Technical University of Munich

Department of Informatics

Bachelor's Thesis in Information Systems

Programming Language Development for Microfluidic Design

Qingyu Li

Technical University of Munich

Department of Informatics

Bachelor's Thesis in Information Systems

Programming Language Development for Microfluidic Design

Entwicklung der Programmiersprache für Mikrofluidikentwurf

Qingyu Li

Supervisor: Prof. Dr. sc.techn. Andreas Herkersdorf

Advisor: Dr.-Ing. Tsun-Ming Tseng

Submission Date: 15.02.2019

Confirmation

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Zusammenfassung

Um praktische Mikrofluidik-Module zu entwerfen, wurde Columba, das Co-Layout Synthesewerkzeug, entwickelt. Benutzer können über Columba verschiedene Module definieren, die Beziehung zwischen den Modulen bestimmen und dann das entsprechende Konstruktionsdiagramm generieren. Columba verfügt jedoch noch nicht über eine Reihe von Spezifikationen, anhand derer die Benutzer die von ihnen benötigten Module definieren können. Die Benutzer werden bei der Verwendung von Columba auf viele unnötige Schwierigkeiten stoßen. Die Arbeit konzentriert sich auf die Gestaltung einer Programmiersprache Co, um dieses Problem zu lösen. Ich werde die Merkmale der gängigen Programmiersprachen analysieren und vorstellen, wie die neu entworfene Sprache Co die Vorteile verschiedener Programiersprachen kombiniert, um die Benutzer dabei zu unterstützen, Columba besser zu nutzen.

Abstract

In order to design practical microfluidic modules, Columba, the co-layout synthesis tool, was created. Users can define different modules through Columba and determine the relationship between modules, and then generate the corresponding design diagram. However, Columba still lacks a set of specifications to help users define the modules they need, and thus users will encounter many unnecessary difficulties when they are using Columba. This thesis proposes a programming language Co to solve this problem. I will analyze the characteristics of current mainstream programming languages and introduce how the proposed language Co combines the advantages of various main-stream languages to help users easily use Columba.

Contents

1 Overview **1**

2 Data preparation for designing a continuous-flow microfluidic biochip **5**

2.1 Rule . . . 5

2.2 Module . . . 6

2.3 Netlist . . . 7

2.4 Conflict . . . 7

2.5 Parallel . . . 7

3 Designing language grammar of Co **8**

3.1 Programming Paradigm . . . 8

3.1.1 Object-oriented Programming (OOP) . . . 8

3.1.2 Functional Programming . . . 9

3.2 VHDL . . . 10

3.2.1 Entity Declaration . . . 10

3.3 JavaScript . . . 11

3.4 Hardware design language . . . 12

3.5 R . . . 14

4 Core Co **16**

4.1 Lexical Structure . . . 16

4.1.1 Character Set . . . 16

4.1.2 Reserved Words . . . 17

4.2 Types, Values, and Variables . . . 17

4.2.1 Variable Declaration . . . 18

4.3 Expressions . . . 18

4.3.1 Parallel . . . 19

5 Compiling of Co **19**

5.1 Lexical analysis . . . 21

5.2 Syntactic analysis . . . 24

5.3 Semantic analysis . . . 26
5.3.1 Detection of module . . . 26
5.3.2 Detection of Parallel's Definition . . . 27
5.3.3 Detection of Netlist's Definition . . . 28
5.3.4 Detection of Connected Graph . . . 28
5.4 Code generation . . . 29
5.5 How to develop Co . . . 31
5.5.1 New module type . . . 31
5.5.2 New type of relationship between modules . . . 31

6 Conclusion **32**

A Design from Columba S **35**

List of Figures

1 The design of a nucleic acid processor automatically synthesized by the proposed tool Columba. (a) A switch. (b) A rotary mixer. 1
2 Design scripts for a continuous-flow microfluidic biochip. (a) An example of the input for Columba S. (b) A script written in Co 2
3 Web-based Columba Platform 3
4 Advanced Edition on Web-based Columba Platform 4
5 Two-dimensional array for Ring 6
6 An example of entity declaration with VHDL 10
7 A simple schematic diagram 13
8 A simple gate-level netlist Script 14
9 Gate and it‘s Input 14
10 Gate and it‘s Output 15
11 Table of flow liquid 15
12 Execution flow of Co’s Language processor 20
13 TOKEN Type Table 21
14 Code Example of Co 22
15 String stream from example code 22
16 Syntax diagram 26

1 Overview

Continuous-flow microfluidic biochips are the mainstream microfluidic technology for cell-based applications, including cell sorting, single cell analysis, and DNA amplification [1]. Figure 1 shows an example of continuous-flow microfluidic biochip. This biochip consists of switches (Figure 1(a)) and mixers (Figure 1(b)) which are functional modules for application execution. Because of the complex structure of biochips, manually designing a biochip involves lots of work, which leads to the need for design automation. Based on this growing demand, Dr.-Ing. Tsun-Ming Tseng developed Columba for automatic design of continuous-flow microfluidic biochips.

