Pipeline

•••

A Presentation by Team Pipeline

Ben Lai
Brandon Bakhshai
Jeffrey Serio
Somya Vasudevan

What is pipeline

Pipeline is an asynchronous programming language that uses an event-driven architecture. Pipeline's event-loop is powered by libuv, the same library used by nodejs to implement it's asynchronous architecture. Pipeline uses the idea of a pipe as a kind of user-friendly asynchronous thread-like interface, which allows the user to write a block of synchronous statements and execute that block asynchronously from the rest of the code in the program.

Quick-start guide

Types

Int

Float

String

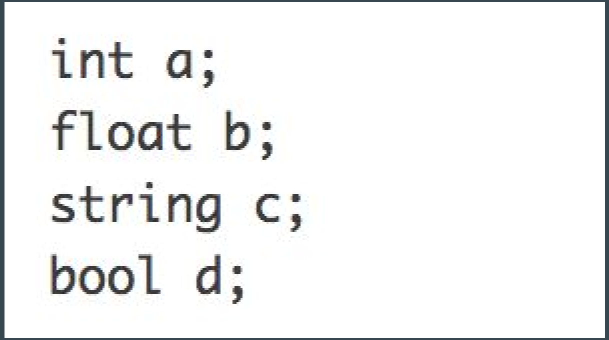
Struct

Bool

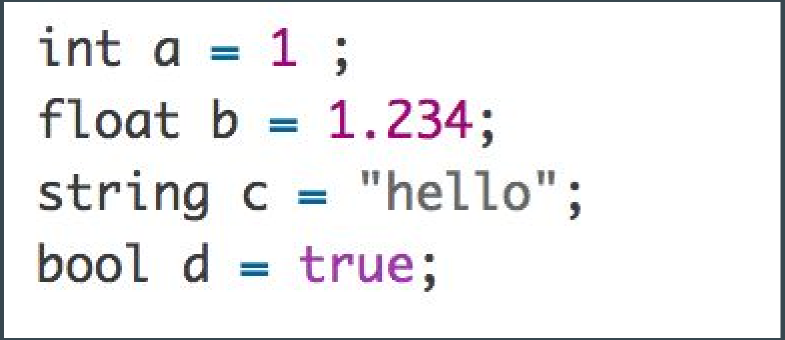
File

Variable Declaration and Assignment

Type ID; or Type ID = expr;



```
int a;
float b;
string c;
bool d;
```



```
int a = 1 ;
float b = 1.234;
string c = "hello";
bool d = true;
```

Strings

String supports the following operations:

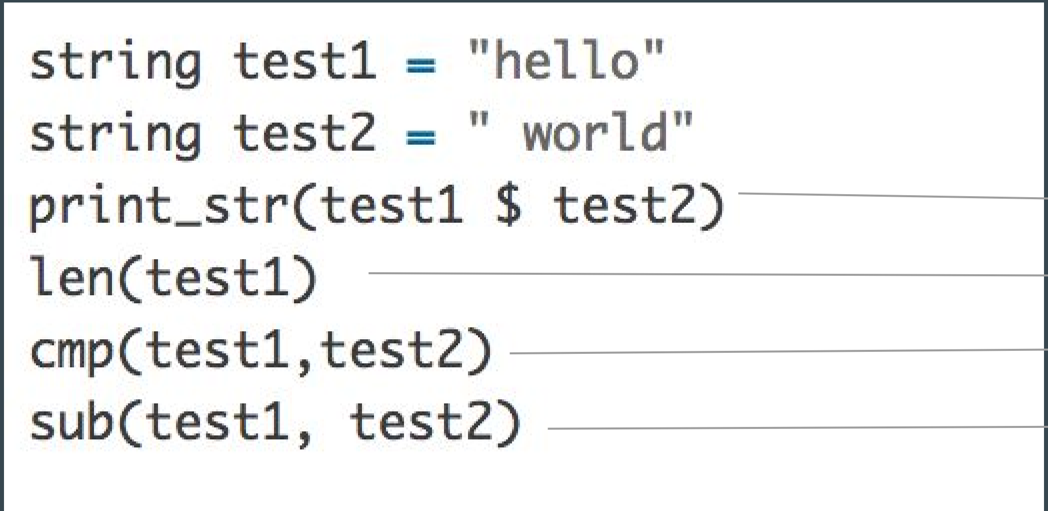
len(string): returns the length of a given string

cmp(string, string): compare two given strings and returns True when equal and False otherwise

sub(string1, string2): returns True if string2 is a substring of string1, return False otherwise

