Fly Language

Final Report

Shenlong Gu, Hsiang-Ho Lin, Carolyn Sun, Xin Xu
sg3301, hl2907, cs3101, xx2168

Fly me to the moon
Let me play among the stars
Let me see what spring is like
On Jupiter and Mars
- *Frank Sinatra*

Table of Content

Introduction

The Fly language is a general purpose programming language. Fly draws inspiration from Go (golang) and functional programming languages, with the aim of simplifying the development of network applications and distributed systems. The Fly compiler outputs C++ code, which is then compiled to target executable.

Fly supports similar concurrent programming features in Go such as goroutine, a light-weight thread, and channels, which are synchronized FIFO buffers for communication between light-weight threads. Fly also features asynchronous event-driven programming, type inference and extensive functional programming features such as lambda and closure. Aside from these features, Fly's syntax and semantics resemble those of common imperative languages.

These features allow simplified implementation of various types of distributed network services and parallel computing.

Motivation

In an age of increasing deployment of distributed systems in software industry, it is always challenging to come up with a programming paradigm that fits the nature of distributed systems. When it comes to building distributed systems, a lot of challenges must be taken care of. Developers need to think about the network response model, thread management, concurrency, and the resources shared by different threads. Go language (golang) is well-known for its concurrency primitives that make building network applications simple. However, there are still some features that are missing in Go, such as event-driven and functional programming paradigm, which we believe will significantly empower developers in tackling the challenges in the realm of distributed systems.

Features

- Type inference
- Concurrency primitives to create light-weight threads and synchronized FIFO buffers
- Event-driven primitives for asynchronous network request handling
- Thread-safe container types
- Capability for code to be distributed and executed across systems
- Support for syntax in common functional programming language such as lambda, clojure
- Network Library

Language Tutorial

Getting Started

Inside the Fly project folder, type “make”. This will create the fly compiler named “fly”, which takes a file in fly language and outputs C++ code to both standard output and “tmp.cc” in the same project folder. Then type “make install” (may require sudo privilege) to copy related header files and the fly binary into search path. To generate c++ code, type “fly < <fly_file_name>” (replace <fly_file_name> with your fly language file) and then tmp.cc will be generated. With tmp.cc, type “g++ -pthread -o <exec_name> -std=c++11 tmp.cc” to compile tmp.cc to the target binary file (replace <exec_name> with your binary file name).

Another way to automate the whole process is, inside the project folder, type “make”, “sudo make install” and then type “make bin src=<fly_file_name> bin=<exec_name>”. This will first generate the fly compiler, and then take the fly code named <fly_file_name>, compile it down to the executable <exec_name>.

The following sample Fly code, helloworld.fly, demonstrates the following features:

```
func main () {
    print("Hello World");
    return 0;
}
```

- The mandatory main function, with no parameters and return 0 as success, otherwise as failure.
- Calling the built-in print function

To compile the sample code above, type:

```
> make bin src=helloworld.fly bin=test_bin
```

The output will be:

```
> ./test_bin
Hello World
```

Data Operation

Type Inference

Fly's declaration and assignments work as follows: A variable is declared when it is assigned a value. The value it is assigned to determines its type. Later in the scope the variable lives, it can be assigned to a different value of the same type.

```
func main() {
    a = 1;
    a = 2;
    b = "3";
    c = 4.1;
    d = true;
```

```
    print(a);
    print(b);
    print(c);
    print(d);

    return 0;
}
```

Output:

```
2
3
4.1
1
```

Primitive Types

Fly supports the following primitive types: Int, Float, String, Bool. Their literal expression and assignment usage are demonstrated as follows.

```
3
a = 2;
b = "3";
c = 4.1;
d = true;
```

Container Types

Fly provides thread-safe array and hashmap with parameterizable element types. The usage is similar to what C++ template offers. Thread-safety

means the container guarantees correctness when it is accessed by multiple threads.

Arrays

Arrays are indexed collections of values, the value it contains must belong to the same type. Arrays can contain primitive types, class types or container types (array and hashmap). Note that in the following example, the get_at function's behavior is undefined when the argument is greater than the max valid index.

```
func main() {
    a = @Array<#Int#>;
    a.push_back(1);
    a.push_back(2);
    a.push_back(3);
    a.push_back(4);

    print(a.size());
    print(a.get_at(3));

    return 0;
}
```

Output:

4

4

Hashmaps

Hashmaps are collections of key and value pairs, the pairs it contains must belong to the same types of key and value. The value type in hashmap pairs can be primitive types, class types or container types (array and hashmap). The key type in hashmap pairs can only be primitive types. The behavior is undefined when the key type is anything other than primitive types.

```
func main() {
    m = @Map<#Int, String#>;
    m.insert(324, "Alex");
    m.insert(38948, "Brian");
    m.insert(9238, "Chris");

    for(k: m) {
        print(k);
        print(m.get(k));
    }

    print(m.exist(324));

    m.delete(324);

    for(k: m) {
        print(k);
        print(m.get(k));
    }
```

```
    print(m.exist(324));

    return 0;
}
```

Output:
324
Alex
9238
Chris
38948
Brian
1
9238
Chris
38948
Brian
0

Control Flow

Control flow statements resemble their counterparts in C++.

For Loop

Similar to C++ except we don't need to specify types when declaring variables in initialization thanks to type inference.

```
for (i = 1; i <= 3; i = i + 1) {
      print(i);
}

While Loop

i = 1;
while(i <= 4) {
   i = i + 1;
}
```

If-Else

Fly supports both IF blocks and IF-ELSE blocks. Note that the curly brackets around the statements are mandatory. We believe this is helpful in spotting mistakes such as gotofail, Apple's SSL/TLS bug (https://gotofail.com).

```
i = 1;
   if (i == 1) {
      /* Do something */
   }
```

```
    if (i == 2) {
        /* Do something */
    } else {
        /* Do something */
    }
```

Function Definition

A function accept multiple arguments and returns one value. If the variable accepting the return value is declared before, it must have the same type as the return value. If the variable accepting the return value is first seen in the scope, it will have the same type as the return value, thanks to type inference.

```
func main() {
    a = 1;
    a = get_int();

    return 0;
}

func get_int() {
    return 2;
}
```

Class

A class is user-defined type or data structure. It is declared with keyword class that its name, data and member functions. It resembles what is offered in C++ but much simpler. Everything in a class can be accessed from outside, meaning everything in a class is public. The variable declaration in a class includes the variable name and type. The member function can be defined just as global functions.

```
class cat {
    name:String;
    age:Int;
    func say_sth() {
        print("I'm " + _string(age) + " years old");
    }
    func play_with(c) {
        print(name + " plays with " + c.name);
    }
}

func main () {
    cat1 = @cat;
    cat1.age = 12;
    cat1.name = "Mark";
}
```

Variable Passing

All the primitive types are passed-by-copy, any other types (class, containers, etc.) are passed-by-reference.

In the following example, we first demonstrate that a Int type variable is passed-by-copy to function change_int. So after the call to change_int the value of a remains the same. Then we show that a variable “kitty” with type cat is passed into change_cat_name by reference, so after the call to change_cat_name its name changes. And then another variable named same_cat is assigned by kitty, making same_cat a reference to kitty. So a call to change_cat_name with same_cat passed in will change the name of kitty. Lastly we show that return_same_cat returns a reference to what is passed in, making the variable still_same_cat a reference to kitty.

```
func change_cat_name(c) {
   c.name = "Almighty " + c.name;
}

func change_int(a) {
   a = a + 1;
}

func return_same_cat(c) {
   return c;
}
```

```
func main () {

    a = 1;
    change_int(a);
    print(a);

    kitty = @cat;
    kitty.age = 63;
    kitty.name = "Mark";
    change_cat_name(kitty);
    print(kitty.name);

    same_cat = kitty;
    change_cat_name(same_cat);
    print(kitty.name);

    still_same_cat = return_same_cat(kitty);
    change_cat_name(still_same_cat);
    print(kitty.name);

    return 0;
}
```

Output:

1

Almighty Mark

Almighty Almighty Mark

Almighty Almighty Almighty Mark

Clojure

We support clojure for a function. When a functions takes less than the number of the necessary parameters, it will return a clojure of this function in which lots of parameters are bounded with the local variables.

Usage:

```
func add(a, b) {
        return a + b;
}
a = add(1);
b = a(3);
```

Finally b will be 4, and a is a clojure with the first parameter binded with the value 1.

Lamda

We support a very basic of the lambda (vars -> expr), takes some vars and return a expr (in the expr there maybe some local variables bounded).

Usage:

```
func sort_arr(a, b) {
    l = a.size();
    for(i = 1; i < l; i=i+1) {
        j = i;
        while(j > 0) {
            if(b(a.get_at(j - 1), a.get_at(j))) {
                break;
            }
            else {
                tmp = a.get_at(j - 1);
                a.set_at(j - 1, a.get_at(j));
                a.set_at(j, tmp);
                j = j - 1;
            }
        }
    }
}

func main () {
    b = (x, y -> x > y);
    c = @Array<#Int#>;
    for (i = 0; i < 10; i = i + 1) {
        c.push_back(i);
    }
    sort_arr(c, b);
    print(_string(c));
    return 0;
}
```

We show an sort array example using lambda, and function passing. We get a lambda b taking two arguments, return true if the first is larger then the second, otherwise false. Then we pass the lambda b to the sort function, and as a result, c will be sort in a decrease order and output

9,8,7,6,5,4,3,2,1,0.

Dispatch and Exec

Dispatch and exec keywords provide the ability for the function to be executed in another machine but the code is written in the similar style as the local function call.

Usage:

```
//Client

func add(a, b) {
        return a + b;
}
a = 1;
b = 2;
c = add(a, b);
d = dispatch add(a, b, “localhost”, 5555);

//Server
```

```
func process(msg) {
        res = exec(msg);
        //exec will
        con.send(res);
}
```

The first add call is a local function call, wait for the return value and c = 3, and the second dispatch call add, the function will be executed in a server that listens to the port 5555.

For the server, the exec function will execute a string in a protocol that contains the function code and the data serialization string and get the return string to be sent back to the client.

Fly

In Fly language, fly is a key word meaning “spawn a thread to execute the following function”. From now on we will refer to “fly a function” as “spawn a thread to execute a function”. The semantic is similar to a normal function call except the function is run by a different, newly-created thread.

The following example flies the say_hello function 10 times, with the second argument as index. In the say_hello function, it prints out its index first and then other string and data manipulations. Because there are 10 different threads executing the function, the order of the result is non-deterministic, but each of their corresponding manipulated data must be correct.

```
func main() {

    for( i = 1; i <= 10; i = i+1) {
        fly say_hello("World", i, i*2, i*3);
    }

    while(true) {

    }

    return 0;
}

func say_hello(str, idx, num1, num2) {

    msg = _string(idx) + " Hello " + str;
    sum = num1 + num2;
    msg = msg + " " + _string(sum);

    print(msg);

    return sum;
}
```

Output (non-deterministic):

3 Hello World 15

1 Hello World 5
7 Hello World 35
5 Hello World 25
6 Hello World 30
4 Hello World 20
8 Hello World 40
2 Hello World 10
9 Hello World 45
10 Hello World 50

Signal

Signal is a way for inter-thread communication. A signal is created after flying a function. When the function returns a object, the object will be sent to the function registered for receiving the signal. If no function is registered for receiving the signal, the signal member function wait() works as barrier, which blocks until the flown function returns. The wait() member function also returns the object passed from the flown function. Signal can be used to pass around different types, including primitive types and user-defined class type. Signal can be thought of a thread-safe blocking queue which is enqueued and dequeued only once respectively.

In the following example, two calc functions are flown and s1.wait() + s2.wait() will blocks until the two calc function returns. After they both return, s1.wait() returns the value passed by the first call to calc() and s2.wait() returns the value passed by the second call to calc(). In this example the signal is used for Int type.

```
func calc(a, b) {
   return a + b;
}
func main() {
   s1 = fly calc(1, 2);
   s2 = fly calc(3, 4);
   ans = s1.wait() + s2.wait();
   print(ans);
}
```

In the following example, both send_hello() and say_something() will be running on their own thread apart from the main thread. When send_hello() returns, say_something() will be called with the second argument being “Mark”. Note that the statement “register s say_something(“Hello”);” binds the first argument of say_something to “Hello”. Also note that there’s nothing blocking the main thread after the register statement, so a call to sleep() is placed after the register statement in case the program exits before say_something is executed. In this example the signal is used for String type.

```
func send_hello (name) {
   return name;
}
func say_something(greating, name) {
   print(greating + " " + name);
}
func main() {
   s = fly send_hello("Mark");
```

```
    register s say_something("Hello");
    sleep(1);
    return 0;
}
```

Output:
Hello Mark

Chan

Chan is another way for inter-thread communication. The name Chan is short for Channel. It can be thought of a thread-safe blocking queue. The way Chan is declared is just as a normal class, but with the type information supplied. It can then be passed into different functions so these functions can enqueue and dequeue element inside a shared Chan even when these functions are flown on different threads. Chan can be used to pass around different types, including primitive types and user-defined class type.

In the following example, both push and pop shares the same Chan while they run on different thread. In push(), 10 dog object are generated and then enqueued to Chan using the “<-” operator. In pop(), the “<-” operator is called to dequeue dog objects, the call to “<-” returns only when there’s already elements in the Chan, otherwise it blocks until it successfully dequeue one element from the Chan.

```
class dog {
    name:String;
    age:Int;
    owner:String;
}

func push(c) {
    for(i = 1; i < 10; i = i + 1) {
        d = @dog;
        d.age = i;
        c <- d;
    }
}

func pop(c) {
    while (true) {
        d <- c;
        print("dog of age: " + _string(d.age));
    }
}

func main() {
    c = chan(dog);
    fly push(c);
    fly pop(c);

    sleep(1);
    return 0;
}
```

Output:

```
dog of age: 1
dog of age: 2
dog of age: 3
dog of age: 4
dog of age: 5
dog of age: 6
dog of age: 7
dog of age: 8
dog of age: 9
```

Thread-Safe Containers

The containers Fly provides, Array and Hashmap are thread-safe. The interfaces of these containers are guaranteed to behave atomically, allowing multi-thread writing and reading. When greater granularity of synchronization is needed, the sync() member function can be called to ensure atomicity of multiple statements.

In the following example, the critical section surrounded by the curly braces is executed atomically. The scope of atomic execution is from when the sync() function is called to when the function falls out of scope (if we imagine it to be a variable).

```
arr  = @Array<#Int#>;
arr.push_back(1);
{
   arr.sync();
   v = a.get_at(0);
   arr.set_at(0, v + 1);
}
```

Language Manual

Introduction

Fly draws inspiration from Go (golang), with the aim of simplifying the development of network applications and distributed systems. Fly supports the concurrent programming features in Go such as goroutine, a light-weight thread, and channels, which are synchronized FIFO buffers for communication between light-weight threads. Fly also features asynchronous event-driven programming, type inference and extensive functional programming features such as lambda, pattern matching, map, and fold. Furthermore, Fly allows code to be distributed and executed across systems. These features allow simplified implementation of various types of distributed network services and parallel computing. We will compile fly language to get the AST and transform it to C++ code. We believe that the template, shared_ptr, auto, etc keywords, boost network libraries can make it easy for us to compile our language to the target executable file.

Lexical Conventions

Identifiers

Identifiers in Fly are case-sensitive strings that represent names of different elements such as integer numbers, classes and functions. Identifiers contain a sequence of letters, digits and underscore '_', but should always start with a letter.

Keywords

Keywords are case-sensitive sequence of letters reserved for use in Fly language and cannot be used as identifiers.

Literals

Literals are the source code representation of a value of some primitive types. Literals include integer literals, float literals, character literals, string literals, boolean literals.

Integer literals are sequence of one or more digits in decimal. For representing negative numbers, a negation operator is prefixed.

Example: 12

Float literals consist of an integer part, a decimal point and a fraction part. For representing negative numbers, a negation operator is prefixed.

Example: 3.14159

Character literals consists of an ASCII character quoted by single quotes. Several special characters are represented with a escape sequence using a backslash preceding another character.

Example:

‘a’

‘\\’ (backslash)

‘\”’ (double quote)

‘\’’ (single quote)

‘\n’ (new line)

‘\r’ (carriage return)

‘\t’ (tab)

String literals are double-quoted sequence of ASCII characters. A string can also be empty. Special characters in a string is also represented with escape sequence.

Example:

“”

“I love Fly language”

“Fly makes your program \”fly\””

Boolean literals are true and false. The former represent logical true and the latter is logical false.

Separators

Separators are used in separating tokens. Separators in Fly language include the following:

( ) { } [ ] ; , .

Operators

The Fly language consists of the following operators:

Operator	Name	Associativity
=	Assign	Right
==	Equal to	-
!=	Unequal to	-
>	Greater than	-
>=	Greater than or equal	-
<	Less than	-
<=	Less than or equal to	-
+	Addition	Left
-	Subtraction	Left
*	Multiplication	Left
/	Division	Left
.	Access	Left
^	Exponentiation	Left
%	Modulo	Left
and	Logical AND	Left
or	Logical OR	Left

not	Logical NOT	Right

The precedence of operators is as follows:

.
* / % ^
+ -
> >= < <=
not
and or
== !=
=

Comments

In Fly language, code between ASCII characters /* and */ are regarded as comments and ignored by Fly compiler. This is a multi-line comment convention as in C and C++.

Data Types

Basic Data Types

Name	Description
int	Integer. Range is from -2147483648 to 2147483648
char	Characters in ASCII.

bool	Boolean value. It can take one of two values: true or false
float	Single precision 32-bit floating point number. Range is 3.4E +/- 38 (7 digits)

null	null represents the absence of data
	Ex:
	if item1 == null {
	//statements }

Collection Data Types

Type	Syntax
String A sequence of characters.	String x = “abc”;

List
List stores a sequence of items, not necessarily of the same type.
Use indices and square brackets to access or update the items in the list.
	list1 = [1, 3,1,2];
	print(list1[1:2]);
	list1[3] = 2;

Dict Dictionary maps each *key* to a *value*, and optimizes element lookups.	dict1 = <”John”: 17, “Mary”: 22>; print(dict1[“John”]); dict1[“Sam”] = 20;
Set Set is an unordered collection of	set1 = Set(“a”, “b”, “c”); set1.add(“d”);

unique elements. Elements are enclosed in two dollar signs.	set1.find(“a”);

Concurrency Data Types

Name	Syntax

chan
A synchronized FIFO blocking queue.
	ch = chan();
	ch <- “sa”; //executed in one thread A1
	<- ch;
	/*executed in thread A2, blocked until ch <- “sa” is executed in A1*/

signal A type supporting event-driven programming. When signal is triggered inside the routine of another thread, the callback function being binded will be executed.	s = fly func1(a, b); register s send_back; /* which means after func1(a,b) executed, the result will be sent to the function send_back to be executed */

Keywords

Basic Keywords

Keywords	Syntax

class
Used for class declaration. It is the same as what it is in C++.
	class MyClass{
	a =0;
	b= []; /* lots of assignments*/
	func1(a,b){
	}/* lots of function declaration */
	}

for
The for keyword provides a compact way to iterate over a range of values like what is in C++.
The second version is designed for iteration through collections and arrays.
	for (i = 0; i < n; ++i) {print i;}
	
	for (a: elems) {print a;}

while The while statement allows continual execution of a block of statements while a particular condition is true.	while (a < b) { a++; print a;}
if… else… Allows program to execute a certain section of code, the codes in the brackets, only if a particular test in the parenthesis after the “if” keyword evaluates to true. The curly braces after “if”, “else if” and “else” are mandatory.	if () {} else if {} else {}

/* */ Provides ways to comment codes. The first is "C-style" or "multi-line" comment.	/* comment */
func Used for function declaration. The function name follows the func keyword. The parameters are listed in the parenthesis.	func abc(type, msg) { }

Network and Distribute Keywords

Name	Syntax

fly
A goroutine keyword.
The keyword fly will put the function to be executed in another thread or an event poll to be executed, which means this statement is non-blocking and we won't wait for the function to finish to execute next instructions.
	func add(a, b) {
	return a + b;
	}
	fly add(2, 4);
	add(1,3);
	/* add(2, 4) and add(1,3) will concurrently execute*/

register An event-driven asynchronous keyword.	register s send_back_to_client(server);

We bind a closure with a signal, and when this signal is triggered, the closure is executed asynchronously.	
dispatch A distributed computing keyword. We dispatch a function with parameters to be executed in a machine with ip and port specified. This statement will return a signal much like usage in Fly keyword, we can bind a function for asynchronous execution when the result from func1 is available.	s = dispatch func1(a, b) “192.168.0.10” “8888”; register s func2;
exec Executing a dispatched function from the remote system. The exec keyword is the back-end support for the dispatching protocol, which executes the dispatched function with the parameters.	exec string;

sync
Making operations to a variable synchronized
	sync a;
	fly foo(a);
	fly bar(a);

Expressions

An expression is composed of one of the following:

- One of the literal mentioned in the Literals section
- Set, Map, Array definition
- Lambda expressions
- List comprehensions
- Function calls
- Assign expr
- Unary and binary operations between expressions

The following are some special expressions that Fly language supports:

Name	Syntax
Lambda Expression Anonymous functions, functions without names.	(v1 v2 ... vn -> expression) ex: (x y -> x + y - 1)
Mapping Applying a function to every element in the list, which returns a list.	map(function, list); ex: map((x -> x + 1), [1, 2, 3]);
List Comprehension Creating a list based on existing lists.	[expression | variable <- list] ex: [x + 1 | x <- [1, 2, 3]];
Pattern Matching	match expression with

Defining computation by case analysis.	| $pattern_1$ -> $expression_1$ | $pattern_2$ -> $expression_2$ | $pattern_3$ -> $expression_3$ ex: match i with | 1 -> “One” | 2 -> “Two” | _ -> “More”;
Fold A family of higher order functions that process a data structure in some order and build a return value.	foldr(function, var, list); ex: foldr((x y -> x + y), 5, [1,2,3,4]);

Closure
A record storing a function together with an environment.
	closure1 = function v1 v2 … vn
	ex:
	func sum (a, b) {
	  return a + b;
	}
	sum1 = sum(1);
	sum1(2);

Assignment Expression

When using assignment to copy a variable, there are some data types that are immutable and only allows deep copy (copy the actual object in the memory). There are another group of data types that are mutable so the variable represents the reference to its object in the memory.