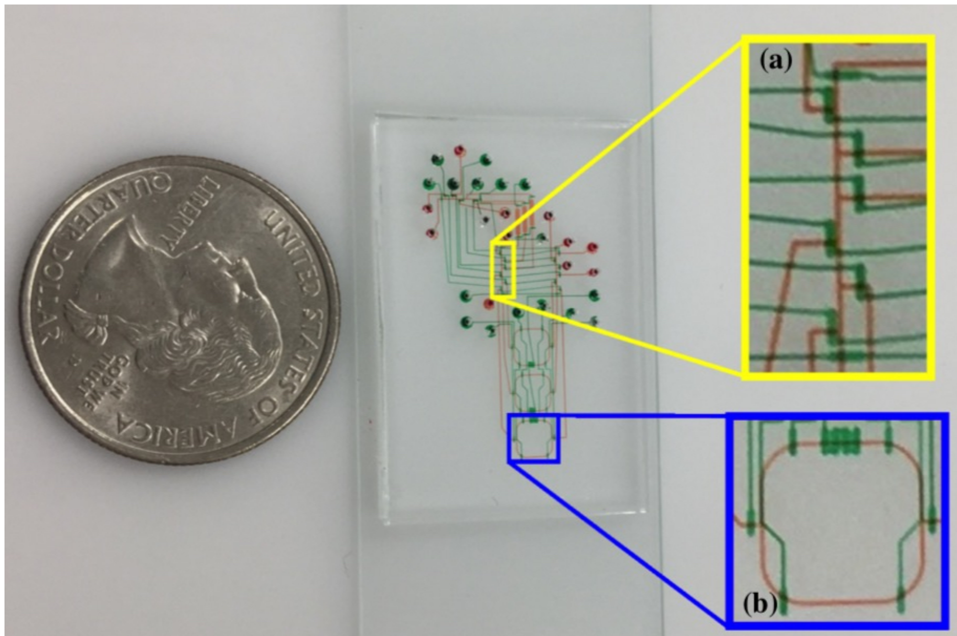

Figure 1: The design of a nucleic acid processor automatically synthesized by the proposed tool Columba. (a) A switch. (b) A rotary mixer.

Columba was originally developed for Linux system. Figure 2 (a) shows an example of the input for Columba S [2] in the original version. Figure 2 (b) shows a script written in Co according to Figure 2 (a). The design generated from this input script is shown in the appendix.

To allow users to design their chips directly on a web browser with various devices, Cloud Columba [3], a web service is provided. In Cloud Columba, users can define modules by filling out forms. Figure 3 shows how the web-based Columba platform creates a new project by filling out the form in the simple edition mode. Nevertheless, when the user wants to be able to define multiple modules at once, it is more convenient and intuitive to define those modules by using programming language. Therefore, the advanced edition mode is also provided in

```
module:

M1 d r 1 1 0 7600 3800 50
M2 d r 1 1 0 7600 3800 50
RC1 d c 0 1 0 0 3000 1200 100
RC2 d c 0 1 0 0 3000 1200 100
i_anti p 1500 1500
i_prob p 1500 1500
i_kin p 1500 1500
o_wf p 1500 1500
o_wb p 1500 1500
o_wf2 p 1500 1500
o_wb2 p 1500 1500
fin
```

```
netlist:

i_anti M1 1
i_anti M2 1
i_prob M1 1
i_prob M2 1
i_kin M1 1
i_kin M2 1
M1 o_wf 0.5
M1 o_wb 0.5
M1 RC1 2
M2 o_wf 0.5
M2 o_wb 0.5
M2 RC2 2
RC1 o_wf2 2
RC1 o_wb2 2
RC2 o_wf2 2
RC2 o_wb2 2
fin
```

```
conflict:
fin

parallel:
2 M1 M2
2 RC1 RC2
fin

group:
fin

testing_module:
OFF
fin
```

(a)

```
1  M1:=Ring(x=7600,y=3800,z=50,pump=true,sv=true,ct=false)
2  M2:=Ring(x=7600,y=3800,z=50,pump=true,sv=true,ct=false)
3
4  RC1:=Chamber(x=3000,y=1200,z=100,pump=true,sv=false,ct=true,spv=false)
5  RC2:=Chamber(x=3000,y=1200,z=100,pump=true,sv=false,ct=true,spv=false)
6
7  i_anti:=Inlet(x=1500)
8  i_prob:=Inlet(x=1500)
9  i_kin:=Inlet(x=1500)
10 o_wf:=Outlet(x=1500)
11 o_wb:=Outlet(x=1500)
12 o_wf2:=Outlet(x=1500)
13 o_wb2:=Outlet(x=1500)
14
15 M1<-(i_anti,i_prob,i_kin)
16 M2<-(i_anti,i_prob,i_kin)
17 o_wf<-(M1,M2)
18 o_wb<-(M1,M2)
19 RC1<-(M1,M2)
20 RC2<-(M1,M2)
21 o_wf2<-(RC1,RC2)
22 o_wb2<-(RC1,RC2)
23 parallel(M1,M2)
24 parallel(RC1,RC2)
```

(b)

Figure 2: Design scripts for a continuous-flow microfluidic biochip. (a) An example of the input for Columba S. (b) A script written in Co

Figure 3: Web-based Columba Platform

the Columba, allowing the user to run the code through the web-based Columba platform by entering the script. Figure 4 shows how the web-based Columba platform creates a new project in the advanced edition mode. If the input script passes the specific detection such as grammar check and meets the design rules, the design drawing will be generated by the back-end server of Cloud Columba.

However, the syntax of the input script is complex. For example, in Figure 2, each row defines a module, where the module name is given first, and after that the parameters required to define the module are given, for example, the length on the x-axis. This syntax is easy for the developer to understand, but for the user, it is not easy to understand and learn. Since the parameter names corresponding to each value are not explicitly given, the user may forget the correct parameter order when inputting the parameters, miss some parameters, or input the wrong parameters. The original compiler for Columba's input does not perform grammatical and numerical verification on the script entered by the user. When the user enters a script, due to the lack of a verification mechanism, the user may need to spend much time for error correction. For example, the user may take a long time to correct the error but end up finding that the error is just made by repeatedly declaring the same variable. To let the user spend more time on designing the chip

