PolyGo

A Polynomial Manipulation Language

Pu Ke pk2532
Jin Zhou jz2792
Yanglu Piao yp2419
Jianpu Ma jm4437

20 December, 2016

Content

1 Introduction

1.1 Background

Polynomial is a commonly used mathematical expression written as the sum of products of coefficient and variables with exponents. Complex polynomial manipulation is always the indispensable part for all scientific practice. Some of its practical using scenarios include missile trajectory, drug effect detection, system stability check and weather forecasting. For single user, however, it might be rather inconvenient to manage polynomials because of all sorts of trivial computational steps. In order to address this, we introduce "PolyGo", the easy polynomial calculation language.

There are certain tools that support the representation and manipulation of polynomials, but in general, they all come as part of inconveniently large applications. For example, programs such as Matlab and Mathematica can indeed work on polynomials, however, they seem not to be straightforward and accessible for elementary users, and the application themselves carry a large amount of overhead in disk space and memory consumption. Users may find it hard to deploy the environments or feel depressed when seeing the high system requirement for such softwares. These issues can certainly be resolved by an simple and lightweight language.

On the other hand, some current softwares also have limitation and inconvenience in certain application scenario. Taking Matlab for example, if a user wants to add two polynomials with different orders, the first thing to do is to explicitly fill "0" as the coefficient to resolve the inconsistency in the power, which might be quite annoying if the formula is large. PolyGo, though it is for sure less powerful than than Matlab, it is far more lightweight, can accomplish many of the same polynomial operations as these other languages with far less expense.

1.2 Language Overview

PolyGo has its own unique polynomial data type, which can greatly facilitate polynomial manipulation. Besides, other primary data type includes int, float, string, boolean, complex and array.

A poly data type is represented by a list of floating numbers enclosed in curly braces representing the coefficients of the given polynomial. The first element in the floating point number list represents the lowest exponent of the polynomial (the constant term), the second element is the order one coefficient and higher degree coefficients follow the sequence. For example, polynomial variable $X^3+15X^2+50X-233$ can be represented as poly[3] a = {-233.0, 50.0, 15.0, 1.0};

Further, simple binary operations like add and minus can also be applied to two polynomials. Finally, a small set of built-in functions and logic expressions along with basic loop controls are implemented in PolyGo, allowing user to work on more complicated polynomial problems.

We can conclude that our PolyGo has the following advantages: flexible manipulation of polynomials, multiple algorithmic customization, applicable and scalable in variable fields.

2 Language Tutorial

2.1 Start with the compiler

To allow PolyGo run properly in your system, run the following commands to compile our PolyGo compiler:

$ make clean
$ make

The first command ensures there is no intermediate files in our working directory. And the second command runs the Makefile and manages the building process of the program.

After the successful making of PolyGo, user can run their own PolyGo using the top level command, which can be implemented as follows:

For printing AST only, add *-a*:
$ ./polygo -a test.pg

For printing LLVM IR and output without checking, add *-l*:
$./polygo -l test.pg

For printing LLVM IR and output with checking, add *-c*:
$./polygo -c test.pg

For default (no *flag*) case, it prints IR and output with checking:
$./polygo test.pg

After execution, the output of the program will be printed in the terminal, and there will be an output file *.out* generated in current directory.

2.2 Program Features

A simple PolyGo program basically consists of two parts: an optional global variable declaration *section* at the beginning of the program and a series of function definition sections followed including a central main function. The main function is a necessary component as the entry point of a program. And each function includes a list of local variable declarations and a list of statements.

For any user with basic knowledge with C or Java, PolyGo is just pretty straightforward to use. Nevertheless, a few thing is noticeable: All local variables must be declared prior to any statements inside the function; type system is static and strict since there is no automatic type conversion; return statement is necessary for every function even when it is void; void cannot be used in any type of declaration but only as the return type of a function.

2.3 Code Example

The following code calculates the greatest common divisor.

```
/* Calculate the greatest common divisor */
int gcd(int x,int y){
	int a = x;

	int b = y;
	while(a!=b){
		if(a>b){
			a = a-b;
		}else{
			b = b-a;
		}
	}
	return a;
}int main(){
	print(gcd(11,121));
	return 0;
}
```

Run *$./polygo test-gcd.pg*
In *test-gcd.out*

```
11
```

The following code demonstrates the how to calculate the maximum volume of air inhaled into the lung giving the polynomial lung volume function V(t) = -0.041t3 + 0.181t2 + 0.202t.

We use the Newton Raphson method to get the root for the polynomial and write function to calculate the the maximum volume value .

```
/* A program using Newton’s Method to solve the maximum of a function */
/* test-lung.pg */

float F(float x)    /* the first derivation */
{
       float a = -0.123 * x * x + 0.362 * x + 0.202;
    return a;
}

float Fd(float x)   /* the second derivation */
{
    return (-0.246 * x + 0.362);
}

float fabs(float x)   /*the absolute value */
{
       float a;
       if( x > 0.0 )
              a = x ;
       else
              a = -x;
       return a;
}

void main()
{
    float x0 = 4.0;
    float h;
    float err = 0.0001;
    float root;
    float x1;
    int miter = 10;
    int iter;
    float fncvalue;
    float max;
    print("Approximation 4,the max error 0.0001, and the maximum number of iterations 
10");
    iter=1;
    while(iter <= miter)
    {
        h = F(x0)/Fd(x0);
        /* calculate f(x)/f'(x)as we do in Newton Raphson method */
        print(iter);
        print(h);
```

```
        x1 = x0 - h;  /* x1=x0-f(x)/f'(x) */
        if(fabs(h) < err) /*If 2 approximations is below the max error */
        {
            root = x1;   /*then make the approximation as the root */
            break;
        }
        else
             x0 = x1;
        ++ iter;
    }
      print("maximum is at:");
    print(root);
    max = -0.041 * root * root * root + 0.181 * root * root + 0.202 * root;
    print("maximum value is");
    print(max);
    return;
}
```

Run *$./polygo test-lung.pg*
In *test-lung.out*

```
Approximation 4,the max error 0.0001, and the maximum number of iterations 10
1
0.511254
2
0.064788
3
0.001075
4
0.000000
maximum is at:
3.422883
maximum value is
1.167821
```

3 Language Reference Manual

PolyGo has the great features of solving polynomial problems, and it also supports complex number operations. Moreover, detailed build-in functions provide more possibilities for the extensible arithmetic customization.

3.1 Lexical convention

3.1.1 Identifier

An identifier is a sequence of alphabetic and numeric characters. The first character must be alphabetic. "_" can also be used in an identifier, but not as the first character identifier.

3.1.2 Keyword

Keywords in PolyGo are special identifiers reserved as part of the programming language itself. They may not be used or referenced in any other way; function definitions and variable naming cannot override keywords.The following identifiers are reserved for use as keywords:

int	float	complex	string	bool	poly
void	true	false	main	if	else
while	for	return	true	false	

3.1.3 Comments

PolyGo supports the block comments enclosed between /* and */

```
/* This is a comment and will not be executed*/

/* This is a multi-line
comment*/
```

3.2 Data Types

PolyGo support the following data types. The name of the type, the type details and declaration and initialization examples are given below.

int	An integer number	int a = 1;
float	A floating number	float a = 1.5;
string	A character string	string a =" Hello";
bool	A boolean sign	bool a = true; bool b = false;
array	A sequence of literal (int, float ,bool)	int[3] a =[2, 21, 14];

complex	A complex number in float form	complex a =<1.0, 2.4> /* 1.0+2.4i */
poly	A sequence of float number that stores the coefficients of a polynomial	poly[2] a = {4.12, 12.2, 9.0} /* 4.12+12.2X+9X^2 */
void	A term represents none	void main( ) /* used in return type of function declaration */

3.3 Constants

3.3.1 Integer constant

An integer constant is a sequence of digits. The integer constant is positive if no sign is designated. “-” makes the constant negative

3.3.2 Float constant

A float constant include an integer part, a decimal point, a fraction part, an e, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part, or the fraction part (not both) may be missing; either the decimal point or the e and the exponent (not both) may be missing. The following sequence are accepted as a float constant: 2., 0.8e-17, .5e+2, 3.5e-4.

3.3.3 String constant

A string is a set of characters enclosed in “ and ”. A string is considered as an array of characters.

3.3.4 Complex constant

A complex constant include two float constants, separated by comma and enclosed in < and >, e.g. <1.5, -2.5>.

3.4 Operators

3.4.1 Basic operators

The basic operators in PolyGo include assignment, additive, multiplicative, relational and boolean operators.

=	Assignment	a = 1;
+, -, *, /	Arithmetic operations	b = (2+2*3)/4 - 1;
%	Modulus	c = 8%7;

++, --	Increment and decrement	d = ++i;
||, &&, !	Logic OR, AND and NOT	a = (true||!true) && false;
<, >, <=, >=, ==, !=	Relational and equality comparisons	if (a>=b) { return a;}

3.4.2 Other operators

;	Terminator for a statements
,	Actual and formal separator, poly/array element separator
“”	String literal declaration
{}	Function block, statement block, poly type assignment cover
( )	Function arguments cover, expression precedence, conditional parameters
[ ]	Array type assignment cover
[[ ]]	Indexing array element and polynomial coefficient

Examples of the above mentioned operators in polynomial and array declaration, initialization, assignment and indexing.

```
/*Assume a polynomial: p =3.0X^2+2.0X-5.0, q=2.0X^2+1.0X+6.0*/
poly[2] p = {-5.0, 2.0, 3.0};
poly[2] q = {6.0, 1.0, 2.0};
int[3] a =[2, 21, 14];

print(p[[1]]);
/*  output 2.0, p[[n]] returned the coefficient of (n+1)th order term*/
print(a[[1]];
/* output 21, a[[[1]] returned the second elements of array a*/
```

3.5 Declarations

3.5.1 Variable declarations

When an assignment statement is preceded by a variable type, the statement is interpreted as a variable declaration. In PolyGo, all the variables must be declared before used, and it can be rewritten. Variables declared within a function must be declared at the beginning of a function, and global variables must be declared at the top of the main area of a program.

Syntax:

Type ID ;	int a;
Type ID = expression;	int a =b+c;
poly[length] ID ={ expression list}	poly[2] x={2.3, 9.2, 1.0};
bool[length] ID =[expression list]	bool[2] y=[true, false, true]

3.5.2 Functions Declaration

Syntax:

```
type fun_name ( parameter1, parameter2, ...)
{statements}
```

Every function will have a return type, if the function does not return a value, then typename should be void. Next will be the function name (funcname), followed by a comma-separated list of types which correspond to the types of the parameters in the function's argument list. This list could be empty if the function takes no parameters.Then comes the statements list.

3.6 Expressions

Expression is a very important part of our language. Many basic operations belong to this part. And the precedence of expressions is something that worth mentioning. Unary operators have the highest precedence, next are multiplicative operators followed by additive operations. Next are logical operators and finally, assignment expressions.

3.6.1 Primary Expressions

Primary expression is a very basic part of expressions including ID, constants function calls and element extractions.

- **ID**

```
int i = 1; /*Here, i is an ID.*/
```

An ID used by itself represents a variable and evaluates to the actual value stored in that variable. It can appear both side of an expression.

- **Constant**

```
int i = 1; /*1 is a constant.*/
```

A constant is an explicit right-value term (number, string) that analyze itself in a given context. Types of constants are shown in the section above.

- **Function Calls**

```
int my_func(int a,int b){return 0;}
```

The parameter includes the type and the identifier, allowing passing arguments from the function call.

As showed earlier, type is the return type of this function, funcname is the name of the function, and parameter1, parameter2, etc. are the real parameters passed to the function. Each parameter must be an expression. Commas are used to separate following parameters.

- **Coefficient Extraction**

```
poly[2] p={4.0,2.0,1.0};
int a;
a=p[[2]];/*Here a is assigned to the coefficient of a poly
variable*/
```

`p` is a poly variable and `a` is an integer variable. This expression extracted the quadratic coefficient of the polynomial `p` and store it into `a`.

- **Constants**

```
int a = 1;
float f = 1.0;
bool b = true;
string s = “Hello world!”;
complex c = <1.0,2.0>;
```

Above are examples of basic constants.