String concat with “$”:

“Hello” $ “ world” = “hello world”



```
string test1 = "hello"
string test2 = " world"
print_str(test1 $ test2)
len(test1)
cmp(test1,test2)
sub(test1, test2)
```

“hello world”

5

false

false

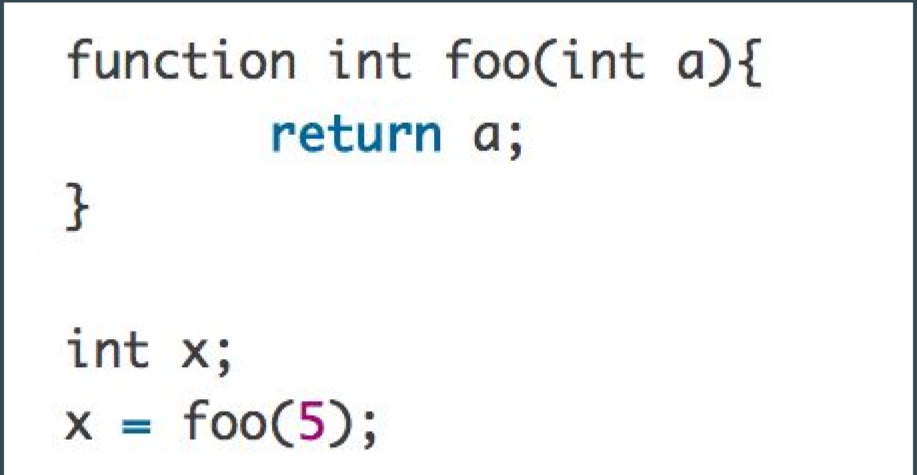
Functions

Function declaration:

Function type ID(formals){stmts}

Function call:

ID(formals)



```
function int foo(int a){
        return a;
}

int x;
x = foo(5);
```

Structs

The structs in pipeline are declared and used the exact same way as the structs in C. The only type that cannot be declared as part of a struct is the List type, because List is not a complete type. Structs also cannot be declared and initialized in the same statement

Struct Definition

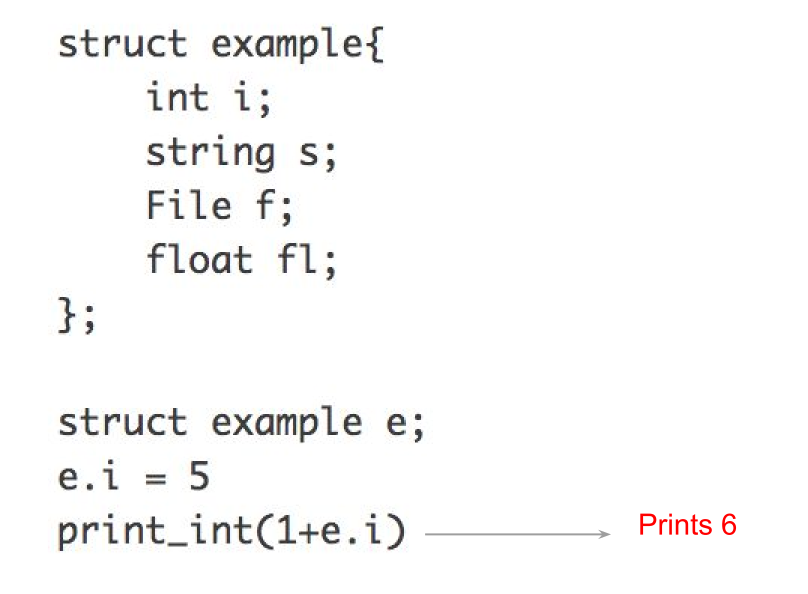
```
struct <struct_name> {
      [statements…]
};
```