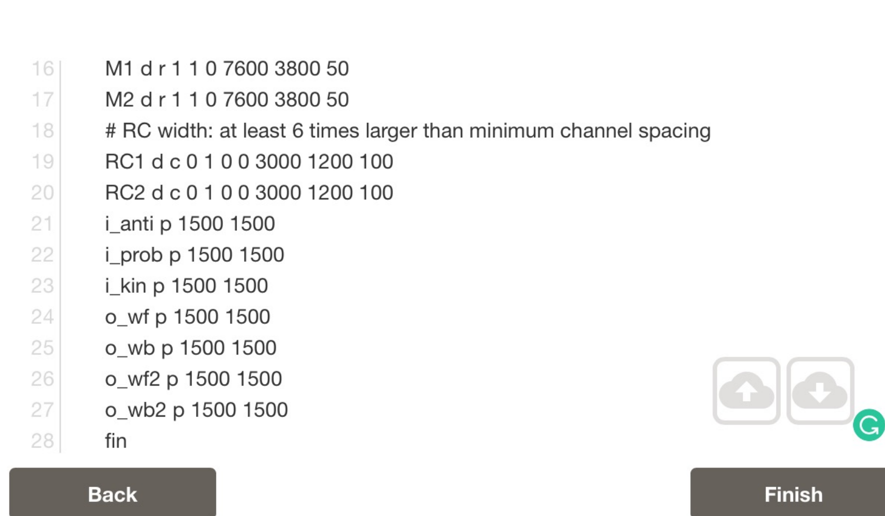

Figure 4: Advanced Edition on Web-based Columba Platform

structure but not on correcting errors, a programming language for writing the input script needs to be developed.

Considering that the users of Columba do not necessarily have programming experience, the designed programming language should be simple and easy to learn. And to achieve that, I propose a new programming language Co in this thesis which refers to many different forms of programming language to adopt their strengths and avoid their weaknesses. The thesis consists of four parts. In the first part, some basic knowledge about Columba will be introduced. In the second part, different programming paradigms and the characteristics of some mainstream programming languages will be analyzed. The third part will detail the characteristics of the proposed language Co. In the final part, the technical details of the implementation of Co will be discussed.

2 Data preparation for designing a continuous-flow microfluidic biochip

Functional modules such as mixers, reaction chambers and switches are components of a continuous flow microfluidic biochip. As a tool for generating continuous-flow microfluidic biochip designs, Columba needs the user to specify the type, number, dimension, and logical connection of the required mixers, reaction chambers, inlets and outlets, as well as particular execution constraints. Therefore, it is necessary to analyze which kind of information Columba needs and how to organize it. In this chapter, all the information that Columba needs will be analyzed and how to make it more structured will be discussed.

2.1 Rule

To improve the chances that the devices will be easily manufactured and operational, some basic design rules should be taken. According to the Stanford Foundry standard [4], Columba takes the following design rules [5]:

1. the width of flow channels and valves is 100µm
2. the width of control channels is 30µm
3. the minimum spacing distance between channels is 100µm
4. the size of a flow inlet and control inlet is 1mm x 1mm
5. the center-to-center distance between inlets/outlets is 2mm

In addition to these parameters, before starting a design, Columba still needs to get user-defined information about the height of flow layer, the height of control layer, the height of overlap layer, the height of flow channel and the height of control channel.

These ten parameters are the primary criteria for the design created by Columba and need to be determined at the very beginning of the user's start of a new design project and will affect the subsequent container definitions.

In the original Columba [3], rules and other definitions like module declarations are separate. The originally set rules refer to the basic design rules set by Stanford University [6]. In the advanced module, users are not allowed to change the design rule settings themselves.

2.2 Module

In Columba, container, inlet and outlet are defined as modules.

Currently, Columba only supports two kinds of containers: chamber and ring. The chamber has four different boolean properties that default to false: whether the pump is open, whether the sieve valve is on, whether the cell trap is on, and whether the normal valves at the end is on. Ring does not have the attribute which defines normal valves at the end is on or off.

In addition, the user needs to define the side length of the container on the x, y, z-axis.

The inlet and outlet differ from the container in that the side length on the x and y-axis has been defined in the rule by the size of a flow inlet and control inlet. In particular, the values of the inlet and outlet on the x and y-axis are exactly the same. In addition, in the original input, Columba will not detect if the input value is within the allowed value range.

Figure 2 (a) shows an example of input from Columba S [5]. By analyzing the contents of the input, each defined module can be saved in a one-dimensional array. For each new module of the same type, such as chamber, it can be saved in the same form, so it is easy to understand that for multiple modules of the same type, a two-dimensional array table can be formed to store the information. According to the example given by Figure 2, a two-dimensional array can be created in the form like Figure 5.

	X	Y	Z	Pump	Sieve_Valve	Sell_Trap
M1	7600	3800	50	TRUE	TRUE	FALSE
M2	7600	3800	50	TRUE	TRUE	FALSE

Figure 5: Two-dimensional array for Ring

2.3 Netlist

Columba users need to describe the logical connections among microfluidic modules. In this section I research how to describe the connections between modules in a concise language.

Originally, Columba accepts a simple script description of netlist as input. Columba users need to define the flow direction of the liquid, i.e., specify the modules that send the liquid and the modules that accept the liquid.

2.4 Conflict

T.-M. Tseng, et al. [2] mentioned in their article that if users want to achieve precise control of fluid manipulations on Microfluidic large-scale integration (mLSI), it is first necessary to define which modules can not share the pressure. The definitions of conflict and parallel are similar to the definition of netlist. They define the relationship between modules and modules.

The only difference is that it is not necessary to indicate the direction of the relationship when users define which modules are parallel or conflict in the script.

2.5 Parallel

Mengchu, Li, et al. [7] mentioned in their article that a way to improve throughput is to perform several replicate operations at the same time on the same chip. In this case, we call the replicated operations parallel operations. Contrary to the definition of conflict, parallel defines a set of modules that can share pressure to perform parallel operations. However, those modules that share pressure must be a set of modules with the same attributes. In the original Columba, there is no error reporting mechanism to verify this rule.

Parallel and conflict are essentially defining the relationship between containers. ”Container” in this article refers to the general name of all module types except inlet and outlet in microfluid-chip design.

Co’s language design is for Columba S [2], a version of the Columba family of tools. Columba S

applies a new architectural framework and a straight channel routing discipline, and synthesizes multiplexers for efficient and reconfigurable valve control. It enables the design of a platform for large and complex applications.