- **Polylit, Arrlit**

```
int[2] a = [1,2];
poly[2] f = {1.0,2.0,3.0};
```

These are expressions to declare array and polynomial respectively. Be aware that although they are both declared as length 2, polynomial type actually has to be declared with 3 elements. Because we need to declare coefficient for its constant term. I.e., here we are declaring an int array `a` with [1,2] as its element, and polynomial `f` with {1.0,2.0,3.0} as its element, which means `f=1.0*x^2+2.0*x+3.0`.

3.6.2 Unary operators

Unary operators are associated with an expression.

- **Neg**

```
int i = 1;
int j = -i;/*This is a Neg operator. */
```

- **Not**

```
bool b1 = true;
bool b2 = !b1;/*This is a Not operator. */
```

- **Addone**

```
int i = 1;
int j = ++i;/*This is an Addone operator. */
```

- **Subone**

```
int i = 1;
int j = --i;/*This is a Subone operator. */
```

- **sqrt**

```
float f = 1.0;
float g = sqrt(f);
/*This is a sqrt operator, returns positive square root of f */
```

3.6.3 Binary operators

Binary operators are associated with two expressions.

- **Add, Sub, Mult, Div, Asn**

```
int i = 1;
int j = i + i;
j = i - i;
j = i * i;
j = i / i;
/*Above are basic arithmetic binary operators.*/
```

- **Equal, Neq, Less, Leq, Greater, Geq, And, Or**

```
bool b1 = (i==i);
bool b2 = (i!=i);
bool b3 = (i<i);
bool b4 = (i<=i);
bool b5 = (i>i);
bool b6 = (i>=i);
bool b7 = (b1&&b1);
bool b8 = (b1||b1);
/*These are basic logical operators.*/
```

- **Modulus**

```
float f = 2.0;
float g = f % 3.0;
/*This is modulus operator.*/
```

3.7 Statements

Statements are the fundamental elements of a function. A sequence of statements will be executed sequentially, unless the flow-control statements intervene. Flow -control statement are represented as if, for, while and break in PolyGo, all statements end with a semicolon “;”.

3.7.3 Conditional statements

if Statement

```
if ( expression ) { statements }
```

The expression after if is a logic expression. statements represents a list of language statements. In an if statement, if the logical expression is evaluated to be true, then the (statements) will be executed. If not, ignore the statements and don’t execute them.

if-else Statement

```
if ( expression ) { statement1 } else { statement2 }
```

The expression after if is a logical expression that can decide which statement to be executed. In an if-else statement, if the condition expression is evaluated to be true, then the statement1 is executed and the statement2 is ignored. If false, statement2 is executed and the first list is ignored. Break statement is allowed in any statement list to jump of the loop.

3.7.4 Iterative statements

while statement

```
while ( expression ) { statements (break) }
```

The while statement first evaluates the logic expression inside the parenthesis. If it evaluates to true, statement is executed, and then the same expression is evaluated again. statement continues to execute repeatedly as long as it is true after each execution of statement. User can jump of the while loop using break.

for statement

```
for (expression1; expression2; expression3) { statements (break) }
```

The for statement can be expressed as:

```
expression1 ;
while ( expression2 )
{
statement;
expression3 ;
}
```

There must be an expression in each of the three positions.

3.7.5 Return statement

```
return expression/ return;
```

Expression represents an result being returned by a function. The type the expression evaluates should match the function's return type. A return with no expression is used to exit a function that returns void.

3.8 Built-in Functions

3.8.1 print Function

Syntax: print (a)
The print function will print the expression to standard output. The argument can be either int type, float type, boolean type, string type and complex type.

3.8.2 print_n Function

Syntax: print_n (c)
The print function will print the imaginary part of a complex type. The argument can only be complex type.

3.8.3 order Function

Syntax: order (p)
The order function will return the order of a polynomial type. The argument can only be complex constant.

3.5.4 modulus Function

Syntax: mod (p)
The modulus function will return the modulus of a complex constant. The argument can only be polynomial constant.

3.9 Other issues

• Statement after return will not be executed.
• Re-assignment is supported to a previous variable.
• The element inside a polynomial type can only be float type.
• The element list inside an array can be void , e.g intarr e=[2, 21, 1]; e=[]; array e now is empty (This is for roots representation for polynomial)
• For element extraction of an array, number n in the index location indicates the (n+1)th element in an array, since indices count from 0.
• For element extraction of a polynomial, number n in the index location indicates the coefficient with n as its variable exponent.

4 Project Plan

4.1 Teamwork

Our group consists of 4 people , Jin Zhou, Pu Ke, Jianpu Ma and Yanglu Piao from EE and CE. We met twice a week after class on Monday and Wednesday. Before milestone (LRM, hello world), we spent the weekend together in lehman library ,working on the project together.

Throughout this project we used incremental strategy together with iterative development process. We splitted the entire program into four part as: scanner, parser, semantic checker and code generator. For each component, we performed iterative tactic to hit our goal. We assigned roles for each team member and hold weekly discussion to push the development.

We wrote regression tests to guarantee that our most fundamental features still worked after each submission. Once a member uploaded something, others will be noticed and do the review. This ensured the high quality in terms of style, and prevented logic errors in our code.

4.2 Development Process

A rough outline of the our timetable was designed at the beginning. This includes hard deadlines of each milestone needed to be completed so that the next milestone could be started. As the day went, more specific timelines were settled for completion of features and resolution of issues and problems. Basically we worked from scanner to parser, AST, semantic checking to target code generation. The first two layer went rather smoothly, and we could scan the code, parse it and use pretty printer to recheck the code. However, when we worked on code generator, we found a lot of things different from what we expected, so we spent a lot of time refining our design and made specific change to previous code, which lowered down our paces a little bit. So the final version of our language is essentially a superset of what we specified in our LRM.

4.3 Testing Process

We had a few unit tests and integration test before the "Hello word" program. Later on, we worked on test-driven process. If any single new test case failed, we would work on together to fix that problem. And each our test case was carefully created to test the core functions or regulation of our language.

4.4 Team Responsibilities

The general team responsibilities were assigned to four members as described in the table below. However, since we worked together all the time, there was no strict division of responsibilities as multiple members contributed to multiple parts, depending on the stage of the project.

Pu Ke	Scanner, Parser, AST, Code generation, Demo
Jin Zhou	Parser, AST, Semantic checking, Test case creation, Demo
Yanglu Piao	Code generation, Testing
Jianpu Ma	Test validation, Slide, Documentation

4.5 Project Timeline

Our timeline was carefully laid out from the start:

Sep 24th	Proposal due date
Oct 7th	Specific syntax created
Oct 14th	Scanner/parser unambiguous and working
Oct 20th	LRM first draft
Oct 27th	LRM due date
Nov 10th	Early version of Architectural design

Nov 22nd	Architectural design finalized
Dec 15th	Compiler works, all tests pass
Dec 17th	Minor changes and write examples
Dec 18th	Arrange project slide and report
Dec 20th	Final report due date

5 Language Evolution

5.1 Language Initialization

We got our language inspiration from the polynomial implementation in Matlab and National Instrument. In digital and controlling fields which we engineering students are interacted with every day, polynomial and complex are used frequently. Thus, starting from the assumption to design a language closely related to daily life, we aim at implementing the language that can be used directly to express polynomial and utilized to address both simple and complex problems.

5.2 Language Style Identification

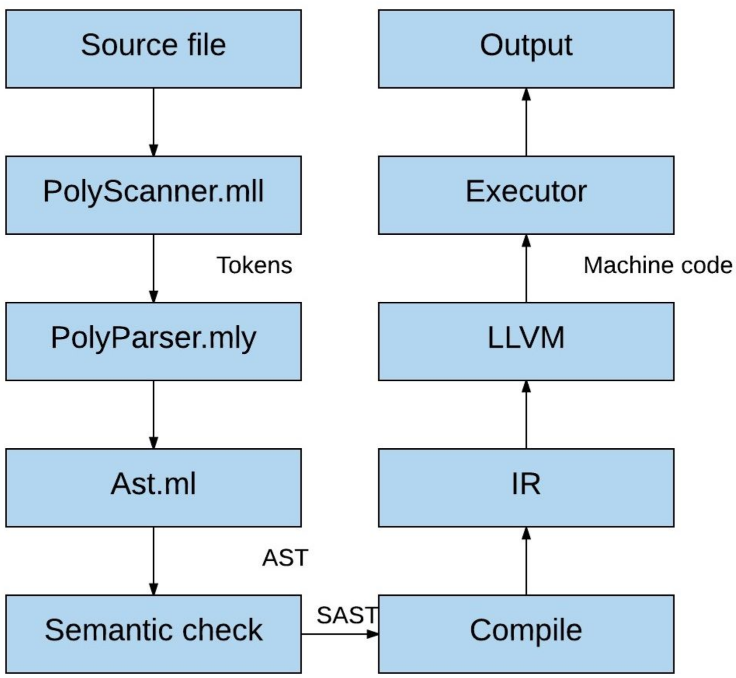
As having little knowledge about the overall complexity and certain bottlenecks of writing a compiler, we consulted with TA jacob to know the difficulty of implementing certain language functions. In the first language reference manual (the version we submitted in the coursework), we described data types of our languages containing primitive ones such as int, float, boolean, complex and polynomial, and derived data type containing array, with certain fundamental functions such as findroot, in which roots expressed as complex numbers are returned in an array, and several fancy functions such as drawing curves indicated by polynomial and complex.

5.3 Language Implementation

With more practice of setting down to realize the compiler, we gradually found that it is a little out of scope to fill in all the functions proposed in the first-version language reference manual. So we wrote our second version of language reference manual. Also, we realized that without declaring size of polynomial type which we insisted on in the beginning as viewing it as a primitive data type and significant feature of our language, it's quite difficult for the code generator to allocate memories for it. Therefore, we make it implemented quite similar with the array type, both of which are declared with size clarified in the same style. And the ways to extract element from them are also the same, with a double brackets following the variable. The only difference is that the polynomial literal is expressed in braces while the array literal is expressed in brackets. In addition, along the way we walk through the realization of compiler, we updated our language reference manual step by step, balancing what we expect to do and what we can make in practice.

6 Architecture Design

The system architecture generally follows the MicroC design discussed in lecture, which contains the basic parse and compile part, and it uses LLVM to produce the execute machine code and generate the output file. The architecture graph is as below：



Scanner

The Poly scanner scan our input file and tokenizes the code. Then whitespaces (including space, newline and tab) and comments are discarded. And if there exists any illegal character combinations, such as malformed escape sequences, a Scanner_error exception is raised. The scanner is in polygo/scanner.mll

Parser

Parser scans the tokens passed to it by the scanner and constructs an abstract syntax tree(AST) based on the definitions provided and the input tokens. The top level of the abstract syntax tree is a structure containing all classes and a structure containing all include statements.

Meanwhile, all imports are resolved and the function definitions in imported source files are combined into local function definition list, And after that, the program is ready for semantic checking. The parser is in polygo/parser.mly, and AST data types are defined in poly/ast.ml.

Semantic checker

The semantic checker will walk through the AST to check the correctness of the statements.The checking contents mainly include : Function declarations, Global, formal and local declarations, Variable initialization, Type of operands, Predicate of for and while loop, Function calls, Return and break statements.

Semantic analysis is performed on each statement and expression in each block of code. So that it can make sure that types are consistent in every expression, variables are declared in the proper scope and number of arguments is corresponded. The purpose is to provide the code generator with data that is organized more similarly to the LLVM code that it will eventually generate. The semantic check functions are defined in polygo/semant.ml.

Code Generator

The code generator uses the semantic checked information to construct the LLVM IR file which contains the final instructions for the program. It constructs expressions bottom-up and develops basic blocks for control-flow statements. With LLVM, the machine could finally generate the executable code for running. Output will be shown in the terminal windows and an output file will be generated in the same directory after execution. The code generator is in polygo/codegen.ml

7 Lesson learned

7.1 Jianpu Ma

There are a couple of things I learned from this project,as my first team project assignment in Columbia, it is frustrating at some point,not only because the opaqueness inside Ocaml and LLVM, but the role as a group member in a team work. Honestly, how to organize the whole project plan so that every member of the team can work on a feasible part is super essential. since it's quite easy to fall behind or lose track. Good time schedule and proper arrangement can make a team job quite joyful. Second thing is, to design an wise and achievable pattern for a language will take much longer time than expected, a lot of changes has to be made through practice. So never underestimate even a single grammar in any language.

