder
       | SCall ("order", [e]) ->
         let order_external_func = L.declare_function "order" (L.function_type i32_t [|poly_t|]) the_module in
         L.build_call order_external_func [| expr builder e |] "order_llvm" builder
       | SCall ("initialize_floats", [e]) ->
         let initialize_floats_external_func = L.declare_function "initialize_floats" (L.function_type float_arr_t [|i32_t|]) the_module in
         L.build_call initialize_floats_external_func [| expr builder e |] "initialize_floats_llvm" builder
       | SCall ("initialize_ints", [e]) ->
         let initialize_ints_external_func = L.declare_function "initialize_ints" (L.function_type int_arr_t [|i32_t|]) the_module in
         L.build_call initialize_ints_external_func [| expr builder e |] "initialize_ints_llvm" builder
       | SCall ("int_to_float", [e]) ->
         let int_to_float_external_func = L.declare_function "int_to_float" (L.function_type float_t [|i32_t|]) the_module in
         L.build_call int_to_float_external_func [| expr builder e |] "int_to_float_llvm" builder
       | SCall ("to_str", [e]) ->
         let poly_to_str_external_func = L.declare_function "poly_to_str" (L.function_type string_t [|poly_t|]) the_module in
         L.build_call poly_to_str_external_func [| expr builder e |] "poly_to_str_llvm" builder
```

```
      | SCall ("tex_document", [e1;e2]) ->
        let print_texdoc_external_func = L.declare_function "generate_texdoc" (L.function_type string_t [|string_arr_t; int_arr_t|]) the_module in
        L.build_call print_texdoc_external_func [| expr builder e1; expr builder e2 |] "print_texdoc_llvm" builder
        | SCall ("print_tex", [e]) ->
        let poly_to_tex_external_func = L.declare_function "poly_to_tex" (L.function_type string_t [|poly_t|]) the_module in
        L.build_call poly_to_tex_external_func [| expr builder e |] "poly_to_tex_llvm" builder
      | SCall ("plot", [e1;e2]) ->
        let e1' = expr builder e1 in
        let e2' = expr builder e2 in
        let len_e1 = L.const_int i32_t (list_length e1) in
        let plot_external_func = L.declare_function "plot" (L.function_type i32_t [|poly_arr_t; i32_t; string_t|]) the_module in
        L.build_call plot_external_func [| e1'; len_e1; e2' |] "plot_llvm" builder
      | SCall ("range_plot", [e1;e2;e3;e4]) ->
        let e1' = expr builder e1 in
        let e2' = expr builder e2 in
        let e3' = expr builder e3 in
        let e4' = expr builder e4 in
        let len_e1 = L.const_int i32_t (list_length e1) in
        let range_plot_external_func = L.declare_function "range_plot" (L.function_type i32_t [|poly_arr_t; i32_t; float_t; float_t; string_t|]) the_module in
        L.build_call range_plot_external_func [| e1'; len_e1; e2'; e3'; e4' |] "range_plot_llvm" builder
      | SCall (f, args) ->
         let (fdef, fdecl) = StringMap.find f function_decls in
    let llargs = List.rev (List.map (expr builder) (List.rev args)) in
    let result = (match fdecl.styp with
                    A.Void -> ""
                  | _ -> f ^ "_result") in
         L.build_call fdef (Array.of_list llargs) result builder
    in
    let add_terminal builder instr =
      match L.block_terminator (L.insertion_block builder) with
   Some _ -> ()
      | None -> ignore (instr builder) in

    (* Build the code for the given statement; return the builder for
       the statement's successor (i.e., the next instruction will be built
       after the one generated by this call) *)

    let rec stmt builder = function
   SBlock sl -> List.fold_left stmt builder sl
      | SExpr e -> ignore(expr builder e); builder
      | SReturn e -> ignore(match fdecl.styp with
                              (* Special "return nothing" instr *)
                              A.Void -> L.build_ret_void builder
                              (* Build return statement *)
                            | _ -> L.build_ret (expr builder e) builder );
                 builder
      | SIf (predicate, then_stmt, else_stmt) ->
         let bool_val = expr builder predicate in
    let merge_bb = L.append_block context "merge" the_function in
         let build_br_merge = L.build_br merge_bb in (* partial function *)

    let then_bb = L.append_block context "then" the_function in
    add_terminal (stmt (L.builder_at_end context then_bb) then_stmt)
      build_br_merge;

    let else_bb = L.append_block context "else" the_function in
    add_terminal (stmt (L.builder_at_end context else_bb) else_stmt)
      build_br_merge;

    ignore(L.build_cond_br bool_val then_bb else_bb builder);
    L.builder_at_end context merge_bb
```

```

     | SWhile (predicate, body) ->
    let pred_bb = L.append_block context "while" the_function in
    ignore(L.build_br pred_bb builder);

    let body_bb = L.append_block context "while_body" the_function in
    add_terminal (stmt (L.builder_at_end context body_bb) body)
      (L.build_br pred_bb);

    let pred_builder = L.builder_at_end context pred_bb in
    let bool_val = expr pred_builder predicate in

    let merge_bb = L.append_block context "merge" the_function in
    ignore(L.build_cond_br bool_val body_bb merge_bb pred_builder);
    L.builder_at_end context merge_bb

     (* Implement for loops as while loops *)
     | SFor (e1, e2, e3, body) -> stmt builder
      ( SBlock [SExpr e1 ; SWhile (e2, SBlock [body ; SExpr e3]) ] )
   in

   (* Build the code for each statement in the function *)
   let builder = stmt builder (SBlock fdecl.sbody) in

   (* Add a return if the last block falls off the end *)
   add_terminal builder (match fdecl.styp with
       A.Void -> L.build_ret_void
     | A.Float -> L.build_ret (L.const_float float_t 0.0)
     | t -> L.build_ret (L.const_int (ltype_of_typ t) 0))
 in

 List.iter build_function_body functions;
 the_module
```

8.10 polywiz.ml (top-level file)

```
(* Top-level of the PolyWiz compiler: scan & parse the input,
   check the resulting AST and generate an SAST from it, generate LLVM IR,
   and dump the module *)

type action = Ast | Sast | LLVM_IR | Compile

let () =
  let action = ref Compile in
  let set_action a () = action := a in
  let speclist = [
    ("-a", Arg.Unit (set_action Ast), "Print the AST");
    ("-s", Arg.Unit (set_action Sast), "Print the SAST");
    ("-l", Arg.Unit (set_action LLVM_IR), "Print the generated LLVM IR");
    ("-c", Arg.Unit (set_action Compile),
      "Check and print the generated LLVM IR (default)");
  ] in
  let usage_msg = "usage: ./polywiz.native [-a|-s|-l|-c] [file.mc]" in
  let channel = ref stdin in
  Arg.parse speclist (fun filename -> channel := open_in filename) usage_msg;

  let lexbuf = Lexing.from_channel !channel in
  let ast = Polywizparse.program Scanner.token lexbuf in
  match !action with
    Ast -> print_string (Ast.string_of_program ast)
  | _ -> let sast = Semant.check ast in
    match !action with
      Ast    -> ()
```

```
    | Sast   -> print_string (Sast.string_of_sprogram sast)
    | LLVM_IR -> print_string (Llvm.string_of_llmodule (Codegen.translate sast))
    | Compile -> let m = Codegen.translate sast in
   Llvm_analysis.assert_valid_module m;
   print_string (Llvm.string_of_llmodule m)
```

8.11 library_functions.c (functions in C for linking)