Immutable Data Types: int, float, string, bool, char

Mutable Data Types: class, map, set, array, chan, signal

Statements

A statement is a unit of code execution.

Expression Statement

An expression statement is an expression followed by a semicolon. An expression statement causes the expression in the statement to be evaluated.

Declaration Statement

Variables in Fly language follow type inference and the Fly language is statically typed. When declaring variables, some value must be assigned to it.

Example:

```
pi = 3.14;
mylist = [];
```

Control Flow Statement

The if statement is used to execute the block of statements in the if-clause when a specified condition is met. If the specified condition is not met, the statement is skipped over until any of the condition is met. If none of the condition is met, the expressions in the else clause (when specified) will be evaluated.

Example:

```
If (expr) {
        stmt_lists;
```

```
}
else if (expr) {
        stmt_lists;
}
else {
        stmt_lists;
}
```

Loop Statement

The while statement is used to execute a block of code continuously in a loop until the specified condition is no longer met. If the condition is not met upon initially reaching the while loop, the code is never executed. The general structure of a while loop is as follows:

```
While (expr) {
        stmt_lists;
}
for (id : id) {
        stmt_lists;
}
for (expr;expr;expr) {
        stmt_lists;
}
```

Function

Function Definitions

A function definition defines executable code that can be invoked, passing a fixed number of values as parameters. func is the keyword for function definition. func_name is the identifier for the function. The parameters are listed in the parenthesis. The body of the function is in the braces after the parameter list. A typical function definition is shown below:

```
func func_name(parameter1, parameter2, ...){
    stmt_list; /* end with return statement or not (which means a
    void function */
}
```

Parameters of primitive data types: int, float, string, bool, and char are passed by value. Parameters of non-primitive data types: class, map, set, array, chan, and signal are passed by reference.

See Scope for the scope of parameters and local variables.

Calling Functions

A member function is declared as a member of a class. It should only be invoked by an instance of the class in which it is declared, as in

```
val = obj1.func_name1(parameter1, parameter2,...);
```

A static function should be invoked with the class name, without the need for creating an instance of the class, as in

```
val = func_name2(parameter1, parameter2,...);
```

Scope

Scope refers to which variables, functions and classes are accessible at a given point of the program. Broadly speaking, variables can be declared at three places:

1. **Local variables** are declared inside a function or a block. They can be used only inside the function or the block.
2. **Formal parameters** are declared in a function definition. The scope of formal parameters starts at the beginning of the block defining the function, and persists through the function.
3. **Global variables** are declared outside all functions, usually at the top of the program. They are available throughout the entire program.

Variable within its own scope must have consistent type. For example, the following code has syntax error:

```
a = null;

if (/*Statement*/){
    a = 1;
} else {
    a = "abc";
}
```

The If-Else statement is inside the scope of a, but a is assigned with values of different types. The following code is valid, because each a is local to its own block:

```
if (/*Statement*/){
    a = 1;
} else {
    a = "abc";
}
```

Basic Syntax

```
//basic syntax
func gcd(a, b) {
    if (b == 0) {
        return a;
    }
    else if (a < b) {
        return gcd(b, a);
    }
    else {
        return gcd(b, a % b);
    }
}

func main(){
    a = [[3,6], [4, 20], [36,45]];
    b = [gcd(item[0], item[1]) | item <- a];
    print(a);
    print(b);
}
```

Goroutine Syntax

```
//copied goroutine
void produce(a, b) {
    while (true) {
        time.sleep(1);
        a <- 3;
        b.append(1);
    }
}

void consume(a, b) {
    while (true) {
        d <- a;
        b.append(c);
    }
}

func main(){
    a = chan(int);
    b = [];
    sync b;
    fly produce(a, b);
    fly consume(a, b);
    while(true) {

    }
}
```

Network Application

A dispatcher which accepts connection and randomly dispatch computing steps to one of three other machines.

```
random_ip = ["192.168.0.1", "192.168.0.2", "192.168.0.3"];
port = 8000;

//send back msg to client
func send_back_msg(conn, msg) {
    conn.send(msg);
}

func handle_connect(conn) {
    while(true) {
        msg = conn.get();
        if (msg == null) {
            break;
        }
        randn = rand_int(3);
        //dispatch computing to another machine and non-block
        s1 = dispatch deal_msg(msg) randn port;
        //if result is back send back to client
        register s1 send_back_msg(con);
    }
}

func deal_msg(msg) {
    a = 1..10;
    b = [x + 1 | x <- a];
    return json.encode(b);
}

func main(){
    server = net.listen(8000);
    s = fly server.accept();
    //register handle_connect function callback
    register s handle_connect;
    //just hold
    while(true) {
    }
}
```

Project Plan

Planning Process

During the development of the project we set our milestones by different features. We initially start from the basic framework without too much language features: parser, scanner, type inference and codegen. And then we iteratively add features into our language: basic concurrency, class, lambda, closure, containers and inter-thread communications.

Specification Process

Our initial specification is similar to what we stated in our proposal. We have several features we want our language to have and some syntax ideas. Most of our final implementation stays very close to what we came up with at the time. The language reference manual helps us to think more specifically about our syntax and things like operator precedences, and helps us to have a more concrete specification. Later in our development, we do added something we haven't thought of at the beginning and thus we still changed specifications as needed.

Development Process

The initial basic framework started from the front-end to the back-end without adding too much feature that we actually needed. We started with the scanner and parser and then the semantics checker and type inference, then

finally the code generator. Then each features are added iteratively from front-end to back-end.

Testing Process

Throughout the project we always use integration testing. At first we don't have automated testing environment as the basic framework does not have any language features. But every time a new feature is added to our language we write some test cases for the features we have added. As the codebase becomes more complex we then build the automated test environment and put all the previously written test cases into the test suite, in addition to other newly created test cases. Our later stages rely heavily on the automated tests, because it does help us to avoid breaking the previously working code.

Programming Style Guide

For formatting and indents, we used four space for indentation in all files. The "in" keyword is placed at a single line for better readability. Before each function definition we have short comments stating the parameter type, return type and the functionality of the function.

Project Timeline

3/20	Type Inference
3/27	Basic Concurrency
4/03	Class
4/17	Lambda, Closure
4/24	Map, Array
5/01	Chan
5/08	Dispatch

Project Log

3/23	Type Inference
4/01	Basic Concurrency
4/10	Class
4/15	Lambda, Closure
4/24	Map, Array
5/01	Chan
5/08	Dispatch

Team Responsibilities

Carolyn Sun	Testing automation, Debug module, Documentation
Hsiang-Ho Lin	Compiler Front end, Code generation, C++ Library, Test case creation, Documentation
Shenlong Gu	Compiler Front end, Semantics, Code generation, C++ Library, Documentation
Xin Xu	Test case creation, Debug module, Documentation

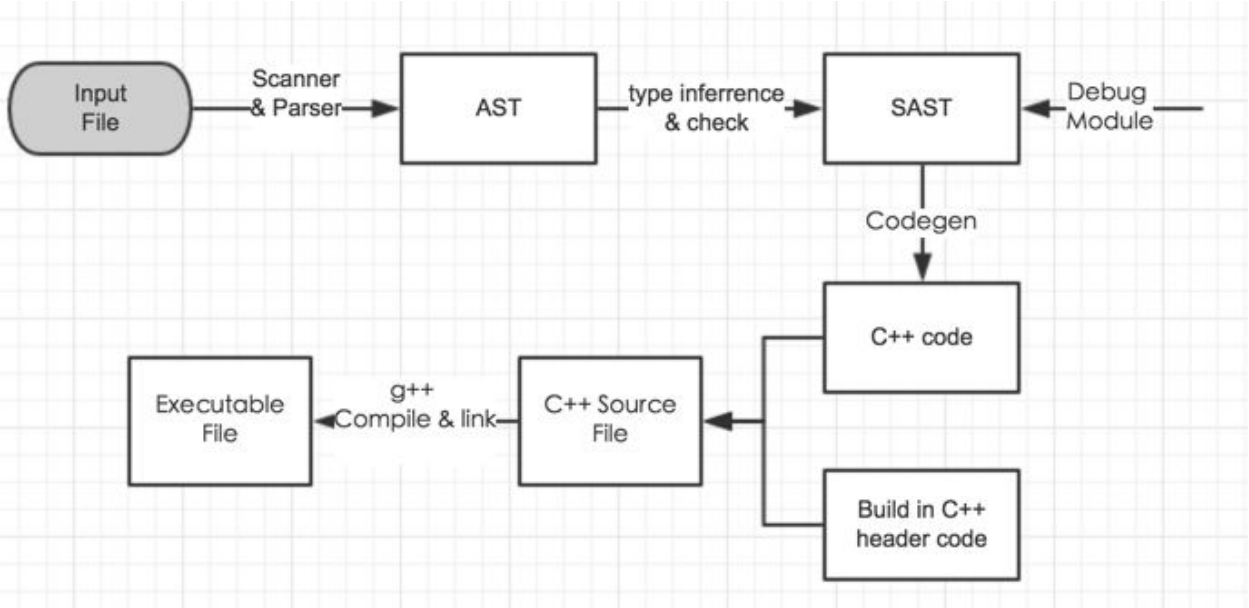
Software Development Environment

We had the following programming and development environment:

Programming language for building compiler : Ocaml version 4.02.3, Ocamlyacc and Ocamllex extensions were used for compiling the scanner and parser front end. GCC version 5.3.0 is for compiling the generated C++ file to binary executable.

Development environments: Different members preferred to code in different linux environments including: Ubuntu and Arch. Text editors include: vim and sublime.

Architectural Design



First, the fly compiler get the input file and use the scanner & Parser module to parse it to get a defined AST.

Then the type inference & check module traverse the AST to make each expression and function typed and finally output a SAST (the debug module prints the content of the SAST used to check the correctness of the type inference & check module.

Then the codegen.ml will accept the input SAST and transform it into C++ code (including c++ class, c++ clojure class, c++ function forward referrence and c++ function definition).

Finally, wrapper the c++ build in header code and the code generated by codegen module, we get the final c++ source file and use g++ to compile and link it to get the finally executable file.

Shenlong Gu implemented part of the scanner & parser module, infer.ml (type inferrence & check module), part of build in functions, part of the codegen module.

Xin Xu implemented part of the debug module, part of build in functions

Carolyn Sun implemented part of the debug module, part of the build in functions.

Hsiang-Ho Lin implemented part of the scanner & parser module, part of the codegen module, part of build in functions.

Test Plan

The test suite contains 48 test cases in total, with 34 positive (success) and 14 negative (fail) test cases. Success tests are used to ensure that compiled programs are correctly operational according to the language specifications. They also ensures that the semantic checker does accept semantically correct programs. Fail tests are used to ensure that the semantic checker can identify semantic errors and prohibits semantically incorrect programs from compiling. New test cases are created as features are added or modified.

Unit testing

The test cases test all aspects of the language; they cover basics syntaxes, including identifier, keywords, literals, operators, and expressions; data types, including simple, collection, and concurrency data types; and language structures, including control flow, function, class, scope, and network application features. Many test cases are designed to be as short and focused as possible, so that each tests a small, specific aspect of the language. Therefore, whenever a test fails, the problem within the language can be can be easily determined.

Integration testing

Longer test cases are used to test more complex programs in order to ensure that both simple operations and fully functional programs work as expected.

Automation

Automation of testing becomes necessary as the project is moving forward. The automation of the testing process is carried out using a python script. The script iterates through all test cases. For each test case, the correspondingly c++ file would be generated, and the expected result for the test case would be compared with the result obtained from executing the c++ program. If the two results match, the test is considered to have passed. Otherwise the test is considered to have failed. A single line is printed to stdout indicating whether the

test has passed or failed. After running all tests in the suite, the number of tests that passed and failed, and the total number of tests would be displayed.

Test suites

Fly to C++ fibonacci2.fly

```
func main() {

   print(fib(12));

   return 0;
}

func fib(n) {

   if (n == 1 || n == 2) {
      return 1;
   }

   return fib(n-1) + fib(n-2);
}
```

C++ output:

```
#include<fly/util.h>
#include<fly/func.h>



#include<fly/class.h>
#include<fly/fly.h>
#include<fly/exec.h>
```

```
class fib_clojure;

class fib_clojure_int;

int main () ;
int fib ( int n) ;
class fib_clojure {
public:

   int call(int n);
};

class fib_clojure_int {
public:

   int __n;
};

int fib_clojure::call(int n)
{
        return fib ( n )  ;
}
int fib ( int n)
{
        if ( ( ( n == 1 )  || ( n == 2 )  )  )
        {
        return 1 ;
        }
```

```
            else
            {
            }
            return ( fib ( ( n - 1 ) ) + fib ( ( n - 2 ) ) ) ;
    }
    int main ()
    {
            print ( fib ( 12 ) ) ;
            return 0 ;
    }
```

chan_dog.fly

```
    func push(c) {
        for(i = 1; i < 10; i = i + 1) {
            d = @dog;
            d.age = i;
            c <- d;
        }
    }

    func pop(c) {
        while (true) {
            d <- c;
            print("dog of age: " + _string(d.age));
```

```
    }
}

func main() {
    c = chan(dog);
    fly push(c);
    fly pop(c);

    sleep(1);
    return 0;
}
```

C++ output:

```
 #include<fly/util.h>


#include<fly/func.h>


#include<fly/class.h>
#include<fly/fly.h>
#include<fly/exec.h>

class dog;
```

```
int main () ;
void push ( shared_ptr <Chan<dog>> c) ;
void pop ( shared_ptr <Chan<dog>> c) ;
class dog;

class dog {
public:

   string name;
   int age;
   string owner;
};

void
push_chan_class_dog_signal_void
( shared_ptr <Chan<dog>> c, shared_ptr <Signal<void>>
push_chan_class_dog_signal_void_sig)
{
push(c);
}
void
pop_chan_class_dog_signal_void
( shared_ptr <Chan<dog>> c, shared_ptr <Signal<void>>
pop_chan_class_dog_signal_void_sig)
{
pop(c);
}
void push ( shared_ptr <Chan<dog>> c)
{
```

```
for ( int i =  1 ; ( i < 10 )  ;  i =  ( i + 1 )  )
{
shared_ptr <dog> d =  shared_ptr <dog>(new dog()) ;
d->age =  i ;
c->push(d) ;
}
}
void pop ( shared_ptr <Chan<dog>> c)
{
while ( true )
{
shared_ptr <dog> d=c->wait_and_pop() ;
print ( ( "dog of age: " + _string ( d->age )  )  )  ;
}
}

int main ()
{
shared_ptr <Chan<dog>> c =  shared_ptr < Chan <dog> >(new Chan < dog
>()) ;
thread( push_chan_class_dog_signal_void , c , shared_ptr <Signal<void>>
(new Signal<void>()) ).detach() ;
thread( pop_chan_class_dog_signal_void , c , shared_ptr <Signal<void>>
(new Signal<void>()) ).detach() ;
sleep ( 1 )  ;
return 0 ;
}
```

Simple_adapt.fly

```
func say_to(a, b) {
    return a + b;
}

func adapt(f, a, b) {
    return f(a, b);
}

func main() {
    res = adapt(say_to, "Hello", " World");

    print(res);

    a = say_to("Hello");
    res = a(" Steve");

    print(res);

    return 0;
}
```

C++ output:

```
 #include<fly/util.h>


#include<fly/func.h>


#include<fly/class.h>
#include<fly/fly.h>
#include<fly/exec.h>

class say_to_clojure_string_string;

class say_to_clojure;

class say_to_clojure_string;

class adapt_clojure;

class adapt_clojure_func_say_to_string_string;

int main () ;
string say_to ( string a, string b) ;
string adapt ( shared_ptr <say_to_clojure> f, string a, string b) ;
class say_to_clojure_string_string {
public:

        string __a;
        string __b;
};
```

```
class say_to_clojure {
public:

	string call(string a,string b);
	shared_ptr <say_to_clojure_string> call(string a);
};

class say_to_clojure_string {
public:

	string __a;
	string call(string b);
};

class adapt_clojure {
public:

	string call(shared_ptr <say_to_clojure> f,string a,string b);
};

class adapt_clojure_func_say_to_string_string {
public:

	shared_ptr <say_to_clojure> __f;
	string __a;
	string __b;
};
```

```
string say_to_clojure::call(string a,string b)
{
return say_to ( a,b )  ;
}
shared_ptr <say_to_clojure_string> say_to_clojure::call(string a)
{
shared_ptr <say_to_clojure_string> tmp = shared_ptr <say_to_clojure_string>(new say_to_clojure_string()) ;
tmp->__a =  a ;
return tmp ;
}
string say_to_clojure_string::call(string b)
{
return say_to ( __a,b )  ;
}
string adapt_clojure::call(shared_ptr <say_to_clojure> f,string a,string b)
{
return adapt ( f,a,b )  ;
}
string say_to ( string a, string b)
{
return ( a + b )  ;
}
string adapt ( shared_ptr <say_to_clojure> f, string a, string b)
{
return f ->call( a,b )  ;
}
int main ()
{
```

```
string res = adapt ( shared_ptr<say_to_clojure>(new say_to_clojure()),"Hello"," World" ) ;
print ( res ) ;
shared_ptr <say_to_clojure_string> a = (shared_ptr<say_to_clojure>(new say_to_clojure())) ->call( "Hello" ) ;
 res = a ->call( " Steve" ) ;
print ( res ) ;
return 0 ;
}
```

Testing Roles

Xin Xu, Carolyn Sun, and Hsiang-Ho Lin created the testing script. Everyone added test cases to the test suite, and reported bugs to the member responsible for the code (Hsiang-Ho Lin or Shenglong Gu), who would then solve the reported error.

Lessons Learned and Advice

Carolyn Sun

Writing code in pairs: you may need to spend several hours investigating an error when working alone; however, having someone else look at it could help you identify the problem within minutes. Testing: use automated testing and don't commit broken code.

Advice: Like everyone says, start early and meet regularly, plan ahead and don't wait until the last minute to do the work.

Hsiang-Ho Lin

Plan source code architecture head. While we are lucky to not need to rewrite basic architecture, planning ahead will still be nice so our code can be more clean and benefit from code-reuse. Automated integration testing is a big part for making sure the modification does not break anything. Because as the code base get larger we will be less confident about the change of code. Automated integration testing assures the correctness to some degree.

Advice: plan ahead and think through the implementation details of different features.

Shenlong Gu

As a team, we need to communicate with each other to decide the features our language need to support. If two teammates think differently on one feature, the code and the interface will not match between components. After deciding what features our language is going to support, We need to carefully plan the system. Because once we design a right architecture, it is easy to add some small features into the system. Third, we need to consider about the test cases to check the correctness. Because some features we support may not be complete, we need to add more robust test cases to check the robustness of our language translator.

Advice:

Start early, because there is lots of things to do. Start early will help you have more time to think about the design of the architecture, more time to code, and finish a good project.

Xin Xu

My role in the project is the manager. Throughout the project, I learned how vital and important it was to have a manager. As the manager, I realized that it was important to keep everyone on track and also around the same place. I also realized that as the manager, if there was a place that required my attention, then I should help the team out by doing what needed to be done to keep everyone on the same page.

Advice:

Start early, plan ahead, and learn OCaml really well from early on

Appendix

scanner.mll

```
{ open Parser }

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"     { comment lexbuf }           (* Comments *)
| '('      { LPAREN }
| ')'      { RPAREN }
| '{'      { LBRACE }
| '}'      { RBRACE }
| '['      { LMBRACE}
| ']'      { RMBRACE} (*add [] for array init*)
| ';'      { SEMI }
```

```
 | ','     { COMMA }
 | '+'     { PLUS }
 | '-'     { MINUS }
 | '*'     { TIMES }
 | '/'     { DIVIDE }
 | '='     { ASSIGN }
 | ':'     { COLON }
 | '.'     { DOT } (*for oop call*)
 | '|'     { VERTICAL} (* for guard *)
 | '$'     { DOLLAR } (* for set initialization *)
 | "set"    { SET } (* for set definition*)
 | "=="    { EQ }
 | "!="    { NEQ }
 | '<'     { LT }
 | "@"     { AT }
 | "<="    { LEQ }
 | ">"     { GT }
 | ">="    { GEQ }
 | "%"     { MOD }
 | "&&"    { AND }
 | "||"    { OR }
 | "!"     { NOT }
 | "<#"    { LJINHAO}
 | "#>"    { RJINHAO}
 | "->"    { RARROW } (* for lambda expression *)
 | "<-"    { LARROW } (* for chan *)
 | "if"    { IF }
 | "else"   { ELSE }
 | "for"    { FOR }
 | "::" {SCOPE}
 | "while"  { WHILE }
 | "return" { RETURN }
 | "break" {BREAK}
 | "continue" {CONTINUE}
 | "true"   { TRUE }
(*add null support*)
 | "null" {NULL}
 | '^' {SADD}
 | "false"  { FALSE }
 | "class"  { CLASS } (*for class initialization *)
 | "func" {FUNC} (*declaration for function*)
 | "map" {MAP} (*declaration for map*)
 | "chan" {CHAN}
 | "fly" {FLY}
```

```
| "register" {REGISTER}
| "dispatch" {DISPATCH}
| ['0'-'9']+ as lxm { LITERAL(int_of_string lxm) }
(* float scan TODO *)
| ['0'-'9']+ '.' ['0'-'9']+ as lxm {FLOAT(float_of_string lxm)}
| ['\"'] [^'\"']* ['\"'] as lxm {STRING(lxm)}
| ['a'-'z' 'A'-'Z' '_']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

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

parser.mly

```
%{
open Ast
%}

%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA
%token PLUS MINUS TIMES DIVIDE ASSIGN NOT
%token EQ NEQ LT LEQ GT GEQ TRUE FALSE AND OR SADD
%token SET MAP AT
%token MOD
%token LJINHAO RJINHAO
%token NULL SCOPE
%token CHAN FLY REGISTER DISPATCH
%token RETURN IF ELSE FOR WHILE
%token BREAK CONTINUE
%token LARROW RARROW VERTICAL LMBRACE RMBRACE FUNC
%token COLON DOT DOLLAR CLASS
%token <int> LITERAL
%token <string> ID
%token <string> STRING
%token <float> FLOAT
%token EOF

%right ASSIGN
%left OR
```

```
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ SADD
%left MOD
%left PLUS MINUS
%left TIMES DIVIDE
%right NOT NEG
%nonassoc UMINUS
%nonassoc NOELSE
%nonassoc ELSE      /* highest precedence */

