Chapter 1

Language Tutorial

1.1 Setup

1.1.1 install dependencies

```
Download libuv from the following link: http://dist.libuv.org/dist/v1.11.0/
Follow the instructions to install:
 sudo apt-get install automake
 sudo apt-get install libtoolize
 ./autogen.sh
 ./configure
 make
 make check
 sudo make install
 echo "export LD_LIBRARY_PATH=\"/usr/local/lib\"" >> ~/.bashrc
 source ~/.bashrc
```

1.1.2 install ocaml

```
add-apt-repository ppa:avsm/ppa
apt-get update
apt-get install ocaml ocaml-native-compilers camlp4-extra opam
```

1.1.3 make pipeline executable

```
navigate to the src folder
enter command 'make'
```

1.1.4 run tests

```
navigate to the src folder
enter command 'python testall.py'
```

1.1.5 Run files with the pipeline executable as follows:

```
./pipeline -<flag> <filename.pl>
where <flag> can be any of the following
    ("-a", Ast);        (* Print the AST *)
    ("-t", Translate);  (* Translate the program to output c file *)
    ("-c", Compile);    (* Compile the output c file *)
    ("-r", Run);        (* Run the program *)
    ("-d", Compare) ]   (* Compare output and expected output *)
```

1.2 Program composition

All statements in a pipeline program must be in one of the following blocks(function, pipe, anonymous block) with global variables as exception.

```
global int x = 10; // global variable not in a block, allowed



function string hello(){
        return "hello" // statement in a function block, allowed
}


{
int a;   //statements in anonymous block, allowed
print_a
}


pipe{
    print_str("stmt in a pipe")// statement in a pipe block, allowed
}


print_int(x)// illegal, statement that's not in any block and not a global variable
    declaration.
```

1.3 Variable declaration and assignment

1.3.1 primitive types

pipeline has 4 primitive types:

- int
- float
- string
- boolean

variable declaration for primitive types works as follow:

```
int a; // type var-name for variable declaration

int b = 1;// type var-name expr, for variable declaration and initialization

a = 10; //variable assignment
```

1.3.2 string manipulation

String type supports the following operations:

- len
- sub
- cmp
- concat(with $)

```
    string a = "hello";//declare string variable a

    string b = " world";//declare string variable b

    int length;

    length = len(a); // length = 5

    bool str;

    str = cmp(a,b);//str = false

    str = sub(a,b);//str = false

    string c = a$b;// c = "hello world" $
```

1.3.3 global variables

global variables in pipeline is declared with the **global** keyword and assessable throughout the program. Also, global variable declaration need not to be inside a block.

```
    global int a = 10;//global variable a.
    function int foo(){
           return a;//global variable a is accessible.
    }
```

1.3.4 List

List is a compound type that can be applied all primitive types. List cannot be declared globally like primitive types
The pipeline list is a singly likened list supports the the following operation:

- addleft
- addright
- popleft
- list_free
- access

list-declaration

```
int a[]; //int-list declaration

float b[]; //float-list declaration

string c[]; //string-list declaration

bool d[]; //boo-list declaration
```

list-append

```
addleft(a,1) //append 1 to the left of a, a = [1]
addright(a,2) // append 2 to the right of a, a = [1,2]
popleft(a) // pop the left-most element in a, a = [2]
```

list-access

```
// let a = [1,2,3,4,5]
a[0] //1
a[1] //2
a[4] //5
```

If a list is declared, it must be freed in order to prevent memory leak in the program **list-free**

```
list_free(a)//release the memory of a and prevent memory leak
```

1.3.5 struct

Structs are a way for the user to define his/her own aggregate data type. It is a collection of variables that can be called and used for whatever purpose a user needs.
A struct must first be defined in the outermost scope before it can be used. It cannot be defined within a block, function or pipe. A struct can only contain a File, int, float, bool or string type, and only declarations without initialization may be used in the struct. The syntax to define a struct is with the struct keyword followed by the name of the struct and a block of statements between curly-braces and after the last curly brace there is a semicolon.

```
/* struct definition */
struct struct_name {
    [declaration statements]
};
```

To use your struct in the body of your function simply use the `keyword` followed by the name of the variable. A struct variable must be declared in pipeline first, and it can never be initialized with a struct literal.

```
/* proper struct declaration */
```

```
struct struct_name;
/* improper struct declaration */
struct struct_name = { [statements] };
```

the second case will fail compilation. In order to use the variables inside of the struct you simple use the struct variable name followed by dot operator "." and the name of the struct variable.

```
/* struct use */
struct_name.variable = expr;
expr = struct_name.variable;
expr = expr + struct_name.variable;
/* etc... */
```

Here is an example program that uses a struct:

```
/* define the struct */
struct Example {
    int i;
    float f;
    bool b;
    string s;
    struct A a;
    File fl;
};
{
    struct Example e;
    e.s = "Hello World\n";
    e.i = 1;
    int i = e.1 + 1;
    if ( e.i < 3 ) {
        print_string(e.s);
    }
}
```

1.3.6 file

File IO is done with a File object. First you must declare a File object and then use the init file object function to initialize the file object and open the file. A file object cannot be initialized and declared in one statement.

```
/* File declaration and initialization */
    File file_object;
    init_file_obj(file_object, file_name, file_mode);
```

for the init file obj you give it first the name of the file object, then a sting containing the name of the desired file, and lastly a string that represents the mode it can it is opened in. pipeline uses the standard c modes, so the mode must be one of the following strings:

```
    mode    | mode description
    ------------------------------------------------------------------------------
    "r"     | read-only mode
    "rb"    | read-only mode for a binary/unix style file
    "r+"    | read-mode but writing is also allowed
    "rb+"   | the same as "r+" but for binary/unix style files
    "w"     | write-only mode that creates a file or overwrites an existing file
    "wb"    | write-only but for binary/unix files
    "w+"    | write-mode that also allows reading
    "wb+"   | same as "w+" but for binary files
    "a"     | append-mode opens an existing file in write-only mode without overwriting the file
    "ab"    | same as "a" but for binary files
    "a+"    | append-mode with reading allowed
    "ab+"   | same as "a+" but for binary files
```