```c
/*
 *  A function illustrating how to link C code to code generated from LLVM
 */

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <errno.h>
#include <string.h>
#include <float.h>
#include <stdbool.h>
#include <unistd.h>
#include <limits.h>

double pow_operator_ff(double a, double b){
  return pow(a, b);
}

double pow_operator_fi(double a, int b){
  return pow(a, (double) b);
}

double pow_operator_if(int a, double b){
  return pow((double) a,b);
}

int pow_operator_ii(int a, int b){
  return (int) pow((double) a, (double) b);
}

double abs_operator_float(double a){
  return fabs(a);
}

int abs_operator_int(int a){
  return abs(a);
}

int order(double *poly){
  int poly_order = -1;
  int i=0;
  //while poly[i] is not the poly sentinal value, DBL_MIN
  while(poly[i] != DBL_MIN){
    //set order to largest exponent where its constant != 0
    if(poly[i]!=0.0){
      poly_order = i;
    }
    i++;
  }
  return poly_order;
}

double* new_poly(double *consts, int *exponents, int arr_lengths){

  //find the order of the polynomial
```

```c
  int order = -1;
  for(int i=0; i < arr_lengths; i++){
    order = exponents[i]>order ? exponents[i]: order;
  }
  if(order<0) return NULL;

  //initialize the poly array with zeros
  double *poly_arr = malloc( (order+2) * sizeof (double));

  for(int i=0; i <= order; i++)
      poly_arr[i] = 0.0;
  //terminate the poly arr with DBL_MIN
  poly_arr[order+1] = DBL_MIN;

  //fill poly array with inputted constants and exponents
  for(int i=0; i < arr_lengths; i++){
    int exponent = exponents[i];
    double constant = consts[i];
    poly_arr[exponent] = constant;
  }
  return poly_arr;

}

double* poly_addition(double *poly1, double *poly2){
  int poly1_order = order(poly1);
  int poly2_order = order(poly2);

  // poly_sum array will be the size of the largest input array
  int poly_sum_order = poly1_order>poly2_order ? poly1_order : poly2_order;
  double *poly_sum = malloc( (poly_sum_order+2) * sizeof (double));

  for(int i=0; i<=poly_sum_order; i++){
    double poly1_const = i<=poly1_order ? poly1[i]: 0.0;
    double poly2_const = i<=poly2_order ? poly2[i]: 0.0;
    poly_sum[i] = poly1_const + poly2_const;
  }
  poly_sum[poly_sum_order+1] = DBL_MIN;

  return poly_sum;

}

double* poly_subtraction(double *poly1, double *poly2){
  int poly1_order = order(poly1);
  int poly2_order = order(poly2);

  // poly_diff array will be the size of the largest input array
  int poly_diff_order = poly1_order>poly2_order ? poly1_order : poly2_order;
  double *poly_diff = malloc( (poly_diff_order+2) * sizeof (double));

  for(int i=0; i<=poly_diff_order; i++){
    double poly1_const = i<=poly1_order ? poly1[i]: 0.0;
    double poly2_const = i<=poly2_order ? poly2[i]: 0.0;
    poly_diff[i] = poly1_const - poly2_const;
  }
  poly_diff[poly_diff_order+1] = DBL_MIN;

  return poly_diff;

}


double* poly_multiplication(double *poly1, double *poly2){
  int poly1_order = order(poly1);
```

```
  int poly2_order = order(poly2);

  // below code is adapted from https://www.geeksforgeeks.org/multiply-two-polynomials-2/
  // poly_product array will be the size of the sum of the largest exponent on poly1 and poly2
  int poly_product_order = poly1_order+poly2_order;
  double *poly_product = malloc((poly_product_order +2) * (sizeof (double)));

  // Initialize the product polynomial with 0s as constants
  for (int i = 0; i<=poly_product_order; i++)
    poly_product[i] = 0;
  poly_product[poly_product_order+1] = DBL_MIN;

  // Loop through each term of first polynomial
  for (int i=0; i<=poly1_order; i++)
  {
    // Multiply the current term of first polynomial
    // with every term of second polynomial.
    for (int j=0; j<=poly2_order; j++)
        poly_product[i+j] += poly1[i]*poly2[j];
  }

  return poly_product;

}

double* constants_retriever(double *poly){
  int poly_order = order(poly);
  double *poly_consts = malloc((poly_order+1) * sizeof (double));

  // fill in the poly consts array
  for (int i = 0; i<=poly_order; i++)
    poly_consts[i] = poly[i];

  return poly_consts;

}

double eval_poly(double *poly, double x){
  int poly_order = order(poly);

  // evaluate poly at specified value x
  double poly_at_x = 0.0;
  for (int i = 0; i<=poly_order; i++)
    poly_at_x += poly[i] * pow(x, i);

  return poly_at_x;
}

bool equal_compare_poly(double *poly1, double *poly2){
  int poly1_order = order(poly1);
  int poly2_order = order(poly2);

  //if not the same order, not equal
  if(poly1_order != poly2_order)
    return false;

  // check if all poly constants are equal
  bool equal = true;
  for (int i = 0; i<= poly1_order; i++){
    if(poly1[i]!=poly2[i])
      equal = false;
  }

  return equal;

```

```
}

bool nequal_compare_poly(double *poly1, double *poly2){
  int poly1_order = order(poly1);
  int poly2_order = order(poly2);

  //if not the same order, not equal
  if(poly1_order != poly2_order)
    return false;

  // check if all poly constants are equal
  bool equal = true;
  for (int i = 0; i<= poly1_order; i++){
    if(poly1[i]!=poly2[i])
      equal = false;
  }

  return !equal;

}


//poly divison by float
double* poly_division(double *poly1, double denominator){

  // poly_divisor will be order 0 and represents the division by the denominator
  double *poly_divisor = malloc((2) * sizeof (double));
  poly_divisor[0] = 1.0 / denominator;
  poly_divisor[1] = DBL_MIN;

  return poly_multiplication(poly1, poly_divisor);

}



double* poly_composition(double *poly1, double *poly2){
  int poly1_order = order(poly1);
  int poly2_order = order(poly2);

  int composed_poly_order = poly1_order*poly2_order;
  double *composed_poly = malloc((composed_poly_order+2) * sizeof (double));
  // Initialize the composed polynomial with 0s as constants
  for (int i = 0; i<=composed_poly_order; i++)
    composed_poly[i] = 0;
  composed_poly[composed_poly_order+1] = DBL_MIN;

  // Loop through each term of first polynomial
  for (int i=0; i<=poly1_order; i++)
  {
    //compose this term
    int current_term_order = i*poly2_order;
    double *current_term = malloc((current_term_order+2) * sizeof (double));

    // Initialize the current poly with poly2
    for (int i = 0; i<=current_term_order; i++){
      current_term[i] = i<=poly2_order ? poly2[i]: 0.0;
    }
    current_term[current_term_order+1] = DBL_MIN;

    //foil this term
    for(int j=i; j>1; j--){
      current_term = poly_multiplication(current_term, poly2);
    }

```

```
    //handle order 0 term special case
    if(i==0)
      current_term[0] = 1.0;

    //multiply by poly1 constant for this term
    double *multiplier = malloc((2) * sizeof (double));
    multiplier[0] = poly1[i];
    multiplier[1] = DBL_MIN;
    current_term = poly_multiplication(current_term, multiplier);

    //add current term to the composed_poly
    composed_poly = poly_addition(composed_poly, current_term);
  }

  return composed_poly;

}


char* poly_to_str(double *poly){
  int poly_order = order(poly);
  const int max_digits = 350;