%start program
%type <Ast.program> program

%%



program:
   /*test without global stmts*/
   stmt_list cdecls fdecls EOF {Program($2, $3)}

fdecls:
   /*nothing*/ {[]}
   | fdecls fdecl {$2 :: $1}

/*because we use type inferrence so we don't have vdecl*/
fdecl:
   FUNC ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE
   { {
   fname = $2;
   formals = $4;
   body = $7 } }




variable_ref:
   ID COLON typedef {
      ($1, $3)
   }


variable_defs_opt:
   /*nothing*/ {[]}
```

```
    | variable_defs {$1}

variable_defs:
    variable_ref SEMI{[$1]}
    | variable_ref SEMI variable_defs {$1 :: $3}

typedef_list:
    typedef {[$1]}
    | typedef_list COMMA typedef {$3::$1}

typedef_list_opt:
    /* nothing*/ {[]}
    | typedef_list {List.rev $1}

/*typedef description*/
typedef:
    ID {
        match $1 with
        | "Int" -> Int
        | "Bool" -> Bool
        | "Void" -> Void
        | "String" -> String
        | "Float" -> Float
        | "Map" | "Set" -> failwith ("set map init must with parameters")
        | x -> Class x
    }
    | ID LJINHAO typedef_list_opt RJINHAO {
        match $1 with
        | "Set" -> begin
                match $3 with
                |[x] -> Set x
                | _ -> failwith ("set just with one parameter")
                end
        | "Map" -> begin
                match $3 with
                | [x;y] -> Map (x,y)
                | _ -> failwith ("map just two parameter")
                end
        | "Array" -> begin
                match $3 with
                |[x] -> Array x
                | _ -> failwith ("array just with one parameter")
                end
        | _ -> failwith ("not suppport template except set map")
```

```
   }
cdecls:
   /* nothing*/ {[]}
   | cdecls cdecl {$2 :: $1}

/* class definition */
cdecl:
   CLASS ID LBRACE variable_defs_opt fdecls RBRACE {
      {
         cname = $2;
         func_decls = $5;
         member_binds = $4
      }
   }

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

/*no type tagged default Undef, we need type inferrence*/

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


/*need semi otherwise expr expr -> shift/reduce conflict*/
stmt_list: /*split otherwise r/r conflict */
   /*nothing*/ {[]} /*cause 60 conflict here*/
   | stmt_true_list {$1}

ctrlblock:
   FOR LPAREN expr SEMI expr SEMI expr RPAREN LBRACE stmt_list RBRACE
     { For($3, $5, $7, $10) }
   | WHILE LPAREN expr RPAREN LBRACE stmt_list RBRACE { While($3, $6) }
   | LBRACE stmt_list RBRACE {Block($2)}
   /*for test without else*/
   | IF LPAREN expr RPAREN LBRACE stmt_list RBRACE {If ($3, $6, [])}
   | IF LPAREN expr RPAREN LBRACE stmt_list RBRACE ELSE LBRACE stmt_list
RBRACE {If ($3, $6, $10)}
   /*for each*/
   | FOR LPAREN ID COLON expr RPAREN LBRACE stmt_list RBRACE
{Foreach($3, $5, $8)}
```

```
stmt_true_list:
   stmt SEMI {[$1]}
   | stmt SEMI stmt_true_list {$1 :: $3}
   | ctrlblock {[$1]}
   | ctrlblock stmt_true_list {$1 :: $2}

stmt:
   expr {Expr($1)}
   | BREAK {Break}
   | CONTINUE {Continue}
   | RETURN expr {Return($2)}

expr_list:
   /* nothing */ {[]}
   | expr_true_list {$1}

expr_true_list:
   expr {[$1]}
   | expr_true_list COMMA expr {$3::$1}

set:
   SET LPAREN expr_list RPAREN {Set(List.rev $3)}

expr_pair_list:
   /*nothing*/ {[]}
   | expr_pair_true_list {$1}

expr_pair_true_list:
   | expr COLON expr {[($1, $3)]}
   | expr_pair_true_list COMMA expr COLON expr {($3,$5)::$1}

map:
   MAP LPAREN expr_pair_list RPAREN {Map(List.rev $3)}



fly:
   /*function_call*/
   FLY ID LPAREN actuals_opt RPAREN {Fly($2, $4)}
   /*oop_function_call*/
   | FLY ID DOT ID LPAREN actuals_opt RPAREN {Flyo($2, $4, $6)}

register:
   /*function_call*/
```

```
   REGISTER ID ID LPAREN actuals_opt RPAREN {Register($2, $3, $5)}

dispatch:
   DISPATCH ID LPAREN actuals_opt RPAREN {
      let arr = List.rev $4
      in match arr with
      | x::y::z -> Dispatch($2, List.rev z, y, x)
      | _ -> failwith ("dispatch param error")
      }

id_list:
   ID {[$1]}
   | ID COMMA id_list {$1::$3}

lambda_expr:
   /*key word undef here*/
   LPAREN id_list RARROW expr RPAREN { Func ($2, $4)}

array:
   LMBRACE expr_list RMBRACE {Array (List.rev $2)}


list_comprehen:
   LMBRACE expr VERTICAL ID LARROW expr RMBRACE { ListComprehen($2,
$4, $6)}

assign_expr:
   /*assign expr*/
   ID ASSIGN expr   { Assign($1, $3) }
mvar_assign_expr:
   /*member assign expr*/
   ID DOT ID ASSIGN expr {MAssign ($1, $3, $5)}

expr:
   /*basic variable and const*/
   /* TODO add float */
   LITERAL { Literal($1) }
   | TRUE { BoolLit(true) }
   | ID SCOPE NULL { Null($1)}
   | FALSE { BoolLit(false) }
   | STRING { String($1) }
   | FLOAT {Float($1)}
   | ID { Id($1)}
   | set {$1} /* set init */
```

```
    | map {$1} /* map init */
    | array {$1} /* array init */
    | lambda_expr {$1} /* lambda init */
    | list_comprehen {$1} /* list comprehension */
    | assign_expr {$1} /* assign expr */
    | mvar_assign_expr {$1} /* member variale assign expr*/
   /*basic operation for expr*/
    | 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 SADD expr {Binop($1, SAdd, $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 MOD expr {Binop($1, Mod, $3)}
    | MINUS expr %prec UMINUS { Unop(Neg, $2) }
    | NOT expr        { Unop(Not, $2) }
   /*function call*/
    | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
   /*class member function call*/
    | ID DOT ID LPAREN actuals_opt RPAREN {ObjCall($1, $3, $5)}
   /*class member get*/
    | ID DOT ID {Objid($1, $3)}
   /*class generation syntax*/
    | AT typedef {ObjGen($2)}
   /*class generation syntax*/
   /*expression is contained with () */
    | LPAREN expr RPAREN { $2 }
   /*network syntax*/
    | chan_decls {$1}
    | chan_op {$1}
    | fly {$1}
    | register {$1}
    | dispatch {$1}

chan_decls:
   CHAN LPAREN typedef RPAREN {Changen($3)}
```

```
chan_op:
   LARROW ID {Chanunop($2)}
   | ID LARROW ID {Chanbinop($1, $3)}

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

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

ast.ml

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

type uop = Neg | Not

(* define for data type *)
type typ =
   Int | Bool | Void | String | Float (*basic type*)
   | Array of typ (*array*)
   | Set of typ (*set*)
   | Map of typ * typ (*map*)
   | Class of string (* a class variable *)
   | Chan of typ (* a chan that contains which type *)
   | Signal of typ (*signal is like a future obj with typ*)
   | Undef (*which means this is a nulptr*)
   | Func of string * typ list (* function name along witH
      some type clojure*)
   | Cfunc of string * string (*class member function cname * fname*)
   | Lfunc of string * typ list (*lambda function along with some type clojure*)
   (*for built-in defned type*)

let uop_to_string = function
   | Neg -> "-"
   | Not -> "!"

let op_to_string = function
   | Add -> "+"
```

```
    | Mod -> "%"
    | Sub -> "-"
    | Mult -> "*"
    | Div -> "/"
    | Equal -> "=="
    | Neq -> "!="
    | Less -> "<"
    | Leq -> "<="
    | Greater -> ">"
    | Geq -> ">="
    | And -> "&&"
    | Or -> "||"
    | RArrow -> "->"
    | LArrow -> "<-"
    | SAdd -> "+"

(* type to string function used for hash
   _ to concat type
   @ to concat different type
*)
let rec type_to_string = function
    | Int -> "int"
    | Bool -> "Bool"
    | Void -> "void"
    | String -> "string"
    | Float -> "float"
    | Array x -> "array_" ^ (type_to_string x)
    | Set x -> "set_" ^ (type_to_string x)
    | Map (x, y) -> "map_" ^ (type_to_string x) ^ "_" ^ (type_to_string y)
    | Class x -> x
    | Chan x -> "chan_" ^ (type_to_string x)
    | Signal x -> "signal:" ^ (type_to_string x)
    | Undef -> "undef"
    | Func (x, type_list) -> "func_" ^ x ^ (List.fold_left
        (fun str item -> str ^ "_" ^ item) "" (List.map type_to_string type_list))
    | Lfunc (x, type_list) -> "lfunc_" ^ x ^ (List.fold_left
        (fun str item -> str ^ "_" ^ item) "" (List.map type_to_string type_list))
    | _ -> raise (Failure ("ast.ml: type_to_string not yet support this type"))


type expr =
   Literal of int
  | BoolLit of bool
  | Null of string (*nullpointer belong to class s*)
```

```
  | Float of float
  | Id of string (* id token *)
  | Objid of string * string
  | Set of expr list
  | Map of (expr * expr) list
  | Array of expr list
  | String of string (*represent const string*)
  | Binop of expr * op * expr
  | Unop of uop * expr
  | Call of string * expr list
  | ObjGen of typ
  | ObjCall of string * string * expr list (*invoke a method of an object*)
  | Func of string list * expr (*lambda expr*)
  | Assign of string * expr
  | MAssign of string * string * expr
  | ListComprehen of expr * string * expr (*can iterate a tuple?*)
  | Noexpr
 (*below are network specified exprs*)
  | Exec of string
  | Dispatch of string * expr list * expr * expr
  | Register of string * string * expr list
  | Changen of typ(*chan of a type*)
  | Chanunop of string
  | Chanbinop of string * string
  | Fly of string * expr list
  | Flyo of string * string * expr list


type stmt =
   Block of stmt list
  | Expr of expr
  | Return of expr
  | If of expr * stmt list * stmt list
  | For of expr * expr * expr * stmt list
  | Foreach of string * expr * stmt list (*for each*)
  | While of expr * stmt list
  | Break
  | Continue
 (*if for while just with list of stmt*)
 (*need to append lambda stmt, lots of built-in keyword stmt,
 like map func list*)


type func_decl = {
```

```
        fname : string; (*function name*)
        formals : string list; (*function paramters*)
        body : stmt list;
    }


type class_decl = {
        cname : string; (* class name *)
        member_binds:  (string * typ) list; (* member variables*)
        func_decls : func_decl list; (* member functions *)
    }

let get_class_name cdecl = match cdecl with
    | {cname=name;_} -> name

let get_class_member_type cdecl var = match cdecl with
    | {member_binds=binds;_} -> List.assoc var binds

type program = Program of class_decl list * func_decl list
```

sast.ml

```
          open Ast
          type texpr =
              TLiteral of int
            | TBoolLit of bool
            | TFloat of float
            | TNull of typ
            | TId of string * typ(* id token *)
            | TSet of texpr list * typ
            | TMap of (texpr * texpr) list * typ
            | TArray of texpr list * typ
            | TString of string(*represent const string*)
            | TBinop of (texpr * op * texpr) * typ
            | TUnop of (uop * texpr) * typ
            | TCall of (string * texpr list) * typ
            | TObjCall of (string * string * texpr list) * typ(*invoke a method of an
object*)
            | TFunc of (string list * texpr) * typ (*lambda expr*)
            | TAssign of (string * texpr) * typ
```

```
  | TMAssign of (string * string * texpr) * typ
  | TListComprehen of (texpr * string * texpr) * typ (*can iterate a tuple?*)
 (*below are network specified exprs*)
  | TExec of string * typ
  | TDispatch of (string * texpr list * texpr * texpr) * typ
  | TRegister of (string * string * texpr list) * typ
  | TChangen of typ * typ
  | TChanunop of string * typ
  | TChanbinop of (string * string) * typ
  | TFly of (string * texpr list) * typ
  | TFlyo of (string * string * texpr list) * typ
  | TObjGen of typ * typ
  | TObjid of (string * string) * typ

let get_expr_type_info tepr = match tepr with
    | TLiteral _ -> Int
    | TBoolLit _ -> Bool
    | TFloat _ -> Float
    | TNull x -> x (*nullpointer now*)
    | TString _ -> String
    | TId (_, x)  -> x
    | TSet (_, x) -> x
    | TMap (_, x) -> x
    | TArray (_, x) -> x
    | TBinop (_, x) -> x
    | TUnop (_, x) -> x
    | TCall (_, x) -> x
    | TObjCall (_, x) -> x
    | TFunc (_, x) -> x
    | TAssign (_, x) -> x
    | TListComprehen (_, x) -> x
    | TExec (_, x) -> x
    | TDispatch (_, x) -> x
    | TRegister (_, x) -> x
    | TChangen (_, x) -> x
    | TChanunop (_, x) -> x
    | TChanbinop (_, x) -> x
    | TFly (_, x) -> x
    | TFlyo (_, x) -> x
    | TObjGen (_, x) -> x
    | TObjid (_, x) -> x
    | TMAssign (_, x) -> x

type tstmt =
```

```
    TBlock of tstmt list
  | TExpr of texpr
  | TReturn of texpr
  | TIf of texpr * tstmt list * tstmt list
  | TFor of texpr * texpr * texpr * tstmt list
  | TForeach of string * texpr * tstmt list (*for each*)
  | TWhile of texpr * tstmt list
  | TBreak
  | TContinue



(* this is for lambda decl, with type information*)
type t_lambda_decl = {
        ltkey: string; (*for matching*)
        ltfname: string; (* random hash *)
        ltbinds: (string * typ) list;
        ltformals : (string * typ) list;
        ltbody: tstmt list;
        ltret: typ (* the return value*)
    }

type t_func_decl = {
        ttkey: string; (* for matching*)
        tfname: string;
        tformals: (string * typ) list;
        tbody: tstmt list;
        tret: typ (*the return value type*)
    }


(*just raw t_fdecl*)
let new_null_tfdecl() =
    {
        ttkey="";
        tfname="";
        tformals=[];
        tbody=[];
        tret=Undef;
    }

(*raw tfdecl with type*)
let new_raw_type_tfdecl thistype =
    {
```

```
        ttkey="";
        tfname="";
        tformals=[];
        tbody=[];
        tret=thistype;
    }

let compare_and_update tfdecl thistype =
    match tfdecl with
    | {ttkey=a;tfname=b;tformals=c;tbody=d;tret=rtype;}->
        begin match rtype with
        | Undef ->
            {ttkey=a;tfname=b;tformals=c;tbody=d;tret=thistype}
        | x -> if x = thistype then tfdecl
                else failwith ("return with different type")
        end


let get_func_result tfdecl = match tfdecl with
    | {tret=rtype;_} -> rtype

let check_bool this_type =
    if this_type = Bool then ()
    else failwith ("check bool error")

(*from a stmts list get a return stmt and get the return type*)
let rec get_rtype stmt_list = match stmt_list with
    | [] -> Void (*no return stmts just return void*)
    | (TReturn x::y) -> get_expr_type_info x
    | (x :: y) -> get_rtype y


type t_class_decl = {
        tcname: string;
        member_binds: (string * typ) list;
        t_func_decls: t_func_decl list;
        (* member functions with overloading records*)
    }

(* debug code for sast*)
```

checkstruct.ml

```
open Ast
let rec br_con_check ast =
   let rec check_stmt stmt pstmt = match stmt with
       | Block stmt_list -> List.iter (fun item -> check_stmt item pstmt) stmt_list
       | Expr expr -> ()
       | Return expr -> ()
       | If (a, b, c) -> List.iter (fun item -> check_stmt item pstmt) b;List.iter (fun
item -> check_stmt item pstmt) c
       | For (a, b, c, d) -> List.iter (fun item -> check_stmt item true) d
       | Foreach (a, b, c) -> List.iter (fun item -> check_stmt item true) c
       | While (a, b) -> List.iter (fun item -> check_stmt item true) b
       | Break -> if pstmt then () else failwith("break must be inside for/while")
       | Continue -> if pstmt then () else failwith("continue must be inside
for/while")
   in
   let rec check_fdecl fdecl = match fdecl with
       | {body=stmt_list;}->
         List.iter (fun item -> check_stmt item false) stmt_list
   in
   match ast with
   | Program (cdecl_list, fdecl_list) ->
     List.iter check_fdecl fdecl_list
```

env.ml

```
open Ast
open Sast
(* create a new env*)
let get_new_env() =
   let (env : (string, typ) Hashtbl.t) = Hashtbl.create 16
   in env

(* a multi-layer env operation *)
let init_level_env () =
   [get_new_env()]

(* append a new level env to level_env*)
let append_new_level level_env =
   get_new_env() :: level_env
```

```
let update_env level_env k v = match level_env with
    | (this_level :: arr) -> Hashtbl.add this_level k v;this_level :: arr
    | _ -> failwith ("no env internal error")

let rec search_id level_env k = match level_env with
    | [] -> failwith ("variable refered without defined" ^ k)
    | (this_level :: arr) ->
       try
          Hashtbl.find this_level k
       with
       | Not_found -> search_id arr k

let search_key level_env k =
    try
       Some (search_id level_env k)
    with
       | _ -> None

let search_key2 level_env k =
    try
       ignore(Some (search_id level_env k));
       print_string (k ^ " yes;")
    with
       | _ -> print_string (k ^ " no;")

let back_level level_env = match level_env with
    | [] -> failwith ("no level to be back")
    | (this_level :: arr) -> arr

(*debug a level env, just print out to the screeen*)
let debug_level_env level_env =
    let rec inner_debug level_env cnt = match level_env with
       | [] -> ()
       | (this_level :: arr) -> print_endline ("this level: " ^ (string_of_int cnt));
    in
    inner_debug level_env 0

let rec level_env_copy level_env = match level_env with
    | [] -> []
    | (x::y) -> Hashtbl.copy x :: level_env_copy y
```

util.ml

```
(*code for generate random unique string*)
open Ast
let global_cnt = ref(0)

let rec generate_str num =
   if num < 26 then String.make 1 (Char.chr (num + 97))
   else
      let extra = (num mod 26)
      in let this_pos = String.make 1 (Char.chr (extra + 97))
      in this_pos ^ (generate_str (num / 26))

let next_random_string () =
   let nxt = (!global_cnt) + 1
   in global_cnt := nxt;
   generate_str nxt


let gen_hash_key fname type_list =
   fname ^ (List.fold_left
      (fun str item -> str ^ "_" ^ item) "" (List.map type_to_string type_list))

let rec explode = function
   "" -> []
   | s  -> (String.get s 0) :: explode (String.sub s 1 ((String.length s) - 1));;

let tablize arr =
   List.map (fun item -> "\t" ^ item) arr

let list_join string_arr join_string = match string_arr with
   | (x::y) ->
      x ^ (List.fold_left (fun res item -> res ^ join_string ^ item) "" y)
   | [] -> ""

let rec zip arr1 arr2 = match arr1, arr2 with
   | [], _ -> []
   | _, [] -> []
   | (x1::y1), (x2::y2) -> (x1, x2) :: zip y1 y2

let rec drop_first arr n = match arr, n with
   | _, 0 -> arr
   | (x::y), _ ->  drop_first y (n - 1)
   | _, _ -> failwith ("drop_first error not enought arr elements")
```

infer.ml

```
(*infer type and do a static syntax checking*)
open Ast
open Sast
open Util
open Debug
open Env
open Buildin
open Checkstruct
let (func_binds : (string, func_decl) Hashtbl.t) = Hashtbl.create 16
(*it is the class_binds*)
let (class_binds : (string, class_decl) Hashtbl.t) = Hashtbl.create 16

(*typed function bindings*)
let (t_func_binds: (string, t_func_decl) Hashtbl.t) = Hashtbl.create 16

(*typed class definition*)
let (t_class_binds: (string, t_class_decl) Hashtbl.t) = Hashtbl.create 16

(*clojure call binds*)
let (clojure_calls: (string, (typ list * typ list) list) Hashtbl.t) = Hashtbl.create 16



let rec get_or_create funcname =
   try
      Hashtbl.find clojure_calls funcname
   with
   | Not_found -> Hashtbl.add clojure_calls funcname []; get_or_create
funcname

let rec update_if_no exist_calls f_type_list s_type_list =
   match exist_calls with
   | [] -> [(f_type_list, s_type_list)]
   | (x::y) -> if x = (f_type_list, s_type_list) then (x::y) else
      x :: (update_if_no y f_type_list s_type_list)

let update_clojure_calls funcname f_type_list s_type_list =
   let exist_calls = get_or_create funcname
   in let new_calls = update_if_no exist_calls f_type_list s_type_list
   in Hashtbl.replace clojure_calls funcname new_calls
```

```
let find_t_func name =
   try
      Some (Hashtbl.find t_func_binds name)
   with
   | Not_found -> None


let find_func name =
   try
      Hashtbl.find func_binds name
   with
   | Not_found -> failwith ("not this function:" ^ name)


let find_class name =
   try
      Hashtbl.find class_binds name
   with
   | Not_found -> failwith ("not this class:" ^ name)


let find_cfunc cname fname =
   let cdecl = find_class cname
   in match cdecl with
      | {cname=name;member_binds=binds;func_decls=fdecls} ->
        let rec search_func_by_name fdecls fname = begin
            match fdecls with
            | [] -> None
            | (x::y) -> begin match x with
                | {fname=thisfname;_} -> if thisfname = fname then Some x else
search_func_by_name y fname
            end
            end
        in search_func_by_name fdecls fname

let find_t_class name =
   try
      Hashtbl.find t_class_binds name
   with
   | Not_found -> failwith ("not this class:" ^ name)

let init_tclass () =
```

```
    let f k v = match v with
        | {cname=name;member_binds=binds;_} ->
          Hashtbl.add t_class_binds k
{tcname=name;member_binds=binds;t_func_decls=[]}
    in Hashtbl.iter f class_binds

let update_tclass tcdecl  tfdecl =
    match tcdecl with
    | {tcname=name;member_binds=binds;t_func_decls=tfdecls} ->
       {tcname=name;member_binds=binds;t_func_decls=tfdecl::tfdecls}

let rec search_hash_key tfdecls hash_key = match tfdecls with
    | [] -> None
    | (x :: y) -> begin match x with
       | {ttkey=this_hash_key;_} -> if this_hash_key = hash_key then Some x else
search_hash_key y hash_key
       end

let find_t_mfunc cname hashkey =
    let tclass = find_t_class cname
    in match tclass with
       | {tcname=name;member_binds=binds;t_func_decls=tfdecls} ->
          search_hash_key tfdecls hashkey

let check_non_exist_func name =
    try
       ignore(Hashtbl.find func_binds name);
       failwith ("exist this bind func:" ^ name)
    with
    | Not_found -> ()

let check_non_exist_class name =
    try
       ignore(Hashtbl.find class_binds name);
       failwith ("exist this bind class:" ^ name)
    with
    | Not_found -> ()

let check_not_void thistype = match thistype with
    | Void -> failwith ("void error")
    | _ -> ()

