Project Proposal:
The Pipeline Language

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

February 8, 2017

1 Team Structure

Architecture Guru: Brandon Bakhshai (bab2209)
Language Guru: Ben Lai (bl2633)
Team Manager: Jeffrey Serio (jjs2240)
Tester: Somya Vasudevan (sv2500)

2 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. 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.

Our intent with Pipeline is 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.

3 Description of Pipeline

Pipeline is a structured inperative C-style language that incorporates the asynchronous programming model of Node.js in the form of a pipeline. Pipeline expands on the idea of Javascript's promises, and maked this concept central to the design of the language in the form of a pipeline. A pipeline allows the programmer to chain functions together that must run synchronously and handle them asynchronously from the body of code in which it resides – a manner similar to Promises from Javascript, except with a more convienient syntax.

3.1 Implementation Plans

Pipeline is designed to provide an easy to use mechanism for concurrent programming. The language is designed mostly to be web-based programming, leveraging features like non-blocking I/O for external client communication.

LLVM IR Pipeline will compile into LLVM IR, which is capable of running on many achitectures, as well as a built-in optimizer and a third-party parallelization optimizer called Polly.

Libraries We plan to use LLVM IR's ability to link C and C++ libraries to support Socket I/O, complex data structures, and web-developement tools.

4 Pipeline's Features and syntax

In pipeline there are two methods to create and use a pipeline. The first is an anonymous pipeline, created with the pipe keyword, the function and the use of the pipe symbol, "|" from bash shell, in between function calls. This creates a non-blocking asynchronous chain of blocking functions which rely on each other, and is written as follows:

```
pipe 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 second way is to create a named pipeline, which allows the programmer the flexibility of interleving function calls that belong to a single pipe, as well as utilize the coupiling feature, which will allow one named pipe to be connected with another one from a specific point. This will allow the progorammer to write a traditional synchrounous program, test it, then group together function calls that could be ran asynchronously at a later time. A named Pipeline is declared with the following syntax:

```
Pipeline <name_of_pipeline>;
/* this will be the syntax for the error handler function*
<name_of_pipeline> || errorHandler();
```

Once created any function or anonymous pipe with the name of the Pipeline in front is logically grouped together as a normal pipeline.

```
<name_of_pipeline> foo(param);
<name_of_pipeline> foo'(_)| foo''(_);
<name_of_pipeline>
/* they will be logically grouped together and work just like this:
pipe foo(param)| foo'(_) | foo''(_);
```

A coupling will be a way for one named pipeline to feed into another one, and in a blocking manner but at that point. Meaning, at the point in the code in which the coupling is placed, the pipeline that arrives at that point will wait there untill it is able to send or recieve that information, and then both will proceed asynchronously. The syntax will be like so:

```
Pipeline <From>;
Pipeline <Into>;
<From> foo(param);
<From> coupling <Into> foo'(_);
<Into> foo''(_);
<From> foo'(_)| foo''(_);
/* this will allow A to */
```

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, with the folloing syntax:

```
function function_name(type parameter_name, ...) (return_type [optional name]) {
    #statements }
```

4.0.a. Compound type

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

4.1 Control Flow:

if - else - if else:
if condition { #statements }
else if condition { #statements }
else condition { #statements }
for -
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.

4.2 Operators

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

4.3 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

5 Example Programs

5.1 simple program

This program is to give the reader an example of how we intend the syntax to be. It is a simple GCD function, and it shows how the pipes are to run asynchronously from eachothe.

```
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 error(String err_message)() {
    printf(err_message);
    exit(1);
}

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

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

    /* Here are the named pipelines */
    Pipeline A;
    Pipeline B;
    A gcd(a, b);
    A coupling B gcd(1031940, _); /* Here is the coupling mechanism */
    B gcd(49980, _);
    A gcd(49980, _);
    A printf("gcd_A: %d", _);
    A || error("oops");
    B || error("oops");


    return 0;
}
```