  //empty poly
  if(poly_order<0){
    char *poly_str = malloc( sizeof (char));
    poly_str[0] = '\0';
    return poly_str;
  }

  //this allocates the max amount of space that could possibly be needed (should probably be optimized)
  char *poly_string = malloc(poly_order* (3+(2*max_digits))* sizeof (char));
  char *poly_str_ind = poly_string;
  for (int i = poly_order; i>=0; i--){
    if(poly[i]==0.0)
      continue;

    //order 0 poly
    if(i==0){
      poly_str_ind += sprintf(poly_str_ind, poly[i]>0.0 ? "+%f" : "%f", poly[i]);
    }
    //order 1 polynomial
    else if(i==1){
      poly_str_ind += sprintf(poly_str_ind, poly[i]>0.0 ? "+%fx" : "%fx", poly[i]);
    }
    //higher order polynomials
    else{
      poly_str_ind += sprintf(poly_str_ind, i==poly_order ? "%fx^%i" : poly[i]>0.0 ? "+%fx^%i" : "%fx^%i",
          poly[i], i);
    }
  }

  //printf(poly_string);

  return poly_string;

}

char* poly_to_tex(double *poly){
  int poly_order = order(poly);
  const int max_digits = 350;

  //empty poly
  if(poly_order<0){
    char *poly_str = malloc( sizeof (char));
```

```
    poly_str[0] = '\0';
    return poly_str;
  }

  //this allocates the max amount of space that could possibly be needed (should probably be optimized)
  char *poly_string = malloc(poly_order* (7+(4*max_digits))* sizeof (char));
  char *poly_str_ind = poly_string;

  poly_str_ind += sprintf(poly_str_ind, "$$");

  for (int i = poly_order; i>=0; i--){
    if(poly[i] == 0.0 || !poly[i])
      continue;

    //order 0 poly
    if(i==0){
      poly_str_ind += sprintf(poly_str_ind, poly[i]>0.0 ? "+%f" : "%f", poly[i]);
    }
    //order 1 polynomial
    else if(i==1){
      poly_str_ind += sprintf(poly_str_ind, poly[i]>0.0 ? "+%fx" : "%fx", poly[i]);
    }
    //higher order polynomials
    else{
      poly_str_ind += sprintf(poly_str_ind, i==poly_order ? "%fx^{%i}" : poly[i]>0.0 ? "+%fx^{%i}" : "%fx^{%i}",
          poly[i], i);
    }
  }

  poly_str_ind += sprintf(poly_str_ind, "$$");

  return poly_string;

}

char* generate_texdoc(char **texdocbody, int *imgindices){

  //header and footer of the body
  char header[] = "\\documentclass{article}\n\\usepackage{graphicx}\n\\begin{document}";
  char footer[] = "\n\\end{document}";

  //header and footer to wrap filepath of image
  char imgheader[] = "\\begin{figure}[h]\n\\centering\n\\includegraphics[width=2.5in]{";
  char imgfooter[] = "}\n\\label{fig_sim}\n\\end{figure}";

  //find length of everything
  int len = strlen(header) + strlen(footer) + ((sizeof(imgindices)/sizeof(int)) * (strlen(imgheader) +
      strlen(imgfooter))) + sizeof(texdocbody);
  int num_elems = sizeof(texdocbody)/sizeof(*texdocbody);

  int j = 0;
  while(texdocbody[j]){
    len = len + strlen(texdocbody[j]) + 2;
    j = j + 1;
  }

  //now, actually make the string
  char *texdoc_str = malloc(len + 100);
  char *texdoc_str_ind = texdoc_str;

  //print header
  texdoc_str_ind += sprintf(texdoc_str_ind, "%s", header);

  int i = 0;
  while(texdocbody[i]){
```

```c
    int isimg = 0;

    //check if it is an image
    for(int j = 0; j < (sizeof(imgindices)/sizeof(int)); j++){
        if((imgindices[j] == i && imgindices[j] != 0) || i == 0 && imgindices[0] == 0){
            isimg = 1;
            break;
        }
    }
    //handle non-image case
    if(texdocbody[i] && isimg == 0){
        char* s1 = texdocbody[i];
        texdoc_str_ind += sprintf(texdoc_str_ind, "\n");
        while(*s1){
          texdoc_str_ind += sprintf(texdoc_str_ind, "%c", *s1);
          s1 = s1 + 1;
        }
        texdoc_str_ind += sprintf(texdoc_str_ind, "\\\\");
    }

    //handle image case
    else if(isimg == 1 && imgindices[0] != -1){
        texdoc_str_ind += sprintf(texdoc_str_ind, "\n%s%s%s", imgheader, texdocbody[i], imgfooter);
    }
    isimg = 0;
    i = i + 1;
  }

  //print footer
  texdoc_str_ind += sprintf(texdoc_str_ind, "%s", footer);

  return texdoc_str;
}

//get poly const at ind
double poly_at_ind(double *poly, int ind){
  return poly[ind];
}

int syscall_gnuplot(char *scriptpath) {
  char buf[100];
  sprintf(buf, "gnuplot %s", scriptpath);
  return system(buf);
}

int plot(double **polynomials, int num_polynomials, char *filepath) {

  FILE *fp = fopen("polypoints.txt","w");
  double range_bottom = -5.0;
  double range_top = 5.0;
  for (double x_val = range_bottom; x_val < range_top; x_val += 0.2) {
    fprintf(fp, "%lf", x_val);
    double **polypointer = polynomials;
    for (int i = 0; i < num_polynomials; i++) {
      double *temp_poly = *polypointer;

      double y_val = eval_poly(temp_poly, x_val);
      fprintf(fp, "\t %lf", y_val);

      polypointer++;
    }
    fprintf(fp, "\n");
  }
  fclose(fp);

```

```
  char *plot_script = "gnu_multiplot_script";
  FILE *sp = fopen(plot_script,"w");
  fprintf(sp, "set term pngcairo; set output '%s';\nplot ", filepath);
  for (int i = 0; i < num_polynomials; i++) {
    fprintf(sp, "'polypoints.txt' using 1:%d w l title 'poly %d', \\\n", i+2, i+1);
  }
  fclose(sp);

  int return_code = syscall_gnuplot(plot_script);

  system("rm gnu_multiplot_script");
  system("rm polypoints.txt");

  return return_code;

}

int range_plot(double **polynomials, int num_polynomials, double range_bottom, double range_top, char *filepath) {

  FILE *fp = fopen("polypoints.txt","w");
  double num_points = 100.0;
  double counter = (range_top - range_bottom) / num_points;
  for (double x_val = range_bottom; x_val < range_top; x_val += counter ) {
    fprintf(fp, "%lf", x_val);
    double **polypointer = polynomials;
    for (int i = 0; i < num_polynomials; i++) {
      double *temp_poly = *polypointer;

      double y_val = eval_poly(temp_poly, x_val);
      fprintf(fp, "\t %lf", y_val);

      polypointer++;
    }
    fprintf(fp, "\n");
  }
  fclose(fp);

  char *plot_script = "gnu_multiplot_script";
  FILE *sp = fopen(plot_script,"w");
  fprintf(sp, "set term pngcairo; set output '%s';\nplot ", filepath);
  for (int i = 0; i < num_polynomials; i++) {
    fprintf(sp, "'polypoints.txt' using 1:%d w l title 'poly %d', \\\n", i+2, i+1);
  }
  fclose(sp);

  int return_code = syscall_gnuplot(plot_script);

  system("rm gnu_multiplot_script");
  system("rm polypoints.txt");

  return return_code;
}

//checks if int is inside int array
bool int_arr_contains(int x, int *arr) {
  int i = -1;
  while (arr[++i] != INT_MIN) {
    if(arr[i] == x) {
      return true;
    }
  }
  return false;
}