(*create a copy of the function env*)
let func_level_env () =
```

```
    let (env : (string, typ) Hashtbl.t) = Hashtbl.create 16
    in let f k v = match v with
        | {fname=name;_} -> Hashtbl.add env k (Func(name,[]))
    in Hashtbl.iter f func_binds;
    env


(* debug the global function *)
let debug_t_func_binds () =
    let f x y = print_endline x;
        match y with
        | {tret=rtype;_} -> print_endline ("return type:" ^ (type_to_string rtype));
    in
    (* print out all the hash key*)
    Hashtbl.iter f t_func_binds

let check_return_of_func func_decl =
    let rec check_return_of_stmts = function
        | [Return (_)] -> ()
        | [] -> ()
        | (Return (_) :: arr) -> failwith("return is not the last statement")
        | (x :: arr) ->
          begin
             match x with
             | If (_, f_branch, s_branch) ->
                check_return_of_stmts f_branch;
                check_return_of_stmts s_branch;
             | While (_, branch) ->
                check_return_of_stmts branch;
             | For (_, _, _, branch) ->
                check_return_of_stmts branch;
             | Foreach (_, _, branch) ->
                check_return_of_stmts branch;
             | _ -> ();
             ;
             check_return_of_stmts arr
          end
    in
    match func_decl with
        | {body = stmts;_} ->
          check_return_of_stmts stmts

(* get a name to func_decl binding of *)
(*meanwhile check the name appear once in the binds*)
```

```
let bind_name (ast : program) = match ast with
    | Program (cdecl_list, fdecl_list) ->
       begin
       List.iter (fun fdecl ->
          ignore(check_return_of_func fdecl);
          match fdecl with
          | {fname = name ; _} ->
             check_non_exist_func name;
             check_non_exist_class name;
             Hashtbl.add func_binds name fdecl;
             ) fdecl_list;
       List.iter (fun cdecl ->
          match cdecl with
             | {cname=name;_} ->
                check_non_exist_func name;
                check_non_exist_class name;
                Hashtbl.add class_binds name cdecl;
                ) cdecl_list;
       end

(*just add to function bind for semantic check not codegen*)
let add_build_in_func () =
    let print_func = {fname="print";formals=["a"];body=[]}
    in let build_in_funcs = [print_func]
    in List.iter (
       fun item -> begin
          match item with
          | {fname=name;_} -> Hashtbl.add func_binds name item
       end
       ) build_in_funcs

let check_type_same type_list cmp_type=
    List.iter (fun item -> if cmp_type = item then () else failwith ("type is not
same")) type_list

let check_non_empty = function
    | [] -> failwith ("empty error")
    | (x :: y) -> ()

let check_type_in_arr this_type check_type_list =
    List.exists (fun item -> this_type = item) check_type_list

let rec check_valid_type thistype = match thistype with
    | Class x ->
```

```
        begin
        if check_build_in_class x then true
        else try
            let _ = find_class x
            in true
        with _ -> false
        end
    | Array x -> check_valid_type x
    | Map (x, y) -> check_valid_type x && check_valid_type y
    | Set x -> check_valid_type x
    | _ -> true

(*check no null*)
(* we don't permit type null to be get evaluated*)
(*T/F*)
let is_null = function
    | Undef -> false
    | _ -> true


(*function from a string to type*)
let string_to_type s = match s with
    | "Int" -> Int
    | "Bool" -> Bool
    | "String" -> String
    | "Float" -> Float
    | _ -> failwith("this type not support")

(* infer the function result given input params*)
(*let rec infer fdecl env *)
(* return a t_func_decl can be inside a class*)
let rec infer_func fdecl hash_key type_list level_env =
   (*level_env is an ref variable modified by infer_expr and infer_stmt*)
   let ref_create_env ()=
      level_env := append_new_level (!level_env)
   in
   let ref_update_env k v =
      level_env := update_env (!level_env) k v
   in
   let ref_search_id k =
      try
         Some (search_id (!level_env) k)
      with
       | _ -> None
```

```
in
let ref_back_env ()=
   level_env := back_level (!level_env)
in
(*return a texpr*)
let rec infer_expr epr = match epr with
    | Literal x -> TLiteral x
    | BoolLit x -> TBoolLit x
    | Null x -> TNull (string_to_type x)
    | Id (a) -> TId (a, search_id (!level_env) a)
    | Float x -> TFloat x
    | Set (expr_list) ->
      let texpr_list =
         List.map (fun item -> infer_expr item) expr_list
      in
      let expr_types = List.map get_expr_type_info texpr_list
      in
      begin
         match expr_types with
            | [] -> TSet (texpr_list,Undef)
            | (x :: y) ->
               check_type_same expr_types x;
               TSet (texpr_list, x)
      end
    | Map (expr_pair_list) ->
      let texpr_pair_list =
         List.map (fun (litem, ritem) -> (infer_expr litem,
            infer_expr ritem)) expr_pair_list
      in let expr_k_types = List.map
               (fun item -> get_expr_type_info (fst item)) texpr_pair_list
         and expr_v_types = List.map
               (fun item -> get_expr_type_info (snd item)) texpr_pair_list
      in
      begin
         match expr_k_types, expr_v_types with
            | [], _ -> TMap (texpr_pair_list, Map (Undef, Undef))
            | _, [] -> TMap (texpr_pair_list, Map (Undef, Undef))
            | (x1 :: y1), (x2 :: y2) ->
               check_type_same expr_k_types x1;
               check_type_same expr_v_types x2;
               TMap (texpr_pair_list, Map(x1, x2))
      end
    | Array (expr_list) ->
      let texpr_list =
```

```
        List.map (fun item -> infer_expr item) expr_list
      in
      let expr_types = List.map get_expr_type_info texpr_list
      in
      begin
        match expr_types with
          | [] -> TArray (texpr_list, Undef)
          | (x :: y) ->
            check_type_same expr_types x;
            TArray (texpr_list, x)
        end
  | String (str) -> TString (str)
  | Binop (f_expr, bop, s_expr) ->
      let
        t_f_expr = infer_expr f_expr
        and t_s_expr = infer_expr s_expr
      in
      let f_expr_type = get_expr_type_info t_f_expr
        and s_expr_type = get_expr_type_info t_s_expr
      in
      begin
        match bop with
        | Add -> begin
          match f_expr_type, s_expr_type with
          | Int, Int -> TBinop ((t_f_expr, bop, t_s_expr), Int)
          | Float, Float -> TBinop ((t_f_expr, bop, t_s_expr), Float)
          | Float, Int -> TBinop ((t_f_expr, bop, t_s_expr), Float)
          | Int, Float -> TBinop ((t_f_expr, bop, t_s_expr), Float)
          | String, String -> TBinop ((t_f_expr, bop, t_s_expr), String)
          | _, _ -> failwith ("wrong type binop with each other")
          end
        | Sub | Mult | Div ->  begin
          match f_expr_type, s_expr_type with
          | Int, Int -> TBinop ((t_f_expr, bop, t_s_expr), Int)
          | Float, Float -> TBinop ((t_f_expr, bop, t_s_expr), Float)
          | Float, Int -> TBinop ((t_f_expr, bop, t_s_expr), Float)
          | Int, Float -> TBinop ((t_f_expr, bop, t_s_expr), Float)
          | _, _ -> failwith ("wrong type binop with each other")
          end
        | Equal | Neq | Less | Leq | Greater | Geq -> begin
          match f_expr_type, s_expr_type with
          | Int, Int
          | Float, Float
          | Float, Int
```

```
            | Int, Float
            | String, String
              -> TBinop ((t_f_expr, bop, t_s_expr), Bool)
            | _, _ -> failwith ("wrong type binop with each other")
          end
          | And | Or ->
            begin
            match f_expr_type, s_expr_type with
            | Bool, Bool -> TBinop ((t_f_expr, bop, t_s_expr), Bool)
            | _, _ -> failwith ("wrong type binop with each other")
            end
          | SAdd ->
            begin
            match f_expr_type, s_expr_type with
            | String, String -> TBinop ((t_f_expr, bop, t_s_expr), String)
            | _, _ -> failwith ("wrong type binop with each other")
            end
            (*TODO chan operation*)
          | Mod ->
            begin
            match f_expr_type, s_expr_type  with
            | Int, Int -> TBinop ((t_f_expr, bop, t_s_expr), Int)
            | _, _ -> failwith ("wrong type binop with each other")
            end
          | _ -> failwith ("chan not implemented,
                undefined binop for binop -> <-")
        end
      | Assign (varname, epr) ->
        let tepr = infer_expr epr
        in let expr_type = get_expr_type_info tepr
        in check_not_void expr_type;let var = ref_search_id varname
        in  begin
          match var with
          | None -> ref_update_env varname expr_type;
            TAssign ((varname, tepr), expr_type)
          | Some x -> if expr_type = x then
            TAssign ((varname, tepr), expr_type)
            else
            failwith ("redefine " ^ varname ^ " with different type")
          end
      | MAssign (varname, mname, epr) ->
        let tepr = infer_expr epr
        in let expr_type = get_expr_type_info tepr
        in check_not_void expr_type;let var = ref_search_id varname
```

```
          in begin match var with
              | Some (Class cname) ->
                let cdecl = find_class cname
                in let mvartype = get_class_member_type cdecl mname
                in if mvartype == expr_type then TMAssign ((varname, mname,tepr),
mvartype)
                else failwith ("type not consistent in the obj assign:" ^ (type_to_string
mvartype) ^ "," ^ (type_to_string expr_type))
              | None -> failwith ("class obj meber assign without init obj")
              | _ -> failwith (mname ^ " not exist in the class: " ^ varname)
             end
       | Unop (unop, epr) ->
          let tepr = infer_expr epr
          in let expr_type = get_expr_type_info tepr
          in  begin
             match unop with
             | Not -> if expr_type != Bool then failwith ("not with not bool") else
                TUnop ((unop, tepr),Bool)
             | Neg -> if expr_type != Int && expr_type != Float then failwith ("neg
with not int or float")
                   else  TUnop ((unop, tepr), expr_type)
             end
       | Func (param_list, epr) ->
          (*lambda expression*)
          (*we don't evaluate the expression '
          but get the bindings and create new fdecl*)
          let rec get_inner_bindings epr =
             match epr with
             | Id a -> if List.mem a param_list then []
                     else
                      let a_bind = ref_search_id a
                       in begin
                       match a_bind with
                       | Some x -> [(a,x)]
                       | None -> []
                       end
             | Set expr_list ->
                List.concat (List.map get_inner_bindings expr_list)
             | Map expr_pair_list ->
                let expr_single_list = List.fold_left (fun arr (item1, item2) ->
item2::item1::arr) [] expr_pair_list
                in List.concat (List.map get_inner_bindings expr_single_list)
             | Array expr_list ->
                List.concat (List.map get_inner_bindings expr_list)
```

```
            | Binop (f_expr, thisop, s_expr) ->
              let left_binds = get_inner_bindings f_expr
              and right_binds = get_inner_bindings s_expr
              in List.append left_binds right_binds
            | Unop (thisnop, thisexpr) ->
              get_inner_bindings thisexpr
            | Call (name, expr_list) ->
              if List.mem name param_list then []
              else
                 let name_bind = ref_search_id name
                 in  begin
                 match name_bind with
                  | None -> List.concat (List.map get_inner_bindings expr_list)
                  | Some x -> (name, x) :: (List.concat (List.map get_inner_bindings
expr_list))
                 end
            | _ -> []
        in
        let inner_params_binds = get_inner_bindings epr
        in
        let inner_param_name_binds = List.map (fun (name, thistype) -> name)
inner_params_binds
        in
        let inner_param_type_binds = List.map (fun (name, thistype) -> thistype)
inner_params_binds
        in
        let new_lambda_name =
           "_" ^ next_random_string()
        in
        let new_lambda_func =
           {
              fname=new_lambda_name;
              (*lambda function __begin not to shadow*)
              formals=(List.append inner_param_name_binds param_list);
              body=[Return epr]
           }
        in
        (*add this func to func_binds*)
        Hashtbl.add func_binds new_lambda_name new_lambda_func;
        let texpr_list = List.map (fun (name, thistype) -> TId (name, thistype))
inner_params_binds
        (*replace with a clojure call*)
        and type_list = List.map (fun (name, thistype) -> thistype)
inner_params_binds
```

```
          in (* update to clojure call binds*)
          update_clojure_calls new_lambda_name [] inner_param_type_binds;
          TCall ((new_lambda_name, texpr_list), Func (new_lambda_name,
type_list))
        | Fly (name, expr_list) ->
          let ftype = ref_search_id name
          in
             begin
             match ftype with
             | None -> failwith("unknow refer" ^ name)
             | Some (Func (fname, arr)) -> (*with*)
                let texpr_list = List.map infer_expr expr_list
                in let expr_types = List.map get_expr_type_info texpr_list
                in let fdecl = find_func fname (* find the function*)
                in let binding_len = List.length arr
                in
                   begin
                   match fdecl with
                   | {formals = param_list;_} -> (*set env*)
                   let param_len = List.length param_list and true_len = List.length
expr_types
                   in if param_len = true_len + binding_len then (* actual a function
call *)
                      let rtype = get_func_result (infer_func_by_name fname
(List.append arr expr_types))
                      in TFly ((name, texpr_list), Signal rtype)
                      else failwith ("fly with not a true function call ")
                      (* a clojure which just a function bind less than true
parameters*)
                   end
             | _ -> failwith  ("not a clojure or function obj when functioncall")
             end
        | Register (signal_name, name, expr_list) ->
          (*check signal name is a signal*)
          let signal_type = ref_search_id signal_name
          in
          begin
          match signal_type with
          | Some (Signal x) ->
             begin
             let ftype = ref_search_id name
             in
             match ftype with
             | None -> failwith ("unknow refer" ^ name)
```

```
            | Some (Func (fname, arr)) ->
              let texpr_list = List.map infer_expr expr_list
              in let expr_types = List.map get_expr_type_info texpr_list
              in let fdecl = find_func fname (* find the function*)
              in let binding_len = List.length arr
              in
                 begin
                 match fdecl with
                 | {formals = param_list;_} ->
                 let param_len = List.length param_list and true_len = List.length
expr_types
                 in if param_len = true_len + binding_len + 1 then
                    (*generate t_funcdecl for this call*)
                    let _ = infer_func_by_name fname (arr @ expr_types @ [x])
                    (*always void for register*)
                    in TRegister ((signal_name, name, texpr_list), x)
                    else failwith ("param num not consistent")
                 end
            | _ -> failwith ("not a clojure or function when register call")
            end
        | _ -> failwith ("no signal type can not c")
        end
    | ObjGen typename ->
    begin match typename with
    | Class x ->
      (* check build in class first*)
      let if_build_in_class = check_build_in_class x
      in if if_build_in_class then
         TObjGen (Class x, Class x)
      else
         let cdecl = find_class x
         in TObjGen (Class x, Class (get_class_name cdecl))
    | Array x ->
      (*check class name exist *)
      if check_valid_type x then
         TObjGen (typename, typename)
      else
         failwith ("not valid type name")
    | Map (x, y) ->
      (* check type exist*)
      if check_valid_type x && check_valid_type y
      then TObjGen (typename, typename)
      else
      failwith ("not valid type name")
```

```
      | _ -> failwith ("no support for other gen")
      end
      | Objid (x, y) ->
        let ctype = ref_search_id x
        in begin
          match ctype with
          | Some (Class cname) ->
            let cdecl = find_class cname
            in let mvartype = get_class_member_type cdecl y
            in TObjid ((x, y), mvartype)
          | None -> failwith ("var used without defined: " ^ x)
          | _ -> failwith ("not class obj can not dot id: " ^ x)
          end
      (*clojure*)
      | Call (name, expr_list) ->
        (* find in bindings*)
        let texpr_list = List.map infer_expr expr_list
        in let expr_types = List.map get_expr_type_info texpr_list
        in let ftype = ref_search_id name
        in
          begin
          match ftype with
          | None ->
            (*maybe build in no support for clojure build in now*)
            let test_build_in_func = match_build_in name expr_types
            in begin match test_build_in_func with
             | None -> failwith ("no refer to " ^ name)
             | Some tfdecl ->
               begin
               match tfdecl with
               | {tret=rtype;_} ->
               TCall ((name, texpr_list), rtype)
               end
            end
          | Some (Func (fname, arr)) -> (*with*)
            let fdecl = find_func fname (* find the function*)
            in let binding_len = List.length arr
            in
              begin
              match fdecl with
              | {formals = param_list;_} -> (*set env*)
              let param_len = List.length param_list and true_len = List.length
expr_types
              in
```

```
                    (*update clojure calls if not build in*)
                    if check_build_in_name fname then () else update_clojure_calls
fname arr expr_types;
                    if param_len = true_len + binding_len then (* actual a function
call *)
                    let rtype = get_func_result (infer_func_by_name fname
(List.append arr expr_types))
                    in TCall ((name, texpr_list), rtype)
                    else
                    if param_len < true_len + binding_len then
                       failwith ("too many args")
                    else
                    TCall ((name, texpr_list), Func (fname, List.append arr
expr_types))
                    (* a clojure which just a function bind less than true
parameters*)
                  end
            | _ -> failwith  ("not a clojure or function obj when functioncall")
            end
      (*Haha OBJ CALL!*)
      | ObjCall (varname, fname, expr_list) ->
         (* find in bindings*)
         (*get expr type*)
         let texpr_list = List.map infer_expr expr_list
         in let expr_types = List.map get_expr_type_info texpr_list
         in
         let ftype = ref_search_id varname
         in (*check class variable*)
            begin
            match ftype with
            | Some (Class cname) ->
               (*check build in_class call*)
               if check_build_in_class cname then
                  let match_tfdecl = match_build_in_objcall cname fname
expr_types
                  in begin match match_tfdecl with
                  | None -> failwith("build in call error plz search ref mannue")
                  | Some x ->
                  let rtype = get_func_result x
                  in TObjCall ((varname, fname, texpr_list), rtype)
                  end
               else
               let somefdecl = find_cfunc cname fname
               in begin match somefdecl with
```

```
                | None -> failwith ("this member func not found: " ^ cname ^ " " ^
fname)
                | Some fdecl ->
                  (*check list length, don't consider meberfunc clojure now*)
                  begin match fdecl with
                  | {formals=param_list;_} ->
                  let param_len = List.length param_list and true_len = List.length
expr_types
                  in if param_len = true_len then
                     let rtype = get_func_result (infer_cfunc_by_name cname fname
expr_types)
                     in TObjCall ((varname, fname, texpr_list), rtype)
                  else
                     failwith ("args number not the same")
                  end
               end
            | None -> failwith ("var used without defined: " ^ varname)
            | Some (Array x) ->
              (*check support array functions*)
              let rtype = get_arr_call_ret (Array x) fname expr_types
              in TObjCall ((varname, fname, texpr_list), rtype)
            | Some (Map (x,y)) ->
              (*check support map functions*)
              let rtype = get_map_call_ret (Map (x,y)) fname expr_types
              in TObjCall ((varname, fname, texpr_list), rtype)
            | Some (Signal (x)) -> (*signal wait*)
              begin match fname with
              | "wait" -> if List.length expr_list == 0 then TObjCall ((varname,
fname, texpr_list), x)
                 else failwith ("wait must 0 params")
              | _ -> failwith ("not support signal option")
              end
            | _ -> failwith ("not class obj can not objcall: " ^ varname)
           end
      | Changen thistype ->
        TChangen(thistype, Chan thistype)
      | Chanunop x ->
        let ctype = ref_search_id x
        in begin match ctype with
        | Some (Chan containtype) -> TChanunop(x, containtype)
        | None -> failwith("obj usage with out def")
        | _ -> failwith ("<- not suitable for non-chan obj")
        end
      | Chanbinop (x, y) ->
```

```
        let ctypex = ref_search_id x and ctypey = ref_search_id y
        in begin match ctypex, ctypey with
        | Some(Chan (containtypex)), Some(containtypey) ->
           if containtypex = containtypey
           then TChanbinop((x, y), containtypex)
           else failwith ("type not consistent in Chanbinop")
        | Some(containtypex), Some(Chan (containtypey)) ->
           if containtypex = containtypey
           then TChanbinop((x, y), containtypex)
           else failwith ("type not consistent in Chanbinop")
        | None, Some(Chan (containtypey)) ->
           (*update env*)
           ref_update_env x containtypey;
           TChanbinop((x, y), containtypey)
        | None, _ -> failwith ("obj usage without def")
        | _, None -> failwith ("obj usage without def")
        | _ -> failwith("<- not support for non-chan")
        end
      | Dispatch (name, expr_list, ip, port) ->
        (*just support self-defined non-dependent function*)
        let texpr_list = List.map infer_expr expr_list
        in let expr_types = List.map get_expr_type_info texpr_list
        in let ftype = ref_search_id name
        in let ip_texpr = infer_expr ip
        in let port_texpr = infer_expr port
        in let ip_type = get_expr_type_info ip_texpr
        in let port_type = get_expr_type_info port_texpr
        in
           if ip_type = String && port_type = Int
           then
              begin match ftype with
              | None ->failwith ("not support for not-defined function")
              | Some (Func (fname, arr)) ->
                 let l = List.length arr
                 in if l = 0 then
                    let fdecl = find_func fname
                    in begin match fdecl with
                    | {formals = param_list;_}->
                    let param_len = List.length param_list and true_len = List.length
expr_types
                    in if param_len = true_len then
                       let rtype = get_func_result (infer_func_by_name fname
expr_types)
                       in TDispatch ((name, texpr_list, ip_texpr, port_texpr), rtype)
```

