Fly Language

Project Proposal

Shenlong Gu, Hsiang-Ho Lin, Carolyn Sun, Xin Xu

sg3301, hl2907, cs3101, xx2168

[Motivation](#)

[Description](#)

[Features](#)

[Syntax](#)

- [Primitive Data Types](#)
- [Supported Data Types](#)
- [Basic Keywords](#)
- [Functional Syntax](#)
- [Network and Distribute Syntax](#)
  - [Basic Types](#)
  - [Keywords](#)

[Sample Code](#)

- [Basic Syntax](#)
- [Network Application](#)

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 paradigm, which we believe will significantly empower developers in tackling the challenges in the realm of distributed systems.

Description

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.

Features

- Concurrency primitives to create light-weight threads and synchronized FIFO buffers
- Event-driven primitives for asynchronous network request handling
- Capability for code to be distributed and executed across systems
- Support for syntax in common functional programming language such as lambda, pattern matching, map and fold

Syntax

Primitive Data Types

Name	Description
int	Integer
char	Character or small integer
bool	Boolean value. It can take one of two values: true or false
float	Single precision 32-bit floating point number
double	Double precision 64-bit floating point number
null	null represents the absence of data Ex: if item1 == null {

	//statements }

Supported 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 unique elements. Elements are enclosed in two dollar signs.	set1 = $”a”, ”b”,”c”$; set1.add(“d”); set1.find(“a”);

Basic Keywords

Keywords	Syntax

class Used for class declaration. It is the same as what it is in C++.	class MyClass{ //class body }

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.	if () {} else if {} else {}
/* */ **//** Provides ways to comment codes. The first is "C-style" or "multi-line" comment. The second is “C++-style" or "single-line" comment.	**/*** comment ***/** **//** comment\n

start
This is the keyword like the main in C++ which indicates an entry point for the program.
	start {
	  //statements
	}

func
Used for function declaration. The function name follows the func keyword. The parameters are listed in the parenthesis. Body of the function goes after |.
	func abc(type, msg) | type == “receive” { //guard }
	
	func abc(count, msg) | count > 2 { //guard }
	
	func abc(type, x:xs) { //pattern match }

Functional Syntax

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 Defining computation by case analysis.	match expression with | $pattern_1$ -> $expression_1$ | $pattern_2$ -> $expression_2$ | $pattern_3$ -> $expression_3$ 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);

Network and Distribute Syntax

Basic 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	s = fly func1(a, b); register s send_back;

inside the routine of another thread, the callback function being binded will be executed.	/* which means after func1(a,b) executed, the result will be sent to the function send_back to be executed */

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. We bind a closure with a signal, and when this signal is triggered, the closure is executed asynchronously.	register s send_back_to_client(server);
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);

Sample Code

Basic Syntax

```
//basic syntax
func gcd(a, b) | b == 0{
    return a;
}

func gcd(a, b) | b > a{
    return gcd(b, a);
}

func gcd(a, b) {
    return gcd(b, a % b);
}

start {
    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) {
    while (true) {
        time.sleep(1);
        a <- 3;
    }
}

void consume(a) {
    while (true) {
        b <- a;
        print(b);
    }
}

start {
    a = chan(int);
    fly produce(a);
    fly consume(a);
    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) | msg == "abc" {
    a = 1..10;
    b = [x + 1 | x <- a];
    return json.encode(b);
}
func deal_msg(msg) | msg == "def" {
    return "undefined";
}

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