7.2 Yanglu Piao

I've learned many things from this project. Firstly, everything should have begun early. We spent too much time discussing something not that important before really doing something. Which means our scanner, parser and ast are built up very late. So time for code generation and semantic checker was very limited. I had to stay up for nights to write code generation. Secondly, our github is a disaster. I bet each one of us has at least three different versions of our project locally. I have to learn how to use github more efficiently. Thirdly, I'd better read more references before doing a project. Because I spent too much time on a wrong direction. For example, the way to store array type in our language was stupid, and this caused some further bad influence. Luckily I fixed the bug with the help of my teammates. And finally, I should have split the work of code generation early. This was really not something that could be done alone. Many things we wrote in our proposal were not able to be implemented because of the lack of time.

7.3 Jin Zhou

This project is definitely challenging, nauseating but somewhat appealing to work on with my dear teammates. The excitement I had when our language finally did something (a simple gcd and an ugly Newton's Method) overwhelmed other happiness in the past few months. A cool language with concise syntax and support to mathematical operations. OCaml and LLVM are all cool stuffs to play around. However, we didn't do several things well: our type system is still messy; the polynomial type is not strong enough to support complex coefficient; the built-in functions are limited. In terms of the development process, there were twist and turns deciding all kinds of greatness of our polynomial type. However, it finally turned out to be too unrealistic though we simplified them again and again. Another lesson is learned: do not underestimate the complexity of an unfamiliar developing environment.

7.4 Pu Ke

Although time is limited and there have been a lot of functions not realized in our compiler, the experience of implementing our own language is still so fantastic and I learned a lot from this first formal group programming project for me. Here are the lessons I learned:

1. Start early. Then you will have time to realize as many functions as possible after you really start to know things.
2. Always try to learn things ahead of time. And do it by hands since doing is different from just reading and trying to understand. For example, when I started to write parser, I know little about semantic checking, which at that time just gave me a sense it can do little things. So I felt I should write parser as 'tight' as possible, resulting in a 'layers within layers' tree. And it`s afterwards that I found there was no need to do things like that. And it`s the same with the semantic checking and codegen. When implementing assignment for a field in an array, codegen can check automatically if a type of value can be assigned to that kind of array, since it will be an

error if the allocated memory is not matched with the type passed in. Thus, there is no need to create arrayint, arrayfloat, arrayboolean, so that semantic checking can work.
3. Learn more about using git. Working on different codes without updating git or being buried in merging conflicts can be really annoying.
4. There are always some spirits gotten from reading other projects in previous years.
5. Testing procedure is a little chaotic and changing always generates other bugs which we didn`t check out in time.
6. Play more with ocaml and try to be as familiar with it as possible. There are a lot of time wasted on misuse of ocaml grammar whose error information cannot always help. For example, the compulsory use of “else” in “if then else” wasted me a lot of time when writing “print” in codegen.
7. Learn from teammates and discuss more frequently with TA rather than just once a week.
8. In terms of our compiler, apart from demanding more fancy built-in functions and richer libraries, there are some fundamental problems waited to be solved. For example, various data types should be supported in our polynomial, which now can only have floats in it; and for array, which can contain the same kind of data as int, float or boolean, should also support complex. For function, its formal type should support poly array and complex, and so it is with its return type, both of which just support int, float, and boolean now. User input function should be added, which is so important to a language.
9. Make more commands of the expectation of what we can do with our language and what we can implement in our practice. Don`t fancy too much in the beginning or take the easiest way in the end.
10. Unify coding style among group members, and communicate more with whom is working with you on the same piece, which may yield a more systematic management involving other managing softwares rather than just git.
I love this course by the way! And it is not as scary at all as I was told.

8 Appendix 1 (Source Code)

8.1 Parser

```
/*
Project:  COMS S4115, PolyGo Compiler
Filename: src/parser.mly
Authors:   Pu Ke            pk2532
           Jin Zhou         jz2792
           Yanglu Piao      yp2419
```

```
           Jianpu Ma        jm4437

Purpose: * Ocamlyacc parser for PolyGo
*/

%{ open Ast %}

%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA
%token LBRACKET RBRACKET LLBRACKET RRBRACKET
%token PLUS MINUS TIMES DIVIDE PLUSONE MINUSONE MODULUS VB ASSIGN SQRT ORDER
%token EQ NEQ LT LEQ GT GEQ TRUE FALSE AND OR NOT
%token RETURN IF ELSE FOR WHILE PASS BREAK
%token INT FLOAT BOOL COMPLEX POLY STRING VOID

%token <int> INTLIT
%token <float> FLOATLIT
%token <string> ID
%token <string> STRINGLIT
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE MODULUS
%right PLUSONE MINUSONE
%right NOT NEG

%start program
%type <Ast.program> program

%%

/* array and poly element can only be: complex, int, float --primary_ap, typ
array is initiated in the beggging, and remain unchanged, that means the value
of the element can be extracted,
      but cannot be assigned or changed. Size must be indicated */
program:
```

```
      decls EOF { $1 }

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

fdecl:/*??? no void type for fdecl */
   typ ID LPAREN formal_list_opt RPAREN LBRACE vdecl_list_opt stmt_list_opt
RBRACE
     {{ ftyp = $1;
       fname = $2;
       formals = $4;
       locals = List.rev $7;
       body = List.rev $8  }}

typ: /* primary type */
        INT { Int }
      | FLOAT { Float }
      | COMPLEX { Complex }
      | BOOL { Bool }
      | STRING { String }
      | VOID { Void }
      | POLY { Poly }

formal:
        typ ID                                    { Prim_f_decl( $1, $2 ) }
      | typ LBRACKET RBRACKET ID               { Arr_f_decl( $1, $4) }

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

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

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

vdecl:
```

```
      typ ID SEMI                                     { Primdecl($1, $2) }
      | typ ID ASSIGN expr SEMI                  { Primdecl_i($1, $2, $4) }
      | typ LBRACKET INTLIT RBRACKET ID SEMI                           {
Arr_poly_decl($1, $5, $3) }
      | typ LBRACKET INTLIT RBRACKET ID ASSIGN LBRACKET expr_list_opt RBRACKET
SEMI { Arrdecl_i($1, $5, $3, List.rev $8) }
      | typ LBRACKET INTLIT RBRACKET ID ASSIGN LBRACE expr_list_opt RBRACE SEMI
{ Polydecl_i( $1, $5, $3, List.rev $8) }
      | typ LBRACKET INTLIT RBRACKET ID ASSIGN ID SEMI
{Arr_poly_decl_i($1,$5,$3,$7)}

stmt_list_opt:
      PASS SEMI       {[]}
  | stmt_list  { $1 }

  stmt_list:
    stmt  { [$1] }
  | stmt_list stmt { $2 :: $1 }

stmt:
      expr SEMI { Expr $1 }
      | RETURN SEMI { Return Noexpr }
      | RETURN expr SEMI { Return $2 }
      | LBRACE stmt_list_opt 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) }
      | BREAK SEMI             { Break }



expr:
        INTLIT                         { Intlit( $1 ) }
      | ID                                             { Id($1) }
      | ID LLBRACKET expr RRBRACKET           { Extr( $1, $3 ) }
      | ORDER LPAREN expr RPAREN {Order($3)}
      | FLOATLIT                         { Floatlit( $1 ) }
      | STRINGLIT                        { Strlit( $1 ) }
      | FALSE            { Boollit( false ) }
      | TRUE             { Boollit( true ) }
      | LT FLOATLIT COMMA FLOATLIT GT       { Complexlit( $2, $4 ) }
```

```
      | LBRACE expr_list_opt RBRACE            { Polylit($2 ) }
       /* array, the whole array can be void, but any of the element cannot be 
void */
      | LBRACKET expr_list_opt RBRACKET { Arrlit($2 )}
      | LPAREN expr RPAREN { $2 }
      /* Binop */
      | expr PLUS   expr { Binop($1, Add,   $3) }
      | expr MINUS  expr { Binop($1, Sub,   $3) }
      | expr TIMES  expr { Binop($1, Mult,  $3) }
      | expr DIVIDE expr { Binop($1, Div,   $3) }
      | expr EQ     expr { Binop($1, Equal, $3) }
      | expr NEQ    expr { Binop($1, Neq,   $3) }
      | expr LT     expr { Binop($1, Less,  $3) }
      | expr LEQ    expr { Binop($1, Leq,   $3) }
      | expr GT     expr { Binop($1, Greater, $3) }
      | expr GEQ    expr { Binop($1, Geq,   $3) }
      | expr AND    expr { Binop($1, And,   $3) }
      | expr OR     expr { Binop($1, Or,    $3) }
      | expr MODULUS expr { Binop($1, Modu,    $3) }
      | VB expr VB           { Mod($2) }
      /*  one operand */
      | MINUS expr %prec NEG { Unop(Neg, $2) }
      | NOT expr         { Unop(Not, $2) }
      | PLUSONE   expr  { Unop( Addone, $2 ) }
      | MINUSONE expr { Unop( Subone, $2 ) }
      | SQRT LPAREN expr RPAREN {Unop(Sqrt,$3)}
      /* function call */
      | ID LPAREN expr_list_opt RPAREN { Call( $1, $3 ) }
      /* assignment */
      | expr ASSIGN expr { Asn( $1, $3 ) }


/* expr_list_opt:
                     { [] }
      | expr_list {  List.rev $1 }
expr_opt:
                         { Noexpr }
      | expr { $1 }

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

```
      */
expr_list_opt:
        expr_opt { [$1] }
      | expr_list_opt COMMA expr_opt { $3 :: $1 }

expr_opt:
                         { Noexpr }
      | expr { $1 }
```

8.2 Scanner

```
(*
Project:  COMS S4115, PolyGo Compiler
Filename: src/scanner.mll
Authors:   Pu Ke            pk2532
           Jin Zhou         jz2792
           Yanglu Piao      yp2419
           Jianpu Ma        jm4437

Purpose:  * Scan an inputted PolyGo file
*)

{ open Parser }

let Exp = 'e'['+' '-']?['0'-'9']+

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

```
| ')'      { RPAREN }
| '{'      { LBRACE }
| '}'      { RBRACE }
| '['    { LBRACKET }
| ']'    { RBRACKET }
| "[["         { LLBRACKET }
| "]]"         { RRBRACKET }
| '>'      { GT }
| '<'      { LT }
| '%'    { MODULUS }
| "++"         { PLUSONE }
| "--"         { MINUSONE }
| "|"    { VB }
| '+'      { PLUS }
| '-'      { MINUS }
| '*'      { TIMES }
| '/'      { DIVIDE }
| '='      { ASSIGN }
| "sqrt"   { SQRT }
| "=="     { EQ }
| "!="     { NEQ }
| "<="     { LEQ }
| ">="     { GEQ }
| "&&"     { AND }
| "||"     { OR }
| '!'      { NOT }
| "if"     { IF }
| "else"   { ELSE }
| "for"    { FOR }
| "while"  { WHILE }
| "return" { RETURN }
| "int"    { INT }
| "bool"   { BOOL }
| "void"   { VOID }
| "true"   { TRUE }
| "false"  { FALSE }
| "float"  { FLOAT }
| "complex"{ COMPLEX }
| "string" { STRING }
| "poly"   { POLY }
| "order"  { ORDER }
| "pass"   { PASS }
```

```
| "break"  { BREAK }
| ['0'-'9']+ as lxm { INTLIT(int_of_string lxm) }
| ('.'['0'-'9']+Exp? | ['0'-'9']+('.'['0'-'9']*Exp? | Exp ) ) as lxm {
FLOATLIT(float_of_string lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
| '"'[^'\n']*'"' as lxm { STRINGLIT(lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

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

8.3 AST

```
(*
Project:  COMS S4115, PolyGo Compiler
Filename: src/ast.ml
Authors:   Pu Ke            pk2532
           Jin Zhou         jz2792
           Yanglu Piao      yp2419
           Jianpu Ma        jm4437