```
                    else failwith ("args not match in dispatch")
                    end
                else failwith ("not support for clojure dispatch now")
            | _ -> failwith ("not a function in dispatch")
            end
        else
            failwith ("ip or port type error")
    (* TODO
    | ListComprehen (f_expr, varname, s_expr) -> *)
    | _ -> TLiteral 142857
in
(* return a tstmt*)
let rec infer_stmt smt = match smt with
    | Block stmt_list -> ref_create_env();
      let tstmt_list = List.map infer_stmt stmt_list
      in ref_back_env(); (*back this env*)
         TBlock tstmt_list
    | Expr epr ->
      TExpr (infer_expr epr)
    | Return epr ->
      (*TODO may update return type*)
      let tepr = infer_expr epr
      in let tepr_type = get_expr_type_info tepr
      in let tfdecl = Hashtbl.find t_func_binds hash_key
      in let new_tfdecl = compare_and_update tfdecl tepr_type
      in Hashtbl.replace t_func_binds hash_key new_tfdecl;
         TReturn tepr
    | If (judge_expr, f_stmt_list, s_stmt_list) ->
      let judge_t_expr = infer_expr judge_expr
      in
         check_bool (get_expr_type_info judge_t_expr);
         ref_create_env();
         let t_f_stmt_list = List.map infer_stmt f_stmt_list
         in ref_back_env();
            ref_create_env();
            let t_s_stmt_list = List.map infer_stmt s_stmt_list
            in ref_back_env();
               TIf (judge_t_expr, t_f_stmt_list, t_s_stmt_list)
    | For (init_expr, judge_expr, loop_expr, inner_stmt_list) ->
      (* new env *)
      ref_create_env();
      let init_texpr = infer_expr init_expr
      in let judge_texpr = infer_expr judge_expr
      in let judge_texpr_type = get_expr_type_info judge_texpr
```

```
          in begin
          match judge_texpr_type with
          | Bool ->
            let loop_texpr = infer_expr loop_expr
            in let tstmt_list = List.map infer_stmt inner_stmt_list
            in ref_back_env();
               TFor (init_texpr, judge_texpr, loop_texpr, tstmt_list)
          | _ -> failwith ("judge expr not bool type")
          end
        | While (judge_expr, stmt_list) ->
          let judge_texpr = infer_expr judge_expr
          in let judge_texpr_type = get_expr_type_info judge_texpr
          in begin
          match judge_texpr_type with
          | Bool ->
            ref_create_env();
            let tstmt_list = List.map infer_stmt stmt_list
            in ref_back_env();
               TWhile (judge_texpr, tstmt_list)
          | _ -> failwith("judge expr not bool type")
          end
        | Break ->
          TBreak
        | Continue ->
          TContinue
        | Foreach (varname, base_expr, stmt_list) ->
          (* easy to change to a TFor stmt*)
          (*TODO*)
          let base_texpr = infer_expr base_expr
          in let base_texpr_type = get_expr_type_info base_texpr
          in let base_container_name = begin match base_expr with
              | Id x -> x
              | _ -> failwith ("not support with dynamic create of container")
             end
          in begin match (base_texpr_type:typ) with
              | Array x ->
                let iter_var_name = "i"
                in let init_expr = Assign (iter_var_name, Literal 0)
                in let stop_expr =
                   Binop (Id iter_var_name, Less, ObjCall(base_container_name,
"size", []))
                in let inc_expr =
                   Assign (iter_var_name, Binop(Id iter_var_name, Add, Literal 1))
                in
```

```
                (*new ref*)
                ref_create_env();
                let init_texpr = infer_expr init_expr
                in let stop_texpr = infer_expr stop_expr
                in let inc_texpr = infer_expr inc_expr
                in
                (*carete varname assign*)
                let var_assign_stmt = Expr (Assign (varname,
ObjCall(base_container_name, "get_at", [Id iter_var_name])))
                in let modify_stmt_list = var_assign_stmt :: stmt_list
                in let tstmt_list = List.map infer_stmt modify_stmt_list
                in ref_back_env(); (*back env*)
                   TFor (init_texpr, stop_texpr, inc_texpr, tstmt_list)
            | Map (x, y) ->
                (*now begin support map foreach*)
                ref_create_env();
                (*add (varname,x) type info*)
                ref_update_env varname x;
                let tstmt_list = List.map infer_stmt stmt_list
                in ref_back_env();
                   TForeach(varname, base_texpr,tstmt_list)
            | _ -> failwith ("not support now")
          end
   in
   match fdecl with
   | {body = stmt_list;formals = param_list;fname = func_name} ->
      (* scan twice to check return type*)
      (*save env*)
      let saved_env = level_env_copy !level_env
      in
      let _ = List.map (fun item -> try
         infer_stmt item
      with
      | _ -> TExpr (TLiteral 0)) stmt_list
      (*sequencely infer each stmt with level env *)
      in (*recover*)
      level_env := saved_env;
      let tstmt_lists = List.map infer_stmt stmt_list
      in let t_param_list = List.map2 (fun item1 item2 -> (item1, item2))
param_list type_list
      in let rtype = get_func_result (Hashtbl.find t_func_binds hash_key)
      in (*if undef then set void*)
         if rtype == Undef then
            {ttkey = hash_key;tfname = func_name;tformals = t_param_list;tbody =
```

```
tstmt_lists;tret = Void}
          else
             {ttkey = hash_key;tfname = func_name;tformals = t_param_list;tbody =
tstmt_lists;tret = rtype}
       (*generate a t func decl*)


(* when we see a fname and para with type
   we call this function to put a record to a global type info
   and return the t_func_decl
*)

and infer_func_by_name fname type_list =
   let hash_key = gen_hash_key fname type_list
   in let hash_value = find_t_func hash_key
   (*this is the place for check in*)
   in let test_build_in_func = match_build_in fname type_list
   in match test_build_in_func with
      | Some x -> x
      | None ->
      begin
      match hash_value with
      | None ->
         let fdecl = find_func fname
         in
            begin
            match fdecl with
            | {formals=param_list;_} ->
               (*first create a binding*)
               Hashtbl.add t_func_binds hash_key (new_null_tfdecl());
               (*create func env*)
               let func_env =
                  func_level_env()
               in
               (*create env and add param type*)
               let new_func_level_env =
                  List.fold_left2 (fun env param_name this_type -> update_env env
param_name this_type) (init_level_env()) param_list type_list
               in
               let ref_new_func_level_env = ref(List.rev
(func_env::new_func_level_env))
               in let tfdecl = infer_func fdecl hash_key type_list
ref_new_func_level_env
               in
```

```
                (*store in the global hash*)
                Hashtbl.replace t_func_binds hash_key tfdecl;
                tfdecl
              end
        | Some x ->
          let rtype = get_func_result x
          in if rtype == Undef then failwith ("no stop recurisve call" ^hash_key)
          else x
       end

(*infer a class member function call*)
and infer_cfunc_by_name cname fname type_list =
   let hash_key = gen_hash_key fname type_list
   in let hash_value = find_t_mfunc cname hash_key
   in match hash_value with
      | None ->
        let somefdecl = find_cfunc cname fname
        in begin match somefdecl with
        | None -> failwith ("no this member func")
        | Some fdecl ->
          (*bind the env*)
          begin match fdecl with
             | {formals=param_list;_}->
            (*first create a binding*)
            Hashtbl.add t_func_binds hash_key (new_null_tfdecl());
            let tcdecl = find_t_class cname
            in
            (*create func env*)
            let func_env =
               func_level_env()
            in
            (*create env and add param type*)
            let new_func_level_env =
               List.fold_left2 (fun env param_name this_type -> update_env env
param_name this_type) (init_level_env()) param_list type_list
            in
            let new_func_level_env =
               (*add class member variable*)
               match tcdecl with
               | {member_binds=binds;_} ->
                  List.fold_left (fun env (varname, thistype) -> update_env env
varname thistype) new_func_level_env binds
            in
            let ref_new_func_level_env = ref(List.rev
```

```
(func_env::new_func_level_env))
                in let tfdecl = infer_func fdecl hash_key type_list
ref_new_func_level_env
                in
                let new_tcdecl = update_tclass tcdecl tfdecl
                in
                   (*replace in the global t_class_binds hash*)
                   Hashtbl.replace t_class_binds cname new_tcdecl;
                   tfdecl
             end
          end
       | Some x ->
          let rtype = get_func_result x
          in if rtype = Undef then failwith ("no class stop recurisve call" ^hash_key)
          else x

let debug_ast_cdecl ast = match ast with
    | Program (cdecls, _) -> List.iter (fun item -> print_endline (debug_cdecl item))
cdecls

(* perform static type checking and inferrence*)
let infer_check (ast : program) =
    (*first check continue/break*)
    br_con_check ast;
    add_build_in_func(); (*first add some build in func name*)
    bind_name ast; (*second bind name*)
    init_tclass();
    (*just infer the main function and recur infer all involved functions *)
    let _ =  infer_func_by_name "main" []
    in
    (*
    debug_ast_cdecl ast;
    debug_t_func_binds();
    print_endline (debug_t_fdecl main_fdecl);
    *)
    (t_func_binds, t_class_binds, clojure_calls, func_binds, t_func_binds)
    (* search main function and do a static type infer*)
```

buildin.ml

```
open Ast
open Sast

(* to_float, to_int, to_string, sleep, exit *)

let build_in_code =
[
"
#include<fly/func.h>
"
]


let str_is_int = {
    ttkey = "";
    tfname ="str_is_int";
    tformals = [("a", String)];
    tbody = [];
    tret = Bool;
}

let str_split = {
    ttkey = "";
    tfname ="str_split";
    tformals = [("a", String)];
    tbody = [];
    tret = Array(String);
}

let len = {
        ttkey = "";
        tfname = "len";
        tformals = [("a", String)];
        tbody = [];
        tret = Int;
}

let int_to_float = {
        ttkey = "";
        tfname = "_float";
        tformals = [("int", Int)];
        tbody = [];
        tret = Float;
}
```

```
let string_to_float = {
        ttkey = "";
        tfname = "_float";
        tformals = [("string", String)];
        tbody = [];
        tret = Float;
}

let print_bool = {
        ttkey = "";
        tfname = "print_bool";
        tformals = [("bool", Bool)];
        tbody = [];
        tret = Void;
}

let print_float = {
        ttkey = "";
        tfname = "print";
        tformals = [("float", Float)];
        tbody = [];
        tret = Void;
}

let print_int = {
        ttkey = "";
        tfname = "print";
        tformals = [("int", Int)];
        tbody = [];
        tret = Void;
}

let print_str = {
        ttkey = "";
        tfname = "print";
        tformals = [("string", String)];
        tbody = [];
        tret = Void;
}

let string_to_int = {
        ttkey = "";
        tfname = "_int";
```

```
        tformals = [("string", String)];
        tbody = [];
        tret = Int;
}

let int_to_string = {
        ttkey = "";
        tfname = "_string";
        tformals = [("int", Int)];
        tbody = [];
        tret = String;
}

let float_to_string = {
        ttkey = "";
        tfname = "_string";
        tformals = [("float", Float)];
        tbody = [];
        tret = String;
}

let exit_func = {
        ttkey = "";
        tfname = "_exit";
        tformals = [("int", Int)];
        tbody = [];
        tret = Void;
}

let vect_int_to_string = {
   ttkey = "";
   tfname = "_string";
   tformals = [("vint", Array (Int))];
   tbody = [];
   tret = String;
}

let sleep_func = {
        ttkey = "";
        tfname = "sleep";
        tformals = [("int", Int)];
        tbody = [];
        tret = Void;
}
```

```
let exec_func = {
    ttkey = "";
    tfname = "exec";
    tformals = [("string", String);("string", String)];
    tbody = [];
    tret = String;
}
let string_vint_func = {
    ttkey = "";
    tfname = "_vector_int";
    tformals = [("string", String)];
    tbody = [];
    tret = Array (Int);
}
let get_argv = {
    ttkey = "";
    tfname = "get_argv";
    tformals = [("int", Int)];
    tbody = [];
    tret = String;
}
let build_in_func =
    [int_to_string;
    float_to_string;
    string_to_int;
            string_to_float;
            int_to_float;
            exit_func;
            sleep_func;
            print_str;
            print_int;
            print_float;
            print_bool;
    str_split;
    str_is_int;
    len;
    exec_func;
    string_vint_func;
    vect_int_to_string;get_argv]



let rec match_build_in fname type_list =
        let rec inner_func funcs fname type_list = match funcs with
```

```
        | [] -> None
        | (x::y) -> begin match x with
                | {tfname=thisfname;tformals=binds;_}->
                        let thistype_list = List.map snd binds
                        in if type_list = thistype_list && fname = thisfname
                        then Some x
                        else inner_func y fname type_list
                end
        in
        inner_func build_in_func fname type_list

let rec check_build_in_name fname =
        List.exists (fun item -> match item with
        | {tfname=thisfname;_} -> if fname = thisfname then true else false)
build_in_func

(*string to_string(void a){} *)
(*string to_string(array a){} *)
(*string to_string(set a ){} *)

(* The following defines built-in class and their member functions *)

let connection_is_alive = {
   ttkey = "";
   tfname = "is_alive";
   tformals = [];
   tbody = [];
   tret = Bool;
}

let connection_recv = {
   ttkey = "";
   tfname = "recv";
   tformals = [];
   tbody = [];
   tret = String;
}

let connection_send = {
   ttkey = "";
   tfname = "send";
   tformals = [("msg", String)];
   tbody = [];
   tret = Bool;
```

```
}

let connection_close = {
    ttkey = "";
    tfname = "close";
    tformals = [];
    tbody = [];
    tret = Void;
}

let connection = {
    tcname = "connection";
    member_binds = [];
    t_func_decls = [connection_recv; connection_close; connection_send;
connection_is_alive];
}

let server_listen = {
    ttkey = "";
    tfname = "listen";
    tformals = [("port", Int)];
    tbody = [];
    tret = Void;
}

let server_accept = {
    ttkey = "";
    tfname = "accept";
    tformals = [];
    tbody = [];
    tret = Class("connection");
}

let server = {
    tcname = "server";
    member_binds = [];
    t_func_decls = [server_listen; server_accept];
}

let client_connect = {
    ttkey = "";
    tfname = "connect";
    tformals = [("server_ip", String);("port", Int)];
    tbody = [];
```

```
    tret = Class("connection");
}

let client = {
    tcname = "client";
    member_binds = [];
    t_func_decls = [client_connect];
}

let build_in_class =
    [server; connection; client]

let check_build_in_class cname =
        List.exists (fun item -> match item with
                | {tcname=thiscname;_}-> if cname = thiscname then true else
false) build_in_class

let match_build_in_objcall cname fname type_list =
        let match_func tfdecl fname type_list = match tfdecl with
                | {tfname=thisfname;tformals=binds;_} ->
                        let thistype_list = List.map snd binds
                        in if thisfname = fname && thistype_list = type_list
                        then true
                        else false
        in
        let rec match_funcs tfdecls fname type_list = match tfdecls with
                | [] -> None
                | (x::y) -> if match_func x fname type_list then Some x else
match_funcs y fname type_list
        in
        let rec inner_func classes cname fname type_list = match classes with
                | [] -> None
                | (x::y) ->
                        begin
                        match x with
                        | {tcname=thiscname;t_func_decls=tfdecls;_} ->
                                if thiscname = cname
                                then match_funcs tfdecls fname type_list
                                else inner_func y cname fname type_list
                        end
        in
        inner_func build_in_class cname fname type_list
```

```
(*get the return type of array, fail if not ok*)
let get_arr_call_ret (thistype:typ) fname expr_types = match thistype with
    | Array x ->
       let expr_len = List.length expr_types
       in
       begin match fname with
       | "push_back" ->
          if expr_len = 1 then
             if [x] = expr_types then Void
             else failwith ("type not consistent: get_arr_call_ret")
          else
             failwith ("push_back not 1 element: get_arr_call_ret")
       | "push_vec" ->
          if expr_len = 1 then
             let y = List.hd expr_types
             in
             match y with
             | Array z -> if x = z then Void
                else failwith ("type not consistent: get_arr_call_ret")
             | _ -> failwith ("type not consistent: get_arr_call_ret")
          else
             failwith ("push_vec not 1 element: get_arr_call_ret")
       | "get_at" ->
          if expr_len = 1 then
             if [Int] = expr_types then x
             else failwith ("type not consistent: get_arr_call_ret")
          else
             failwith ("get_at not 1 element: get_arr_call_ret")
       | "set_at" ->
          if expr_len = 2 then
             if [Int;x] = expr_types then Void
             else failwith ("type not consistent: get_arr_call_ret")
          else
             failwith ("get_at not 1 element: get_arr_call_ret")
       | "size" ->
          if expr_len = 0 then
             Int
          else
             failwith("size should 0 element: get_arr_call_ret")
       | "sync" ->
          if expr_len = 0 then
             Void
          else
             failwith("sync should 0 element: get_arr_call_ret")
```

```
      | _ ->
        failwith ("not support build in array function")
      end
   | _ -> failwith ("not array error")

(*get the return type of map, fail if not ok*)
let get_map_call_ret (thistype:typ) fname expr_types = match thistype with
   | Map (x,y) ->
      let expr_len = List.length expr_types
      in
      begin match fname with
      | "insert" ->
        if expr_len = 2 then
          if [x;y] = expr_types then Void
          else failwith ("type not consistent: get_map_call_ret")
        else
          failwith ("insert not 2 element: get_map_call_ret")
      | "get" ->
        if expr_len = 1 then
          if [x] = expr_types then y
          else failwith ("type not consistent: get_map_call_ret")
        else
          failwith ("get_at not 1 element: get_map_call_ret")
      | "size" ->
        if expr_len = 0 then
          Int
        else
          failwith("size should 0 element: get_map_call_ret")
      | "delete" ->
        if expr_len = 1 then
          if [x] = expr_types then Void
          else failwith ("type not consistent: get_map_call_ret")
        else
          failwith("delete should 1 element: get_map_call_ret")
      | "exist" ->
        if expr_len = 1 then
          if [x] = expr_types then Bool
          else failwith  ("type not consistent: get_map_call_ret")
        else
          failwith("exist should be 1 element: get_map_call_ret")
      | "sync" ->
        if expr_len = 0 then
          Void
        else
```

```
            failwith("sync should 0 element: get_map_call_ret")
        | _ ->
          failwith ("not support build in map function")
        end
    | _ -> failwith ("not array error")

let build_in_class_code = ["
#include<fly/class.h>
#include<fly/fly.h>
#include<fly/exec.h>
"]
```

codegen.ml

```
open Ast
open Sast
open Env
open Util
open Infer
open Buildin

type sigbind = {
    vn : string;
    vt : typ;
}

type regibind = {
    vn : string; (* sig var name *)
    vt : typ;    (* Signal(t) *)
    rn : string; (* var name to wait for value *)
}

type fkey_fd_bind = {
    fkey : string;
    fd   : t_func_decl;
}

type objfly_bind = {
```

```
    objname : string;
    classname : string;
    fname : string;
    paramt : typ list;
    rtype : typ;
}

let gen_ofly_fkey oname fname =  ""

(* store signal funcs *)
let (objsignal_funcs : (string, objfly_bind) Hashtbl.t) = Hashtbl.create 16

(* store signal funcs *)
let (signal_funcs : (string, string) Hashtbl.t) = Hashtbl.create 16

(* store register funcs *)
let (register_funcs : (string, string) Hashtbl.t) = Hashtbl.create 16


let gen_clojure_class_name funcname type_list =
    funcname ^ (List.fold_left (fun res item -> res ^ "_" ^ (type_to_string item))
"_clojure" type_list)


let (dispatch_funcs : (string, string) Hashtbl.t) = Hashtbl.create 16

let gen_dispatch_code () =
    Hashtbl.fold (fun k v res -> v::res) dispatch_funcs []

let gen_clojure_classes clojure_calls func_binds t_func_binds =
    let find_func name =
        try
            Hashtbl.find func_binds name
        with
        | Not_found -> failwith ("not this function:" ^ name)
    in
    let gen_clojure_class fname call_list =
        let (clojure_class_hash : (string, t_class_decl) Hashtbl.t) = Hashtbl.create 16
        in let fdecl = find_func fname
        in let init_tcdecl fname type_list = begin match fdecl with
            | {formals=param_list;_} ->
                let _ = List.length type_list
                in let modify_param_list = List.map (fun item -> "__" ^ item) param_list
                in let binds = zip modify_param_list type_list
```

```
            in let clojure_name = gen_clojure_class_name fname type_list
            in {tcname=clojure_name;member_binds=binds;t_func_decls=[]}
          end
      in let rec get_or_init fname type_list =
          let clojure_name = gen_clojure_class_name fname type_list
          in
          try
            Hashtbl.find clojure_class_hash clojure_name
          with
          | _ -> Hashtbl.add clojure_class_hash clojure_name (init_tcdecl fname
type_list);get_or_init fname type_list
      in let rec update_if_no tfdecls tfdecl = match tfdecls with
          | [] -> [tfdecl]
          | (x::y) -> begin match x, tfdecl with
            | {ttkey=key1;_}, {ttkey=key2;_} ->
              if key1 = key2 then y else x :: (update_if_no y tfdecl)
            end
      in let gen_tfdecl fname f_type_list s_type_list =
          let clojure_name = gen_clojure_class_name fname f_type_list
          in let _ = get_or_init fname (List.concat [f_type_list;s_type_list]) and tcdecl
= get_or_init fname f_type_list
          in begin match fdecl with
          | {formals=param_list;_} ->
            let l1 = List.length f_type_list
            in let l2 = List.length s_type_list
            in let lt = List.length param_list
            in let f_binds = zip param_list f_type_list
            in let s_param_list = drop_first param_list l1
            in let s_binds = zip s_param_list s_type_list
            in
              let new_tfdecl =
              if l1 + l2 = lt then
                let tfkey = gen_hash_key fname (List.concat
[f_type_list;s_type_list])
                in
                let tfdecl = Hashtbl.find t_func_binds tfkey
                in let rtype = get_func_result tfdecl
                in
                let ftexprs = List.map (fun (varname, thistype) -> TId("__" ^
varname, thistype)) f_binds
                in let stexprs = List.map (fun (varname, thistype) -> TId(varname,
thistype)) s_binds
                in let ttexprs = List.concat [ftexprs;stexprs]
                in let body = [TReturn (TCall ((fname, ttexprs), rtype))]
```

