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

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.

0.1 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.

0.2 Your First Programs in Pipeline

0.2.1 An Oldie But a Goodie: "Hello World"

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

0.2.2 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 1

Chapter

This is some text

1.1 Section

1.1.1 subsection

subsubsection

Chapter 2

Data Types

This is some text

2.1 Primitive Types

2.2 Complex Types

Chapter 3

Lexical Conventions

3.1 Identifiers

3.2 Literals

3.3 Tokens

3.4 Punctuation

Chapter 4

Expressions and Operatos

4.1 Assignment

4.2 Arithmetic Operations

4.3 Boolean Expressions

4.4 Pointers and References

4.5 Array access

4.6 Operator Precedence and Associativity

Chapter 5

Statements

5.1 Declaration

5.2 Expressions and Assignment

5.3 Control Flow

5.3.1 Conditionals

5.3.2 Loops

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

6.2 Built-in Functions

6.3 Standard Library

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