To write to a file you use the fwrite str function, which takes a string and a file object as an argument, and then writes that string to a file.

```
    fwrite_str(given_string, file_object);
```

To read a line from a file use the fread line function which takes only the file object and returns a string. It reads up to the next newline character. To read up to 4095-bytes from a file in one chunk use the freadn function which takes a file object and an integer as an argument.

```
    string s = fread_ln(file_object);
    s = freadn(file_object, number_of_bytes);
```

When you are done you must close the file to both prevent memory leaks and to ensure that the contents of the buffer are fully read into the file. To close a file object, use the function close file which takes a file object as an argument and closes the file.

```
    close_file(file_object);
```

In order to avoid memory leaks or file problems it is recomended that for write operations you open the file in either "w" or "a" mode write your strings and then close the file before you do any read operations, and for reading it is recommended to open in "r" or "rb" mode and do your read operations and then close the file before you do any write operations.
here is an example program:

```
{
    File test_file;
    init_file_obj(test_file, "test-fileIO.txt", "w+");
    fwrite_str("Hello World\n", test_file);
    close_file(test_file);
    init_file_obj(test_file, "test-fileIO.txt", "r+");
    string s = fread_line(test_file);
    print_str(s);
    close_file(test_file);
    init_file_obj(test_file, "test-fileIO.txt", "r+");
    s = freadn(test_file, 4);
    print_str(s);
    print_str("\n");
    close_file(test_file);
}
```

1.4 control flow

control flow in pipeline works similar as C, it has for and wile loop and if, else selection

```
{
    int a;
    for(a = 0;a<10;a=a+1){// a for loop that loops over 10 times
        print_int(a);
    }
    a = 0;
    while(a<10){// a while loop that loops over 10 times
        print_int(a);
    }
    if(true){// an if selection that will always print false
        print_bool(false);
    }else{
        print_bool(true);
    }
}
```

The following program prints the lyrics for 99-bottles of beers.

```
 {
 int i;
 for (i = 99; i >0; i = i-1)
     {

     string wall = " bottles of beer on the wall";
     string beer = " bottles of beer.";
     string pass = "Take one down and paas it around, ";
     string nomore = "No more bottle of beer on the wall";
     if(i>1){
         print_int(i);
         print_str(wall$", ");
         print_int(i);
         print_str(beer$"\n");
         print_str(pass);
         print_int(i-1);
         print_str(wall$".\n\n");
         }
     else{
         print_int(i);
         print_str(wall$",");
         print_int(i);
         print_str(beer$"\n");
         print_str(pass);
         print_str(nomore$"\n\n");
         print_str(nomore $ ", no more bottle of beer"$"\n");
         print_str("Go to the store and buy some more, 99 bottles of beer on the wall.\n");
         }
     }
     }
```

1.5 function

while pipeline has many buildin functions, it also allows custom function declaration. The fallowing function calculates the gcd of two given numbers and returns the result.

```
function int gcd(int a, int b){
    while(a != b){
        if(a>b){
            a = a-b;
        }else{
            b = b-a;
        }
    }
    return a;
}
{
int result = gcd(14,21)// result = 7
}
```

pipeline also supports recursive functions.

```
function int fib(int x){//function the calculates the nth Fibonacci numbers

     if(x==0){
         return 0;
     }
     if (x==1){
         return 1;
     }
     return fib(x-1)+fib(x-2);
 }
 {
 int c = fib(4);

 print_int(c);
 }
```

1.6 pipe

Pipes are created to enable asynchronous programming using the event-driven architecture. Ideally the code that is blocking, and the variables dependent on it, go inside a pipe. Multiple pipes can be created in the program.

```
pipe{
    int a ;
    int b ;
    while(1);
    int c = a + b;
    print_int(a+b);
}
```

1.7 Routing

Pipeline language supports the following HTTP functions - LISTEN, GET, PUT, POST. All these functions are supported only inside a pipe. The Listen function has to go first inside the pipe before anything else, and the rest of the HTTP functions require LISTEN to be present for them to execute. The listen function takes a string (IP Address) and an integer (the port number). The other HTTP functions take “GET”, “POST”, "PUT", "DELETE" as the 1st argument; the route as the 2nd argument; and the callback function(function name is passed as string) as the 3rd argument.

```
function string get_user_handler(){
    return "You sent me a GET request !!!???! ";
}
function string put_user_handler(){
    return "You sent me a PUT request !!!???! ";
}
function string post_user_handler(){
    return "You sent me a POST request !!!???! ";
}
function string delete_user_handler(){
    return "You sent me a DELETE request !!!???! ";
}
pipe {
    listen("127.0.0.1",8080);
    http("GET","/user","get_user_handler");
    http("PUT","/user","put_user_handler");
    http("POST","/user","post_user_handler");
    http("DELETE","/user","delete_user_handler");
}
```

1.8 build-in functions

build-in functions provided by the pipeline language:

- print_int(int) - print a int to stdout
- print_float(float) - print a float to stdout
- print_bool(bool) - print a bool to stdout
- print_string(string) - print a string to stdout
- print_error(string) - print a string to stderr
- sleep(int) - block the program for a given number of seconds
- exit(int) - exit the program with given status