```
                    in {
                        ttkey=gen_hash_key "call" s_type_list;
                        tfname="call";
                        tformals=s_binds;
                        tbody = body;
                        tret = rtype
                    }
                else
                let res_class_name = gen_clojure_class_name fname (List.concat
[f_type_list;s_type_list])
                in let new_obj_stmt = TExpr (TAssign (("tmp", TObjGen(Class
res_class_name, Class res_class_name)), Class res_class_name))
                in let f_assign_stmts =
                    List.map (fun (varname, thistype) -> TExpr (TMAssign(("tmp", "__"
^ varname, TId (varname, thistype)), thistype))) f_binds
                in let s_assign_stmts =
                    List.map (fun (varname, thistype) -> TExpr (TMAssign(("tmp", "__"
^ varname, TId (varname, thistype)), thistype))) s_binds
                in let return_stmt =
                    TReturn (TId ("tmp", Class res_class_name))
                in {
                        ttkey=gen_hash_key "call" s_type_list;
                        tfname="call";
                        tformals=s_binds;
                        tbody = [new_obj_stmt] @ f_assign_stmts @ s_assign_stmts
@[return_stmt];
                        tret = Class res_class_name
                    }
                in begin match tcdecl with
                    | {tcname=tmp1;member_binds=tmp2;t_func_decls=tfdecls;_} ->
                        let new_tfdecls = update_if_no tfdecls new_tfdecl
                        in Hashtbl.replace clojure_class_hash clojure_name
{tcname=tmp1;member_binds=tmp2;t_func_decls=new_tfdecls}
                end
            end
    in
        List.iter (fun (f_type_list, s_type_list) -> gen_tfdecl fname f_type_list
s_type_list) call_list;
        Hashtbl.fold (fun k v arr -> (k,v)::arr) clojure_class_hash []
    in
        List.concat (Hashtbl.fold (fun k v arr -> (gen_clojure_class k v) :: arr)
clojure_calls [])
```

```
let (fundone : (string, string) Hashtbl.t) = Hashtbl.create 16

let add_hash ht k v =
   Hashtbl.add ht k v

let find_hash ht key =
   try
      Some (Hashtbl.find ht key)
   with
    | Not_found -> None


let remove_hash ht key =
   Hashtbl.remove ht key


let clean_up_hash ht =
   Hashtbl.iter (fun k v -> remove_hash ht k) ht


let rec type_to_func_string = function
    | Int -> "int"
    | Bool -> "bool"
    | Void -> "void"
    | String -> "string"
    | Float -> "float"
    | Signal(x) -> "signal_" ^ (type_to_func_string x)
    | Chan(x) -> "chan_" ^ (type_to_func_string x)
    | Class(x) -> "class_" ^ x
    | Array(x) -> "array_" ^ (type_to_func_string x)
    | Map(x,y) -> "map_" ^ (type_to_func_string x) ^ "_" ^ (type_to_func_string y)
    | Func(_, tlist) -> List.fold_left (fun ret ele -> ret ^ "_" ^ (type_to_func_string
ele)) "closure" tlist
    | Lfunc(_) -> raise (Failure ("type_to_func_string not yet support Lfunc")) (*
TODO *)
    | Set(_) -> raise (Failure ("type_to_func_string not yet support Set")) (* TODO
*)
    | Undef -> raise (Failure ("type_to_func_string not yet support Undef")) (*
TODO *)
    | Cfunc(_) -> raise (Failure ("type_to_func_string not yet support Cfunc")) (*
TODO *)
```

```
let rec type_to_code_string x = begin match x with
    | Int -> "int"
    | Bool -> "bool"
    | Void -> "void"
    | String -> "string"
    | Float -> "float"
    | Signal(x) -> "shared_ptr <Signal<" ^ (new_type_to_code_string x) ^ ">>"
    | Class x -> "shared_ptr <" ^ x ^ ">"
    | Array x -> "shared_ptr < flyvector <" ^ (type_to_code_string x) ^ "> >"
    | Map (x, y) -> "shared_ptr < flymap <" ^ (type_to_code_string x) ^ "," ^
(type_to_code_string y) ^ "> >"
    | Chan x -> "shared_ptr <Chan<" ^ (new_type_to_code_string x) ^ ">>"
    | Func (x, type_list) -> "shared_ptr <" ^ (gen_clojure_class_name x type_list) ^
">"
    | _ -> raise (Failure ("type_to_code_string not yet support this type"))
   end
and new_type_to_code_string x = begin match x with
    | Class x -> x
    | Array x -> "flyvector <" ^ (type_to_code_string x) ^ ">"
    | Map (x, y) ->  "flymap <" ^ (type_to_code_string x) ^ "," ^
(type_to_code_string y) ^ ">"
    | Int -> "int"
    | Bool -> "bool"
    | Void -> "void"
    | String -> "string"
    | Float -> "float"
    | Chan x -> "Chan<" ^ (new_type_to_code_string x) ^ ">"
    | x -> print_endline (type_to_string x);failwith ("not support for other
new_type_to_code_string")
   end

(* take a string list and concatenate them with interleaving space into a single
string *)
let rec cat_string_list_with_space sl =
   match sl with
    | [] -> ""
    | hd::tl -> hd ^ " " ^ (cat_string_list_with_space tl)

(* take a string list and concatenate them with interleaving comma into a single
string *)
let rec cat_string_list_with_comma sl =
   let tmp = List.fold_left (fun ret ele -> ret ^ ele ^ ",") "" sl in
   let len = (String.length tmp) in
```

```
    if len > 0 then (String.sub tmp 0 (len-1)) else tmp

let rec merge_string_list sl = match sl with
    | [] -> ""
    | (x::y) -> x ^ (merge_string_list y)

let get_typelist_from_fm fm =
    List.fold_left
    (fun ret (str_, type_) -> ret @ [type_])
    [] fm

(* take a formal and generate the string *)
let handle_fm formals refenv =
    let fstr =
        List.fold_left
        (fun ret (str_, type_) ->
            ignore(update_env (!refenv) str_ type_);
            ret ^ " " ^ (type_to_code_string type_) ^ " " ^ str_ ^ ",") "" formals in
    let len = (String.length fstr) in
    let trimed = if len > 0 then (String.sub fstr 0 (len-1)) else fstr in
    "(" ^ trimed ^ ")"

(* generate fly wrapper, return string list *)
let handle_fd_fly fd refenv =
    match fd with
    | {tret=rt; tfname=name; tformals=fm; tbody=body ;_} ->
        let ret = ["void"] in
        let rtstr = type_to_code_string rt in
        let tlist = get_typelist_from_fm fm in
        let fname = List.fold_left
                (fun ret type_ -> ret ^ "_" ^ (type_to_func_string type_))
                name
                (tlist @ [Signal(rt)]) in
        let sigvar = fname ^ "_sig" in
        let fmstr = handle_fm (fm @ [(sigvar, Signal(rt))]) refenv in
        let param = cat_string_list_with_comma (List.map (fun (n,_) -> n) fm) in
        let getvar = fname ^ "_var" in
        let body =
            match rt with
            | Void -> ["{"] @ [name ^ "(" ^ param ^ ");"] @ ["}"]
            | Class (_) | Array (_) | Map (_) | Chan (_) ->
                ["{"] @ ["auto " ^ getvar ^ " = " ^ name ^ "(" ^ param ^ ");"] @
                [sigvar ^ "->notify(" ^ getvar ^ ");"] @
                ["}"]
```

```
          | _ ->
            ["{"] @
            [(type_to_code_string rt) ^ " " ^ getvar ^ " = " ^ name ^ "(" ^ param ^ ");"]
@
            [sigvar ^ "->notify(shared_ptr<" ^ rtstr ^ ">(new " ^ rtstr ^ "(" ^ getvar ^
"")));"] @
            ["}"]
      in
      ret @ [fname] @ [fmstr] @ body

(* generate register wrapper, return string list *)
let handle_fd_register fd refenv =
   match fd with
   | {tret=rt; tfname=name; tformals=fm; tbody=body ;_} ->
      let (getvar, sigty) = match List.rev fm with
         | (var, ty)::tl -> (var, ty)
         | _ -> raise (Failure ("This register function doesn't have param")) in
      let sigvar = name ^ "_sig" in
      let nfm = match List.rev fm with
         | _::tl -> (List.rev tl) @ [(sigvar, Signal(sigty))]
         | _ -> [] in
      let rtstr = type_to_code_string rt in
      let tlist = get_typelist_from_fm nfm in
      let fname = List.fold_left
            (fun ret type_ -> ret ^ "_" ^ (type_to_func_string type_))
            name
            tlist in
      let fmstr = handle_fm nfm refenv in
      let param = cat_string_list_with_comma (List.map (fun (n,_) -> n) fm) in
      let body =
         match sigty with
         | Class (_) | Array (_) | Map (_) | Chan (_) ->
            ["{"] @
            [(type_to_code_string sigty) ^ " " ^ getvar ^ " = " ^ sigvar ^ "->wait();"] @
            [name ^ "(" ^ param ^ ");"] @
            ["}"]
         | _ ->
            ["{"] @
            [(type_to_code_string sigty) ^ " " ^ getvar ^ " = *" ^ sigvar ^ "->wait();"]
@
            [name ^ "(" ^ param ^ ");"] @
            ["}"]
         in
      [rtstr] @ [fname] @ [fmstr] @ body
```

```
(* take signal name and fly call, return a string list *)
let rec handle_fly_expr signame expr refenv =
   let syncfunc = "detach()" in
   match expr with
   | TFly((fn, texpr_list), st) ->
         let expr_types_list = List.map get_expr_type_info texpr_list in
         let nfn = (List.fold_left (fun ret et -> ret ^ "_" ^ (type_to_func_string et)) fn
expr_types_list)
            ^ "_" ^ (type_to_func_string st) in
         let param = [cat_string_list_with_comma (List.fold_left (fun ret ex ->
ret@(handle_texpr ex refenv)) [] texpr_list)] in
         let param2 =
         (
            match param with
            | [""] -> []
            | _ -> param @ [","]
         ) in
         ["thread(";nfn;","] @ param2 @ [signame;")." ^ syncfunc]
   | _ -> raise (Failure ("Assigning something to Signal other than TFly"))

(* take one expr and return a string list *)
and handle_texpr expr refenv =
   match expr with
   | TLiteral(value) -> [string_of_int value]
   | TBoolLit(value) -> if value then ["true"] else ["false"]
   | TFloat(value) -> [string_of_float value]
   | TId(str, _) ->
      (* TODO if this is a func passing, we convert it into our self defined obj*)
      begin
      try
         let _ = Hashtbl.find func_binds str
         in let raw_clojure_class_name = gen_clojure_class_name str []
         in ["shared_ptr<" ^ raw_clojure_class_name ^ ">(new " ^
raw_clojure_class_name ^ "())"]
      with
      | _ -> [str]
      end
   | TSet(_) -> [] (* TODO *)
   | TMap(_) -> [] (* TODO *)
   | TArray(_) -> [] (* TODO *)
   | TString(str) -> [str]
   | TBinop((texpr1, op, texpr2), _) ->
      [cat_string_list_with_space
```

```
        (["("] @ (handle_texpr texpr1 refenv) @ [op_to_string op] @
(handle_texpr texpr2 refenv) @ [")"])
      ]
  | TUnop((uop, texpr), _) -> [cat_string_list_with_space (["("] @ [uop_to_string
uop] @ (handle_texpr texpr refenv) @ [")"])]
  | TCall ((fn, texpr_list), t) ->
      (
      let expr_types = List.map get_expr_type_info texpr_list
      in let if_check_in = match_build_in fn expr_types
      in match if_check_in with
      | Some x ->
        [
        cat_string_list_with_space
        ([fn;"("]@
        [cat_string_list_with_comma (List.fold_left (fun ret ex ->
ret@([merge_string_list (handle_texpr ex refenv)])) [] texpr_list)]@
        [")"])
        ]
      (* above are built-in functions *)
      | _ ->
        begin
        try
          let fdecl = Hashtbl.find func_binds fn
          in begin match fdecl with
          | {formals=binds;_} ->
            let bind_len = List.length binds and expr_len = List.length texpr_list
            in
            if bind_len = expr_len then
            [
              cat_string_list_with_space
              ([fn;"("]@
              [cat_string_list_with_comma (List.fold_left (fun ret ex ->
ret@(handle_texpr ex refenv)) [] texpr_list)]@
              [")"])
            ]
            else
            let raw_clojure_class_name = gen_clojure_class_name fn []
            in let func_obj =
              "(shared_ptr<" ^ raw_clojure_class_name ^ ">(new " ^
raw_clojure_class_name ^ "()))"
            in
            [
            cat_string_list_with_space
            ([func_obj;"->call("]@
```

```
                [cat_string_list_with_comma (List.fold_left (fun ret ex ->
ret@(handle_texpr ex refenv)) [] texpr_list)]@
                [")"])
                ]
            end
          with | _ ->
            [
            cat_string_list_with_space
            ([fn;"->call("]@
            [cat_string_list_with_comma (List.fold_left (fun ret ex ->
ret@(handle_texpr ex refenv)) [] texpr_list)]@
            [")"])
            ]
          end
      )
   | TObjCall ((varname, mfname, texpr_list), ty) ->
      let res = search_key (!refenv) varname
      in begin match res with
      | Some x ->
         begin match (x:typ) with
         | Array _ ->
            if mfname = "sync" then
               ["std::unique_lock<std::recursive_mutex> lk(" ^ varname ^
"->v_mutex);"]
            else
            let arr_code_gen varname mfname texpr_list =
               let newfname = mfname
               in
               let fn = varname ^ "->" ^ newfname
               in
               [
                  cat_string_list_with_space
                  ([fn;"("]@
                  [cat_string_list_with_comma (List.fold_left (fun ret ex ->
ret@(handle_texpr ex refenv)) [] texpr_list)]@
                  [")"])
               ]
            in
               arr_code_gen varname mfname texpr_list
         | Map _ ->
            let map_code_gen varname mfname texpr_list =
               let normal_gen fn =
                  [
                  cat_string_list_with_space
```

```
                    ([fn;"("]@
                    [cat_string_list_with_comma (List.fold_left (fun ret ex ->
ret@(handle_texpr ex refenv)) [] texpr_list)]@
                    [")"])
                    ]
                in
                begin match mfname with
                | "get" ->
                    normal_gen (varname ^ "->operator[]")
                | "delete" ->
                    let epr = List.hd texpr_list
                    in
                    [varname ^ "->erase("
                        ^ (merge_string_list (handle_texpr epr refenv)) ^ ")"]
                | "exist" ->
                    let epr = List.hd texpr_list
                    in
                    ["( " ^ varname ^ "->find("
                        ^ (merge_string_list (handle_texpr epr refenv)) ^ ") != " ^
varname ^ "->end() )"]
                | "size" ->
                    (* change to int*)
                    ["int(" ^ (merge_string_list (normal_gen (varname ^"->size"))) ^ ")"]
                | "insert" ->
                    begin match texpr_list with
                    | [x;y] ->
                        let key_code = merge_string_list (handle_texpr x refenv)
                        in let value_code = merge_string_list (handle_texpr y refenv)
                        in
                        [varname ^ "->insert(" ^ key_code ^ "," ^ value_code ^ ")"]
                    | _ -> failwith ("not support for insert map")
                    end
                | "sync" ->
                    ["std::unique_lock<std::recursive_mutex> lk(" ^ varname ^
"->m_mutex);"]
                | _ -> failwith ("not support map function")
                end
            in
                map_code_gen varname mfname texpr_list
        | Signal(t) ->
            let fn = (
                match t with
                | Class (_) | Array (_) | Map (_) | Chan (_) -> varname ^ "->" ^
mfname
```

```
                | _ -> "*" ^ varname ^ "->" ^ mfname
            )
            in
            [
                cat_string_list_with_space
                ([fn;"("]@
                [cat_string_list_with_comma (List.fold_left (fun ret ex ->
ret@(handle_texpr ex refenv)) [] texpr_list)]@
                [")"])
            ]
          | _ ->
            let fn = varname ^ "->" ^ mfname
            in
            [
                cat_string_list_with_space
                ([fn;"("]@
                [cat_string_list_with_comma (List.fold_left (fun ret ex ->
ret@(handle_texpr ex refenv)) [] texpr_list)]@
                [")"])
            ]
          end
      | _ -> failwith ("inner error")
      end
   | TFunc(_) -> [] (* TODO *)
   | TAssign((str, expr), ty) ->
      let res = (search_key (!refenv) str) in
      (
         let type_code = type_to_code_string ty in
         let decl_type_code = if res = None then type_code else "" in
         ignore(update_env !refenv str ty);
         (
            match ty with
            (* deal with signal assignment from fly *)
            | Signal(x) ->
               (* flying a no return function is not allowed *)
               ignore(if x = Void then raise (Failure ("Function should return
something to signal")));
               (
               let type_str =
                  match x with
                  | Class(class_type) -> class_type
                  | Array(sometype) -> "flyvector<" ^ type_to_code_string sometype
^ ">"
                  | Map(t1, t2) -> "flymap<" ^ type_to_code_string t1 ^ "," ^
```

```
type_to_code_string t2 ^ ">"
                    | Chan(sometype) -> "Chan<" ^ type_to_code_string sometype ^ ">"
                    | _ -> type_to_code_string x
                in
                [decl_type_code ^ " " ^ str ^ " = " ^ type_code ^ "(new Signal<" ^
(type_str) ^ ">());";] @ handle_fly_expr str expr refenv
                )
            (* normal *)
            | _ -> [decl_type_code ^ " " ^ str ^ " = "] @ handle_texpr expr refenv
        )
    )
  | TListComprehen(_) -> [] (* TODO *)
  | TExec(_) -> [] (* TODO *)
  | TDispatch ((fname, texpr_list, ip, port), rtype) ->
    let texpr_types = List.map get_expr_type_info texpr_list
    in let key = gen_hash_key fname texpr_types
    in begin match (find_hash t_func_binds key) with
    | None -> failwith ("conflict with infer tdispatch")
    | Some (tfdecl) ->
      let code_str = merge_string_list (handle_fdecl key tfdecl (ref
(init_level_env())))
      in let wrap_dispatch_name = "_dispatch_" ^ key
      in
      let merge_texpr_list = texpr_list @ [ip;port]
      in
      let call_stmt =
      [
      cat_string_list_with_space
      ([wrap_dispatch_name;"("]@
      [cat_string_list_with_comma (List.fold_left (fun ret ex ->
ret@(handle_texpr ex refenv)) [] merge_texpr_list)]@
      [")"])
      ]
      in
      match (find_hash dispatch_funcs wrap_dispatch_name) with
      | None ->
        let inner_names =
["_dispatch_a";"_dispatch_b";"_dispatch_c";"_dispatch_d";"_dispatch_e";"_dispatc
h_f"]
        in let split_type = "string split_type = \"\\x01\";\n"
        in let split_var = "string split_var = \"\\x02\";\n"
        in let code_str_assign = "string code_string=\"" ^ fname ^ "\\x01" ^
code_str ^ "\";"
        in let varname_type_list = zip inner_names texpr_types
```

```
            in let encoding_assign = List.map (fun (varname, thistype) ->
               let assign_stmt = "string _str" ^ varname  ^ " = string(\""  ^
(type_to_code_string thistype) ^ "\")+split_type+_string("^ varname ^ ");\n"
               in assign_stmt )
            varname_type_list
            in let all_str = "code_string"::(List.map (fun (varname, _) -> "_str" ^
varname) varname_type_list)
            in let join_by_split = List.rev (List.fold_left (fun res item ->
item::"split_var"::res) [] all_str)
            in let packet_assign = "string _packet = " ^ (list_join join_by_split "+" ) ^
";"
            in let var_defs = (List.map (fun (varname, thistype) ->
(type_to_code_string thistype) ^ " " ^ varname) varname_type_list) @ ["string
ip";"int port"]
            in let var_defs_str = list_join var_defs ","
            in let func_def = (type_to_code_string rtype) ^ " " ^
wrap_dispatch_name ^ " (" ^ var_defs_str ^ "){\n"
            in let tablize_assign = tablize
(code_str_assign::split_type::split_var::encoding_assign) @[packet_assign]
            in let packet_send_conv =
               ["shared_ptr <client> _client =  shared_ptr <client>(new
client());\n";"shared_ptr <connection> _con =  _client->connect (ip,port);";
                  "string _msg;\nif (_con->is_alive ()  ) {_con->send (_packet);_msg =
_con->recv ();}"]
            in
            let return_stmt = begin match rtype with
            | Int -> "return _int(_msg);"
            | Array (Int) -> "return _vector_int(_msg);"
            | _ -> "just support int vector <int> now"
            end
            in
            Hashtbl.add dispatch_funcs wrap_dispatch_name (merge_string_list
(func_def::[List.fold_left (fun res item -> res ^ item) "" tablize_assign] @
packet_send_conv @[return_stmt;"}"]));
            call_stmt
          | Some x -> call_stmt
       end
   | TChangen(containtype, x) ->
      let containname = new_type_to_code_string containtype
      in ["shared_ptr < Chan <" ^ containname ^ "> >(new Chan < " ^
containname ^ " >())"]
   | TChanbinop((x, y), containtype) ->
      (* according to different type wrap or unwrap shared_ptr*)
      begin match containtype with
```

```
      | Int | String | Float ->
        let res = search_key (!refenv) y
        in begin match res with
        | Some (Chan (a)) -> (* push*)
          let checkx = search_key (!refenv) x
          in if checkx = None then
          [(type_to_code_string containtype) ^ " " ^ x ^ "=*(" ^ y ^
"->wait_and_pop())"]
          else
          [x ^ "=*(" ^ y ^ "->wait_and_pop())"]
        | Some (a) ->
          [x ^ "->push(make_shared<" ^ (type_to_code_string a) ^ ">(" ^ y ^ "))"]
        | None -> failwith ("conflict binop")
        end
      | Class (c) ->
        let res = search_key (!refenv) y
        in begin match res with
        | Some (Chan (a)) -> (* push*)
          let checkx = search_key (!refenv) x
          in if checkx = None then
          [(type_to_code_string containtype) ^ " " ^ x ^ "=" ^ y ^
"->wait_and_pop()"]
          else
          [x ^ "=" ^ y ^ "->wait_and_pop()"]
        | Some (a) ->
          [x ^ "->push(" ^ y ^ ")"]
        | None -> failwith ("conflict binop")
        end
      | _ -> failwith ("not support for other chan now")
      end
  | TChanunop(x, containtype) ->
      [x ^ "->wait_and_pop()"] (* TODO *)
  | TFly((fn, texpr_list),st) ->
      let type_str =
      match st with
        | Signal(Class(tstr)) -> tstr
        | Signal(Array(sometype)) -> "flyvector<" ^ type_to_code_string sometype
^ ">"
        | Signal(Map(t1,t2)) -> "flymap<" ^ type_to_code_string t1 ^ "," ^
type_to_code_string t2 ^ ">"
        | Signal(Chan(sometype)) -> "Chan<" ^ type_to_code_string sometype ^
">"
        | Signal(t) -> type_to_code_string t
        | _ -> raise (Failure ("Fly type error"))
```