Purpose: * Generate abstract syntax tree
         * Functions for printing the abstract syntax tree for checking
*)

type op = Add | Sub | Mult | Div | Modu | Equal | Neq | Less | Leq | Greater |
Geq |
          And | Or
type unop = Neg | Not | Addone | Subone | Sqrt
type typ = Int | Float | Complex | Bool | String | Void | Poly
type bind = typ * string


type expr =
    Id of string
  | Order of expr
```

```
  | Extr of string * expr
  | Intlit of int
  | Floatlit of float
  | Boollit of bool
  | Strlit of string
  | Complexlit of float * float
  | Polylit of expr list
  | Arrlit of expr list
  | Binop of expr * op * expr
  | Unop of unop * expr
  | Asn of expr * expr
  | Call of string * expr list
  | Mod of expr
  | 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
  | Break

type formaldecl =
   Prim_f_decl of typ * string
  |Arr_f_decl of typ * string

type variabledecl =
   Primdecl of typ * string
  |Primdecl_i of typ * string * expr
  |Arr_poly_decl of typ * string * int
  |Arrdecl_i of typ * string * int * expr list
  |Polydecl_i of typ * string * int * expr list
  |Arr_poly_decl_i of typ * string * int * string

type functiondecl =
  {
  ftyp: typ;
  fname: string;
```

```
  formals: formaldecl list;
  locals: variabledecl list;
  body: stmt list;
  }

type program = variabledecl list * functiondecl list

(* Pretty-printing functions *)

let string_of_op = function
    Add -> "+"
  | Sub -> "-"
  | Mult -> "*"
  | Div -> "/"
  | Modu -> "%"
  | Equal -> "=="
  | Neq -> "!="
  | Less -> "<"
  | Leq -> "<="
  | Greater -> ">"
  | Geq -> ">="
  | And -> "&&"
  | Or -> "||"

let string_of_unop = function
    Neg -> "-"
  | Not -> "!"
  | Addone -> "++"
  | Subone -> "--"
  | Sqrt -> "sqrt"

let string_of_typ = function
    Int -> "int"
  | Float -> "float"
  | Complex -> "complex"
  | Bool -> "bool"
  | String -> "string"
  | Poly -> "poly"
  | Void -> "void"

let rec string_of_expr = function
   Intlit(l) -> string_of_int l
```

```
  | Floatlit(f) -> string_of_float f
  | Id(s) -> s
  | Extr(s, e) -> s ^ "[[" ^ string_of_expr e ^ "]]"
  | Complexlit(a,b) ->  "<" ^ string_of_float a ^ "," ^ string_of_float b  ^ ">"
  | Boollit(true) -> "true"
  | Boollit(false) -> "false"
  | Polylit(p) -> "{" ^ String.concat "," (List.map string_of_expr p) ^ "}"
  | Strlit(p) -> p
  | Arrlit (arr) -> "[" ^ String.concat "," (List.map string_of_expr arr) ^ "]"
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
  | Unop(o, e) -> string_of_unop o ^ "(" ^ string_of_expr e ^ ")"
  | Mod(e) -> "|" ^ string_of_expr e ^ "|"
  | Order(e)->"order(" ^ string_of_expr e ^ ")"
  | Asn(v, e) -> string_of_expr v ^ " = " ^ 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(e) -> string_of_expr e ^ ";\n";
  | Return(e) -> "return " ^ string_of_expr e ^ ";\n";
  | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s
^ "\n"
  | 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  ^ ")\n" ^ string_of_stmt s
  | While(e, s) -> "while (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s
  | Break -> "break;\n"

let string_of_formaldecl = function
  Prim_f_decl(t, s) -> string_of_typ t ^ " " ^ s
  | Arr_f_decl(t, s) ->  string_of_typ t ^ " " ^ s ^ "[]"

let string_of_variabledecl = function
    Primdecl(a,b) -> string_of_typ a ^ " " ^ b ^ ";"
  | Arr_poly_decl_i(a,b,c,d) -> string_of_typ a ^ " " ^ " [" ^ string_of_int c ^
"]" ^ b ^ " = " ^  d ^ ";"
```

```
  | Primdecl_i (a,b,c) -> string_of_typ a ^ " " ^ b ^ " = " ^ string_of_expr c ^
";"
  | Arr_poly_decl(a,b,c) -> string_of_typ a ^ " [" ^ string_of_int c ^ "]" ^ b ^
";"
  | Polydecl_i (a,b,c,d) -> string_of_typ a ^ " [" ^ string_of_int c ^ "]" ^ b ^
" = " ^ "{" ^ String.concat ", " (List.map string_of_expr d) ^ "}" ^ ";"
  | Arrdecl_i (a,b,c,d) -> string_of_typ a ^ " [" ^ string_of_int c ^ "]" ^ b ^
" = " ^ "[" ^ String.concat ", " (List.map string_of_expr d) ^ "]" ^ ";"

let string_of_fdecl fdecl =
  string_of_typ fdecl.ftyp ^ " " ^
  fdecl.fname ^ "(" ^ String.concat ", " (List.map string_of_formaldecl
fdecl.formals) ^
  ")\n{\n" ^
  String.concat "\n" (List.map string_of_variabledecl fdecl.locals) ^ "\n" ^
  String.concat "" (List.map string_of_stmt fdecl.body) ^
  "}\n"

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

8.4 Semantic Checker

```
(*
Project:  COMS S4115, PolyGo Compiler
Filename: src/semant.ml
Authors:   Pu Ke            pk2532
           Jin Zhou         jz2792
           Yanglu Piao      yp2419
           Jianpu Ma        jm4437

Purpose: * Semantic checking for the PolyGo compiler
         * Returns void if successful. Otherwise throws exception.
*)

open Ast
```

```
module StringMap = Map.Make(String)

let semant_check ast  =
      let check_duplicate l err=
            let rec helper list = match list with
                  n1 :: n2 :: _ when n1 = n2 -> raise (Failure (err^n1))
            | _ :: t -> helper t
            | [] -> ()
            in helper l
      in
      let function_decls =
            let built_in_decls =  StringMap.add "print"
                  { ftyp = Void; fname = "print"; formals = [Prim_f_decl(Int,
"x")];
                  locals = []; body = [] } (StringMap.empty)
            in   (* built-in function how to check the polymorphism of print???
*)
            List.fold_left (fun m fd -> StringMap.add fd.fname fd m)
built_in_decls (List.rev (snd ast))
      in
      let function_decl s = try StringMap.find s function_decls
            with Not_found -> raise (Failure ("undefined function " ^ s))
      in
      let check_func ast  =
            let fname_list = List.fold_left (fun l fd -> fd.fname :: l)
["print"] (List.rev (snd ast))
            in
            let _ = function_decl "main"
            in (* Ensure "main" is defined *)
            check_duplicate (List.sort compare fname_list) ("duplicate function
definition (or conflict with built-in function): ")
      in
      let typ_of_identifier s map =
            try StringMap.find s map
            with Not_found ->
                  raise (Failure ("undecalred identifier " ^ s))
      in
      let check_assign lv rv err = (* Besides lv=rv, poly and array indexing is
also considered *)
            if lv = rv then lv else
```

```
            if lv = "poly_co" && (rv = "int" || rv = "float" || rv = "complex")
then rv else
            if (lv = "int array" || lv = "float array" || lv = "bool array" ||
lv = "complex array") && rv = "array" then lv
            else raise err
      in
      let typ_arrtyp s =           (* When an array is declared, its type is
transformed before being stored in map *)
            if s = "int" then "int array" else
            if s = "float" then "float array" else
            if s = "bool" then "bool array" else
            if s = "complex" then "complex array"
            else "array"
      in
      let arrtyp_typ s =
            if s = "int array" then "int" else
            if s = "float array" then "float" else
            if s = "bool array" then "bool" else
            if s = "complex array" then "complex"
          else "float"
      in
      let check_expr expression m =   (* Give the type of the expression *)
            let build_formal_list l e = match e with    (* Build the formal list
for Call expression *)
                        Prim_f_decl(t, id) -> (string_of_typ t, id) :: l
                  | Arr_f_decl(t, id) -> (typ_arrtyp(string_of_typ t), id) :: l
            in
            let check_extr t expr =
                  if t <> "poly" && t <> "int array" && t <> "float array" && t
<> "bool array" && t <> "complex array" then
                  raise (Failure("type " ^ t ^ " is illegal for indexing: " ^
expr ^ ". poly or array type is expected."))
            in
            let rec expr_typ expr map = match expr with
                   Intlit _ -> "int"
                  | Floatlit _ -> "float"
                  | Complexlit (_, _) -> "complex"
                  | Polylit _ -> "poly"
                  | Boollit _ -> "bool"
                  | Strlit _ -> "string"
                  | Arrlit l as e -> (match l with
                                    [] -> "array"
```

```
                                          | [a] -> typ_arrtyp (expr_typ a map)
                                          | head :: tail -> ignore(List.iter (fun
arr -> ignore(check_assign (expr_typ head map) (expr_typ arr map)
(Failure(string_of_expr e ^
                                                ": type of the elements in array
assignment list is inconsistent ")))) tail); typ_arrtyp (expr_typ head map))
                  | Binop(e1, op, e2) as e -> let t1 = expr_typ e1 map
                                                           and t2 = expr_typ e2
map in
                                                           (match op with
                                                                 Add | Sub | Mult
| Div when t1 = t2 -> t1
                                                                 | Equal | Neq
when t1 = t2 -> "bool"
                                                                 | Less | Leq |
Greater | Geq when t1 = t2 && (t1 = "int" || t1 = "float" || t1 = "bool") ->
"bool"
                                                                 | And | Or when
t1 = "bool" && t2 = "bool" -> "bool"
                                                                 | Modu when t1 =
"int" && t2 = "int" -> "int"
                                                                 | _ -> raise
(Failure ("illegal binary operator " ^ t1 ^ " " ^ string_of_op op ^
                                                                       " " ^ t2 ^
" in " ^ string_of_expr e))
                                                           )
                  | Unop(op, e1) as e -> let t = expr_typ e1 map in
                                                     (match op with
                                                           Neg when t = "int" ||
t = "float" -> t
                                                           | Not when t = "bool"
-> "bool"
                                                           | Addone | Subone
when t = "int" -> "int"
                                                           | Sqrt when t =
"float" -> "float"
                                                           | _ -> raise (Failure
("illegal unary operator " ^ string_of_unop op ^
                                                                       " " ^ t ^
" in " ^ string_of_expr e))
                                                     )
                  | Order(op) as ex-> if (expr_typ op m) = "poly" then "int"
```

```
                                  else raise (Failure ("illegal type " ^
expr_typ op m ^ " in " ^ string_of_expr ex ^ ". Poly is expected."))
                 | Id s -> typ_of_identifier s map
                 | Extr (s, e) as ex -> if expr_typ e map = "int" then
      (* array and poly indexing *)
                                  let t = typ_of_identifier s map in
                                  ( ignore(check_extr t (string_of_expr
ex)); (* check indexing is legal or not *)
                                  arrtyp_typ (typ_of_identifier s map))
                            else raise (Failure("The index cannot be type
" ^ expr_typ e map ^ " in " ^ string_of_expr ex))
                 | Asn(extr, e) as ex -> let lt = expr_typ extr map
                                        and rt = expr_typ e map in
                                        check_assign lt rt (Failure
("illegal assignment " ^ lt ^ " = " ^ rt ^ " in " ^ string_of_expr ex))
                 | Mod c as e -> let t = expr_typ c map in
                                  (if t = "complex" then "float"
                                        else raise (Failure ("illegal
type " ^ t ^ " in Mod expression " ^ string_of_expr e)))
                 | Call (fname, actuals) as call-> if fname = "print" then
"int" else
                                                            (let fd =
function_decl fname
                                                            and l =
List.fold_left build_formal_list [] (function_decl fname).formals
                                                            in
                                                            if List.length
actuals <> List.length l then
                                                                  raise (Failure
("expecting " ^ string_of_int
                                                                  (List.length l)
^ " arguments in " ^ string_of_expr call))
                                                            else
                                                                  List.iter2 (fun
(ft, _) e ->
                                                                  let et =
expr_typ e map in
                                                                  ignore
(check_assign ft et (Failure ("illegal actual argument found in " ^
string_of_expr call)))) l (List.rev actuals);
                                                            string_of_typ
fd.ftyp)
```

```
                  | Noexpr -> "void"
            in
            expr_typ expression m
      in
      let check_decl table decl_list =   (* Check the variable declarations *)
            let check_function m variabledecl=
                  let check_void t e =   (* all decl type can not be void *)
                        if t = "void"  then raise (Failure ("Illegal variable
type voidma in " ^ string_of_variabledecl e))
                  in
                  let check_arr t expr = (* check if the type can be declared as
an array *)
                        if t <> "int" && t <> "float" && t <> "bool" && t <>
"complex" then
                        raise (Failure("type " ^ t ^ " is not supported as an
array type in " ^ expr))
                  in
                  let check_arr_init t id i l map=   (* check the type and size
of array initializer *)
                        if List.length l <> i
                        then raise (Failure ("array " ^ id ^ ": length of the
initializer doesn't match the array size"));
                        List.iter (fun e -> ignore(check_assign (string_of_typ
t) (check_expr e map) (Failure("array " ^ id ^
                              ": type of the element " ^ check_expr e map ^ " in
initialization " ^ "[" ^ String.concat "," (List.map string_of_expr l) ^ "]"
                              ^ " doesn't match the array type " ^ string_of_typ
t )))) l
                  in
                  let check_poly_init i id l map decl =    (* check the type and
size of poly initializer *)
                        if List.length l <> i+1
                        then raise (Failure ("poly " ^ id ^ ": length of the
initializer doesn't match the poly size"));
                        List.iter (fun e -> let t = check_expr e map in
                                                      if t <> "int" && t <>
"float" && t <> "complex" then
                                                      raise(Failure("type " ^ t ^
" of " ^ string_of_expr e ^ "can not be used in the initilization of " ^
string_of_variabledecl decl))) l
                  in
                  let check_arr_decl t expr =
```

```
                       if t <> "poly" && t <> "int" && t <> "float" && t <>
"bool" && t <> "complex" then
                       raise (Failure("type " ^ t ^ " is illegal in " ^ expr ^
". poly or acceptable array type is expected."))
                 in
                 let create vdecl = match vdecl with           (* check the
declarations *)
                       Primdecl(t, id) as decl -> ignore(check_void
(string_of_typ t) decl);
                                                     StringMap.add id
(string_of_typ t) m
                 | Primdecl_i(t, id, e) as decl -> ignore(check_void
(string_of_typ t) decl);     (* primitive type with initialization *)
                                                           let rt = check_expr e
m in
                                                           ignore( check_assign
(string_of_typ t) rt
                                                           (Failure ("illegal
assignment " ^ string_of_typ t ^
                                                          " = " ^ rt ^ " in " ^
string_of_variabledecl decl)));
                                                           StringMap.add id
(string_of_typ t) m
                 | Arr_poly_decl (t, id, _) as decl -> check_arr_decl
(string_of_typ t) (string_of_variabledecl decl);

StringMap.add id (typ_arrtyp (string_of_typ t)) m
                 | Arrdecl_i (t, id, i, l) as decl -> ignore(check_arr
(string_of_typ t) (string_of_variabledecl decl));   (* arry decl with
initialization *)
                                                           ignore(check_arr_init
t id i l m);
                                                           StringMap.add id
(typ_arrtyp (string_of_typ t)) m
                 | Polydecl_i (t, id, i, l) as decl -> if string_of_typ t <>
"poly" then        (* poly decl with initialization *)

raise(Failure("type " ^ string_of_typ t ^ " is illegal in " ^
string_of_variabledecl decl))
                                                                 else
check_poly_init i id l m decl;
```

```
                                                              StringMap.add id
(string_of_typ t) m
                 | Arr_poly_decl_i (t, id1, _, id2) -> ignore(typ_of_identifier
id2 m); StringMap.add id1 (typ_arrtyp (string_of_typ t)) m
                 in
                 create variabledecl
           in
           List.fold_left check_function table decl_list
     in
     let check_body func map =
           let check_bool e = if check_expr e map <> "bool"
                                 then raise (Failure ("expected Boolean
expression in " ^ string_of_expr e ))
           in
           let rec check_stmt counter s = match s with
                 Expr e -> ignore (check_expr e map); counter (*counter is used
to check if the break statement is legal*)
           | If (e, s1, s2) -> ignore(check_bool e); ignore(check_stmt counter
s1); ignore(check_stmt counter s2); counter
           | For (e1, e2, e3, s) -> ignore (check_expr e1 map);
ignore(check_bool e2); ignore(check_expr e3 map); ignore(check_stmt 1 s);
counter
           | While (e, s) -> ignore(check_bool e); ignore(check_stmt 1 s);
counter
           | Return e -> if string_of_typ func.ftyp <> check_expr e map then
                             raise (Failure ("Invalid return type " ^ check_expr
e map ^ " in function " ^ func.fname ^
                                   ". It should be " ^ string_of_typ
func.ftyp)); counter
           | Break -> if counter > 0 then counter-1
                       else raise (Failure("illegal Break statement in function
" ^ func.fname))
           | Block e -> let rec check_block blk cnt = match blk with
                             [Return _ as st] -> ignore(check_stmt cnt st); cnt
                       | Return _ :: _ -> raise (Failure "nothing can follow a
Return statement")
                       | Block sl :: ss -> check_block (sl @ ss) cnt
                       | st :: ss -> ignore(check_block ss (check_stmt cnt
st)); cnt
                       | [] -> cnt
                 in ignore(check_block e counter); counter
           in
```

```
            check_stmt 0 (Block func.body)
      in
      let formal_table global func =
            let check_void_formal t e =
                  if string_of_typ t = "void"
                  then raise (Failure ("illegal type void in formal decl " ^
string_of_formaldecl e))
            in
            let build_list l e = match e with
                        Prim_f_decl(_, id) -> id :: l
                  | Arr_f_decl(_, id) -> id :: l
            in
            let formal_create map e = match e with
                  Prim_f_decl(t, id) ->  ignore(check_void_formal t e);
                                                StringMap.add id (string_of_typ
t) map
            | Arr_f_decl(t, id) -> ignore(check_void_formal t e);
                                          StringMap.add id (typ_arrtyp
(string_of_typ t)) map
            in
            ignore(check_duplicate (List.sort compare (List.fold_left build_list
[] func.formals))
                  ("duplicate formal variables in function " ^ func.fname ^ ":
"));
            List.fold_left formal_create global func.formals
      in
      check_func ast;  (* check if there is duplicate function definition first
*)
      let global = check_decl StringMap.empty (List.rev (fst ast)) in (*check
global variables*)
      List.iter (fun func -> ignore(check_body func (check_decl (formal_table
global func) func.locals))) (List.rev (snd ast));;
```