//checks if float is inside float array
```

```c
bool float_arr_contains(double x, double *arr) {
  int i = -1;
  while (arr[++i] != DBL_MIN) {
    if(arr[i] == x) {
      return true;
    }
  }
  return false;
}

bool string_arr_contains(char *x, char **arr) {
  int i = -1;
  while (arr[++i]) {
    if(arr[i] == x) {
      return true;
    }
  }
  return false;
}

//checks if poly is inside poly array
bool poly_arr_contains(double *poly, double **poly_arr) {
  int i = -1;
  while (poly_arr[++i]) {
    if(equal_compare_poly(poly, poly_arr[i])) {
      return true;
    }
  }
  return false;
}

//get int array element at ind
int arr_at_ind_i(int *arr, int ind) {
  return arr[ind];
}
//get double array element at ind
double arr_at_ind_f(double *arr, int ind) {
  return arr[ind];
}
//get bool array element at ind
bool arr_at_ind_b(bool *arr, int ind) {
  return arr[ind];
}

//set array value at ind for float arrays
double* set_arr_at_ind_f(double *arr, double el, int ind) {
  arr[ind] = el;
  return arr;
}
//set array value at ind for int arrays
int* set_arr_at_ind_i(int *arr, int el, int ind) {
  arr[ind] = el;
  return arr;
}
//set array value at ind for bool arrays
bool* set_arr_at_ind_b(bool *arr, bool el, int ind) {
  arr[ind] = el;
  return arr;
}

//instantiate float array with zeros
double* initialize_floats(int length){
  double *arr = malloc(length * sizeof (double));
  for(int i=0; i<length; i++)
    arr[i] = 0.0;
```

```
  return arr;
}
//instantiate int array with zeros
int* initialize_ints(int length){
  int *arr = malloc(length * sizeof (int));
  for(int i=0; i<length; i++)
    arr[i] = 0;
  return arr;
}

//convert an int to a float
double int_to_float(int number){
  return (double) number;
}


#ifdef BUILD_TEST
int main()
{
  double a = pow(1.0, 2.0);
  char s[] = "HELLO WORLD09AZ";
  char *c;
  double curve[] = {4.0, 5.0, 7.0, 8.0, DBL_MIN};
}
#endif
```

8.12 testall.sh (full testing script)

```
#!/bin/sh

# Regression testing script for PolyWiz
# Step through a list of files
#  Compile, run, and check the output of each expected-to-work test
#  Compile and check the error of each expected-to-fail test

# Path to the LLVM interpreter
LLI="lli"
#LLI="/usr/local/opt/llvm/bin/lli"

# Path to the LLVM compiler
LLC="llc"

# Path to the C compiler
CC="cc"

# Path to the polywiz compiler. Usually "./polywiz.native"
# Try "_build/polywiz.native" if ocamlbuild was unable to create a symbolic link.
POLYWIZ="./polywiz.native"
#POLYWIZ="_build/polywiz.native"

# Set time limit for all operations
ulimit -t 30

globallog=testall.log
rm -f $globallog
error=0
globalerror=0

keep=0

Usage() {
    echo "Usage: testall.sh [options] [.wiz files]"
    echo "-k   Keep intermediate files"
```

```
    echo "-h   Print this help"
    exit 1
}

SignalError() {
    if [ $error -eq 0 ] ; then
   echo "FAILED"
   error=1
    fi
    echo " $1"
}

# Compare <outfile> <reffile> <difffile>
# Compares the outfile with reffile. Differences, if any, written to difffile
Compare() {
    generatedfiles="$generatedfiles $3"
    echo diff -b $1 $2 ">" $3 1>&2
    diff -b "$1" "$2" > "$3" 2>&1 || {
   SignalError "$1 differs"
   echo "FAILED $1 differs from $2" 1>&2
    }
}

# Run <args>
# Report the command, run it, and report any errors
Run() {
    echo $* 1>&2
    eval $* || {
   SignalError "$1 failed on $*"
   return 1
    }
}

# RunFail <args>
# Report the command, run it, and expect an error
RunFail() {
    echo $* 1>&2
    eval $* && {
   SignalError "failed: $* did not report an error"
   return 1
    }
    return 0
}

Check() {
    error=0
    basename=`echo $1 | sed 's/.*\\///
                         s/.wiz//'`
    reffile=`echo $1 | sed 's/.wiz$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

    echo -n "$basename..."

    echo 1>&2
    echo "###### Testing $basename" 1>&2

    generatedfiles=""

    generatedfiles="$generatedfiles ${basename}.ll ${basename}.s ${basename}.exe ${basename}.out" &&
    Run "$POLYWIZ" "$1" ">" "${basename}.ll" &&
    Run "$LLC" "-relocation-model=pic" "${basename}.ll" ">" "${basename}.s" &&
    Run "$CC" "-o" "${basename}.exe" "${basename}.s" "library_functions.o -lm" &&
    Run "./${basename}.exe" > "${basename}.out" &&
    Compare ${basename}.out ${reffile}.out ${basename}.diff

```

```
    # Report the status and clean up the generated files

    if [ $error -eq 0 ] ; then
   if [ $keep -eq 0 ] ; then
      rm -f $generatedfiles
   fi
   echo "OK"
   echo "###### SUCCESS" 1>&2
    else
   echo "###### FAILED" 1>&2
   globalerror=$error
    fi
}

CheckFail() {
    error=0
    basename=`echo $1 | sed 's/.*\\///
                             s/.wiz//'`
    reffile=`echo $1 | sed 's/.wiz$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

    echo -n "$basename..."

    echo 1>&2
    echo "###### Testing $basename" 1>&2

    generatedfiles=""

    generatedfiles="$generatedfiles ${basename}.err ${basename}.diff" &&
    RunFail "$POLYWIZ" "<" $1 "2>" "${basename}.err" ">>" $globallog &&
    Compare ${basename}.err ${reffile}.err ${basename}.diff

    # Report the status and clean up the generated files

    if [ $error -eq 0 ] ; then
   if [ $keep -eq 0 ] ; then
      rm -f $generatedfiles
   fi
   echo "OK"
   echo "###### SUCCESS" 1>&2
    else
   echo "###### FAILED" 1>&2
   globalerror=$error
    fi
}

while getopts kdpsh c; do
    case $c in
   k) # Keep intermediate files
      keep=1
      ;;
   h) # Help
      Usage
      ;;
    esac
done

shift `expr $OPTIND - 1`

LLIFail() {
  echo "Could not find the LLVM interpreter \"$LLI\"."
  echo "Check your LLVM installation and/or modify the LLI variable in testall.sh"
  exit 1
}
```

```
which "$LLI" >> $globallog || LLIFail

if [ ! -f library_functions.o ]
then
    echo "Could not find library_functions.o"
    echo "Try \"make library_functions.o\""
    exit 1
fi

if [ $# -ge 1 ]
then
    files=$@
else
    files="tests/test-*.wiz tests/fail-*.wiz"
fi

for file in $files
do
    case $file in
   *test-*)
       Check $file 2>> $globallog
       ;;
   *fail-*)
       CheckFail $file 2>> $globallog
       ;;
   *)
       echo "unknown file type $file"
       globalerror=1
       ;;
    esac
done

exit $globalerror
```

8.13 Full Suite of Test Files

8.13.1 fail-assign1

```
def int main()
{
  int i;
  bool b;

  i = 42;
  i = 10;
  b = true;
  b = false;
  i = false; /* Fail: assigning a bool to an integer */
}
```

8.13.2 fail-assign2