3 Designing language grammar of Co

3.1 Programming Paradigm

Programming languages can be classified by their features, classifying them into programming paradigms is a representative way. Some paradigms are concerned more about the style of syntax and grammar while some paradigms are more concerned about how to organize the code or implications for the execution model of the language. For example, whether side effects should be allowed. There are a number of programming paradigms, such as object-oriented, event-driven, imperative, declarative, and language-oriented. Although there are multiple programming paradigms [8], a programming language must not only combine one programming paradigm. In practice, multi-paradigm programming languages are better suited for building tools for large and complex heterogeneous systems [9].

3.1.1 Object-oriented Programming (OOP)

Timothy, Budd [10] wrote in his book that in object-oriented programming(OOP), objects are initiated by passing messages to the object which is responsible for the behavior. The message contains the parameters required to complete the request. When the receiver receives the message, the receiver responds to the message and executes the corresponding function to complete the request.

Alan Curtis Kay [11], an American computer scientist summarized the basic characteristics of OOP listed below.

1. Everything can be seen as an object.
2. When objects are related to each other, the object can request other objects to perform

calculations by executing a specific function. The way to communicate among objects is to send and receive messages to each other. The content of the message is a request for a particular function and the parameters needed to complete the request.

3. Each object has its own storage space for storing other objects.

4. Each object is an instance of a class. Class is a group of similar objects.

5. Multiple instance objects of the same class can execute the same function.

6. Classes can have an inheritance hierarchy.

Objects will encapsulate state (data values) and behavior (functions). Object-oriented programming is very versatile. Both subtle and complex problems can be solved through it. Because in the future version of Columba, users can customize the module, that means, users can define a module consisting of multiple containers. This process matches the third and the sixth characteristics of object-oriented programming above.

3.1.2 Functional Programming

Michael Feathers [12] pointed out in his book that the difference between functional programming and object-oriented programming is that object-oriented programming makes people easier to understand the code by encapsulating uncertain factors, while functional programming makes the code more understandable by minimizing uncertainties. OOP encourages developers to build different data structures for specific problems, and to combine specialized operations with these data structures through functions.

However, functional programming languages advocate the use of highly optimized operations designed for these limited data structures on a limited number of key data structures, such as list, set, and map, to form the basis of programming [13]. Developers then organize their own data structures and use higher-order functions to adjust these basic parts according to their own tasks.

According to the above analysis of the data structure of Columba's input, the new design language Co also hopes to avoid variable states as much as possible, and convert and store the contents of input with limited data structures, such as array. This is the reason why Co does not allow default values. When defining each module, all parameter values need to be given,

which facilitates compilation of the compiler, reduces compile time and reduces the possibility of unknown errors. And all the parameters need to be given according to a certain order, which is convenient for the compiler to convert the input script into the required data format.

3.2 VHDL

Stephen Brown and Zvonko Vranesic [14] detailed how to design digital logic circuits with VHDL language in their book. In the VHDL language, the basic unit of design is a basic design entity. A basic design entity can be a simple logic gate, such as an AND gate, or a microprocessor or even a system. A design entity is composed of two parts, the Entity Declaration and the Architecture Body. The Entity Declaration specifies the input and output interface signals or pins of the design entity, while the structure part defines the specific construction and operation (behavior) of the design entity.

Co is also a language developed for the design of physical microfluidic chip. The language format of VHDL can be referred to design Co.

3.2.1 Entity Declaration

The entity declaration of any design entity will contain a generic parameter description which specifies parameters such as (m:TIME:=1ns) and port descriptions to describe the design entity and its external interface. Figure 6 shows a simple example:

```
Entity Name of Entity is
    port(
    a : IN STD_LOGIC;
    b : OUT STD_LOGIC
    );
End Name of Entity;
```

Figure 6: An example of entity declaration with VHDL

The following forms are used when defining a port:

Port (port name {, port name}: Direction Type of the data);

When referring to the form of VHDL, in Columba, each container can be regarded as a design entity. Each container also has inputs and outputs, but unlike VHDL, the form of inputs and outputs is not complicated at present, so we can simply define that the flow direction viewed from the container has only two options: inflow or outflow.

We can write pseudo code to show how to define the flow of liquid between modules by using the pseudo VHDL form. The following pseudo code is an example written with reference to VHDL according to the input example of Figure 2.

```
1  Entity M1 is
2    port(
3    p_anti  : IN;
4    p_kin   : IN;
5    wf      : OUT;
6    wb      : OUT;
7    RC1     : OUT);
8     End M1;
```

According to the analysis of the input data in the second section, when designing Co, there will be two lists, an inflow list and an outflow list for each module. The inflow list contains all the names of the modules that send liquid flows to the current module. The outflow list contains all the names of the modules which accept the liquid from the current module.

3.3 JavaScript

JavaScript is the programming language of the Web [15]. Most of the modern websites use JavaScript. Cloud Columba is a web platform, so JavaScript is also an option for Co.

The following code shows an example of how to write code to define a Chamber class for Columba with JavaScript. This code defines a class called ChamberNode, and the last line shows how to generate a concrete object.

```
function ChamberNode(name,x,y,z,p,s,c,spv) {
                this.id=name;
                this.x=x;
                this.y=y;
                this.z=z;
                this.pump=p;
                this.sieve_valve=s;
                this.cell_trap=c;
                this.seperate_valve=spv;
                this.inflow=[];
                this.outflow=[];
            }
m1=new ChamberNode(name,x,y,z,pump,sv,ct,spv)
```

When writing JavaScript code, programmers may encounter a lot of problems because they don't pay attention to the different scopes of variables. There is no such problem in Co because Co does not have the difference between global variables and local variables. Variables can be referenced throughout the script. Moreover, there is no problem with the order of declarations in Co. This means that the appearance order in the code has no effect.