8.5 Code Generation

```
(*
Project:  COMS S4115, PolyGo Compiler
Filename: src/codegen.ml
```

```
Authors:   Pu Ke           pk2532
           Jin Zhou        jz2792
           Yanglu Piao     yp2419
           Jianpu Ma       jm4437

Purpose: * Translates semantically checked PolyGo AST to LLVM IR
*)

(* 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

module StringMap = Map.Make(String)

(* Translate to llvm type *)
let translate (globals, functiondecl) =
  let context = L.global_context () in
  let the_module = L.create_module context "PolyGo"
  and i32_t  = L.i32_type  context
  and i8_t   = L.i8_type   context
  and i1_t   = L.i1_type   context
  and d64_t  = L.double_type context
  and void_t = L.void_type context in

  let ltype_of_typ = function
      A.Int -> i32_t
    | A.Float -> d64_t
    | A.Bool -> i1_t
    | A.Void -> void_t
    | A.String  -> L.pointer_type i8_t
```

```
    | A.Complex -> d64_t
    | A.Poly -> d64_t
  in

(* TODO-more global variables *)
let type_of_global= function
      A.Primdecl    (t,s)-> (t,s)
    | A.Primdecl_i  (t,s,_) -> (t,s)
    | A.Arrdecl_i    (t,s,_,_) ->(t,s)
    | A.Polydecl_i (t,s,_,_) -> (t,s)
    | A.Arr_poly_decl (t,s,_)->(t,s)
    | A.Arr_poly_decl_i (t,s1,_,_) -> (t,s1)in

(* Initialization of global variables *)
  let init t= (match t with  A.Int -> L.const_int (ltype_of_typ t) 0
                                | A.Float -> L.const_float (ltype_of_typ t) 0.0
                                | A.Bool -> L.const_int (ltype_of_typ t) 0
                                | A.Void -> L.const_null (ltype_of_typ t)
                                | A.String -> L.const_pointer_null (ltype_of_typ
t)
                                | A.Complex -> L.const_float (ltype_of_typ t)
0.0
                                | A.Poly -> L.const_float (ltype_of_typ t) 0.0
                                )in

  (* Declare each global variable; remember its value in a map *)
  let global_vars =
    let global_var m global =
        let (t,s) = type_of_global global in
    StringMap.add s ((L.define_global s (init t) the_module),0,0) m in
  List.fold_left global_var StringMap.empty globals in


  (* Declare printf(), which the print built-in function will call *)
  let printf_t = L.var_arg_function_type i32_t [| L.pointer_type i8_t |] in
    let printf_func = L.declare_function "printf" printf_t the_module in
(*  let printf_s = L.var_arg_function_type i32_t [| L.pointer_type i8_t |] in
    let printf_func_s = L.declare_function "printf" printf_s the_module in *)
  let printf_f = L.var_arg_function_type i32_t [| L.pointer_type i8_t |] in
    let printf_func_f = L.declare_function "printf" printf_f the_module in

  (* TODO-more formal types *)
```

```
 let type_of_formaldecl = function
    A.Prim_f_decl (t,s) -> (ltype_of_typ t,s)
  | A.Arr_f_decl (t,s) -> (ltype_of_typ t,s)
in
(* Define each function (arguments and return type) so we can call it *)
let function_decls =
  let function_decl m fdecl =
    let typ'= List.map type_of_formaldecl fdecl.A.formals in
    let name = fdecl.A.fname and
        formal_types = Array.of_list (List.map (fun (t,_) ->t) typ') in
    let ftype = L.function_type (ltype_of_typ fdecl.A.ftyp) formal_types in
    StringMap.add name (L.define_function name ftype the_module, fdecl) m in
  List.fold_left function_decl StringMap.empty functiondecl
in
(* Fill in the body of the given function *)
let build_function_body fdecl =
  let (the_function,_) = StringMap.find fdecl.A.fname 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 "%f\n" "float" builder
  and str_format_str = L.build_global_stringptr "%s\n" "str" builder
  and real_format_str = L.build_global_stringptr "%.3f+" "real" builder
  and image_format_str = L.build_global_stringptr "%.3fi\n" "image" 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 *)
(* Some usefule functions *)
  let rec range a b =
                  if a > b then []
                  else a :: range (a+1) b in

  let rec zeros_int length= if length < 0 then []
                  else 0 :: zeros_int (length-1) in
  let rec zeros_float length= if length < 0 then []
                  else  0.0 :: zeros_float (length-1)in

  let init_val t expr = match expr with A.Intlit i -> [L.const_int i32_t i]
                                      | A.Floatlit f -> [L.const_float d64_t f]
                                      | A.Complexlit (e1,e2) -> [L.const_float
d64_t e1;L.const_float d64_t e2]
```

```
                                      | _ -> (match t with A.Int -> [L.const_int
i32_t 0]
                                                        | A.Float ->
[L.const_float d64_t 0.0]
                                                        | A.String ->
[L.const_pointer_null (ltype_of_typ t)]
                                                        | A.Bool -> [L.const_int
i1_t 0]
                                                        | A.Poly ->
[L.const_float d64_t 0.0]
                                                        | _ ->
raise(Failure("Invalid Type123")))
                                      (*  *)  in
    let init_local t length = (match t with  A.Int -> List.map (fun i ->
A.Intlit i) (zeros_int length)
                                | A.Bool -> [A.Intlit 0]
                                | A.Float -> List.map (fun f -> A.Floatlit f)
(zeros_float length)
                                | A.Complex -> List.map (fun e -> A.Floatlit
e)(zeros_float length)
                                | A.Poly -> List.map (fun f -> A.Floatlit f)
(zeros_float length)
                                | A.String -> [A.Strlit ""]
                                | _ -> raise(Failure("Invalid Type"))

                                )in

  (* Type of local variables *)
    let type_of_locals = function
        A.Primdecl    (t,s)-> (match t with A.Complex -> (t,s,0,(init_local t
1),0)| _ -> (t,s,0,(init_local t 0),0))
      | A.Primdecl_i  (t,s,e) -> let mark = (match e with A.Id _-> 2| A.Intlit _
-> 1| A.Floatlit _-> 1| A.Complexlit (_,_) -> 1|_->2) in
                          (match t with A.Complex -> let ee = (match e with
A.Complexlit (e1,e2)->[A.Floatlit e1;A.Floatlit e2]

|_ -> (init_local t 1) ) in (t,s,0,ee,mark)
                                      | _ -> (t,s,0,[e],mark))
      | A.Arr_poly_decl      (t,s,i)->(match t with A.Poly ->(t,s,i,(init_local
t i),0)| _ -> (t,s,i,(init_local t (i-1)),0))
      | A.Arrdecl_i    (t,s,i,e) -> let mark = (match List.hd e with A.Id _-> 2|
_ -> 2) in (t,s,i,e,mark)
```

```
      | A.Polydecl_i (t,s,i,e) -> let mark = (match List.hd e with A.Id _-> 2| _
-> 2) in (t, s,i,e,mark)
      | A.Arr_poly_decl_i (t,s1,i,_) ->(match t with A.Poly
->(t,s1,i,(init_local t i),2)| _ -> (t,s1,i,(init_local t (i-1)),2)) in

  (* Store formal and local variables *)
    let local_vars =
      let add_formal m (formal_typ, name) param = L.set_value_name name param;

      let local = L.build_alloca (ltype_of_typ formal_typ) name builder in
      ignore (L.build_store param local builder);
      StringMap.add name (local,0,0) m in

  let store_val typ name length expr _ builder= (match length with 0 -> (match
typ with A.Complex -> (let addr = L.build_array_alloca d64_t (L.const_int i32_t
2) name builder in

let r = range 0 1 in

let i  = List.map (fun index -> [|L.const_int i32_t index|]) r in

let addr' = List.map (fun i -> L.build_in_bounds_gep addr i "comp_addr"
builder)i in

let asn_e = (List.map (fun e -> List.hd (init_val typ e)) expr) in

ignore(List.map2 (fun addr e-> L.build_store e addr builder)addr' asn_e);addr)
                                                                   | _ -> let
addr = L.build_alloca (ltype_of_typ typ) name builder in

ignore(L.build_store (List.hd (init_val typ (List.hd expr))) addr builder);addr)
                                      | _ ->(match typ with A.Poly -> (let addr
= L.build_array_alloca d64_t (L.const_int i32_t (length+1)) name builder in

let r = range 0 length in

let i  = List.map (fun index -> [|L.const_int i32_t index|]) r in

let addr' = List.map (fun i -> L.build_in_bounds_gep addr i "poly_addr"
builder)i in

let asn_e = List.map (fun e -> List.hd (init_val typ e)) (List.rev expr) in
```

```
ignore(List.map2 (fun addr e-> L.build_store e addr builder)addr' asn_e);addr)

                                                            | _ ->  (let addr =
L.build_array_alloca (ltype_of_typ typ) (L.const_int i32_t (length)) name
builder in

let r = range 0 (length-1) in

let i  = List.map (fun index -> [|L.const_int i32_t index|]) r in

let addr' = List.map (fun i ->  L.build_in_bounds_gep addr i "arr_addr"
builder)i in

let asn_e = List.map (fun e -> List.hd (init_val typ e)) expr in

ignore(List.map2 (fun addr e-> L.build_store e addr builder)addr' asn_e);addr))
                       ) in
  let add_local m (local_typ, name,length,e,mark) =
    let local_var =   store_val local_typ name length e mark builder in
     (match local_typ with A.Poly -> StringMap.add name
(local_var,(length+1),mark) m
                        | A.Complex -> StringMap.add name (local_var,2,mark) m
                        | _ -> StringMap.add name (local_var,length,mark) m) in

    let my_formals = function
      A.Prim_f_decl (t, s) -> (t,s)
    | A.Arr_f_decl (t,s) ->(t,s)
    in
  (* Store into the map *)
  let formall = List.map my_formals fdecl.A.formals in
  let locall = List.map type_of_locals fdecl.A.locals in
  let formals  = List.fold_left2 add_formal StringMap.empty formall
(Array.to_list (L.params the_function)) in
  List.fold_left add_local formals locall  in

   (* Lookup name, size and if this variable is initialized *)
  let lookup_name name = (fun (s,_,_)->s)(try StringMap.find name local_vars
                 with Not_found -> StringMap.find name global_vars) (*raise
(Failure ("SBaa"))*)
  and lookup_size name = (fun (_,l,_)->l)(try StringMap.find name local_vars
```

```
                 with Not_found -> StringMap.find name global_vars) (*raise
(Failure ("SBaa"))*)
  and check_init name = (fun (_,_,i)->i)(try StringMap.find name local_vars
                 with Not_found -> StringMap.find name global_vars) (*raise
(Failure ("SBaa"))*)
  in
  let get_expr_type expr = L.type_of (List.hd expr) in

  (* Assign complex, poly and array *)
  let asn_extr_value s index_expr value builder =
    let i  = [|index_expr|]in
    let addr' = L.build_in_bounds_gep (lookup_name s) i "storeArr" builder in
    L.build_store value addr' builder
  in
  let build_addr size s =
     let r = range 0 (size-1) in
     let i  = List.map (fun index -> [|L.const_int i32_t index|]) r in
     List.map (fun i -> L.build_in_bounds_gep (lookup_name s) i "tmp_addr"
builder)i in

    (* Construct code for an expression; return its value *)
    let rec expr builder = function
      A.Asn (ex,e) -> (match ex with A.Extr (s,index_expr) -> let e' =
List.hd(expr builder e) in let index = List.hd(expr builder index_expr) in

[asn_extr_value s index e' builder]
                                   | A.Id s ->  (let size = lookup_size s in let
vals = expr builder e in
                                         (match size with 0 -> [L.build_store
(List.hd vals) (lookup_name s) builder]
                                                        | _ -> let addrs =
build_addr size s in
                                                               List.map2 (fun
value addr -> L.build_store value addr builder)vals addrs))
                                   | _ -> raise(Failure("Invalid asign"))

                       )
     | A.Intlit i -> [L.const_int i32_t i]
     | A.Floatlit f -> [L.const_float  d64_t f]
     | A.Strlit s -> [L.build_global_stringptr (String.sub s 1 ((String.length
s) - 2)) "" builder]
     | A.Boollit b -> [L.const_int i1_t (if b then 1 else 0)]
```

```
      | A.Polylit pl -> (match pl with [] -> [L.const_int i32_t 0]
                               | _ ->List.map (fun e -> List.hd e)(List.map
(expr builder) pl))
    | A.Arrlit al -> List.map (fun e -> List.hd e)(List.map (expr builder) al)
    | A.Id s -> let size = lookup_size s in (match size with 0 -> [L.build_load
(lookup_name s) "PrimValue" builder]
                  | _ ->  let addrs = build_addr size s in List.map (fun addr
-> L.build_load addr "tmp_val" builder) addrs )

    | A.Extr(s,index) -> let e' = List.hd(expr builder index) in
                         let addr = L.build_in_bounds_gep (lookup_name s)
[|e'|] "storeIdx" builder  in
                                                                [L.build_load
addr "tmp" builder]

    | A.Complexlit (e1,e2) -> (* let ee1 = List.hd (expr builder e1) in let ee2
= List.hd (expr builder e2) in
                            [ee1;ee2] *)[L.const_float  d64_t e1;L.const_float
d64_t e2]
    | A.Mod e->
                let real = L.float_of_const (L.build_fmul (List.hd (expr builder
e)) (List.hd (expr builder e))"tmp" builder)in
                let out1 = (match real with None -> raise(Failure("Invalid
