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;
    }
}

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

fun 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 */

}
```

Chapter 2

Data Types

This is some text

2.1 Primitive Types

2.2 Complex Types

Chapter 3

Lexical Conventions

Chapter 4

Operators and Expression

4.1 Expression

In pipeline, a expression must contain at least one operand with any number of operators. Each operator has either one or two operand. Pipeline does not support the (inline if) operator.A expression must be a typed object.

Examples of Expression:

```
100;
100+10;
sqrt(10);
```

Group of subexpressions are done by parentheses, the innermost expression is evaluated first. Outermost parentheses is optional.

Example of expression grouping:

```
(1+(2+3)-10)*(1-2+(3+1))
```

4.2 Assignment Operator

A assignment operator stores the value of the right operand in the left operand with "=" operator. The left operand must be a variable identifier with the same type of the right operand.

Examples of assignment:

```
int x = 10;
float y =  0.5;
float z = 1.0 + 2.5;
```

Pipeline also supports compound assignment that combines arithmetic evaluation and assign the result to the left operand.

Supported compound assignment operators:

- +=
  Adds the two operands together, and then assign the result of the addition to the left operand.

- -=
  Subtract the right operand from the left operand, and then assign the result of the subtraction to the left operand.

- *=
  Multiply the two operands together, and then assign the result of the multiplication to the left operand.

- /=
  Divide the left operand by the right operand, and assign the result of the division to the left operand.

Example with compound assignment:

```
a += b /* the same as a = a + b */
a *= b /* the saem as a = a * b */
```

4.3 Increment and decrement

Pipeline supports increment operator "++" and decrement operator "–". The operand must be one of the primitive types or a pointer. The operators can only be applied after the operand.Operand will be evaluated before incrementation.
The result of pointer increment will depends on the type of the pointer.

Examples:

```
int x;
int@ p = &x;
x++    /* same as x = x+1 */
p++   /* same as p = &x + sizeof(x) */
```

4.4 Arithmetic Operations

Pipeline provides 4 standard arithmetic operations (addition, subtraction, multiplication, and division) as well as modular division and negation.

Examples:

```
Addition:
a = 1 + 2;
x = a + b;
y = 1 + x;
```

```
Subtraction:
a = 5 - 1;
b = x - y;

Multiplication:
a = x * y;
b = 10 * 5;

Division:
x = 5 / 3;
/*The result of division will be promoted to float even when the result is integer*/
y = 10 / a;

Modular division:
a = x % b;
b = x % 2;

Negation:
x = -a;
b = -10;
```

4.5 Comparison Operator

Pipeline supports Comparison Operator to determine the relationship between two operands. The result of a comparison operator will either be 1 or 0. Two operands of the comparison operator must be comparable types. Char type will be compared on their integer reference on ASCII encoding.

Examples:

```
Equality:
x == y;
a == 10;

Inequality:
x != 1;
y != a;

Less than:
x < 10;
y < a;

Less or equal than:
x <= 10;
y <= b;

Greater than:
x > 10;
y > a;

Greater or equal than:
x >= 100;
y >= x;
```

4.6 Logical Expression

Logical Expression will evaluate both operands and compute the truth value of those two operands. In Pipeline, only 0 will be evaluated to False. AND and OR are two logical expression in pipeline.
AND operator "and":
The expression will be evaluated to 1 if and only if both operands are true.

```
int x = 1;
int y = 0;
x and y /* This expression will be evaluated to 0*/
```

OR operator "or":
The expression will be evaluated to 1 if and only if at least one of the two operands is true.

```
int x = 1;
int y = 0;
x or y /* This expression will be evaluated to 1*/
```

4.7 Pointers Operator

There are two pointer operators in Pipeline, "@" for dereference and "&" reference.
"@" operator will dereference the value in the pointer address.
"&" operator will return the memory address of a varable.

Example:

```
int x = 10;
int@ p;
p = &x; /* the memory address of x will be stored in pointer variable p*/
int@@ ptr;
ptr = &p; /* the address of the address of the pointer to x will be stored in ptr*/
```

The pointer type will indicate the size of the object stored in the given address. If the type of the object in the address is not known, a void pointer can be used. However, a void pointer cannot be dereferenced since the machine won't know how many byte to read in the address.

4.8 type casting

Pipeline supports type casting between scalar types(int,float,pointer).Type casting operator "<type>" cast the operand to the type indicated.

Example:

```
int x = 10;
float y;
y = <float>(x);

int@ p;
float@ q;
float y = 10.5;
p = &<int>(y);
q = <float@>(p);
```

4.9 Array access

Pipeline uses subscript to access element in an array. A[i] will access the ith element in array A. Pipeline uses 0 indexing, the first element in an array has index 0.

Example:

```
x [10]int = [1,2,3,4,5,6,7,8,9,10];
x[0]; /*This expression will be evaluated to 1*/
x[9]; /*This expression will be evaluated to 10*/
```

4.10 Operator Precedence and Associativity

Chapter 5

Control Flow

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>
}
```

Chapter 6

Functions

6.1 Anatomy of a Function

A function in pipeline is defined with the keyword `fun` followed by the function name, the parameters and then the return type or types if applicable. The function definition must contain these elements in order to be a viable function, and the body of the function must be enclosed in curly braces ('{' '}'). In general the function definition has the following Grammar and syntax:

```
fun function_name(P_type_1 p_name_1, ..., P_type_n p_name_n)(ret_type)
{
    <body of code>
}
```

A function need not take any parameters, nor need it return any values.

6.1.1 Declaration

In Pipeline, as in C, the function must exist before it is called to be used. Meaning, you cannot call any function within any other function unless it has been explicitly stateted previous to that function was defined. A function declaration allows the programmer to declare a function exists before he/she has defined the function itself. A function declaration is almost identical to the above function definition, except that instead of curly braces and a code body, there is only a semicolon:

```
fun function_name(P_type_1 p_name_1, ..., P_type_n p_name_n)(ret_type)
{
    <body of code>
}
```

6.1.2 Parameters and Return values

Pipeline is a pass by value language, meaning that whenever a value is passed to another function or from another function, it is merely a copy. This is why Pipeline provides pointers like C, so that they can pass and manipulate a given object in memory, and not a copy of that object that keeps the original intact. Therefore the scope of any variable is the function in which is was declared, and in order for it to exist external to that function a pointer to its memory location must be provided.

6.1.3 The "main" function

The main function is the function that is executed at run time. The main function has as parameters char *string and int argc, which are related to the command-line arguments provided at run-time. The argv variable holds a pointer to the array of the strings typed by the user at run-time, the first of which is always the name of the executable, and the argc variable is the number of those arguments including the executable name.

6.2 Scope

The scope of a variable is from the point it was declared, until the end of the translation unit in which it resides. By translation unit, I am refering to control flow code bodies, functions, pipelines and ,in the case of a global variable, the program itself. The scope a parameter is from the point at which the function code block starts, until the end of the function.

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:

```
pipe {
    var a = readFile('/home/user/you/data.txt');
    var b = processData(a);
    saveProcessedData(b, '/home/user/you/processedData.txt');
}
```

Formally, the definition of a pipeline is as such:

```
pipe {
    tuple a0 = function0(Type param0, ..., Type paramN);
    tuple a1 = function1(a0, Type param0, ..., Type paramN);
    tuple a2 = function2(a1, Type param0, ..., Type paramN);
    ...
    functionN(anminus1, 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
- *tuple ax* is a tuple holding the results of the corresponding function, which can be used by the next function in sequence

It's extremely important for the programmer to understand the code within the curly brackets above would not run the way it should if it were placed outside a pipeline. Because any of the functions may be blocking, if they are not placed in a pipeline, the main thread will not wait for the function to return, so any code immediately after *tuple a0 = function0(Type param0, ..., Type paramN);* that makes use of variable *a0* could be reading a null or junk value. Blocking functions must be placed within pipelines, which will handle the control flow so that blocking functions must return before subsequent code runs.

Note that 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 of the main thread, and then on the main thread continues on to execute *pipeline1*. The functions in pipeline1 will execute until one blocks, at which point this pipeline1 will also be queued and moved off the main thread. The main thread will continue executing any code after pipeline1. When the blocking function in pipeline0 or pipeline1 returns, the corresponding pipeline resumes execution.

7.3 Data and Pipelines

If there is a blocking I/O operation, it must be put in a pipeline or the return value for the blocking function may be filled with junk or a null value, and this erroneous value could be used immediately if the next line makes use of the variable. Pipelines should be treated as islands of data, in that functions which depend on (take as arguments) values returned from blocking functions can only execute after the blocking function returns, which is unpredictable. In that sense, functions that are data dependent and relate to a particular instance or type of I/O operation should likely be encapsulated in a single pipeline.

7.4 Grammars

```
pipe { statement-list }

statement-list:
    statement
    statement statement-list
```

Appendix A

The Pipe Directive

A.1 Standard Library

A.1.1 Built-in functions

Pipeline comes with several built-in functions in its standard library.

A.1.2 I/O

A.1.3 Memory