Struct declaration:

```
struct <struct_name>;
```

Struct use:

```
<struct_name>.<struct_variable>
```



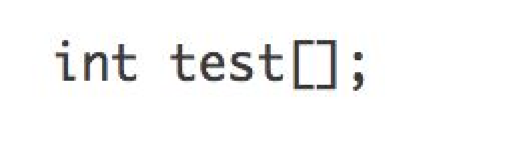
```
struct example{
    int i;
    string s;
    File f;
    float fl;
};

struct example e;
e.i = 5
print_int(1+e.i)
```

Prints 6

List

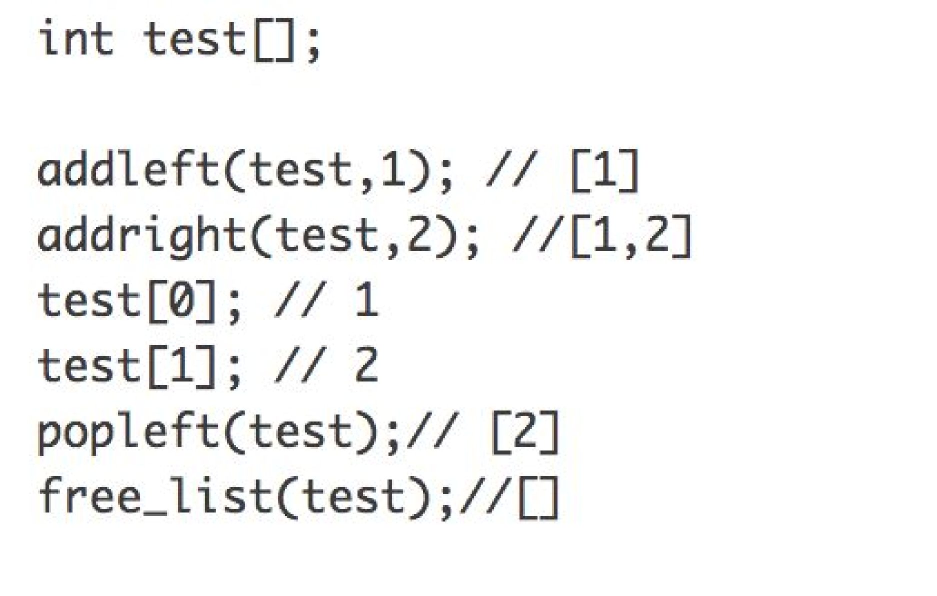
Singly linked list operates on the heap.

List decl:



```
int test[];
```

List operations:
addleft , addright, popleft, access, free_list



```
int test[];

addleft(test,1); // [1]
addright(test,2); //[1,2]
test[0]; // 1
test[1]; // 2
popleft(test);// [2]
free_list(test);//[]
```

File I/O

For file IO a File type is first declared and then initialized. Initialization opens a file in a given mode, to be used for reading and writing to files. The supported file operations are read line from the file, read n bytes (up to 4095) from the file, and write a string to the file. After the user is finished he/she must use close_file(file_obj) to close the file.

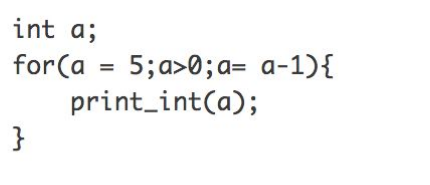
File declaration:
File f;

File initialization:
init_file_obj(f, file_name, file_mode);

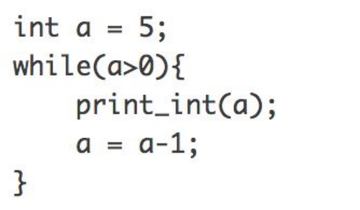
Supported operations:
function string string fread_line(File f);
reads a line from a file and returns a string.
function string freadn(File f, int n);
reads n bytes from the file up to 4095 bytes.
function void fwrite_string(string s, File f);
writes a string to a to the file
function close_file(File f);
closes the file