complex value1."))
                            | Some v1 -> v1) in
                let image = L.float_of_const (L.build_fmul (List.hd(List.tl
(expr builder e))) (List.hd(List.tl (expr builder e)))"tmp" builder)in
                let out2 = (match image with None -> raise(Failure("Invalid
complex value2."))
                            | Some v2 -> v2) in
                            expr builder (A.Floatlit (sqrt(out1+.out2)))
    | A.Order e -> [L.const_int i32_t ((List.length (expr builder e))-1)]

    | A.Binop (e1, op, e2) ->
    (let e1' = (expr builder e1)
    and e2' = (expr builder e2) in
    let int_op =(match op with
      A.Add     -> L.build_add
    | A.Sub     -> L.build_sub
    | A.Mult    -> L.build_mul
    | A.Div     -> L.build_sdiv
    | A.And     -> L.build_and
```

```
    | A.Or      -> L.build_or
    | A.Equal   -> L.build_icmp L.Icmp.Eq
    | A.Neq     -> L.build_icmp L.Icmp.Ne
    | A.Less    -> L.build_icmp L.Icmp.Slt
    | A.Leq     -> L.build_icmp L.Icmp.Sle
    | A.Greater -> L.build_icmp L.Icmp.Sgt
    | A.Geq     -> L.build_icmp L.Icmp.Sge
    | A.Modu    -> L.build_srem) in
    let float_op =(match op with
      A.Add     -> L.build_fadd
    | A.Sub     -> L.build_fsub
    | A.Mult    -> L.build_fmul
    | A.Div     -> L.build_fdiv
    | A.And     -> L.build_and
    | A.Or      -> L.build_or
    | A.Equal   -> L.build_fcmp L.Fcmp.Oeq
    | A.Neq     -> L.build_fcmp L.Fcmp.One
    | A.Less    -> L.build_fcmp L.Fcmp.Ult
    | A.Leq     -> L.build_fcmp L.Fcmp.Ole
    | A.Greater -> L.build_fcmp L.Fcmp.Ogt
    | A.Geq     -> L.build_fcmp L.Fcmp.Oge
    | A.Modu -> L.build_frem ) in
    let typ1 = get_expr_type e1' and typ2 = get_expr_type e2' in
    let opp = (if typ1 = i32_t then (if typ2= i32_t then int_op else float_op)
                              else float_op) in
    match List.length e1' with 1 -> let x = List.hd e1' in
                               List.map (fun a -> opp x a "tmp" builder)e2'
                          | _ ->(let a1 = List.hd e1'
                                and a2 = List.hd(List.tl e1')
                                and b1 = List.hd e2'
                                and b2 = List.hd(List.tl e2') in
                                match op with A.Mult ->
                                let first = L.build_fsub (opp a1 b1 "tmp"
builder) (opp a2 b2 "tmp" builder)"tmp" builder
                                and second = L.build_fadd (opp a1 b2 "tmp"
builder) (opp a2 b1 "tmp" builder)"tmp" builder in
                                [first;second]
                                            | A.Div ->
                                let molecular1 = L.build_fadd (L.build_fmul
a1 b1 "tmp" builder) (L.build_fmul a2 b2 "tmp" builder)"tmp" builder
                                and molecular2 = L.build_fsub (L.build_fmul
a2 b1 "tmp" builder) (L.build_fmul a1 b2 "tmp" builder)"tmp" builder
```

```
                                    and denominator = L.build_fadd (L.build_fmul
b1 b1 "tmp" builder) (L.build_fmul b2 b2 "tmp" builder) "tmp" builder in
                                    let first = opp molecular1 denominator "tmp"
builder
                                    and second = opp molecular2 denominator
"tmp" builder in
                                    [first;second]
                                                | _ -> List.map2 (fun a b ->opp
a b "tmp" builder)e1' e2'
                                  )
    )
    | A.Unop(op, e) ->let e' = List.hd (expr builder e ) in
                                let var_opt = L.float_of_const e' in
                                let var = (match var_opt with None -> 0.0
                                                          | Some v1 -> v1) in
                                let typ = get_expr_type [e'] in
                                let neg = (if typ = i32_t then (L.build_neg e'
"tmp" builder )else (L.build_fneg e' "tmp" builder)) in
                                let addone = (if typ = i32_t then ((L.build_add
e' (L.const_int i32_t 1)"tmp" builder)) else ((L.build_fadd e' (L.const_float
d64_t 1.0)"tmp" builder)))in
                                let subone = (if typ = i32_t then ((L.build_sub
e' (L.const_int i32_t 1)"tmp" builder)) else ((L.build_fsub e' (L.const_float
d64_t 1.0)"tmp" builder)))in
                                let sqrt = (if var > 0.0 then [List.hd (expr
builder (A.Floatlit (sqrt(var))))(* ;List.hd (expr builder (A.Floatlit
(sqrt(-.var)))) *)] else
                                                [List.hd (expr builder
(A.Floatlit (sqrt(-.var))))]) in
    (match e with A.Id s ->(match op with A.Neg     -> [neg ];
                                              | A.Not     -> [L.build_not e'
"tmp" builder];
                                              | A.Sqrt -> sqrt;
                                              | A.Addone  ->
                                              ignore(L.build_store addone
(lookup_name s) builder);[addone]
                                              | A.Subone  ->
                                              ignore(L.build_store subone
(lookup_name s) builder);[subone]
                                )
              | _ ->(match op with  A.Neg     -> [neg]
                                | A.Not     -> [L.build_not e' "tmp" builder]
```

```
                                | A.Sqrt -> sqrt
                                | A.Addone  -> [addone]
                                | A.Subone  -> [subone]
              ))
    | A.Call ("print", [e]) ->
    let e' = expr builder e in
 ( match List.length( e' ) with 2 -> ( [(L.build_call printf_func_f
        [| image_format_str; (List.hd (List.tl(e'))) |] "printf"
builder);(L.build_call printf_func_f
        [| real_format_str; (List.hd (e')) |] "printf" builder)])
        | _ ->
  ( let format_type =
    let e_type = L.type_of ( List.hd (e')) in
      ( if e_type = i32_t then int_format_str
                        else ( if e_type = d64_t then float_format_str
                                                 else str_format_str ))

                      in  [L.build_call printf_func [| format_type ; (List.hd
(e')) |] "printf" builder] ))
 |  A.Call ("print_n", [e]) -> [L.build_call printf_func_f [| image_format_str;
(List.hd (expr builder e)) |] "printf" builder]
    | A.Call (f, act) ->
         let (fdef, fdecl) = StringMap.find f function_decls in
   let actuals = List.rev (List.map (fun l -> List.hd (expr builder l))
(List.rev act)) in
   let result = (match fdecl.A.ftyp with A.Void -> ""
                                      | _ -> f ^ "_result") in
         [L.build_call fdef (Array.of_list actuals) result builder]
    | A.Noexpr -> [L.const_int i1_t 0]
    in
    (* Assign variables which declared by an expresstion list *)
    let rec generate_zeros_int length typ= if length = 0 then []
                      else (L.const_int i32_t 0) :: generate_zeros_int
(length-1) typ in
    let rec generate_zeros_float length typ= if length = 0 then []
                      else (L.const_float d64_t 0.0) :: generate_zeros_float
(length-1) typ in

    let init t= (match t with  A.Int -> [L.const_int (ltype_of_typ t) 0]
                                | A.Float -> [L.const_float (ltype_of_typ t)
0.0]
                                | A.Bool -> [L.const_int (ltype_of_typ t) 0]
```

```
                                | A.Void -> [L.const_null (ltype_of_typ t)]
                                | A.String -> [L.const_pointer_null
(ltype_of_typ t)]
                                | A.Poly -> [L.const_float (ltype_of_typ t) 0.0]
                                | A.Complex -> [L.const_float (ltype_of_typ t)
0.0;L.const_float (ltype_of_typ t) 0.0])in


    let get_asn_local = function
        A.Primdecl_i  (t,s,e) -> (t,s,0,expr builder e)
      | A.Primdecl    (t,s)      -> (t,s,0,init t)
      | A.Arr_poly_decl      (t,s,length)->(match t with A.Int ->
(t,s,length,(generate_zeros_int (length) t))
                                                       | A.Float ->
(t,s,length,(generate_zeros_float (length) t))
                                                       | A.Poly ->
(t,s,(length+1),(generate_zeros_float (length+1) t))
                                                       | _ ->
raise(Failure("Invalid type"))
                                            )
      | A.Arrdecl_i    (t,s,length,e) -> (let arr_decl = List.map (fun e ->
List.hd e)(List.map (expr builder) (List.rev e))in
                                              (t,s,length,arr_decl))
      | A.Polydecl_i (t,s,length,e) ->  (let poly_decl = List.map (fun e ->
List.hd e)(List.map (expr builder) (e))in
                                              (t,s,(length+1),poly_decl))
      | A.Arr_poly_decl_i (t,s1,length,s2) -> let decl = (expr builder (A.Id
s2)) in
                                              (match t with A.Int ->
(t,s1,length,decl)
                                                       | A.Float ->
(t,s1,length,decl)
                                                       | A.Poly ->
(t,s1,(length+1),decl)
                                                       | _ ->
raise(Failure("Invalid type"))

                                            )
    in
    let asn_local = List.map get_asn_local fdecl.A.locals in
    let asn_val s vals =
      let size = lookup_size s in
```

```
      (match size with 0 -> [L.build_store (List.hd vals) (lookup_name s)
builder]
                     | _ -> let addrs = build_addr size s in
                            List.map2 (fun value addr -> L.build_store value
addr builder)vals addrs
      )
       in
    let asn= function
      (_,s,_,e)->if (check_init s) =2 then ignore(asn_val s e) else () in

     ignore(List.map (asn) asn_local);
     (* Add block terminal *)
    let add_terminal builder f =
      match L.block_terminator (L.insertion_block builder) with
        Some _ -> ()
      | None -> ignore (f builder) in

    (* Build the code for the given statement; return the builder for
       the statement's successor *)
    let rec stmt (builder,break_b) = function
        A.Block sl -> List.fold_left stmt (builder,break_b) sl
      | A.Expr e -> ignore (expr builder e); (builder,break_b)
      | A.Break ->
        ignore(add_terminal builder (L.build_br break_b));
        let new_block = L.append_block context "after.break" the_function in
        let builder = L.builder_at_end context new_block in (builder, break_b)
      | A.Return e -> ignore (match fdecl.A.ftyp with
                  A.Void -> L.build_ret_void builder
                  | _ -> L.build_ret (List.hd (expr builder e)) builder);
(builder,break_b)
      | A.If (predicate, then_stmt, else_stmt) ->
         let bool_val = List.hd (expr builder predicate) in
   let merge_bb = L.append_block context "merge" the_function in

   let then_bb = L.append_block context "then" the_function in
   let b = L.builder_at_end context then_bb in
   let (tmp1,_) = (stmt (b,break_b) then_stmt) in
   add_terminal tmp1 (L.build_br merge_bb);

   let else_bb = L.append_block context "else" the_function in
   let b = L.builder_at_end context else_bb in
   let (tmp1,_) = (stmt (b,break_b) else_stmt) in
```

```
   add_terminal tmp1 (L.build_br merge_bb);

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

      | A.While (predicate, body) ->
    let pred_b = L.append_block context "pred" the_function in
        ignore (L.build_br pred_b builder);
        let body_b = L.append_block context "body" the_function in
        let merge_b = L.append_block context "merge.block" the_function in
        let break_builder = merge_b in
        let b = L.builder_at_end context body_b in
        let (temp1,_)= stmt (b, break_builder) body in
        ignore(add_terminal temp1 (L.build_br pred_b));
        let pred_builder = L.builder_at_end context pred_b in
        let bool_val = match expr pred_builder predicate with p->List.hd p in
        ignore (L.build_cond_br bool_val body_b merge_b pred_builder);
        ((L.builder_at_end context merge_b), break_builder)
    |
    A.For (e1, e2, e3, body) ->
        stmt (builder, break_b)
        ( A.Block [A.Expr e1 ; A.While (e2, A.Block [body ; A.Expr e3]) ] )
    in

  (* Build the code for each statement in the function *)
    let dummy_bb = L.append_block context "dummy.toremove.block" the_function in
    let break_builder = dummy_bb in
    let (builder, _) = (stmt (builder, break_builder) (A.Block fdecl.A.body))
in
    (* Add a return if the last block falls off the end *)
    add_terminal builder (match fdecl.A.ftyp with
        A.Void -> L.build_ret_void
      | A.Int -> L.build_ret (L.const_int i32_t 0)
      | A.Float -> L.build_ret (L.const_float d64_t 0.0)
      | _ -> L.build_ret_void);
    ignore(L.builder_at_end context dummy_bb);
    ignore(L.block_terminator dummy_bb);
    ignore(L.delete_block dummy_bb);
  in

  List.iter build_function_body functiondecl;
  the_module
```

```
```

