Project Proposal:
The Pipeline Language

Team: Brandon Bakhshai, Ben Lai, Jeffrey Serio, and Somya Vasudevan
UNIs: bab2209, bl2633, jjs2240, and sv2500

February 7, 2017

1 Motivation

Concurrent programming has become a very important paradigm in modern times, with mainstream languages such as Java, Python, and C++ offering concurrent programming mechanisms as part of their APIs. However, they tend to be complicated - sometimes necessarily - and invite a host of additional concerns like atomicity. And of course, complexity of code is strongly correlated with bugs.

Node.js has emerged as a framework with a unique approach toward asynchronous programming - the single-threaded (but not really) asynchronous programming. The event-driven architecture of Node.js and nonblocking I/O API makes it a perfect fit for backend web development.

We want to build a simple language that encompasses these features from Javascript and the Node.js framework - easy asynchronous programming using the event-driven architecture and a speedy I/O API.

2 Description of Pipeline

The asynchronous programming model takes after that of Node.js, while the syntax is more like that of Go. The way chains of functions which depend on the results of the previous function are handled is in a manner similar to Promises from Javascript, except we use the pipe ("—") syntax from bash shell. A chain of blocking functions which rely on each other is written as follows:

```
firstDoThis() | secondDoThis(_) | thirdDoThis(_) | fourthDoThis(_) || errorHandler();
```

As soon as firstDoThis() blocks, the program will continue on to the next line, without executing secondDoThis(). Only when firstDoThis() returns does secondDoThis() run with the return value of firstDoThis(). If there is an error in any function, then the pipeline jumps to errorHandler().

The language is designed for web-based programming, leveraging features like non-blocking I/O for external client communication.

3 Pipeline's Features and syntax

Data types

a. Primatives

int:Basic integer type, 4 byte in size
char: ASCII characters, 1 bytet in size
float:Floating point numbers, 4 bytes in size
pointer:A memory address stored in a 8 byte
function:Takes in parameters and return if needed

3.0.a. Compound type

String:String type, support basic string manipulation.
List:A collection of data supports basic list operation.
Struct: set of grouped variables.

3.1 Control Flow:

if - else - if else:
if condition {
#statements
} else if condition {
#statements
} else condition {
#statements
}
for - It is a control flow statement for iterations. Syntax :
for init; condition; inc/dec { statements }
while - while (condition) {statements;}
break - breaks the control flow out of loop
continue - forces the next iteration of the loop to take place, skipping any code in between.
return terminates the execution of a function and returns control to the calling function.

3.2 Operators

Arithmetic: we use the standard C operators +,=,*,/,%
Assignment: =, +=, =, =, /=, %=
Comparison: ==, !=, ¡=, ¿=
Increment/Decrement: ++, - -
Logical: (these are the only operators that differ from c) and, or, not

3.3 Keywords

null - The return statement terminates the execution of a function and returns control to the calling function.
/*....*/ - Comments
@ - Dereferencing
*** -** Referencing
| − Pipeline operator
|| − Error Handler at the end of a pipelineg
pipe - feed returned pipe data into here
. - Name space indicator

4 Example Programs

4.1 simple program

```
function gcd(int a, int b)(int)
{
   if a < 0 {
```

```
        a = -a;
    }
    if b < 0 {
        b = -b;
    }
    if b > a {
        int temp = a;
        a = b;
        b = temp;
    }
    while 1 {
        if b == 0 {
            return a;
        }
        a = a % b;
        if a == 0 {
            return b;
        }
        b = b % a;
    }
}


function main(void)(int)
{
    /* Here is one way to type it */
    pipe gcd(a, b)| gcd(1031940, pipe)| gcd(49980, pipe)| printf("gcd: %d", pipe) ||
        printf("invalid numbers");

    /* Here is an alternative way to type the same thing*/
    pipe gcd(a, b)|
    gcd(1031940, pipe)|
    gcd(49980, pipe)|
    printf("gcd_2: %d", pipe)||
    printf("invalid numbers");

    /* the idea is that these two pipes will be executed asynchronously, but
     * the functions inside the pipe will be executed synchronously */
}
```

4.2 Simple Server Example

```
/* assuming listen, and reply are already defined socket api
 * and that stripe has an api for Pipeline*/
function main()(int)
{
    while (1) {
        listen(80)|
        stripe.charges.create({
            "amount": 1000,
            "currency": "usd",
            "description": "Example charge",
            "source": token,})|
            reply(pipe); /*assumes stripe.charges.create returns
                         *everything needed for the reply */
```

```
    }

    listen(80)|
    stripe.charges.create({
        "amount": 1000,
        "currency": "usd",
        "description": "Example charge",
        "source": token,})|
    goback(8)||
    perror("pipe failed");

}
```

5 Example Programs

5.1 simple program

```
function gcd(int a, int b)(int)
{
    if a < 0 {
        a = -a;
    }
    if b < 0 {
        b = -b;
    }
    if b > a {
        int temp = a;
        a = b;
        b = temp;
    }
    while 1 {
        if b == 0 {
            return a;
        }
        a = a % b;
        if a == 0 {
            return b;
        }
        b = b % a;
    }
}


function main(void)(int)
{
    /* Here is one way to type it */
    pipe gcd(a, b)| gcd(1031940, pipe)| gcd(49980, pipe)| printf("gcd: %d", pipe);

    /* Here is an alternative way to type the same thing*/
    pipe gcd(a, b)|
    gcd(1031940, pipe)|
    gcd(49980, pipe)|
    printf("gcd_2: %d", pipe);

    /* the idea is that these two pipes will be executed asynchronously, but
```

```
    * the functions inside the pipe will be executed synchronously */
}
```

5.2 Simple Server Example

```
/* assuming listen, and reply are already defined socket api
 * and that stripe has an api for Pipeline*/
function main()(int)
{
    while (1) {
        listen(80)|
        stripe.charges.create({
            "amount": 1000,
            "currency": "usd",
            "description": "Example charge",
            "source": token,})|
            reply(pipe); /*assumes stripe.charges.create returns
                        *everything needed for the reply */
    }

    listen(80)|
    stripe.charges.create({
        "amount": 1000,
        "currency": "usd",
        "description": "Example charge",
        "source": token,})|
    goback(8)||
    perror("pipe failed");

}
```