PIPELINE
LANGUAGE RESOURCE MANUAL

By Team Pipe Dream:
Brandon Bakhshai (bab2209)
Ben Lai (bl2633)
Jeffrey Serio (jjs2240)
Somya Vasudevan (sv2500)

Spring 2017 COMS W4115: Programming Languages and translators

Contents

Chapter 1

Introduction

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.

1.1 About 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.

1.2 Your First Programs in Pipeline

1.2.1 An Oldie But a Goodie: "Hello World"

```
function main(char** argv, int argc)(void){
    printf("Hello World");
}
```

1.2.2 Getting to know the Pipeline with GCD

```
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;
}
```

Chapter 2

Data Types

This is some text

2.1 Primitive Types

2.2 Complex Types

Chapter 3

Lexical Conventions

Chapter 4

Expressions and Operatos

Chapter 5

Control Flow and Pipelines

5.1 Control Flow

As with any imperative programming language, Pipeline has control flow mechanisms. Control flow mechanisms control the order in which statements are executed within a program. Although this definition does not fully apply to Pipeline by design, control flow mechanisms in Pipeline still control the order in which statements are executed both inside of and, to an extent, outside of pipelines.

5.1.1 Conditionals

A conditional is a way to decide what action you wish to take based on the given situation. A conditional relies on some given expression that can be evaluated to true or false, or rather in Pipeline, like in C, any expression that can be evaluated to an integer. Pipeline uses the conventions established in C for evaluating an integer as true or false, where ANY non-zero value evaluates to true and only zero(0) evaluates to false. A conditional statement is written with the `if` statement followed by the aforementioned conditional statement and encloses the body of its text in curly braces ('{' and '}').

```
if <conditional expression> { <body_of_statements> }
```

It can then be followed by the optional `else if` statement, wich executes its body if the if statement evaluates to false and the `else if` condition evaluates to true, and/or the `else` statement which acts as a catch all and executes when no previous condition was true.

```
if <conditional expression> {
    <body_of_statements>
} else if {
    <body_of_statements>
} else {
    <body_of_statements>
}
```

5.1.2 Loops

Loops are exactly what they sound like; they execute a body of code repeatedly. They are one of the most important tools in an imperative language for they allow the user to automate repetitive processes in an intuitive manner.
Pipeline uses the two most standard types of loops, while and for loops. The while loop works exactly like the conditional if statement, except after it executes its body of code it re-evaluates the expression and if it remains true (non-zero) it repeats the block of code. This continues until the condition becomes false,

at which point the program continues past the loop. The loop is constructed with the `while` keyword as follows:

```
while <conditional expression> {
    <body_of_statements>
}
```

5.2 Pipelines

There are two flavors of pipelines in Pipeline: The anonymous pipeline, and the named pipeline. The anonymous pipeline contains less features than its named counterpart, but it is concise. The named variety offers more features and is more flexible than its anonymous counterpart. The purposes of this section is merely an introduction to the vocabulary of the asynchronous pipeline features, but the use and explaination of asynchronous programming is located in chapter 7.

5.2.1 Anonymous Pipelines

An anonymous pipeline creates a non-blocking asynchronous chain of functions which themselves are executed synchronously. Anonymous pipelines are created with the `pipe` keyword followed by one or more functions joined by the pipe symbol, "|" from bash shell. At bare minimum requires the `pipe` keyword and one initial function, and the syntax of an anonymous pipeline is 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().

5.2.2 Named Pipelines

The second variety is the named pipeline, which allows the programmer the flexibility of interleving function calls that belong to a single pipe, as well as other features, such as coupling, not scene in the Anonymous pipeline. 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''(_);
```

5.2.3 Pipeline Control Flow

goback

Coupling

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.

Chapter 6

Functions and the Standard Library

6.1 Anatomy of a Function

6.1.1 Declaration

6.1.2 Definition

6.1.3 Return values

6.1.4 Function in a pipeline

6.1.5 Variable Scope

6.2 Decorators and Function Pointers

6.2.1 Function Pointers

6.2.2 Decorators

6.3 Standard Library

6.3.1 Built-in functions

6.3.2 I/O

File I/O

6.3.3 Memory

Chapter 7

Asynchronous Programming with Pipeline

Async control flow is incredibly useful when dealing with I/O operations, which are the foundation of web-based programming. When restricted to a single-threaded and single-process model, I/O operations in a programming language are blocking and a program can wait for an unbounded time. Introduce multi-threading or multi-process models, and a programming language becomes much more complex. The single-threaded asynchronous control flow model simplifies dealing with I/O operations such that the program does not halt because of them.

7.1 My First Pipeline

Consider the following example pipeline:

```
readFile('/home/user/you/data.txt')
| processData(_)
| saveProcessedData(_, '/home/user/you/processedData.txt')
```

Formally, the definition of a pipeline is as such:

```
function0(Type param0, ..., Type paramN) |
function1(_, Type param0, ..., Type paramN) |
function2(_, Type param0, ..., Type paramN) | ... |
functionN(_, Type param0, ..., Type paramN)
```

How to interpret the above: "Type" references some actual type "Type param1, ..., Type paramN" references some (potentially zero-length) sequence of parameters to a function "_" acts as a placeholder for whatever value will be provided by the function one to the left in the pipeline

> Note that:
> The first function in the pipeline, function0, has no placeholder "_" because it has no function one to the left from which to take a value. The sequence of parameters can be of any finite length, including zero.

7.2 And Then There Were Two

Consider the terms "pipelineX" to refer to a sequence of functions arranged in a pipeline as detailed in the section "My First Pipeline," where X is a number serving as a unique ID for the pipeline. Now consider the two distinct pipelines, "pipeline0" and "pipeline1." Both pipelines contain functions which are blocking, waiting for the results of an I/O operation. The two pipelines are arranged as such in code:

```
pipeline0;
```

```
pipeline1;
```

Let's mimic the flow of a real program as it executes the two lines above. "Pipeline0" is executed and runs until there is a blocking operation. As soon as a blocking operation is encountered, the program moves it off to a worker thread, and then on the main thread continues on to execute "pipeline1."

Now we have a handful of different cases to consider.

7.3 Brief Summary of the Architecture?

Functions in a pipeline do not return. Rather, an entire pipeline may be thought of a series of nested functions, where what might be mistaken for the return value is simply passed as a parameter to the next nested function, and so on.

Therefore, if there is data you wish to manipulate or use after a function yields it, this manipulation/use should be done within the same pipeline.

The same state is kept throughout an entire pipeline

This is some text

Chapter 8

Tools for Web Developement

Pipeline is a language designed for backend web development. As such, there are certain tools necessary for the backend web programmer.

8.1 Comprehensive List of Tools for Web Development

8.1.1 subsection

subsubsection

Appendix A

The Pipe Directive