Control structures

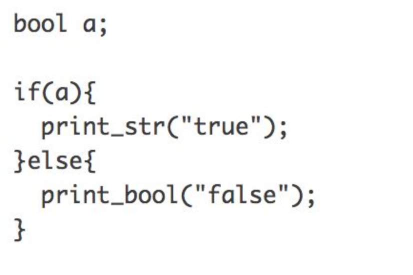
for , while, if, else



```
int a;
for(a = 5;a>0;a= a-1){
    print_int(a);
}
```



```
int a = 5;
while(a>0){
    print_int(a);
    a = a-1;
}
```



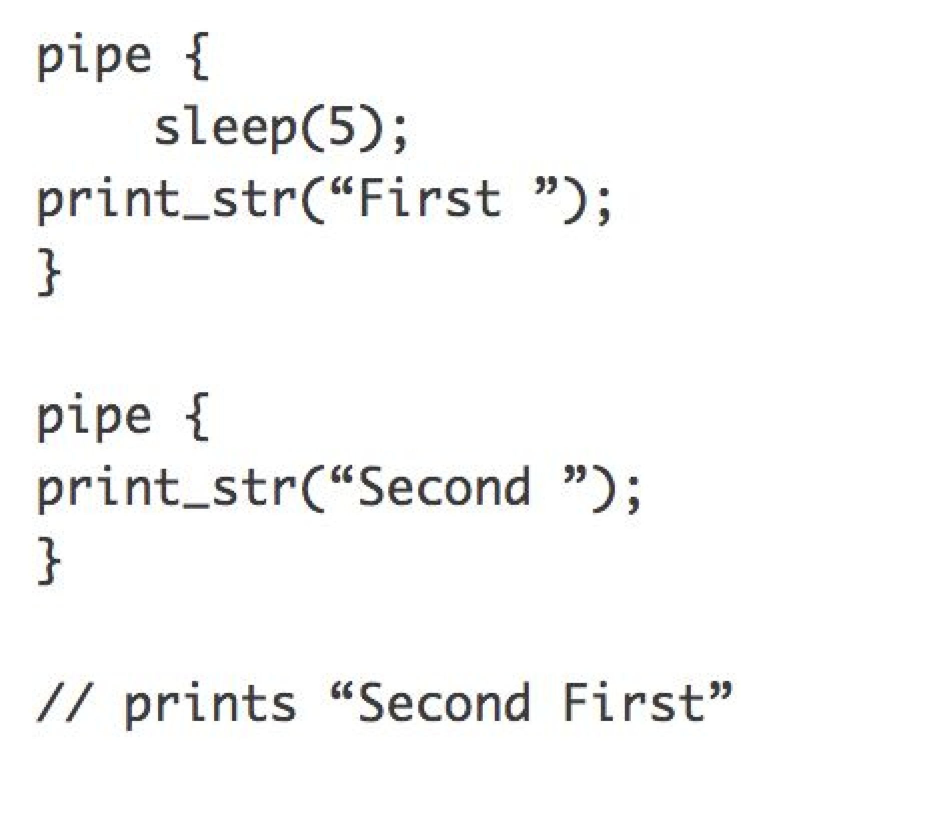
```
bool a;

if(a){
  print_str("true");
}else{
  print_bool("false");
}
```

Pipes

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.

Example Program



```
pipe {
    sleep(5);
print_str(“First ”);
}

pipe {
print_str(“Second ”);
}

// prints “Second First”
```