3.4 Hardware design language

Newnes [16] showed how engineers can define the relationship between modules in a standardized language in his book. When we design the language grammar for defining netlist, we can refer to how the hardware design language describes the relationship between modules. According to Newnes's method [16], I made a simple schematic diagram (Figure 7). A textual description of this diagram is called a gate-level netlist. According to the above diagram and the format of the statements mentioned in the book, the relationship between modules can be clearly seen in Figure 8.

It can be clearly seen from Figure 8 that when the user defines the relationship between the modules, which input and output are defined first, and then define which input and output are connected to each module, the graph can be converted into a text description very clearly. At the same time, when the name of the input and output is passed as a parameter to the module, it is necessary to indicate what type it is.

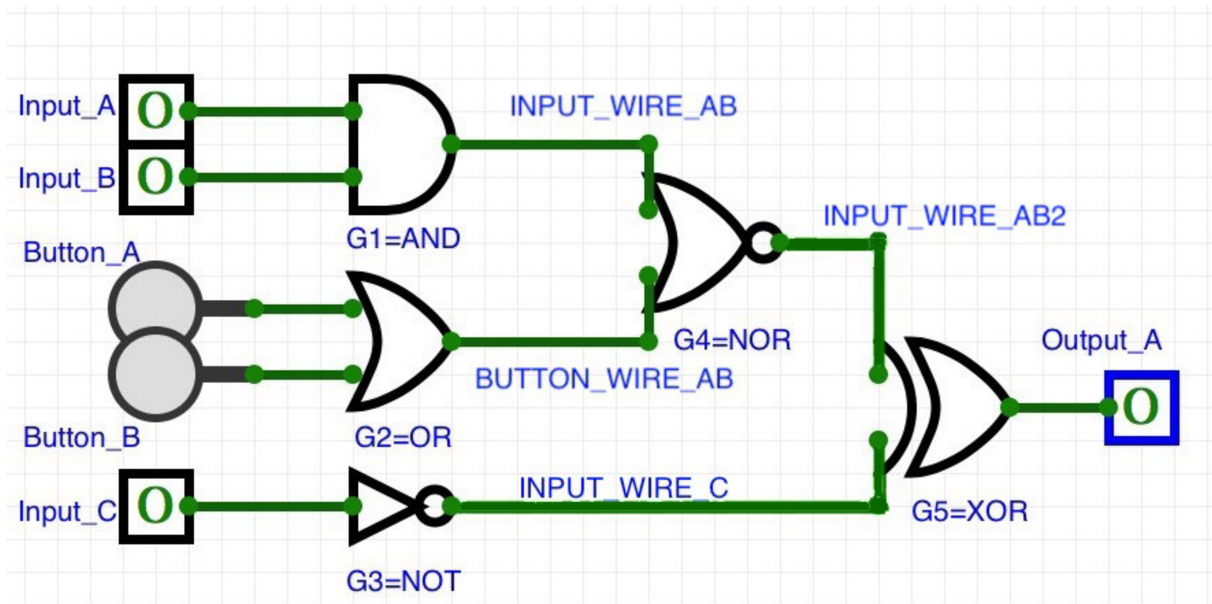

Figure 7: A simple schematic diagram

The textual description may still be less intuitive. We found that the existence of links between modules can be broken down into binary relationships. It would be more intuitive to use a two-dimensional array to display. With two module names, whether there is an association between the two modules can be determined.

From the Figure 8 we can get the relationship between gate and its input, and gate and its output. The following two boolean tables (Figure 9 and Figure 10) can clearly show whether there is any association between modules.

Similarly, we can make each container in Co have its own input and output tables according to the input from users to determine whether there is a liquid flow relationship. The advantage of doing this is that it is possible to quickly determine whether the design is a connected graph by a certain algorithm when each module has its own adjacency list.

Figure 2 (a) shows an example of input from Columba S [2]. According to the above analysis, we can also obtain the relationship between each two modules. Although the resulting design is an undirected graph, the design is considered to be a directed graph when the user specifies the direction of liquid flow. Figure 11 shows that the first column is the module that flows out of the liquid, and the first row defines the module that accepts the liquid.

```
BEGIN CIRCUIT=TEST
    INPUT Input_A, Input_B, Input_C, Button_A, Button_B;
    OUTPUT Output_A;
    WIRE INPUT_WIRE_AB, BUTTON_WIRE_AB, INPUT_WIRE_C;

    GATE G1=AND  (IN1=Input_A, IN2=Input_B, OUT1=INPUT_WIRE_AB);
    GATE G2=OR   (IN1=BUTTON_A, IN2=BUTTON_B, OUT1=BUTTON_WIRE_AB);
    GATE G3=NOT  (IN1=Input_C, OUT1=INPUT_WIRE_C);
    GATE G4=XOR  (IN1=Input_WIRE_AB2, IN2=Input_WIRE_C, OUT1=Output_A);

END CIRCUIT=TEST;
```

Figure 8: A simple gate-level netlist Script

	Input_A	Input_B	Input_C	Button_A	Button_B	Output_A	INPUT_WIRE_AB	INPUT_WIRE_AB2	INPUT_WIRE_C	BUTTON_WIRE_AB
G1	1	1	0	0	0	0	0	0	0	0
G2	0	0	0	1	1	0	0	0	0	0
G3	0	0	1	0	0	0	0	0	0	0
G4	0	0	0	0	0	0	0	1	1	0

Figure 9: Gate and it‘s Input

3.5 R

R is a programming language for data analysis and statistics and is widely used in every field that requires statistics or data analysis, including finance, marketing, medicine, etc.

In R, variables are assigned by <- or =. Programmers do not need to declare them before assigning values to variables. Co refers to one of these two features, using <- to assign the value of the Netlist property of the variable. Because the arrow can show directionality very intuitively. For example, in the flow chart, the arrow indicates the starting point of the process to each subsequent node until the endpoint.