```
def int main()
{
  int i;
  bool b;

  b = 48; /* Fail: assigning an integer to a bool */
}
```

8.13.3 fail-assign3

```
def void myvoid()
{
  return;
}

def int main()
{
  int i;

  i = myvoid(); /* Fail: assigning a void to an integer */
}
```

8.13.4 fail-dead1

```
def int main()
{
  int i;

  i = 15;
  return i;
  i = 32; /* Error: code after a return */
}
```

8.13.5 fail-dead2

```
def int main()
{
  int i;

  {
    i = 15;
    return i;
  }
  i = 32; /* Error: code after a return */
}
```

8.13.6 fail-expr1

```
int a;
bool b;

def void foo(int c, bool d)
{
  int dd;
  bool e;
  a + c;
  c - a;
  a * 3;
  c / 2;
  d + a; /* Error: bool + int */
}

def int main()
{
  return 0;
}
```

8.13.7 fail-expr2

```

int a;
bool b;

def void foo(int c, bool d)
{
  int d;
  bool e;
  b + a; /* Error: bool + int */
}

def int main()
{
  return 0;
}
```

8.13.8 fail-expr3

```
int a;
float b;

def void foo(int c, float d)
{
  int d;
  float e;
  b + a; /* Error: float + int */
}

def int main()
{
  return 0;
}
```

8.13.9 fail-float1

```
def int main()
{
  -3.5 and 1; /* Float with AND? */
  return 0;
}
```

fail-float2

```
def int main()
{
  -3.5 and 2.5; /* Float with AND? */
  return 0;
}
```

8.13.10 fail-for1

```
def int main()
{
  int i;
  for ( ; true ; ) {} /* OK: Forever */

  for (i = 0 ; i < 10 ; i = i + 1) {
```

```
    if (i == 3) return 42;
  }

  for (j = 0; i < 10 ; i = i + 1) {} /* j undefined */

  return 0;
}
```

8.13.11 fail-for2

```
def int main()
{
  int i;

  for (i = 0; j < 10 ; i = i + 1) {} /* j undefined */

  return 0;
}
```

8.13.12 fail-for3

```
def int main()
{
  int i;

  for (i = 0; i ; i = i + 1) {} /* i is an integer, not Boolean */

  return 0;
}
```

8.13.13 fail-for4

```
def int main()
{
  int i;

  for (i = 0; i < 10 ; i = j + 1) {} /* j undefined */

  return 0;
}
```

8.13.14 fail-for5

```
def int main()
{
  int i;

  for (i = 0; i < 10 ; i = i + 1) {
    foo(); /* Error: no function foo */
  }

  return 0;
}
```

8.13.15 fail-func1

```
def int foo() {}

def int bar() {}

def int baz() {}

def void bar() {} /* Error: duplicate function bar */

def int main()
{
  return 0;
}
```

8.13.16 fail-func2

```
def int foo(int a, bool b, int c) { }

def void bar(int a, bool b, int a) {} /* Error: duplicate formal a in bar */

def int main()
{
  return 0;
}
```

8.13.17 fail-func3

```
def int foo(int a, bool b, int c) { }

def void bar(int a, void b, int c) {} /* Error: illegal void formal b */

def int main()
{
  return 0;
}
```

8.13.18 fail-func4

```
def int foo() {}

def void bar() {}

def int printint() {} /* Should not be able to define printint */

def void baz() {}

def int main()
{
  return 0;
}
```

8.13.19 fail-func5

```

def int foo() {}

def int bar() {
```

```
  int a;
  void b; /* Error: illegal void local b */
  bool c;

  return 0;
}

def int main()
{
  return 0;
}
```

8.13.20 fail-func6

```

def void foo(int a, bool b)
{
}

def int main()
{
  foo(42, true);
  foo(42); /* Wrong number of arguments */
}
```

8.13.21 fail-func7

```
def void foo(int a, bool b)
{
}

def int main()
{
  foo(42, true);
  foo(42, true, false); /* Wrong number of arguments */
}
```

8.13.22 fail-func8

```

def void foo(int a, bool b)
{
}

def void bar()
{
}

def int main()
{
  foo(42, true);
  foo(42, bar()); /* int and void, not int and bool */
}
```

8.13.23 fail-func9

```
def void foo(int a, bool b)
```

```
{
}

def int main()
{
  foo(42, true);
  foo(42, 42); /* Fail: int, not bool */
}
```

8.13.24 fail-global1

```
int c;
bool b;
void a; /* global variables should not be void */


def int main()
{
  return 0;
}
```

8.13.25 fail-global2

```
int b;
bool c;
int a;
int b; /* Duplicate global variable */

def int main()
{
  return 0;
}
```

8.13.26 fail-if1

```
def int main()
{
  if (true) {}
  if (false) {} else {}
  if (42) {} /* Error: non-bool predicate */
}
```

8.13.27 fail-if2

```
def int main()
{
  if (true) {
    foo; /* Error: undeclared variable */
  }
}
```

8.13.28 fail-if3

```
def int main()
{
```

```
  if (true) {
    42;
  } else {
    bar; /* Error: undeclared variable */
  }
}
```

8.13.29 fail-nomain

8.13.30 fail-plot_empty_arr

```
def int main()
{
  poly poly1;
  poly poly2;
  poly poly_product;

  poly1 = new_poly([1.0, 2.0, 4.0], [2, 1, 0]);
  poly2 = new_poly([7.0, 4.0], [1, 0]);
  poly_product = poly1 * poly2;

  printint(plot([ ], "plots/plot_single.png"));

  return 0;
}def int main()
{
  poly poly1;
  poly poly2;
  poly poly_product;

  poly1 = new_poly([1.0, 2.0, 4.0], [2, 1, 0]);
  poly2 = new_poly([7.0, 4.0], [1, 0]);
  poly_product = poly1 * poly2;

  printint(plot([ ], "plots/plot_single.png"));

  return 0;
}
```

8.13.31 fail-plot_not_arr

def int main() poly poly1; poly poly2; poly poly$_{p}roduct$;

poly1 = new$_{p}oly([1.0, 2.0, 4.0], [2, 1, 0]); poly2 = new_{p}oly([7.0, 4.0], [1, 0]); poly_{p}roduct = poly1 * poly2;$

printint(plot(poly$_{p}roduct, "plots/plot_{s}ingle.png"$));

return 0;

8.13.32 fail-plot_wrong_type

```
def int main()
{
  poly poly1;
  poly poly2;
  poly poly_product;

  poly1 = new_poly([1.0, 2.0, 4.0], [2, 1, 0]);
  poly2 = new_poly([7.0, 4.0], [1, 0]);
  poly_product = poly1 * poly2;

```

```
  printint(plot([ 1.0, 3.0, 4.0 ], "plots/plot_single.png"));

  return 0;
}
```

8.13.33 fail-printb

```
/* Should be illegal to redefine */
def void printb() {}
```

8.13.34 fail-printint

```
/* Should be illegal to redefine */
def void printint() {}
```

8.13.35 fail-return1

```
def int main()
{
  return true; /* Should return int */
}
```

8.13.36 fail-return2