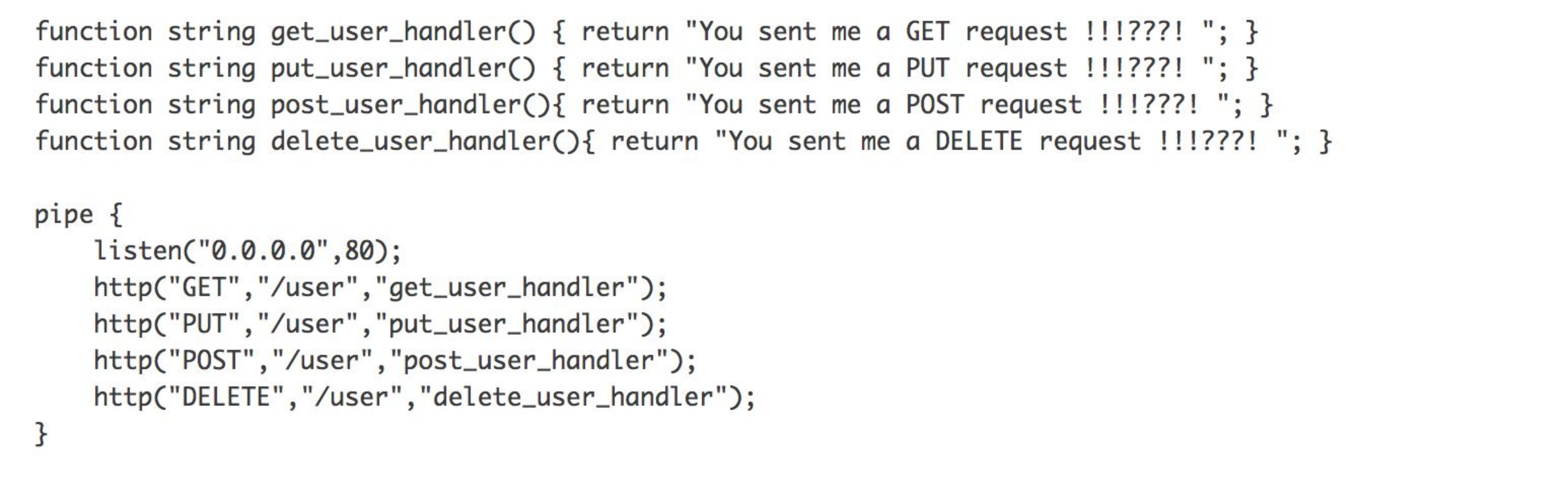
Routing

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.

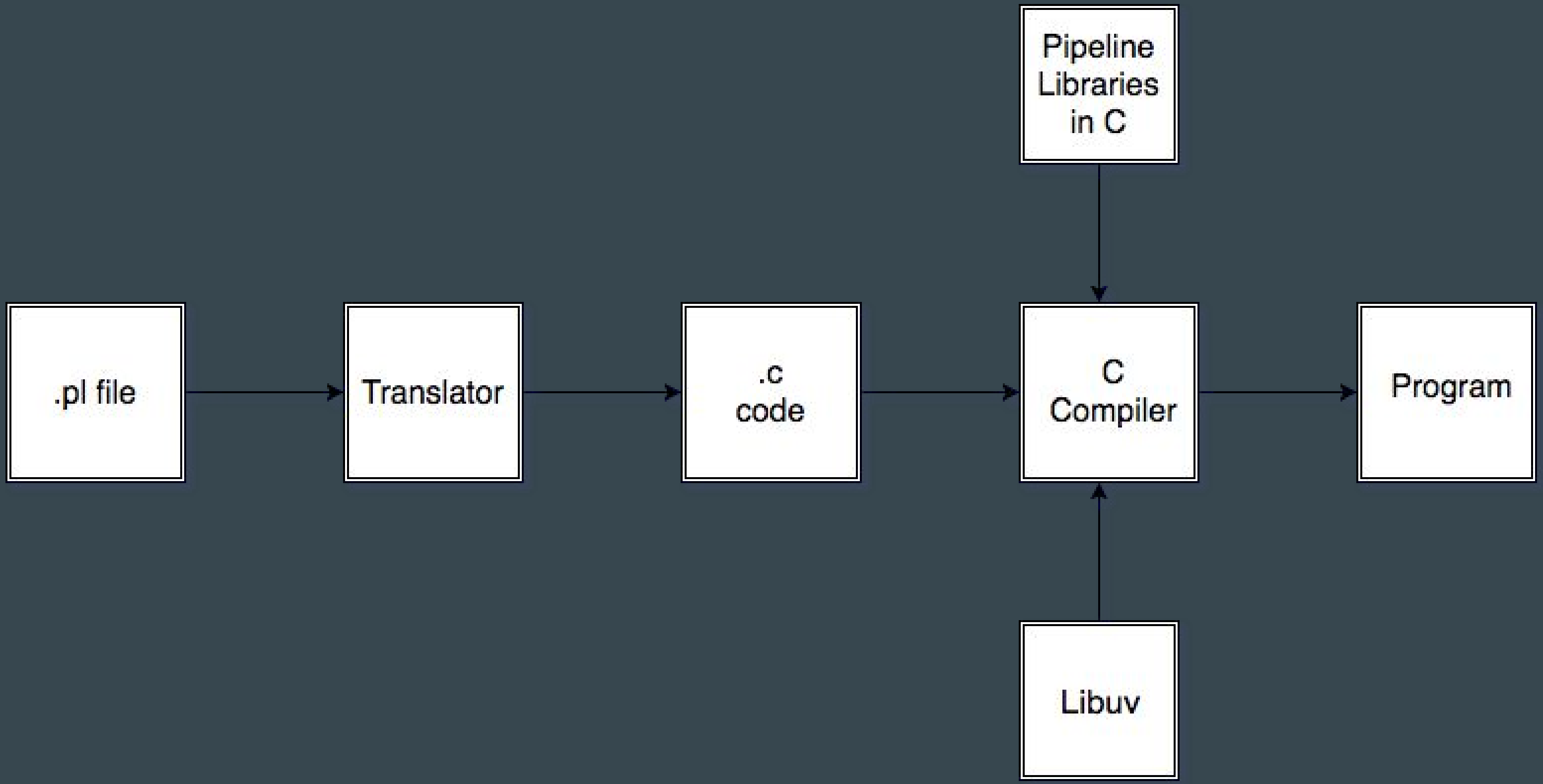
Examples:



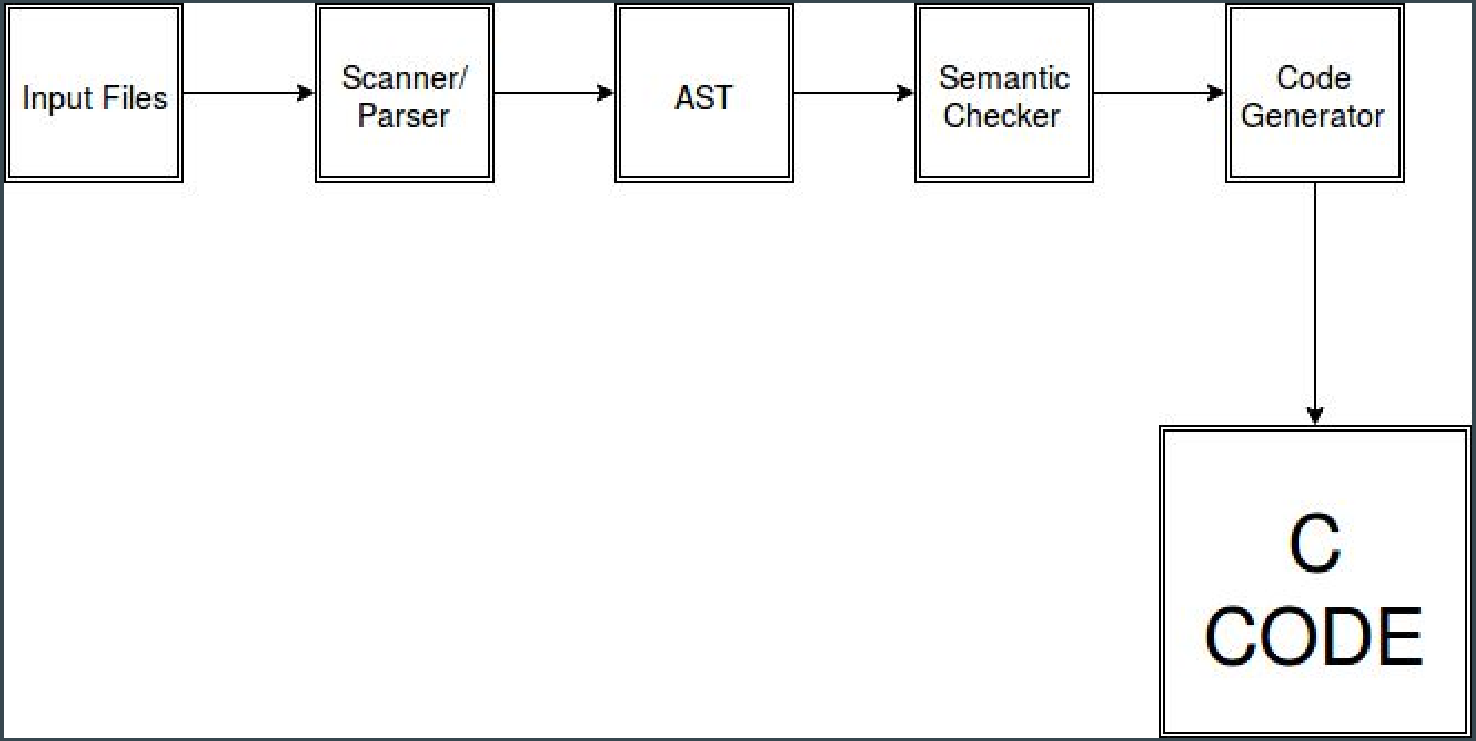
```
function string get_user_handler() { return "You sent me a GET request !!!???! "; }
function string put_user_handler() { return "You sent me a PUT request !!!???! "; }
function string post_user_handler(){ return "You sent me a POST request !!!???! "; }
function string delete_user_handler(){ return "You sent me a DELETE request !!!???! "; }

pipe {
    listen("0.0.0.0",80);
    http("GET","/user","get_user_handler");
    http("PUT","/user","put_user_handler");
    http("POST","/user","post_user_handler");
    http("DELETE","/user","delete_user_handler");
}
```