```
      in
      handle_fly_expr ("shared_ptr <Signal<" ^ type_str ^ ">> (new Signal<" ^
type_str ^">())")
         (TFly((fn, texpr_list),st)) refenv
   | TRegister ((sign, fn, texpr_list), t) ->
      (* must change the function name, appending type *)
      handle_fly_expr sign (TFly((fn, texpr_list), Signal(t))) refenv
   | TFlyo(_) -> [] (* TODO *)
   | TNull(_) -> [] (* TODO *)
   | TObjGen (typename, _) ->
      begin
      match typename with
      | Class x -> ["shared_ptr <" ^ x ^ ">(new " ^ x ^ "())"]
      | Array _ | Map _ -> [type_to_code_string typename ^ "(new " ^
(new_type_to_code_string typename) ^ "())"]
      | _ -> failwith ("not support for other TObjgen now")
      end
   | TObjid((objname, objid), _) -> [objname ^ "->" ^ objid]
   | TMAssign ((varname, mname, expr), ty)->
      begin
         match ty with
         (* deal with signal assignment *)
         | Signal(x) ->
            let tycode = type_to_code_string x
            in let str = varname ^ "->" ^ mname
            in  [str ^ "=shared_ptr <Signal<" ^ tycode ^ ">>(new Signal<" ^ tycode ^
">());";] @
            handle_fly_expr str expr refenv
         (* normal *)
         | x ->
            let str = varname ^ "->" ^ mname
            in [str ^ " = "] @ handle_texpr expr refenv
      end
and handle_tstmt fkey tstmt_ refenv =
   let refnewenv = ref (append_new_level !refenv) in
   match tstmt_ with
   | TBlock(tstmtlist) ->
      ["{"] @
      (List.fold_left (fun ret tstmt_ -> ret @ (handle_tstmt fkey tstmt_ refnewenv))
[] tstmtlist)
      @ ["}"]
   | TExpr(expr) -> [cat_string_list_with_space ((handle_texpr expr refenv) @
[";"])]
   | TReturn(expr) -> [cat_string_list_with_space (["return"] @ (handle_texpr
```

```
expr refenv) @ [";"])]
    | TIf(texp_, tstmtl1, tstmtl2) ->
        [cat_string_list_with_space (["if ("] @ (handle_texpr texp_ refnewenv) @
[")"])] @
        ["{"] @ ((List.fold_left (fun ret tstmt_ -> ret @ (handle_tstmt fkey tstmt_
refnewenv)) [] tstmtl1)) @ ["}"] @
        ["else"] @
        ["{"] @ ((List.fold_left (fun ret tstmt_ -> ret @ (handle_tstmt fkey tstmt_
refnewenv)) [] tstmtl2)) @ ["}"]
    | TFor(exp1, exp2, exp3, tstmtlist) ->
        let f1 = handle_texpr exp1 refnewenv in
        let f2 = handle_texpr exp2 refnewenv in
        let f3 = handle_texpr exp3 refnewenv in
        let tstmtstr = (List.fold_left (fun ret tstmt_ -> ret @ (handle_tstmt fkey tstmt_
refnewenv)) [] tstmtlist) in
        [cat_string_list_with_space (["for ("] @ f1 @ [";"] @ f2 @ [";"] @ f3 @ [")"])] @
        ["{"] @ tstmtstr @ ["}"]
    | TForeach (varname, t_base_expr, tstmt_list) ->
        (*array is changed to TFor, now TForeach is only for the need of map*)
        (*deal with foreach map which can not be done by the change of the for*)
        let base_expr_code = handle_texpr t_base_expr refenv
        in
        let for_code = "for (auto itr = (" ^ (merge_string_list base_expr_code) ^
")->begin(); itr != ("
            ^ (merge_string_list base_expr_code) ^ ")->end(); ++itr){"
        in let var_type = begin match get_expr_type_info t_base_expr with
            | Map (x, y) -> x
            | _ -> failwith ("infer error for tforeach map")
            end
        in let assign_var_code = (type_to_code_string var_type) ^ " " ^ varname ^
"=itr->first;"
        in ignore(update_env !refnewenv varname var_type); (*do a env update*)
        let tstmtstr =  (List.fold_left (fun ret tstmt_ -> ret @ (handle_tstmt fkey
tstmt_ refnewenv)) [] tstmt_list)
        in
        let lock = "std::unique_lock<std::recursive_mutex> lk(" ^ merge_string_list
base_expr_code ^ "->m_mutex);"
        in ["{";lock;for_code;assign_var_code] @ tstmtstr @ ["}"] @ ["}"]
    | TWhile(expr_, tstmtlist) ->
        [cat_string_list_with_space (["while ("] @ (handle_texpr expr_ refnewenv)
@ [")"])] @
        ["{"] @
        (List.fold_left (fun ret tstmt_ -> ret @ (handle_tstmt fkey tstmt_ refnewenv))
[] tstmtlist) @
```

```
      ["}"]
   | TBreak ->
      ["break;"]
   | TContinue ->
      ["continue;"]

(* take tstmt list and return string list *)
(*对stmt list 产生code*)
and handle_body fkey body refenv =
   let refnewenv = ref (append_new_level !refenv) in
   let body_code = List.fold_left (fun ret tstmt_ -> ret @ (handle_tstmt fkey tstmt_
refnewenv)) [] body in
   if fkey = "main"
   then ["{\n___argc=argc;___argv=argv;\n"] @ body_code @ ["}"]
   else
   ["{"] @ body_code @ ["}"]

(* return string list *)
(* take a function key, declaration and generate the string list *)
and handle_fdecl fkey fd refenv =
   let refnewenv = ref (append_new_level !refenv) in
   match fd with
   | {tret=rt; tfname=name; tformals=fm; tbody=body ;_} ->
      if name = ""
      then []
      else
         let fmstr = handle_fm fm refnewenv in
         let bodystr = (handle_body fkey body refnewenv) in
         if name = "main" then
            (*tricy replacement for main func*)
            let new_fmstr = "(int argc, char **argv)"
            in [ cat_string_list_with_space [(type_to_code_string
rt);name;new_fmstr]] @ bodystr
         else
            [ cat_string_list_with_space [(type_to_code_string rt);name;fmstr]] @
bodystr


let code_header = ["
   #include<fly/util.h>
   #include<fly/cli.h>
"]

(* take a texp and return function key list *)
```

```
let rec texp_helper texp_ =
    match texp_ with
    | TBinop ((texpr1, _, texpr2), _) -> (texp_helper texpr1) @ (texp_helper texpr2)
    | TUnop ((_, texpr_), _) -> texp_helper texpr_
    | TCall ((fn, texprlist), _) ->
        (
        match fn with
        | "print" -> []
        (* above are built-in functions *)
        | _ ->
            let expr_types_list = List.map get_expr_type_info texprlist in
            let hash_key = gen_hash_key fn expr_types_list in
            [hash_key] @ (List.fold_left (fun ret exp_ -> ret @ (texp_helper exp_)) []
texprlist)
        )
    | TFly ((fn, texprlist), Signal(t)) ->
        ignore(
            let expr_types_list = List.map get_expr_type_info texprlist in
            let hash_key = gen_hash_key fn expr_types_list in
            add_hash signal_funcs hash_key ""
        );
        texp_helper (TCall((fn, texprlist), t))
    | TRegister ((sign, fn, texpl), t) ->
        let expr_types_list = List.map get_expr_type_info texpl in
        (* register texpl will miss the last t, so append it*)
        let hash_key = gen_hash_key fn (expr_types_list @ [t]) in
        ignore(add_hash register_funcs hash_key "");
        [hash_key]
    (* TObjCall of (string * string * texpr list) * typ TODO*)
    | TObjCall (_) -> []
    (* TFunc of (string list * texpr) * typ *) (* lambda TODO*)
    | TFunc (_) -> []
    (* TAssign of (string * texpr) * typ *)
    | TAssign ((_, e_), _) -> texp_helper e_
    (* TListComprehen of (texpr * string * texpr) * typ (*can iterate a tuple?*)
TODO*)
    | TListComprehen (_) -> []
    (* TExec of string * typ TODO*)
    | TExec (_) -> []
    (* TDispatch of (string * texpr list * string * string) * typ TODO *)
    | TDispatch (_) -> []
    (* TChan of texpr * typ TODO *)
    | TChangen(_) -> []
    (* TChanunop of string * typ TODO *)
```

```
    | TChanunop (_) -> []
    (* TChanbinop of (string * string) * typ TODO *)
    | TChanbinop (_) -> []
    (* | TFlyo of (string * string * texpr list) * typ TODO *)
    | TFlyo (_) -> []
    | _ -> []

(* take a tstmt and return function key list *)
let rec tstmt_helper tstmt_ =
    match tstmt_ with
    | TBlock(tstmtlist) -> List.fold_left (fun ret tstmt_ -> ret @ (tstmt_helper
tstmt_)) [] tstmtlist
    | TExpr(texpr_) -> texp_helper texpr_
    | TReturn(texpr_) -> texp_helper texpr_
    | TIf(texpr_, tstmtlist_a, tstmtlist_b) ->
        (texp_helper texpr_) @
        (List.fold_left (fun ret tstmt_ -> ret @ (tstmt_helper tstmt_)) [] (tstmtlist_a @
tstmtlist_b))
    | TFor(ex1, ex2, ex3, tstmtlist) ->
        (texp_helper ex1) @ (texp_helper ex2) @ (texp_helper ex3) @
        (List.fold_left (fun ret tstmt_ -> ret @ (tstmt_helper tstmt_)) [] tstmtlist)
    | TForeach(_, texpr_, tstmtlist) ->
        (texp_helper texpr_) @ (List.fold_left (fun ret tstmt_ -> ret @ (tstmt_helper
tstmt_)) [] tstmtlist)
    | TWhile(texpr_, tstmtlist) ->
        (texp_helper texpr_) @ (List.fold_left (fun ret tstmt_ -> ret @ (tstmt_helper
tstmt_)) [] tstmtlist)
    | TBreak -> []
    | TContinue -> []


(* take a function key and return string list, which are the code *)
let rec dfs ht fkey refenv =
    let hash_value = find_hash fundone fkey in
    match hash_value with
    | None ->
        let sfd = find_hash ht fkey in
        (
            match sfd with
            | None -> []
            | Some (fd) ->
                ignore(Hashtbl.add fundone fkey "dummy");
                (
                match fd with
```

```
            | {tbody=body; _} ->
              (*get all t_func_decl needed*)
              let fklist = List.fold_left (fun ret tstmt_ -> ret @ (tstmt_helper tstmt_))
[] body in
              (List.fold_left (fun ret key_ -> ret @ (dfs ht key_ refenv)) [] fklist) @
(handle_fdecl fkey fd refenv)
            )
      )
  | _ -> []

(*
let ht_left ht =
  Hashtbl.fold
  (fun k v ret ->
    let sfd = find_hash fundone k in
    match sfd with
    | None ->
      ignore(Hashtbl.add fundone k "dummy");
      (
        match v with
        | {tfname=name; _} -> print_string name
      );
      ret @ [v]
    | _ -> ret
  ) ht []
*)

let gen_rest ht refenv =
  let fcode = Hashtbl.fold
    (
      fun k v code ->
      match (find_hash fundone k) with
      | None ->
        ignore(add_hash fundone k "");
        code @ (handle_fdecl k v refenv)
      | _ ->
      code
    ) ht [] in
  fcode

(* generate signal wrappers *)
let gen_sig_wrapper ht =
  let g_env = init_level_env() in
  let sig_funcs = Hashtbl.fold
```

```
    (
      fun k v ret ->
      match find_hash fundone k with
      | None ->
        ignore(add_hash fundone k "");
        (
          match find_hash ht k with
          | Some(fd) -> ret @ (handle_fd_fly fd (ref g_env))
          | None -> raise (Failure (k ^ " not in fht"))
        )
      | _ -> ret
    ) signal_funcs [] in
    let regi_funcs = Hashtbl.fold
    (
      fun k v ret ->
      match find_hash fundone k with
      | None ->
        ignore(add_hash fundone k "");
        (
          match find_hash ht k with
          | Some(fd) -> ret @ handle_fd_register fd (ref g_env)
          | None -> raise (Failure (k ^ " not in fht"))
        )
      | _ -> ret
    ) register_funcs [] in
    sig_funcs @ regi_funcs

let handle_func_forward fd refenv =
    let refnewenv = ref (append_new_level !refenv) in
    match fd with
    | {tret=rt; tfname=name; tformals=fm;_} ->
      if name = ""
      then []
      else
        let fmstr =
          if name = "main" then
            "(int argc, char **argv)"
          else
          (handle_fm fm refnewenv)
        in
        [ cat_string_list_with_space [(type_to_code_string rt);name;fmstr;] ^ ";"]

let gen_forward ht refenv =
    Hashtbl.fold (fun k v code -> code @ (handle_func_forward v refenv)) ht []
```

```
let gen_sig_wrapper_forward ht refenv = []

(* take ht and return string list, which is code *)
let build_func_from_ht ht =
   let g_env = init_level_env() in
   let refenv = (ref g_env) in
   let res = dfs ht "main" refenv in
   let res2 = gen_rest ht refenv in
   let forward = gen_forward ht refenv in
   ignore(clean_up_hash fundone);
   let sig_wrapper_code = gen_sig_wrapper ht in
   let sig_wrapper_forward = gen_sig_wrapper_forward ht refenv in
   (forward @ sig_wrapper_forward, res2 @ sig_wrapper_code @ res)

(* take t_class_decl and return string list (code) of the class referrence *)
let handle_class_refer tcdecl = match tcdecl with
   | {tcname=cname;member_binds=binds;t_func_decls=tfdecls}->
      let class_header = "class " ^ cname ^ " {\npublic:\n" (*all public*)
      in let var_defs  =
         List.map (fun (varname, thistype) ->
            (type_to_code_string thistype) ^ " " ^ varname ^ ";"
         ) binds
      in let refer_map tfdecl = begin match tfdecl with
         | {tfname=fname;tformals=bind_list;tret=rtype;_} ->
         let var_refs = List.map (fun (varname, thistype) ->
            (type_to_code_string thistype) ^ " " ^ varname ^ ""
         ) bind_list
         in let fstr = list_join var_refs ","
         in
            (type_to_code_string rtype) ^ " " ^ fname ^ "(" ^ fstr ^ ");"
         end
      in let func_refers =
         List.map refer_map tfdecls
      in let tab_var_defs = tablize var_defs
      in let tab_func_refers = tablize func_refers
      in let end_lines = "};"
      in (*concate with \n*)
         let total = List.concat
[[class_header];tab_var_defs;tab_func_refers;[end_lines]]
      in [List.fold_left (fun res item -> res ^ item ^ "\n") "" total]


(* take t_class_decl and return string list (code) of the class definition *)
```

```
let handle_class_def tcdecl = match tcdecl with
    | {tcname=cname;member_binds=binds;t_func_decls=tfdecls}->
      let def_map tfdecl = begin match tfdecl with
          | {tfname=fname;tformals=bind_list;tret=rtype;_} ->
          let var_refs = List.map (fun (varname, thistype) ->
             (type_to_code_string thistype) ^ " " ^ varname ^ ""
          ) bind_list
          in let fstr = list_join var_refs ","
          in
             (type_to_code_string rtype) ^ " " ^ cname ^ "::" ^fname ^ "(" ^ fstr ^ ")"
          end
      in
      let gen_body tfdecl = begin match tfdecl with
          | {tfname=fname;tformals=bind_list;tret=rtype;tbody=stmt_list;_} ->
             (*I don't know what is fkey*)
             (*create a new env with member variables and parameters*)
             let new_env = List.fold_left (
                fun thisenv (varname, thistype) ->
                update_env thisenv varname thistype
                ) (init_level_env()) binds
             in let new_env = List.fold_left (fun thisenv (varname, thistype) ->
                update_env thisenv varname thistype) new_env bind_list
             in let new_env_ref = ref(new_env)
             in handle_body "" stmt_list new_env_ref
             end
      in let gen_all tfdecl =
         let func_def = def_map tfdecl
         in let body = gen_body tfdecl
         in [func_def] @ body
      in List.concat (List.map gen_all tfdecls)


let handle_class_forward tcdecl = match tcdecl with
     | {tcname=cname;member_binds=binds;t_func_decls=tfdecls}->
       ["class " ^ cname ^ ";\n"]

(* take ht of string->class_decl and return string list *)
let build_class_from_ht cht =
   (*first generate forward decl*)
   let code_fw = Hashtbl.fold (fun k v code -> code @ (handle_class_forward v))
cht []
   in let code_v2 = Hashtbl.fold (fun k v code -> code @ (handle_class_refer v))
cht code_fw
   in (code_fw, Hashtbl.fold (fun k v code -> code @ (handle_class_def v)) cht
```

```
code_v2)

let build_clojure_class clojure_classes =
   (*first generate forward decl*)
   let code_v1 = List.fold_left (fun code (k, v) -> code @ (handle_class_forward v))
[] clojure_classes
   in let code_v2 = List.fold_left (fun code (k, v) -> code @ (handle_class_refer v))
[] clojure_classes
   in let code_v3 = List.fold_left (fun code (k,v) -> code @ (handle_class_def v)) []
clojure_classes
   in (code_v1, code_v2, code_v3)

let codegen fht cht clojure_calls func_binds t_func_binds =
   let clojure_classes = gen_clojure_classes clojure_calls func_binds t_func_binds
in
   let (clojure_class_forwards, clojure_class_refers, clojure_class_defs)=
build_clojure_class clojure_classes in
   let (forward_codelist, func_codelist) = build_func_from_ht fht in
   let (class_fw, class_def) = build_class_from_ht cht in
   let dispatch_code = gen_dispatch_code()
   in
   let buffer = code_header @ build_in_code @ build_in_class_code @
clojure_class_forwards @ class_fw @ forward_codelist
   @ clojure_class_refers @ clojure_class_defs @ class_def @dispatch_code
@func_codelist in
   List.fold_left (fun ret ele -> ret ^ ele ^ "\n") "" buffer
```

debug.ml

```
(*below are some debuging function to show some sub-tree of ast
   TODO modified when writing our codes*)
open Ast
open Sast

let string_of_op = function
    | Add -> "add"
    | Sub -> "sub"
    | Mult -> "mul"
    | Mod -> "mod"
    | Div -> "div"
```

```
    | Equal -> "equal"
    | Neq -> "neq"
    | Less -> "less"
    | Leq -> "less or equal"
    | Greater -> "greater"
    | Geq -> "greater or equal"
    | And -> "and"
    | Or -> "or"
    | RArrow -> "->"
    | LArrow -> "<-"
    | SAdd -> "add string"

let string_of_uop = function
    | Neg -> "neg"
    | Not -> "not"

let rec debug_expr = function
    | Null a -> "null:" ^ a
    | Literal a ->"a integer:" ^ (string_of_int a)
    | BoolLit a -> if a = true then "a bool:true" else "a bool:false"
    | Float a -> "a float:" ^ (string_of_float a)
    | Id a -> "an id:" ^ a
    | Set a -> "a set:" ^ (List.fold_left (fun res item -> res ^ "," ^ debug_expr item) ""
a)
    | Map a -> "a map:" ^ (List.fold_left (fun res (item1, item2) -> res ^ ",k:"
^(debug_expr item1)^ "_v:" ^(debug_expr item2)) "" a)
    | Array a -> "an array:" ^ (List.fold_left (fun res item -> res ^ "," ^ debug_expr
item) "" a)
    | String a -> "a string:" ^ a
    | Binop (a, op, b) -> "binop:" ^ (string_of_op op) ^ "_left:" ^ (debug_expr a) ^
"_right:" ^ (debug_expr b)
    | Unop (uop, a) -> "unop:" ^ (string_of_uop uop) ^ "expr:" ^ (debug_expr a)
    | Call (id, exprs) -> "call: " ^ id ^ "_" ^ (List.fold_left (fun res item -> res ^ "," ^
(debug_expr item)) "" exprs);
    | ObjCall (id1, id2, exprs) -> "call by" ^ id1 ^ "." ^ id2 ^ (List.fold_left (fun res
item -> res ^ "," ^ (debug_expr item)) "" exprs)
    | Func (a, b) -> "lambda:" ^ (List.fold_left (fun res item -> res ^ "," ^ item) "" a)
^ "lambda expr:" ^ (debug_expr b)
    | Assign (a, b) -> "assign: " ^ a ^ " by:" ^ (debug_expr b)
    | ListComprehen (a, b, c) -> "list comprehension:" ^ (debug_expr a) ^ b ^
(debug_expr c)
    | Noexpr -> "no expression"
   (*network specified exprs*)
    | Exec(a) -> "exec: " ^ a
```

