Fly Language

Language Reference Manual

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

sg3301, hl2907, cs3101, xx2168

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++. Fly also

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 unique elements. Elements are enclosed in two dollar signs.	set1 = Set(“a”, “b”, “c”); set1.add(“d”); 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. 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	exec string;

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.	

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	map(function, list);

Applying a function to every element in the list, which returns a 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);

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) {
    }
}
```