8.6 Top-level

```
(*
Project:  COMS S4115, PolyGo Compiler
Filename: src/polygo.ml
Authors:   Pu Ke           pk2532
           Jin Zhou        jz2792
           Yanglu Piao     yp2419
           Jianpu Ma       jm4437

Purpose:  * Top-level of the MicroC compiler
          * scan & parse the input, check the resulting AST, generate LLVM IR,
and dump the module
*)

type action =  Ast | LLVM_IR | Compile
let _ =
  let action = if Array.length Sys.argv > 1 then
    List.assoc Sys.argv.(1) [
            ("-a", Ast); (* Use pretty printer *)
            ("-l", LLVM_IR);  (* Generate LLVM, don't check *)
            ("-c", Compile) ] (* Generate, check LLVM IR *)
  else Compile in
  let lexbuf = Lexing.from_channel stdin in
  let ast = Parser.program Scanner.token lexbuf in
  Semant.semant_check ast;
  match action with
    Ast -> print_string (Ast.string_of_program ast)
  | LLVM_IR -> print_string (Llvm.string_of_llmodule (Codegen.translate ast))
  | Compile -> let m = Codegen.translate ast in
    Llvm_analysis.assert_valid_module m;
    print_string (Llvm.string_of_llmodule m)
```

8.7 Top-level Script

```
#!/bin/bash
LLI="/usr/local/opt/llvm38/bin/lli-3.8"
polygo="./polygo.native"
Run (){
if [ $# -eq 1 ]; then
      basename=`echo $1 | sed 's/.*\\///
                              s/.pg//'`
      eval "rm -f ${basename}.ll"
      eval "rm -f ${basename}.out"
      eval "$polygo" "-c" "<" $1 "| tee -a" "${basename}.ll" &&
      eval "$LLI" "${basename}.ll" "| tee -a" "${basename}.out"
elif [ $# -eq 2 ]; then
      basename=`echo $2 | sed 's/.*\\///
                              s/.pg//'`
      eval "rm -f ${basename}.ll"
      eval "rm -f ${basename}.out"
      while getopts ':a:l:c:' flag; do
            case "${flag}" in
                  a) eval "$polygo" "-a" "<" $2;;
                  l) eval "$polygo" "-l" "<" $2 "| tee -a" "${basename}.ll" &&
                        eval "$LLI" "${basename}.ll" "| tee -a"
"${basename}.out";;
                  c) eval "$polygo" "-c" "<" $2 "| tee -a" "${basename}.ll" &&
                    eval "$LLI" "${basename}.ll" "| tee -a" "${basename}.out";;
                *) echo "Input Error. Instruction: to run a PolyGo source code,
try ./polygo [-a|-l|-c] your_code";;
            esac
      done
fi
}

Run $*
if [ $? -eq 1 ] ; then
      echo "Input Error. Instruction: to run a PolyGo source code, try ./polygo
[-a|-l|-c] your_code"
fi
```

8.8 Make file

```
# Project: COMS 4115, Fall 2016, PolyGo
# Author:   Yanglu Piao, Pu Ke, Jin Zhou, Jianpu Ma

# Make sure ocamlbuild can find opam-managed packages: first run
#
# eval `opam config env`

# Easiest way to build: using ocamlbuild, which in turn uses ocamlfind

.PHONY : polygo.native

polygo.native :
	ocamlbuild -use-ocamlfind -pkgs llvm,llvm.analysis -cflags -w,+a-4 \
		polygo.native

# "make clean" removes all generated files

.PHONY : clean
clean :
	ocamlbuild -clean
	rm -rf testall.log *.diff polygo scanner.ml parser.ml parser.mli
	rm -rf *.cmx *.cmi *.cmo *.cmx *.o *.ll *.out

# More detailed: build using ocamlc/ocamlopt + ocamlfind to locate LLVM

OBJS = ast.cmx codegen.cmx parser.cmx scanner.cmx semant.cmx polygo.cmx

polygo : $(OBJS)
	ocamlfind ocamlopt -linkpkg -package llvm -package llvm.analysis $(OBJS)
-o polygo

scanner.ml : scanner.mll
	ocamllex scanner.mll

parser.ml parser.mli : parser.mly
	ocamlyacc parser.mly

%.cmo : %.ml
	ocamlc -c $<
```

```
%.cmi : %.mli
	ocamlc -c $<

%.cmx : %.ml
	ocamlfind ocamlopt -c -package llvm $<

### Generated by "ocamldep *.ml *.mli" after building scanner.ml and parser.ml
ast.cmo :
ast.cmx :
codegen.cmo : ast.cmo
codegen.cmx : ast.cmx
polygo.cmo : semant.cmo scanner.cmo parser.cmi codegen.cmo ast.cmo
polygo.cmx : semant.cmx scanner.cmx parser.cmx codegen.cmx ast.cmx
parser.cmo : ast.cmo parser.cmi
parser.cmx : ast.cmx parser.cmi
scanner.cmo : parser.cmi
scanner.cmx : parser.cmx
semant.cmo : ast.cmo
semant.cmx : ast.cmx
parser.cmi : ast.cmo

# Building the tarball

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

FAILS = assign1 assign2 assign3 dead1 dead2 expr1 expr2 for1 for2     \
    for3 for4 for5 func1 func2 func3 func4 func5 func6 func7 func8    \
    func9 global1 global2 if1 if2 if3 nomain return1 return2 while1   \
    while2

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

TARFILES = ast.ml codegen.ml Makefile polygo.ml parser.mly README scanner.mll \
	semant.ml testall.sh $(TESTFILES:%=tests/%)

polygo-llvm.tar.gz : $(TARFILES)
	cd .. && tar czf polygo-llvm/polygo-llvm.tar.gz \
```

```
            $(TARFILES:%=polygo-llvm/%)
```