In the previous chapter, it is known that the input data should contain the attribute values of multiple different modules. It should also contain all the existing binary relationships between different modules. A one-dimensional array can be used to store a single defined module object. And by adding dimensions, the stored information can be continually increased.

The following code shows how to create some one-dimensional array to store several module's information based on the script of the Columba S shown in Figure 2. For example, the first line creates an array with the variable name "M1", which stores the attribute values given in

	Input_A	Input_B	Input_C	Button_A	Button_B	Output_A	INPUT_WIRE_AB	INPUT_WIRE_AB2	INPUT_WIRE_C	BUTTON_WIRE_AB
G1	0	0	0	0	0	0	1	0	0	0
G2	0	0	0	0	0	0	1	0	0	0
G3	0	0	0	0	0	0	0	0	1	0
G4	0	0	0	0	0	1	0	0	0	0

Figure 10: Gate and it‘s Output

	M1	M2	RC1	RC2	p_anti	p_prob	p_kin	wf	wb	wf2	wb2
M1	0	0	1	0	0	0	0	1	1	0	0
M2	0	0	0	1	0	0	0	1	1	0	0
RC1	0	0	0	0	0	0	0	0	0	1	1
RC2	0	0	0	0	0	0	0	0	0	1	1
p_anti	1	1	0	0	0	0	0	0	0	0	0
p_prob	1	1	0	0	0	0	0	0	0	0	0
p_kin	1	1	0	0	0	0	0	0	0	0	0
wf	0	0	0	0	0	0	0	0	0	0	0
wb	0	0	0	0	0	0	0	0	0	0	0
wf2	0	0	0	0	0	0	0	0	0	0	0
wb2	0	0	0	0	0	0	0	0	0	0	0

Figure 11: Table of flow liquid

parentheses.

```
1 M1<-c("r",1,1,0,7800,3600,50)
2 M2<-c("r",1,1,0,7800,3600,50)
3 RC1<-c("c",0,1,0,0,3000,1200,100)
4 RC2<-c("c",0,1,0,0,3000,1200,100)
5 i_anti<-c("p",1500,1500)
6 i_prob<-c("p",1500,1500)
```

In Co, each defined container is treated as an object. And JavaScript is used to write Co's compiler. In JavaScript, the data structure used to store object information can be thought of as a sparse array that can be expanded infinitely. The way R defines a one-dimensional array can be referred to. In Co, a similar way is also used to define modules separately.

4 Core Co

4.1 Lexical Structure

The lexical structure of a programming language is a set of basic rules that describe how to write a program with the language [17]. It specifies rules such as how variable names are, how to write comments, and how to separate between program statements.

4.1.1 Character Set

Case Sensitivity Co is not a absolute case-sensitive language. Language keywords must not always be typed with a consistent capitalization of letters. However, variables is case-sensitive. For example, ”M1” and ”m1” are two distinct variable names.

Whitespace, Line Breaks Spaces that appear between tokens in programs is ignored by Co. Mostly, Co also ignores line breaks. However, it is not allowed to have a line break inner a complete statement, for example, when defining a module. But between the definitions of two different modules, a line break is allowed.

Comments Co supports only one style of comments. Texts between the characters /* and */ is treated as a comment.

Parentheses Unlike other languages, Co does not require a semicolon as the end of each definition. In Co, when the right parenthesis appears, it means that the definition ends, and line breaks after the right parenthesis will be ignored.

4.1.2 Reserved Words

Co reserves some identifiers as the keywords of the language itself. Users can not use these words as identifiers in their programs:

Ring	Chamber	x	y	z	Pump
SV	CT	SPV	Inlet	Outlet	Parallel
true	false	Netlist			

Co also reserves certain keywords that are not currently used by the language but which might be used in future versions. The following shows some example of reserved words:

if var while print int error

4.2 Types, Values, and Variables

Programs work by manipulating values, such as the number 123 or text ”abc”. Types are the kinds of values that can be represented and manipulated in a programming language.

Types Co types have two categories: primitive types and object types. Co’s primitive types include only numbers and booleans.

Numbers Unlike many languages, all numbers in Co are represented as integer values. Because all the values of the microfluidic chip itself are in nanometers, it is meaningless to be accurate to floating point under the existing manufacturing process.

Variables Any Co value that is not an interger number, a boolean, is an object. An object is a collection of properties where each property has a name and a value.

In co, all objects have several specified properties, and the user only needs to assign values to the corresponding properties.

4.2.1 Variable Declaration

In co, there are only following types of declarations for variables.

```
M1:=Ring(x=5400,y=4600,z=50,pump=true,sv=true,ct=false)
RC1:=Chamber(x=3000,y=1200,z=100,pump=true,sv=false,ct=true,spv=false)
p_i:=Inlet(x=1500)
p_lo:=Outlet(x=1500)
M1<-(p_i, p_lo)
```

At the very beginning of the definition, it is always necessary to first determine which object is to be operated on.

Then use := or <- to determine whether to define the physical properties of this variable or to define the connection between this variable and other variables.

If the user defines the physical properties of the variable, first determine the type of the variable, and secondly give different parameter values, that is, its property value, depending on the type.

If the user defines the connection between the variable and other variables, all the variables that will flow out of the microfluid to this variable are given in the following parentheses.

4.3 Expressions

An expression is a phrase of Co, and a Co interpreter defines a relationship between a set of variables based on an expression. Because when defining the relationship of a set of variables, if users specify a variable as an assignment object each time in the way defined above, it will make people feel that there is a different priority in the relationship of a set of variables. To more clearly indicate that there is no precedence in a set of variables that have a relationship (such as Parallel Set), the expression is used in Co to define a particular relationship of a set of variables.

4.3.1 Parallel

The following code shows how to define a set of variables with parallel relationship. First give the relationship name and then give a set of variable names with this relationship in parentheses.

```
parallel(M1,M2,M3,M4)
```

5 Compiling of Co

The language used to design a program is called a programming language, such as Java, Ruby, C++. Programmers use software that matches each programming language to execute programs written in that language, often referred to as language processors. However, some programming languages can be interpreted and executed directly by hardware without the need for a corresponding language processor. This language is called machine language.