The Pipeline Translator

The Translation Process

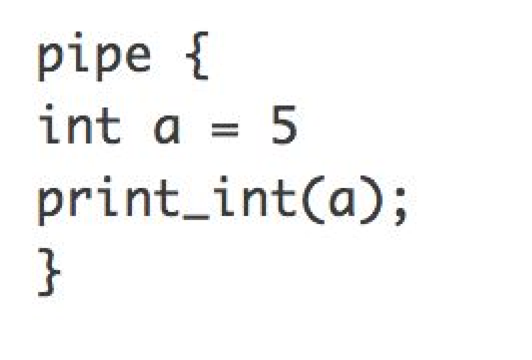


Inside the Translator



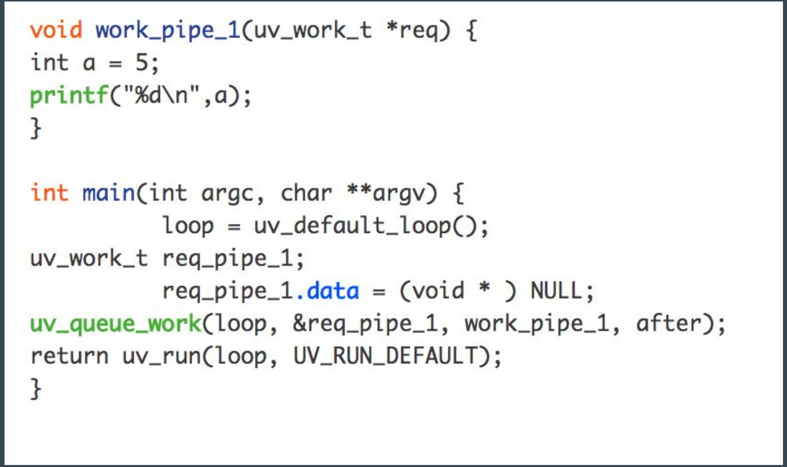
The Translator

Pipeline Code:



```
pipe {
int a = 5
print_int(a);
}
```

C Code:



```
void work_pipe_1(uv_work_t *req) {
int a = 5;
printf("%d\n",a);
}

int main(int argc, char **argv) {
        loop = uv_default_loop();
uv_work_t req_pipe_1;
        req_pipe_1.data = (void * ) NULL;
uv_queue_work(loop, &req_pipe_1, work_pipe_1, after);
return uv_run(loop, UV_RUN_DEFAULT);
}
```