```
def void foo()
{
  if (true) return 42; /* Should return void */
  else return;
}

def int main()
{
  return 42;
}
```

8.13.37 fail-while1

```
def int main()
{
  int i;

  while (true) {
    i = i + 1;
  }

  while (42) { /* Should be boolean */
    i = i + 1;
  }

}
```

8.13.38 fail-while2

```
def int main()
{
  int i;

  while (true) {
    i = i + 1;
  }

  while (true) {
    foo(); /* foo undefined */
  }

}
```

8.13.39 test-add1

```
def int add(int x, int y)
{
  return x + y;
}

def int main()
{
  printint( add(17, 25) );
  return 0;
}
```

8.13.40 test-and

```
def int main()
{
  bool a;
  a = true and true; /* a = true */
  if(a) printint(1);
  return 0;
}
```

8.13.41 test-arith1

```
def int main()
{
  printint(39 + 3);
  return 0;
}
```

8.13.42 test-arith2

```
def int main()
{
  printint(1 + 2 * 3 + 4);
  return 0;
}
```

8.13.43 test-arith3

```
def int foo(int a)
{
  return a;
}

def int main()
{
  int a;
  a = 42;
  a = a + 5;
  printint(a);
  return 0;
}
```

8.13.44 test-arraylit

```
def int main()
{
  float[] new_arr;
  new_arr = [1.0,2.3,3.1];
  printint(1);

  return 0;
}
```

8.13.45 test-complex_program

```
/*

Interesting program that proves the Mean Value Theorem's underlying property.

It does this through the following steps:

1) Creates a poly, called poly_original
2) Builds a function to get poly_original's derivative, called poly_derivative
3) Calculates the average slope between two endpoints of poly_original
4) Iterate over values on poly_derivative to find the point of the average slope
5) If this point is found, for any polynomial, then the property behind
      the Mean Value Theorem holds.

This program also plots poly_original and poly_derivative on the same graph
to help visualize the problem.

*/


/* user-defined function to take derivative of polynomial */
def float[] derivative(float[] consts_arr, int poly_order){
  float[] poly_derivative;
  int i;
  poly_derivative = initialize_floats(poly_order);

  /* use calculus techniques to get derivative of each term */
  for(i=0; i<=poly_order; i=i+1){
    if(i>0){
      poly_derivative[i-1] = consts_arr[i] * int_to_float(i);
    }
  }

  return poly_derivative;
}
```

```

/* user-defined function to calculate the average slope between two points */
def float slope(float x1, float y1, float x2, float y2){
  return (y2-y1)/(x2-x1);
}

/* user-defined function to get severeal values of polynomial */
def float[] poly_values(poly p, float x1, float x2, int num_of_points){
  float[] values;
  float current_x;
  float delta_x;
  float temp;
  int i;
  values = initialize_floats(num_of_points);

  /* guarentee x2 > x1 */
  if(x1 > x2){
    temp = x2;
    x2 = x1;
    x1 = temp;
  }

  current_x = x1;
  delta_x = (x2-x1) / int_to_float(num_of_points);
  for(i=0; i<num_of_points; i=i+1){
    values[i] = p @ current_x;
    current_x = current_x + delta_x;
  }

  return values;
}

/* user-defined function to find if value in arr by some err margin */
def bool approx_in(float value, float[] arr, int arr_len, float err){
  bool in_arr;
  int i;

  in_arr = false;

  /* force err to be a positive value */
  err = | err |;

  /* check if value is in arr by margin of err */
  for(i=0; i<arr_len; i=i+1){
    if( |arr[i]-value| <= err){
      in_arr = true;
    }
  }

  return in_arr;
}


def int main()
{
  /* instantiate variables */
  poly poly_original;
  poly poly_derivative;
  float[] consts_arr;
  int i;
  int poly_original_order;
  float[] poly_derivative_consts;
  int[] poly_derivative_exps;
  float average_slope;
  float x1;
```

```
float x2;
float err;
int return_code;
float[] derivative_values;

/* LaTeX Stuff */
string[] body;
string text1;
string text2;
string text3;
string text4;
string text5;
string poly_original_string;
string poly_derivative_string;
string fp1;
string fp2;
string tdoc;

poly_derivative_exps = initialize_ints(2);

poly_original = new_poly([3.1, 10.0, 4.0], [2, 1, 0], 3);
poly_original_order = order(poly_original);
poly_original_string = print_tex(poly_original);

consts_arr = poly_original#;

poly_derivative_consts = derivative(consts_arr, poly_original_order);

/* create the exponents array of derivative poly */
for(i=0; i<poly_original_order; i=i+1){
  poly_derivative_exps[i] = i;
}

poly_derivative = new_poly(poly_derivative_consts, poly_derivative_exps, poly_original_order);
poly_derivative_string = print_tex(poly_derivative);

x1 = -7.0;
x2 = 7.0;

/* plot poly_original and poly_derivative together */

return_code = range_plot([ poly_original, poly_derivative ], x1, x2, "plots/complexprogram_plot.png");

/* calculate average slope of poly_original in range x1, x2 */
average_slope = slope(x1, poly_original @ x1, x2, poly_original @ x2);

/* collect several values on poly_derivative */
derivative_values = poly_values(poly_derivative, x1, x2, 1000);

err = 0.3;
/* if slope is in poly_derivative's values by margin of err */
if( approx_in(average_slope, derivative_values, 1000, err) ){

  text1 = "Every time I read a LaTeX document, I think, wow, this must be correct! - Prof. Christos
      Papadimitriou \\";
  text2 = "So, let's prove the MVT with Proof By LaTeX and PolyWiz. Consider the polynomial:";
  text3 = "Also, consider its derivative:";
  text4 = "Now, let's plot them both:";
  text5 = "And, as you can observe, the MVT holds! QED via LaTeX and PolyWiz";
  fp1 = "polywizard.png";
  fp2 = "plots/complexprogram_plot.png";

  body = [fp1, text1, text2, poly_original_string, text3, poly_derivative_string, text4, fp2, text5];

  tdoc = (tex_document(body, [0,7]));
```

```
    printstr(tdoc);
  }

  return 0;
}
```

8.13.46 test-else1

```

def int main()
{
  int i;
  i = 3;
  if (i > 5) {
    printint(1);
  } else {
    printint(8);
  }
  return 0;
}
```

8.13.47 test-fib

```

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

def int main()
{
  printint(fib(0));
  printint(fib(1));
  printint(fib(2));
  printint(fib(3));
  printint(fib(4));
  printint(fib(5));
  return 0;
}
```

8.13.48 test-float1

```
def int main()
{
  float a;
  a = 3.14159267;
  printf(a);
  return 0;
}
```

8.13.49 test-float2

```

def int main()
{
  float a;
```

```
  float b;
  float c;
  a = 3.14159267;
  b = -2.71828;
  c = a + b;
  printf(c);
  return 0;
}
```

8.13.50 test-float3

```
  def void testfloat(float a, float b)
{
  printf(a + b);
  printf(a - b);
  printf(a * b);
  printf(a / b);
  printb(a == b);
  printb(a == a);
  printb(a != b);
  printb(a != a);
  printb(a > b);
  printb(a >= b);
  printb(a < b);
  printb(a <= b);
}

def int main()
{
  float c;
  float d;

  c = 42.0;
  d = 3.14159;

  testfloat(c, d);

  testfloat(d, d);

  return 0;
}
```

8.13.51 test-for1

```
 def int main()
{
  int i;
  for (i = 0 ; i < 5 ; i = i + 1) {
    printint(i);
  }
  printint(42);
  return 0;
}
```

8.13.52 test-for2