The essence of machine language is a binary number with a long number of digits. It is very difficult to understand if programmers read it directly. So assembly language was developed to represent this huge number, making it easy to understand. If someone want to execute a program written in assembly language, he or she usually need to use software to convert it into machine language. This kind of software is called assembler, which is a basic language processor.

Language processors can be roughly divided into interpreters and compilers.

Interpreter The interpreter is a computer program that translates high-level programming languages directly and line by line. The interpreter does not translate the entire program at once, just like a ”middleman”. Every time when running a program, the program’s code must be first converted to another language and then be run it again, so the interpreter’s program runs slower. It runs immediately after translating a line of program descriptions, then translates the next line, and then runs, so it keeps going.

Compiler A compiler can convert a program written in one language into a program in another language. Usually, the original program is converted into a machine language program. This behavior of converting the original language into another language is called compilation. If the compiler does not convert the original program directly into machine language, it is generally called source code translator.

A script written in Co will not be directly converted to machine language, but will be converted to Json first, because Co will run on the web platform, and the developer of Columba web platform chooses Json as the format for transferring data, and finally, Json will be converted into the language which Columba can accept. According to the above classification, the language processor developed for Co belongs to the compiler.

Whether it's an interpreter or a compiler, the language processor's initial handling is similar. First, the source code is lexically analyzed, and the input text is divided into more smaller string units. The divided string is called token, and then the processor performs syntax analysis to convert the token arrangement into an abstract syntax tree. From lexical analysis to parsing, the interpreter does the same thing as the compiler, but then the compiler converts the abstract syntax tree into other languages, and the interpreter performs the computation while parsing the abstract syntax tree. Figure 12 shows this process.

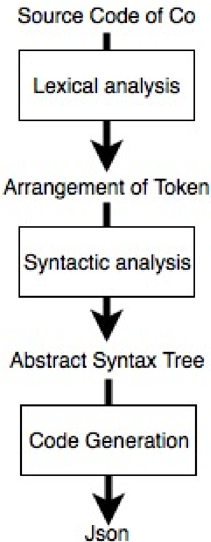

Figure 12: Execution flow of Co's Language processor

5.1 Lexical analysis

In order to translate a program from one language to another, the compiler must first separate the various modules of the program, figure out its structure and meaning, and then combine the modules in another way.

The first step is lexical analysis, which breaks down the input into separate lexical symbols, called ”TOKEN”.

Lexical analysis generates a series of variable names, reserved words, and special symbols based on the input character stream while discarding white space and blank line symbols between words and comments. Lexical words in programming languages can be categorized into a limited set of TOKEN types.

Figure 13 shows all the token types defined in Co. In the actual lexer, more tokens are defined to facilitate future development of Co. All reserved words are not case sensitive and cannot be used to name variables.

Refer to C and Java for the definition of identifiers. The identifier is a sequence of letters and numbers. The first character must be a letter. Users can use an underscore ”_” in the identifier. Blanks, newlines, and comments in the input will be ignored. And the token of the EOF type will be returned at the end to indicate the end of the input stream.

TYPE	EXAMPLE
ID	M1 M2 RC1
NUM	4800 5600 0
DEVICE DEFINE SYMBOL	:=
NETLIST DEFINE SYMBOL	<-
LEFTPAREN	(
RIGHTPAREN	)
ASSIGN SYMBOL	=
COMMA SYMBOL	,
RESERVED WORD	RING CHAMBER
	X Y Z PUMP SV
	CT SPV INLET
	OUTLET PARALLEL
END SYMBOL	EOF

Figure 13: TOKEN Type Table

Lexical analysis will break the code in Figure 15 into the string shown in Figure 14.

```
M1:=RING(x=7600,y=3800,z=50, pump=0,SV=0,CT=0,SPV=0)
M2:=RING(x=7600,y=3800,z=50, pump=0,SV=0,CT=0,SPV=0)
i_anti:=inlet(x=1500)
M1<-(i_anti)
parallel(M1,M2)
```

Figure 14: Code Example of Co

IDENTIFIER (M1) DEVICEDEFINE RING LEFTPAREN X ASSIGN NUM(7600) COMMA Y ASSIGN NUM (3800) COMMA Z ASSIGN NUM(50) COMMA PUMP ASSIGN NUM(0) COMMA SV ASSIGN NUM(0) COMMA CT ASSIGN NUM(0) COMMA SPV ASSIGN NUM(0)RIGHTPAREN IDENTIFIER(M2) DEVICEDEFINE RING LEFTPAREN X ASSIGN NUM(7600) COMMA Y ASSIGN NUM(3800) COMMA Z ASSIGN NUM(50) COMMA PUMP ASSIGN NUM(0) COMMA SV ASSIGN NUM(0) COMMA CT ASSIGN NUM(0) COMMA SPV ASSIGN NUM(0) RIGHTPAREN IDENTIFIER(i_anti) DEVICEDEFINE INLET LEFTPAREN X ASSIGN NUM(1500) RIGHTPAREN IDENTIFIER(M1) NETLIST LEFTPAREN IDENTIFIER(i_anti) RIGHTPAREN PARALLEL LEFTPAREN IDENTIFIER(M1) COMMA IDENTIFIER(M2) RIGHTPAREN EOF

Figure 15: String stream from example code

Columba generates design drawings through a web platform. JavaScript is the best choice for compiling a Co script, because JavaScript is a web-oriented programming language, and all modern web browsers include a JavaScript interpreter.