```
    | Dispatch(a, exprs, b, c) -> "dispatch: "
    | Register (a, b, exprs) -> "register: " ^ a ^ " " ^ b  ^ " " ^ ( List.fold_left (fun str
item -> str ^ "," ^ item) "" (List.map debug_expr exprs))
    | Chanunop (a) -> "chaunop: " ^ a
    | Chanbinop (a, b) -> "chanbinop: " ^ a ^ " " ^ b
    | Fly (a, exprs) -> "fly: " ^ a ^ " " ^ ( List.fold_left (fun str item -> str ^ "," ^ item)
"" (List.map debug_expr exprs)  )
    | Flyo (a, b, exprs) -> "flyo: " ^ a ^ " " ^ b ^ " " ^ ( List.fold_left (fun str item ->
str ^ "," ^ item) "" (List.map debug_expr exprs) )
    | _ -> "not implemented"

let rec debug_stmt = function
     Block stmts -> "block:" ^ (List.fold_left (fun acc item -> acc ^ "," ^ (debug_stmt
item)) "" stmts)
    | Expr a -> "expr:" ^ (debug_expr a)
    | Return a -> "return: " ^ (debug_expr a)
    | If (a, stmts1, stmts2) -> "if:" ^ (debug_expr a) ^ " " ^ (List.fold_left (fun acc
item -> acc ^ "," ^ (debug_stmt item)) "" stmts1) ^ " " ^ (List.fold_left (fun acc item
-> acc ^ "," ^ (debug_stmt item)) "" stmts2)
    | For (a, b, c, stmts) -> "for:" ^ (debug_expr a) ^ " " ^ (debug_expr b) ^ " " ^
(debug_expr c) ^ " " ^ ( List.fold_left (fun acc item -> acc ^ "," ^ item) "" (List.map
debug_stmt stmts) )
    | Foreach (a, expr, stmts) -> "for each:" ^ a ^ " " ^ (debug_expr expr) ^ " " ^ (
List.fold_left (fun acc item -> acc ^ "," ^ item) "" (List.map debug_stmt stmts) )
    | While (expr, stmts) -> "while:" ^ (debug_expr expr) ^ " " ^ ( List.fold_left (fun
acc item -> acc ^ "," ^ item) "" (List.map debug_stmt stmts) )
    | Break -> "break"
    | Continue -> "continue"

let debug_fdecl (fdecl : func_decl) = match fdecl with
    | {fname=name; body=stmts; formals=param_list} ->
       "function name:" ^ name ^ ", " ^ "params:" ^ (List.fold_left (fun res item ->
res ^ "," ^ item) "" param_list)
       ^ ", body:" ^ (List.fold_left (fun res item -> res ^ "," ^ (debug_stmt item)) ""
stmts)

(*debug for class definition*)
let debug_cdecl cdecl = match cdecl with
    | {cname=name;member_binds=binds;_} ->
       "class name:" ^ name ^ "\n" ^ "each member variables:\n"
          ^ (List.fold_left (fun res (mname, mtype) -> res ^ mname ^ ":" ^
(type_to_string mtype) ^ "\n") "" binds)

(*debug for typed things*)
```

```
let rec debug_texpr = function
     TLiteral a -> "literal: " ^ (string_of_int a)
   | TBoolLit a -> if a = true then "bool: true" else "bool: false"
   | TNull a -> "null with type:" ^ (type_to_string a)
   | TFloat a -> "float: " ^ (string_of_float a)
   | TId (name, this_type) -> "id:" ^ name ^ "_withtype_" ^ type_to_string
this_type
         | TSet (a, this_type) -> "set: " ^ (List.fold_left (fun res item -> res ^ "," ^
debug_texpr item) "" a) ^ "_withtype_" ^ type_to_string this_type
         | TMap (a, this_type) -> "map: " ^ (List.fold_left (fun res (item1, item2) ->
res ^ ",k:" ^(debug_texpr item1)^ "_v:" ^(debug_texpr item2)) "" a) ^ "_withtype_"
^ type_to_string this_type
         | TArray(a, this_type) -> "array: " ^ (List.fold_left (fun res item -> res ^ ","
^ debug_texpr item) "" a) ^ "_withtype_" ^ type_to_string this_type
         | TString a -> "string:" ^ a
   | TBinop (binop, this_type) -> (fun (a, op, b) -> "binop: " ^ (string_of_op op) ^ ",
left operand: " ^ (debug_texpr a) ^ ", right operand: " ^ (debug_texpr b)) binop ^
", _withtype_" ^ type_to_string this_type
   | TUnop (unop, this_type) -> (fun (uop, a) -> "unop: " ^ (string_of_uop uop) ^ ",
expr: " ^ (debug_texpr a)) unop ^ "_withtype_" ^ type_to_string this_type
   | TCall(a, this_type) -> (fun (id, texprs) -> "call_" ^ id ^ ": " ^ (List.fold_left (fun
res item -> res ^ (debug_texpr item) ^ ", ") "" texprs) ) a ^ "_withtype_" ^
type_to_string this_type
   | TObjCall (a, this_type) -> (fun (id1, id2, texprs) -> "call by: " ^ id1 ^ "." ^ id2 ^
(List.fold_left (fun res item -> res ^ "," ^ (debug_texpr item)) "" texprs) ) a ^ ",
withtype_" ^ type_to_string this_type
   | TFunc (args, this_type) -> (fun (a, b) -> "lambda:" ^ (List.fold_left (fun res
item -> res ^ "," ^ item) "" a) ^ "lambda expr:" ^ (debug_texpr b) ) args ^
"_withtype_" ^ type_to_string this_type
   | TAssign (args, this_type) -> (fun (a, b) -> "assign " ^ a ^ " by " ^ (debug_texpr
b) ) args ^ "_withtype_" ^ type_to_string this_type
   | TListComprehen(args, this_type) -> (fun (a, b, c) -> "list comprehension: " ^
(debug_texpr a) ^ b ^ (debug_texpr c) ) args ^ "_withtype_" ^ type_to_string
this_type
   | TExec (a, this_type) -> "exec: " ^ a ^ "_withtype_" ^ type_to_string this_type
   | TDispatch (args, this_type) -> (fun (a, exprs, b, c) -> "dispatch: " ^ a ^
(List.fold_left (fun str item -> str ^ "," ^ (debug_texpr item)) "" exprs) ) args ^
"_withtype_" ^ type_to_string this_type
   | TRegister (args, this_type) -> (fun (a, b, exprs) -> "register: " ^ a ^ " " ^ b  ^ " "
^ ( List.fold_left (fun str item -> str ^ "," ^ item) "" (List.map debug_texpr exprs))
) args ^ "_withtype_" ^ type_to_string this_type
   | TChanunop (a, this_type) -> "chaunop: " ^ a ^ "_withtype_" ^ type_to_string
this_type
   | TChanbinop (args, this_type) -> (fun (a, b) -> "chanbinop: " ^ a ^ " " ^ b ) args
```

```
^ "_withtype_" ^ type_to_string this_type
    | TFly (args, this_type) -> (fun (a, exprs) -> "fly: " ^ a ^ " " ^ ( List.fold_left (fun
str item -> str ^ "," ^ item)  "" (List.map debug_texpr exprs)) ) args ^ "_withtype_"
^ type_to_string this_type
    | TFlyo (args, this_type) -> (fun (a, b, exprs) -> "flyo: " ^ a ^ " " ^ b ^ " " ^ (
List.fold_left (fun str item -> str ^ "," ^ item) "" (List.map debug_texpr exprs)) )
args^ "_withtype_" ^ type_to_string this_type
    | _ -> raise (Failure ("debug_texpr not yet support this type"))


(*debug for typed stmts*)
let rec debug_tstmt = function
      TBlock tstmts -> "block: " ^ ( List.fold_left (fun acc item -> acc ^ "," ^ item) ""
(List.map debug_tstmt tstmts) )
    |  TExpr a -> "expr: " ^ (debug_texpr a)
    |  TReturn a -> "return: " ^ (debug_texpr a)
    |  TIf (a, tstmts1, tstmts2) -> "if: " ^ (debug_texpr a) ^ " " ^ (List.fold_left (fun
acc item -> acc ^ "," ^ (debug_tstmt item)) "" tstmts1) ^ " " ^ (List.fold_left (fun
acc item -> acc ^ "," ^ (debug_tstmt item)) "" tstmts2)
    |  TFor (a, b, c, tstmts) -> "for: " ^ (debug_texpr a) ^ " " ^ (debug_texpr b) ^ " " ^
(debug_texpr c) ^ " " ^ ( List.fold_left (fun acc item -> acc ^ "," ^ item) ""
(List.map debug_tstmt tstmts) )
    |  TForeach (a, texpr, tstmts) -> "for each: " ^ a ^ " " ^ (debug_texpr texpr) ^ " "
^ ( List.fold_left (fun acc item -> acc ^ "," ^ item) "" (List.map debug_tstmt tstmts)
)
    |  TWhile (texpr, tstmts) -> "while: " ^ (debug_texpr texpr) ^ " " ^ (
List.fold_left (fun acc item -> acc ^ "," ^ item) "" (List.map debug_tstmt tstmts) )
    | TBreak -> "break"
    | TContinue -> "continue"

(*debug for a typed function call*)
let debug_t_fdecl (tfdecl: t_func_decl) = match tfdecl with
    | {ttkey=key; tfname=name; tformals=param_list; tbody=tstmts; tret=return}
->
       "KEY: " ^ key ^ "\n"
       ^ "FUNCTION NAME: " ^ name ^ "\n"
       ^ "PARAMS:\n" ^ ( List.fold_left (fun acc (str, typ) -> acc ^ "str:" ^ str ^
"_type:" ^ (type_to_string typ) ^ "\n" ) "" param_list )
       ^"BODY:\n" ^ (List.fold_left (fun acc item -> acc ^ (debug_tstmt item) ^ ",\n")
"" tstmts)
       ^ "RETURN TYPE: " ^ (type_to_string return)

let debug_t_lambda_decl (tldecl: t_lambda_decl) = match tldecl with
    | {ltkey=key; ltfname=name; ltbinds=bind_list; ltformals=param_list;
```

```
ltbody=tstmts; ltret=return} ->
        "KEY: " ^ key ^ "\n"
        ^ "FUNCTION NAME: " ^ name ^ "\n"
        ^ "BINDS:\n" ^ ( List.fold_left (fun acc (str, typ) -> acc ^ "str:" ^ str ^ "_type:"
^ (type_to_string typ) ^ "\n") "" bind_list )
        ^ "PARAMS:\n" ^ ( List.fold_left (fun acc (str, typ) -> acc ^ "str:" ^ str ^
"_type:" ^ (type_to_string typ) ^ "\n") "" param_list )
        ^ "BODY:\n" ^ (List.fold_left (fun acc item -> acc ^ (debug_tstmt item) ^ ",\n")
"" tstmts)
        ^ "RETURN TYPE: " ^ (type_to_string return)
```

class.h

```
template <typename T> class Signal {
   public:
      condition_variable data_cond;
      mutex data_mutex;
      queue <std::shared_ptr <T>> data_queue;
      shared_ptr <T> wait() {
         std::unique_lock<std::mutex> lk(data_mutex);
         data_cond.wait(lk, [this]{return !this->data_queue.empty();});
         auto result = data_queue.front();
         data_queue.pop();
         return result;
      }
      void notify(std::shared_ptr <T> res) {
         std::lock_guard<std::mutex> lk(data_mutex);
         data_queue.push(res);
         data_cond.notify_one();
      }
};

template <typename T> class Chan {
private:
   mutable std::mutex mut;
   std::queue <std::shared_ptr <T>> data_queue;
   std::condition_variable data_cond;
public:
   Chan(){
```

```
    }
    std::shared_ptr <T> wait_and_pop() {
        std::unique_lock <std::mutex> lk(mut);
        data_cond.wait(lk, [this]{return !data_queue.empty();});
        std::shared_ptr <T> res = data_queue.front();
        data_queue.pop();
        return res;
    }
    void push(std::shared_ptr <T> tmp) {
        std::lock_guard <std::mutex> lk(mut);
        data_queue.push(tmp);
        data_cond.notify_one();
    }
};

template<class T>
class flyvector
{
    std::vector<T> v;
public:
    std::recursive_mutex v_mutex;

    flyvector () {
    }

    flyvector (istream_iterator<T> first, istream_iterator<T> last) {
        v.assign(first, last);
    }

    int size() {
        std::unique_lock<std::recursive_mutex> lk(v_mutex);
        return v.size();
    }
    T& get_at (const int& n) {
        std::unique_lock<std::recursive_mutex> lk(v_mutex);
        return v[n];
    }
    void set_at (const int& n, const T& val) {
        std::unique_lock<std::recursive_mutex> lk(v_mutex);
        v[n] = val;
    }

    void push_back (const T& val) {
        std::unique_lock<std::recursive_mutex> lk(v_mutex);
```

```
      v.push_back(val);
   }
   void push_vec (shared_ptr < flyvector <T> > another) {
      int l = another->size();
      for (int i = 0; i < l; ++i) {
         push_back(another->get_at(i));
      }
   }
};

shared_ptr<flyvector<string>> str_split(string s) {
   stringstream ss(s);
   istream_iterator<string> begin(ss);
   istream_iterator<string> end;
   return shared_ptr<flyvector<string>>( new flyvector<string>(begin, end));
}


template<class K, class V>
class flymap
{
   std::map<K,V> m;
public:
   std::recursive_mutex m_mutex;

   int size() {
      std::unique_lock<std::recursive_mutex> lk(m_mutex);
      return m.size();
   }

   V& operator[] (const K& k) {
      std::unique_lock<std::recursive_mutex> lk(m_mutex);
      return m[k];
   }

   void erase (const K& k) {
      std::unique_lock<std::recursive_mutex> lk(m_mutex);
      m.erase(k);
   }

   typename std::map<K,V>::iterator find (const K& k) {
      std::unique_lock<std::recursive_mutex> lk(m_mutex);
      return m.find(k);
   }
```

```
    typename std::map<K,V>::iterator end (void) {
        return m.end();
    }

    typename std::map<K,V>::iterator begin (void) {
        return m.begin();
    }

    void insert (const K& k, const V& v) {
        std::unique_lock<std::recursive_mutex> lk(m_mutex);
        m[k] = v;
    }
};

class connection {
private:
    int c_sock = -1;
    FILE *c_fp = NULL;
    bool alive;
public:
    connection(int c, bool al): c_sock(c), alive(al) {};
    string recv();
    bool send(string s);
    void close();
    bool is_alive();
};

bool connection::is_alive() {
    return alive;
}

bool connection::send(string msg) {
    if (c_sock < 0) {
        cout << "connection::send wrong socket " << c_sock << endl;
        alive = false;
        return false;
    }

    msg += "\n";

    int len = msg.length();
    if (::send(c_sock, msg.c_str(), len, 0) != len) {
        cout << "connection::send fail " << endl;
```

```
        alive = false;
        return false;
    }

    return true;
}

string connection::recv() {

    string rmsg;
    char requestLine[1024] = {0};

    if (!c_fp) {
        if (c_sock < 0) {
            cout << "connection::recv wrong socket " << c_sock << endl;
            return rmsg;
        }
        c_fp = fdopen(c_sock, "r");
    }

    if (c_fp == NULL) {
        alive = false;
        return rmsg;
    }

    if (!fgets(requestLine, sizeof(requestLine), c_fp)) {
        alive = false;
        return rmsg;
    }

    rmsg = string(requestLine);
    rmsg.pop_back();

    return rmsg;
}

void connection::close() {
    fclose(c_fp);
    c_fp = NULL;
    c_sock = -1;
    alive = false;
}

class server {
```

```
private:
   int create_server_socket(unsigned short port);
   int serv_sock = 0;
public:
   void listen(int port);
   shared_ptr<connection> accept(void);
};

int server::create_server_socket(unsigned short port)
{
   int servSock;
   struct sockaddr_in servAddr;

   /* Create socket for incoming connections */
   if ((servSock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0) {
      cout << "socket() failed" << endl;
      exit(1);
   }

   /* Construct local address structure */
   memset(&servAddr, 0, sizeof(servAddr));       /* Zero out structure */
   servAddr.sin_family = AF_INET;                /* Internet address family */
   servAddr.sin_addr.s_addr = htonl(INADDR_ANY); /* Any incoming interface */
   servAddr.sin_port = htons(port);              /* Local port */

   /* Bind to the local address */
   if (bind(servSock, (struct sockaddr *)&servAddr, sizeof(servAddr)) < 0) {
      cout << "bind() failed" << endl;
      exit(1);
   }

   /* Mark the socket so it will listen for incoming connections */
   if (::listen(servSock, 5) < 0) {
      cout << "listen() failed" << endl;
   }

   return servSock;
}

void server::listen(int port) {
   signal(SIGPIPE, SIG_IGN);
   serv_sock = create_server_socket(port);
}
```

```
shared_ptr<connection> server::accept(void) {

   struct sockaddr_in clntAddr;
   unsigned int clntLen = sizeof(clntAddr);

   int c_sock = ::accept(serv_sock, (struct sockaddr *)&clntAddr, &clntLen);

   return shared_ptr<connection>(new connection(c_sock, true));
}

class client {
public:
   shared_ptr<connection> connect(string server_ip, int port) {

      int sockfd;
      struct sockaddr_in serv_addr;

      shared_ptr<connection> fail(new connection(sockfd, false));

      if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
      {
         printf("Error : Could not create socket ");
         return fail;
      }

      memset(&serv_addr, '0', sizeof(serv_addr));
      serv_addr.sin_family = AF_INET;
      serv_addr.sin_port = htons((short)port);

      if (inet_pton(AF_INET, server_ip.c_str(), &serv_addr.sin_addr)<=0)
      {
         printf(" inet_pton error occured");
         return fail;
      }

      if ( ::connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0)
      {
         printf(" Error : Connect Failed ");
         return fail;
      }

      return shared_ptr<connection> (new connection(sockfd, true));
   }
};
```

```
vector <string> split(string str, char split_c) {
    int l = int(str.length());
    int last = -1;
    vector <string> res;
    for (int i = 0; i <= l; ++i) {
        if (i == l || str[i] == split_c) {
            if(i - last -1 > 0) {
                res.push_back(str.substr(last+1, i - last - 1));
            }
            last = i;
        }
    }
    return res;
}
shared_ptr < flyvector <int> > _vector_int(string msg) {
    vector <string> vector_str =  split(msg, ',');
    shared_ptr < flyvector <int> > res(new flyvector <int>());
    for (int i = 0; i < vector_str.size(); ++i) {
        res->push_back(_int(vector_str[i]));
    }
    return res;
}
```

cli.h

```
int ___argc;
char ** ___argv;
#include <string>
std::string get_argv(int i) {
    if (i <= ___argc) {
        return ___argv[i];
    }
    else {
        return "";
    }
}
```

exec.h

```
const char split_var = '\x02';
const char split_type = '\x01';
#include<iostream>
#include<vector>
#include<string>
#include<fstream>
#include<string>
#include<cstdlib>
using namespace std;


string join(vector <string> v, string split) {
    string res = "";
    int l = v.size();
    for (int i = 0; i < l; ++i) {
        if (i == 0) {
            res += v[i];
        }
        else {
            res += split;
            res += v[i];
        }
    }
    return res;
}
string exec(string str, string filename) {
    vector <string> func_and_param = split(str, split_var);
    vector <string> func = split(func_and_param[0], split_type);
    vector <string> headers = {"#include <fly/util.h>","#include <fly/func.h>",
"#include <fly/class.h>","#include <fly/fly.h>", "using namespace std;"};

    string header = join(headers, "\n");

    string main_func = "int main(){";
    int l = func_and_param.size();
    vector <string> assigns;
    for (int i = 1; i < l; ++i) {
        vector <string> type_and_content = split(func_and_param[i], split_type);
        std::cout << type_and_content[0] << std::endl;
        if(type_and_content[0] == "int") {
            string tmp = "int ";
            tmp += i + 'a';
            tmp += " = ";
```

```
            tmp += type_and_content[1];
            tmp += ";";
            assigns.push_back(tmp);
        }
        else if (type_and_content[0] == "shared_ptr < flyvector <int> >"){
            string tmp = "shared_ptr < flyvector <int> > ";
            tmp += i + 'a';
            tmp += " = ";
            tmp += "_vector_int(\"" + type_and_content[1] + "\")";
            tmp += ";";
            assigns.push_back(tmp);
        }
        else {

        }
    }
    ofstream ofs;
    ofs.open((filename + "_in.cpp").c_str(), ios::out);
    ofs << header << endl;
    ofs << func[1] << endl;
    ofs << main_func << endl;
    for (int i = 0; i < assigns.size(); ++i)
        ofs << "\t" << assigns[i] << endl;
    //exec
    ofs << "std::cout << _string(" << func[0] << "(";
    for (int i = 1; i < l; ++i) {
        if(i == 1) {
            ofs << char(i + 'a');
        }
        else {
            ofs << "," << char(i + 'a');
        }
    }
    ofs << "));" << endl;
    ofs << "}" << endl;
    ofs.close();
    //execute
    string input_file = filename + "_in.cpp";
    string output_file = filename + "_out";
    system(("g++ -std=c++11 -pthread -o " + filename + " " + input_file).c_str());
    system(("./" + filename + " > " + output_file).c_str());
    ifstream in;
    in.open(output_file.c_str(), ios::in);
    string res;
```

```
    in >> res;
    in.close();
    return res;
}
```

fly.h

```
string _string(shared_ptr <flyvector <int> > v) {
    int l = v->size();
    string res = "";
    for (int i = 0; i < l; ++i) {
        if (i == 0) {
            res += _string(v->get_at(i));
        }
        else {
            res += ",";
            res += _string(v->get_at(i));
        }
    }
    return res;
}
```

func.h

```
bool str_is_int(const string & s)
{
  if(s.empty() || ((!isdigit(s[0])) && (s[0] != '-') && (s[0] != '+'))) return false ;

  char * p ;
  strtol(s.c_str(), &p, 10) ;

  return (*p == 0) ;
}

int len(string a) {
    return a.length();
}

void print_bool(bool a) {
```

```
    std::stringstream stream;
    stream << a << endl;
        cout << stream.str();
}
void print(int a) {
    std::stringstream stream;
    stream << a << endl;
        cout << stream.str();
}
void print(size_t a) {
    print((int)a);
}
void print(string a) {
    std::stringstream stream;
    stream << a << endl;
        cout << stream.str();
}
void print(float a) {
    std::stringstream stream;
    stream << a << endl;
        cout << stream.str();
}
float _float(int a){
        return float(a);
}
float _float(string a) {
        return stof(a);
}

int _int(string a){
    char * p;
    return strtol(a.c_str(), &p, 10);
}

string _string(int a ){
        return to_string(a);
}
string _string(float a ){
        return to_string(a);
}
string _string(string a) {
    return a;
}
void _sleep(int seconds){
```

```
        std::chrono::seconds duration(seconds);
        std::this_thread::sleep_for(duration);
}

void _exit(int exit_code){
        exit(exit_code);
}
```

util.h

```
#include <sstream>
#include <iostream>
#include <iterator>
#include <string>
#include <string.h>     /* for memset() */
#include <thread>
#include <vector>
#include <map>
#include <mutex>
#include <condition_variable>
#include <queue>
#include <sys/socket.h>
#include <arpa/inet.h>  /* for sockaddr_in and inet_ntoa() */
#include <stdlib.h>     /* for atoi() and exit() */
#include <unistd.h>     /* for close() */
#include <signal.h>     /* for signal() */

using namespace std;
```