```

def int main()
{
```

```
  int i;
  i = 0;
  for ( ; i < 5; ) {
    printint(i);
    i = i + 1;
  }
  printint(42);
  return 0;
}
```

8.13.53 test-func1

```
 def int add(int a, int b)
{
  return a + b;
}

def int main()
{
  int a;
  a = add(39, 3);
  printint(a);
  return 0;
}
```

8.13.54 test-func2

```
 def int fun(int x, int y)
{
  return 0;
}

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

  fun(i = 2, i = i+1);

  printint(i);
  return 0;
}
```

8.13.55 test-func3

```

def void printem(int a, int b, int c, int d)
{
  printint(a);
  printint(b);
  printint(c);
  printint(d);
}

def int main()
{
  printem(42,17,192,8);
  return 0;
}
```

8.13.56 test-func4

```
def int add(int a, int b)
{
  int c;
  c = a + b;
  return c;
}

def int main()
{
  int d;
  d = add(52, 10);
  printint(d);
  return 0;
}
```

8.13.57 test-func5

```
def int foo(int a)
{
  return a;
}

def int main()
{
  return 0;
}
```

8.13.58 test-func6

```
def void foo() {}

def int bar(int a, bool b, int c) { return a + c; }

def int main()
{
  printint(bar(17, false, 25));
  return 0;
}
```

8.13.59 test-func7

```
int a;

def void foo(int c)
{
  a = c + 42;
}

def int main()
{
  foo(73);
  printint(a);
  return 0;
}
```

8.13.60 test-func8

```
def void foo(int a)
{
  printint(a + 3);
}

def int main()
{
  foo(40);
  return 0;
}
```

8.13.61 test-func9

```
def void foo(int a)
{
  printint(a + 3);
  return;
}

def int main()
{
  foo(40);
  return 0;
}
```

8.13.62 test-gcd

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

def int main()
{
  printint(gcd(2,14));
  printint(gcd(3,15));
  printint(gcd(99,121));
  return 0;
}
```

8.13.63 test-gcd2

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

def int main()
{
  printint(gcd(14,21));
  printint(gcd(8,36));
  printint(gcd(99,121));
```

```
  return 0;
}
```

8.13.64 test-generate_tex

```

def int main()
{
  string[] body;
  string tex_doc;
  int[] ind;
  string intro;
  string outro;
  string fp;
  ind = [1];
  intro = "Edwards is a good prof";
  outro = "Hans is a chill TA";
  fp = "shrihan.png";

  body = [intro, fp, outro];

  tex_doc = tex_document(body, ind);

  printstr(tex_doc);

  return 0;
}
```

8.13.65 test-global1

```
int a;
int b;

def void printa()
{
  printint(a);
}

def void printbb()
{
  printint(b);
}

def void incab()
{
  a = a + 1;
  b = b + 1;
}

def int main()
{
  a = 42;
  b = 21;
  printa();
  printbb();
  incab();
  printa();
  printbb();
  return 0;
}
```

8.13.66 test-global2

```
bool i;

def int main()
{
  int i; /* Should hide the global i */

  i = 42;
  printint(i + i);
  return 0;
}
```

8.13.67 test-global3

```
int i;
bool b;
int j;

def int main()
{
  i = 42;
  j = 10;
  printint(i + j);
  return 0;
}
```

8.13.68 test-hello

```

def int main()
{
  printint(42);
  printint(71);
  printint(1);
  return 0;
}
```

8.13.69 test-helloworld

```
def int main(){
 printstr("Hello World");
 return 0;
}
```

8.13.70 test-if1

```

def int main()
{
  if (true) printint(42);
  printint(17);
  return 0;
}
```

8.13.71 test-if2

```
def int main()
{
  if (true) printint(42); else printint(8);
  printint(17);
  return 0;
}
```

8.13.72 test-if3

```

def int main()
{
  if (false) printint(42);
  if (true) printint(17);
  return 0;
}
```

8.13.73 test-if4

```
def int main()
{
  if (false) printint(42); else printint(8);
  printint(17);
  return 0;
}
```

8.13.74 test-if5

```
def int cond(bool b)
{
  int x;
  if (b)
    x = 42;
  else
    x = 17;
  return x;
}

def int main()
{
 printint(cond(true));
 printint(cond(false));
 return 0;
}
```

8.13.75 test-if6

```
def int cond(bool b)
{
  int x;
  x = 10;
  if (b)
    if (x == 10)
      x = 42;
  else
```

```
    x = 17;
  return x;
}

def int main()
{
 printint(cond(true));
 printint(cond(false));
 return 0;
}
```

8.13.76 test-in_arrays

```
def int main()
{
  int i;
  float f;
  string s;
  poly p1;
  poly p2;
  poly p3;
  int[] int_arr;
  float[] float_arr;
  string[] string_arr;
  poly[] poly_arr;

  i = 50;
  int_arr = [1, 2, -3, 76, 4, 0, 0, 50];
  if (i in int_arr and -3 in int_arr) {
    printint(1);
  }
  else if (88 in int_arr) {
    printint(0);
  }

  f = 5.9;
  float_arr = [1.6, 2.25, 3.14159267, -2.71828, 42.0, 0.0, 5.9];
  if (f in float_arr) {
    printint(0);
  }
  else if (8.9 in float_arr) {
    printint(1);
  }

  s = "wanker";
  string_arr = ["hello", "world", "", "wanker"];
  if (s in string_arr) {
    printint(1);
  }

  p1 = new_poly([1.0, 2.1, 4.0], [2, 1, 0], 3);
  p2 = new_poly([1.0, -1.0, 3.2, 0.0], [4, 2, 1, 0], 4);
  p3 = new_poly([1.0, -1.0, 3.3, 0.0], [3, 2, 1, 0], 4);
  poly_arr = [p1, p2];
  if (p2 in poly_arr and p1 in poly_arr) {
    printint(1);
  }
  else if (p3 in poly_arr) {
    printint(0);
  }

  return 0;
}
```

8.13.77 test-local1

```
def void foo(bool i)
{
  int i; /* Should hide the formal i */

  i = 42;
  printint(i + i);
}

def int main()
{
  foo(true);
  return 0;
}
```

8.13.78 test-local2

```
def int foo(int a, bool b)
{
  int c;
  bool d;

  c = a;

  return c + 10;
}

def int main() {
 printint(foo(37, false));
 return 0;
}
```

8.13.79 test-ops1

```
def int main()
{
  printint(1 + 2);
  printint(1 - 2);
  printint(1 * 2);
  printint(100 / 2);
  printint(99);
  printb(1 == 2);
  printb(1 == 1);
  printint(99);
  printb(1 != 2);
  printb(1 != 1);
  printint(99);
  printb(1 < 2);
  printb(2 < 1);
  printint(99);
  printb(1 <= 2);
  printb(1 <= 1);
  printb(2 <= 1);
  printint(99);
  printb(1 > 2);
  printb(2 > 1);
  printint(99);
  printb(1 >= 2);
  printb(1 >= 1);
  printb(2 >= 1);
```

```
  return 0;
}
```

8.13.80 test-ops2

```
def int main()
{
  printb(true);
  printb(false);
  printb(true and true);
  printb(true and false);
  printb(false and true);
  printb(false and false);
  printb(true or true);
  printb(true or false);
  printb(false or true);
  printb(false or false);
  printb(not false);
  printb(not true);
  printint(-10);
  printint(--42);
  printf(2.0 ^ 2.0);
  printf(|-2.0|);
}
```

8.13.81 test-or