8.9 Test Script

```
#!/bin/sh

# Regression testing script for polygo
# 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/llvm38/bin/lli-3.8"
LLI="/usr/local/opt/llvm38/bin/lli-3.8"

# Path to the polygo compiler.  Usually "./polygo.native"
# Try "_build/polygo.native" if ocamlbuild was unable to create a symbolic link.
polygo="./polygo.native"
#polygo="_build/polygo.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] [.pg 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/.pg//'`
    reffile=`echo $1 | sed 's/.pg$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

    echo -n "$basename..."

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

    generatedfiles=""

    generatedfiles="$generatedfiles ${basename}.ll ${basename}.out" &&
    Run "$polygo" "<" $1 ">" "${basename}.ll" &&
    Run "$LLI" "${basename}.ll" ">" "${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/.pg//'`
    reffile=`echo $1 | sed 's/.pg$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

    echo -n "$basename..."

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

```
    generatedfiles=""

    generatedfiles="$generatedfiles ${basename}.err ${basename}.diff" &&
    RunFail "$polygo" "<" $1 "2>" "${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 [ $# -ge 1 ]
then
    files=$@
else
    files="tests/test-*.pg tests/fail-*.pg"
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
```

9 Appendix 2 (Example Code)

9.1 GCD Example

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

```
            }
      }
      return a;
}
int main(){
      print(gcd(11,121));
      return 0;
}
```

9.2 Volume of Lung Example

```
float F(float x)
{
      float a = -0.123 * x * x + 0.362 * x + 0.202;
    return a;
}

float Fd(float x)
{
    return (-0.246 * x + 0.362);
}

float fabs(float x)
{
      float a;
      if( x > 0.0 )
            a = x ;
      else
            a = -x;
      return a;
}

void main()
{
    float x0 = 4.0;
    float h;
    float err = 0.0001;
    float root;
    float x1;
    int miter = 10;
    int iter;
```

```
    float fncvalue;
    float max;
    print("Approximation 4,the max error 0.0001, and the maximum number of
iterations 10");
    iter=1;
    while(iter <= miter)
    {
        h = F(x0)/Fd(x0);    /* calculatinf f(x)/f'(x)as we do in Newton Raphson
method */
        print(iter);
        print(h);
        x1 = x0 - h;   /* x1=x0-f(x)/f'(x) */
        if(fabs(h) < err) /*If 2 approximations is below the max error */
        {
            root = x1;   /*then make the approximation as the root */
            break;
        }
        else
            x0 = x1;
        ++ iter;
    }
      print("maximum is at:");
    print(root);
    max = -0.041 * root * root * root + 0.181 * root * root + 0.202 * root;
    print("maximum value is at");
    print(max);
    return;
}
```

9.3 Car Speed Example

```
float distance( float t )
{
poly [3] s = {8.0, 1.0, 2.0, 1.0};
float d = s[[0]] + s[[1]] * t + s[[2]] * t * t + s[[3]] * t * t;
print("Distance increases in the power of:");
print(order(s));
return d;

}
```

```
float velocity( float t )
{
poly [2] v = {24.0, 2.0, 2.0};
float d = v[[0]] + v[[1]] * t + v[[2]] * t * t;
print("Velocity increases in the power of:");
print(order(v));
return d;
}

float acceleration( float t )
{
poly [1] a = {48.0, 1.0};
float d = a[[0]] + a[[1]] * t;
print("Accelerated speed increases in the power of:");
print(order(a));
return d;
}

int main()
{
float t = 2.0;
float d = distance(t);
float v = velocity(t);
float a = acceleration(t);
print("Time t is:");
print(t);
print("After t seconds, the distance is:");
print (d);
print("After t seconds, the velocity is:");
print (v);
print("After t seconds, the accelerated speed is:");
print (a);
return 0;
}
```

10 Appendix 3 (Test Case)

10.1 fail-assign1.pg

```
int main()
{
      int i;
      i = 3.5; /* Fail: assign a float to an int */
      return 0;
}
```

10.2 fail-assign2.pg

```
int main()
{
      float a;
      a = true; /* Fail: assign a bool to an float */
      return 0;
}
```

10.3 fail-assign3.pg

```
int main()
{
      int [2]a;
      a = 1; /* Fail: assign an int to an int array */
      return 0;
}
```

10.4 fail-assign4.pg

```
int main()
{
      int a = 3.5;      /* illegal assignment float to int in initialization */
      return 0;
}
```

```
```

10.5 fail-assign5.pg

```
int main()
{
      int [2]a = [3];   /* length of initialization of array doesn't match */
      return 0;
}
```

10.6 fail-assign6.pg

```
int main()
{
      int [2]a = [3.5, 5];   /* type of initialization of array doesn't match */
      return 0;
}
```

10.7 fail-assign7.pg

```
int main()
{
      int a = 1;
      float b = a;      /* Fail assign int to float */
      return 0;
}
```

10.8 fail-assign8.pg

```
int main()
{
      int [2]a = [1,2];
      a[[1]] = 2.5;     /* Fail assign float to an element in int array */
      return 0;
}
```

10.9 fail-assign9.pg

```
int main()
```

```
{
      int [2]a = [1,2];
      a = [2.5, 3.5];   /* Fail assign float array to int array */
      return 0;
}
```

10.10 fail-assign10.pg

```
int main()
{
      int [2]a = [1,2,3]; /* length of the initializer doesn't match */
      return 0;
}
```

10.11 fail-assign11.pg

```
int main(){
      int a = 1;
      int b = c;  /* identifier undefined */
      return c;
}
```

10.12 fail-expr1.pg

```
int main()
{
      int a = 1;
      float b = 2.5;
      int c;
      return (a+b);     /* illegal operand int + float */
}
```

10.13 fail-expr2.pg

```
int main()
{
      int a = 1;
      return (a+c);     /* undefined identifier c */
}
```

10.14 fail-expr3.pg

```
int main()
{
	int [2]a;
	float b = 2.5;
	a = [2, 1.5+b]; /* return type of 1.5+b is float,can not be assigned to int array */
	return 0;
}
```

10.15 fail-func1.pg

```
int main()
{
	void a;     /* variable can not be void */
	int b;
	b = 1;
	return 0;
}
```

10.16 fail-func2.pg

```
int add(int a, int b)  /* No main function */
{
	return b;
}
```

10.17 fail-func3.pg

```
int add(void a, int b) /* formal can not be type void */
{
	return b;
}

int main()
{
	int a = 1;
	return 0;
}
```

10.18 fail-func4.pg

```
int main()
{
      int a = 1;
      return 0;
      print(1);   /* Statement after return */
}
```

10.19 fail-func5.pg

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

float add(float a, float b)  /* Duplicate function definition */
{
      return (a+b);
}

int main()
{
      add(1, 2);
      return 0;
}
```

10.20 fail-func6.pg

```
void print(int a) /* Conflict with built-in function */
{
      a = a + 1;
      return 0;
}

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

10.21 fail-func7.pg

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

int main()
{
	int a;
	a = add(1.5, 2);  /* wrong actual type */
	return a;
}
```

10.22 fail-func8.pg

```
int add(int a, int a)  /* duplicate formal */
{
	return a;
}

int main()
{
	int a;
	a = add(1, 2);
	return a;
}
```

10.23 fail-global1.pg

```
int a = 1;
float b = 2.5;

int main(){
	int a = b;  /* assign global float variable b to int */
	return b;
}
```

10.24 fail-global2.pg

```
int a = 1;
void b;     /* void global variable */

int main(){
      int a = 1;
      return a;
}
```

10.25 fail-stmt1.pg

```
int main(){
      int a = 1;
      int b = 0;
      for(a; a+1; ++a){       /* a+1 has type int, can not be used as termination condition */
            ++b;
      }
      return b;
}
```

10.26 fail-stmt2.pg

```
int main(){
      int a = 1;
      break;      /* illegal break, not in a for or while loop */
      return a;
}
```

10.27 fail-stmt3.pg

```
int main(){
      int a = 1;
      int b = 0;
      for(a; a < 5; ++a){
            ++b;
            break;
            break;      /* illegal break statement */
      }
```

```
	return b;
}
```

10.28 fail-stmt4.pg

```
int main(){
	int a = 1;
	int b = 0;
	if(a) {		/* a is not a boolean predicate */
		++b;
	}
	return b;
}
```

10.29 fail-stmt5.pg

```
int main(){
	int a = 1;
	int b = 0;
	while(15) {	/* 25 is not a boolean predicate */
		++a;
	}
	return b;
}
```

10.30 test-array1.pg

```
int main(){
	int[3]a=[1,2,3];
	print(a[[1]]);
	return 0;
}
```

10.31 test-assign1.pg

```
int main(){
	int a = 1;
	int a = 2;
	float f = 1.0;
	float f = 1.2;
```

```
	print(a);
	print(f);
}
```

10.32 test-assign3.pg

```
int main(){
	int i = 1;
	string s = "Hello World!";
	print(i);
	print(s);
	return 0;
}
```

10.33 test-assign4.pg

```
int main(){
	int i = 1;
	int j=i;
	print(j);
	return 0;
}
```

10.34 test-complex1.pg

```
int main(){
	complex c = <1.0,2.0>;
	print(c);
	return 0;
}
```

10.35 test-for1.pg

```
int main(){
	int i = 5;
	for(i;i>0;i=i-1)
	print(i);
	return 0;
}
```

10.36 test-func1.pg

```
int main(){
      int a = 1;
      int b = 2;
      int c = add(1,2);
      print(c);
      return 0;
}
int add(int a,int b){
      return (a+b);
}
```

10.37 test-func2.pg

```
float double(float a){
      return a;
}
int main(){
      float a = 4.0;
      print(double(a));
      return 0;
}
```

10.38 test-global1.pg

```
int a;

int main(){
    a = 10086;
      print(a);
      return 0;
}
```

10.39 test-if1.pg

```
int main(){
      if(1==1)
      print(10086);
      return 0;
```

```
}
```

10.40 test-if2.pg

```
int main(){
      int i;
      i = 1;
      if(false)
      print(i);
      print(i+1);
      return 0;
}
```

10.41 test-mod1.pg

```
int main(){
      print(|<1.0,2.0>|);
      return 0;
}
```

10.42 test-order.pg

```
int main(){
      poly[2]p={1.0,2.0,3.0};
      int i = order(p);
      print(i);
      return 0;
}
```

10.43 test-poly1.pg

```
int main()
{
      poly[2]p={1.0,2.0,3.0};
      poly[2]q=p;
      print(q[[0]]);
      return 0;
}
```

10.44 test-poly2.pg

```
int main(){
      poly[1]p={1.0,(-2.0)};
      int[2]a=[1,-2];
      return 0;
}
```

10.45 test-poly3.pg

```
int main()
{
      poly[2]p={1.0,2.0,3.0};
      poly[2]q=p;
      int i = 2;
      print(q[[i]]);
      return 0;
}
```

10.46 test-poly4.pg

```
int main(){
      poly[2]p={1.0,2.0,3.0};
      poly[2]q={1.0,2.0,3.0};
      float f = p[[1]];
      print(f);
      return 0;
}
```

10.47 test-print1.pg

```
int main()
{
  print(1);
  print(1.1);
  print("Hello World!");
  return 0;
}
```

10.48 test-sqrt.pg

```
int main(){
      print(sqrt(4.0));
      return 0;
}
```

10.49 test-while1.pg

```
int main(){
      int i = 10;
      while(i>5){
            print(i);
            i = --i;
      }
      return 0;
}
```

10.50 test-while2.pg

```
int main(){
      int i = 10;
      while(i>5){
            print(i);
            i = --i;
            break;
      }
      return 0;
}
```