```
def int main()
{
  bool a;
  a = false or true; /* a = true */
  if(a) printint(1);
  return 0;
}
```

8.13.82 test-plot_many

```

def int main()
{
  poly poly1;
  poly poly2;
  poly poly_product;

  poly1 = new_poly([1.0, 2.0, 4.0], [2, 1, 0], 3);
  poly2 = new_poly([7.0, 4.0], [1, 0], 2);
  poly_product = poly1 * poly2;

  printint(plot([poly1, poly2, poly_product], "plots/plot_many.png"));

  return 0;
}
```

8.13.83 test-plot_many_range

```
def int main()
{
```

```
  poly poly1;
  poly poly2;
  poly poly_product;

  poly1 = new_poly([1.0, 2.0, 4.0], [2, 1, 0], 3);
  poly2 = new_poly([7.0, 4.0], [1, 0], 2);
  poly_product = poly1 * poly2;

  printint(range_plot([poly1, poly2, poly_product], 1.0, 3.0, "plots/plot_many_range.png"));

  return 0;
}
```

8.13.84 test-plot_single

```
def int main()
{
  poly poly1;
  poly poly2;
  poly poly_product;

  poly1 = new_poly([1.0, 2.0, 4.0], [2, 1, 0], 3);
  poly2 = new_poly([7.0, 4.0], [1, 0], 2);
  poly_product = poly1 * poly2;

  printint(plot([ poly_product ], "plots/plot_single.png"));

  return 0;
}
```

8.13.85 test-plot_single_range

```
def int main()
{
  poly poly1;
  poly poly2;
  poly poly_product;

  poly1 = new_poly([1.0, 2.0, 4.0], [2, 1, 0], 3);
  poly2 = new_poly([7.0, 4.0], [1, 0], 2);
  poly_product = poly1 * poly2;

  printint(range_plot([ poly_product ], -12.0, 7.0, "plots/plot_single_range.png"));

  return 0;
}
```

8.13.86 test-poly_addition

```
def int main()
{
  poly poly1;
  poly poly2;
  poly poly_sum;

  poly1 = new_poly([1.0, 2.1, 4.0], [2, 1, 0], 3);
  poly2 = new_poly([1.0, -1.0, 3.2, 0.0], [3, 2, 1, 0], 3);
  poly_sum = poly1 + poly2;

```

```
  printf(poly_at_ind(poly_sum, 1));

  return 0;
}
```

8.13.87 test-poly_composition

```
def int main()
{
  poly poly1;
  poly poly2;
  poly poly_composed;

  poly1 = new_poly([1.0, 2.0, 4.0], [2, 1, 0], 3);
  poly2 = new_poly([7.0, 4.0], [1, 0], 2);
  poly_composed = poly1 : poly2;

  printf(poly_at_ind(poly_composed, 1));

  return 0;
}
```

8.13.88 test-poly_const_retriever

```
def int main()
{
  poly poly1;
  float[] consts_arr;
  int i;

  poly1 = new_poly([3.1, 10.0, 4.0], [3, 1, 0], 3);

  consts_arr = poly1#;

  for(i=0 ; i <= order(poly1) ; i=i+1){
    printf( consts_arr[i] );
    printstr("\n");
  }

  return 0;
}
```

8.13.89 test-poly_division

```
def int main()
{
  poly poly1;
  float[] consts_arr;
  int i;

  poly1 = new_poly([3.1, 10.0, 4.0], [3, 1, 0], 3);

  consts_arr = poly1#;

  for(i=0 ; i <= order(poly1) ; i=i+1){
    printf( consts_arr[i] );
    printstr("\n");
  }

```

```
  return 0;
}
```

8.13.90 test-poly_division

```
def int main()
{
  poly poly1;
  float denominator;
  poly poly_quotient;

  poly1 = new_poly([1.0, 2.0, 4.0], [2, 1, 0], 3);
  denominator = 2.0;
  poly_quotient = poly1 / denominator;

  printf(poly_at_ind(poly_quotient, 2));

  return 0;
}
```

8.13.91 test-poly_equal_comparison

```
def int main()
{
  poly poly1;
  poly poly2;

  poly1 = new_poly([0.0, 0.0, 4.0], [3, 1, 0], 3);
  poly2 = new_poly([4.0], [0], 1);

  printb(poly1 == poly2);

  return 0;
}
```

8.13.92 test-poly_evaluation

```
def int main()
{
  poly poly1;
  float x;
  float poly1_at_x;

  poly1 = new_poly([1.0, 2.0, 4.0], [2, 1, 0], 3);
  x = 3.0;
  poly1_at_x = poly1 @ x;

  printf(poly1_at_x);

  return 0;
}
```

8.13.93 test-poly_instantiation

```
def int main()
{
  poly poly1;
```

```
  printint(1);

  return 0;
}
```

8.13.94 test-poly_multiplication

```
def int main()
{
  poly poly1;
  poly poly2;
  poly poly_product;

  poly1 = new_poly([1.0, 2.0, 4.0], [2, 1, 0], 3);
  poly2 = new_poly([7.0, 4.0], [1, 0], 2);
  poly_product = poly1 * poly2;

  printf(poly_at_ind(poly_product, 1));

  return 0;
}
```

8.13.95 test-poly_new_poly

```
def int main()
{
  poly poly1;

  poly1 = new_poly([1.1, 2.1, 4.0], [2, 1, 0], 3);

  printf(poly_at_ind(poly1, 1));

  return 0;
}
```

8.13.96 test-poly_order

```
def int main()
{
  poly poly1;

  poly1 = new_poly([1.0, 2.0, 4.0], [2, 1, 0], 3);

  printint( order(poly1) );

  return 0;
}
```

8.13.97 test-poly_print_tex

```
def int main()
{
  poly poly1;
  string poly1_as_string;

  poly1 = new_poly([-1.0, -2.0, -4.0], [2, 1, 0], 3);

```

```
  poly1_as_string = print_tex(poly1);

  printstr(poly1_as_string);

  return 0;
}
```

8.13.98 test-poly_subtraction

```
def int main()
{
  poly poly1;
  poly poly2;
  poly poly_diff;

  poly1 = new_poly([1.0, 2.1, 4.0], [2, 1, 0], 3);
  poly2 = new_poly([1.3, -1.0, 3.2, 0.0], [3, 2, 1, 0], 4);
  poly_diff = poly1 - poly2;

  printf(poly_at_ind(poly_diff, 3));

  return 0;
}
```

8.13.99 test-poly_to_str

```
def int main()
{
  poly poly1;
  string poly1_as_string;

  poly1 = new_poly([-1.0, -2.0, -4.0], [2, 1, 0], 3);

  poly1_as_string = to_str(poly1);

  printstr(poly1_as_string);

  return 0;
}
```

8.13.100 test-string

```
def int main()
{
   printstr("hello world");
   return 0;
}
```

8.13.101 test-var1

```
def int main()
{
  int a;
  a = 42;
  printint(a);
  return 0;
}
```

8.13.102 test-var2

```
int a;

def void foo(int c)
{
  a = c + 42;
}

def int main()
{
  foo(73);
  printint(a);
  return 0;
}
```

8.13.103 test-while1

```
def int main()
{
  int i;
  i = 5;
  while (i > 0) {
    printint(i);
    i = i - 1;
  }
  printint(42);
  return 0;
}
```

8.13.104 test-while2

```
def int foo(int a)
{
  int j;
  j = 0;
  while (a > 0) {
    j = j + 2;
    a = a - 1;
  }
  return j;
}

def int main()
{
  printint(foo(7));
  return 0;
}
```