The following code shows how to define the class token with JavaScript, and thanks to KJlmfe's effort[18]. "type" is used to store a token type such as "DEVICEDEFINE_TOKEN". "text" is used to store error messages when an error occurs.

```
function Token(type, text) {
            this.type = type;
            this.text = text;
        }
```

The following code shows some Examples how to define tokens with JavaScript. "Token.tokens" is a class attribute of Class Token. It is also an object. Token.tokens.DEVICEDEFINE_TOKEN is an attribute of this object. The following code shows how to extend other attributes of this object.

```
Token.tokens.DEVICEDEFINE_TOKEN = Token.tokens.NEWLINE_TOKEN + 1; //:=
Token.tokens.COMMA_TOKEN = Token.tokens.DEVICEDEFINE_TOKEN + 1; //,
Token.tokens.PARALLEL_TOKEN = Token.tokens.COMMA_TOKEN + 1;//parallel
Token.tokens.RING_TOKEN = Token.tokens.PARALLEL_TOKEN + 1;//ring
Token.tokens.CHAMBER_TOKEN = Token.tokens.RING_TOKEN + 1;//chamber
Token.tokens.PUMP_TOKEN = Token.tokens.CHAMBER_TOKEN + 1;//pump
Token.tokens.SV_TOKEN = Token.tokens.PUMP_TOKEN + 1;//sieve_valve
Token.tokens.CT_TOKEN = Token.tokens.SV_TOKEN + 1;//cell_trap
Token.tokens.SPV_TOKEN = Token.tokens.CT_TOKEN + 1;//seperate_valve
Token.tokens.INLET_TOKEN = Token.tokens.SPV_TOKEN + 1;//inlet
Token.tokens.OUTLET_TOKEN = Token.tokens.INLET_TOKEN + 1;//outlet
Token.tokens.X_TOKEN = Token.tokens.OUTLET_TOKEN + 1;//x_axis-length
Token.tokens.Y_TOKEN = Token.tokens.X_TOKEN + 1;//y_axis-length
Token.tokens.Z_TOKEN = Token.tokens.Y_TOKEN + 1;//z_axis-length
Token.tokens.NETLIST_TOKEN = Token.tokens.Z_TOKEN + 1;//netlist
```

The following code shows how to return an Identifier's token. When lexical analysis of a piece of code, the characters in the code will be read one by one, and ”nextChar()” is the function used to return the next character. If each character in a string of characters before a white space is a number or a letter or an underscore, then an ”IDENTIFIER_TOKEN” is returned. And by comparison with the keyword, it is determined whether it is a variable name or a keyword.

```
case Scanner.IDENTIFIER_STATE:
    var next_char = this.reader.nextChar();
if ((next_char == '_') || (next_char >= '0' && next_char <= '9') ||
(next_char >= 'a' &&
        next_char <= 'z') || (next_char >= 'A' && next_char <= 'Z')) {
    this.bufferStr += next_char;
    return this.nextToken();
} else if (next_char == -1) {
    this.state = Scanner.END_STATE;
} else {
    this.state = Scanner.START_STATE;
    this.reader.retract();
}

switch (this.bufferStr.toLowerCase()) {
    case "parallel":
        return this.makeToken(Token.tokens.PARALLEL_TOKEN);
    case "chamber":
        return this.makeToken(Token.tokens.CHAMBER_TOKEN);
```

```
20        case "ring":
21            return this.makeToken(Token.tokens.RING_TOKEN);
22        default:
23            return this.makeToken(Token.tokens.IDENTIFIER_TOKEN, this.bufferStr);
24    }
25    break;
```

5.2 Syntactic analysis

Figure 16 shows the grammatical structure of Co. According to this grammatical structure and Figure 2, we can get the script written by Co.

After lexical analysis, the program has been decomposed into tokens. The main task of parsing is to analyze the relationship between tokens, such as determining which tokens belong to the same expression or statement, and dealing with the pairing of left and right brackets. Grammatical errors will also be checked out at this stage.

In Co, when defining the physical properties of a variable, the order of the parameters is fixed and cannot be changed. If the user changes the order of the input parameters or lacks the assignment to some parameters, it is considered a syntax error. For example, according to Figure 16, to define a module's Netlist, the name of the module must first be given, then an arrow, then a left parenthesis. And then the name of the module that will flow out the liquid into this module should be indicated. If more than one module will flow out the liquid into this module, separate them with commas and end the definition with a closing parenthesis. The following code shows how to perform a syntax check on the Netlist definition statement.

```
1   Parser.prototype.parseNetlistExpression = function(name) {
2       var name = name;
3       this.nextToken();
4       var netlistnode = new NetlistNode(name, this.scanner.currLine);
5       var right = true;
6
7       if (this.lookahead() != Token.tokens.LEFTPAREN_TOKEN) {
8           right = false;
9       } else {
10          this.nextToken(); //cosume ( token
11      }
```

```
12      if (this.lookahead() != Token.tokens.IDENTIFIER_TOKEN) {
13          right = false;
14      } else {
15          this.nextToken(); //cosume id token
16          netlistnode.push(this.currentToken.text);
17      }
18
19      while (this.lookahead() != Token.tokens.RIGHTPAREN_TOKEN &&
20      this.lookahead() != Token.tokens.NEWLINE_TOKEN) {
21          if (this.lookahead() != Token.tokens.COMMA_TOKEN) {
22              right = false;
23              return;
24          } else {
25              this.nextToken(); //cosume ) token
26              if (this.lookahead() != Token.tokens.IDENTIFIER_TOKEN) {
27                  right = false;
28              } else {
29                  this.nextToken(); //cosume id token
30                  netlistnode.push(this.currentToken.text);
31              }
32          }
33      }
34      if (this.lookahead() != Token.tokens.RIGHTPAREN_TOKEN) {
35          right = false;
36      } else {
37          this.nextToken(); //cosume id token
38      }
39      if (!right) { log("Line" + this.scanner.currLine + ": (Syntax Error)
40      Expecting a statement like: ID < -(ID, ID, ID)"); }
41      if (netlistnode != undefined) { return netlistnode; } else return null;
42  }
```

The same principle is used for other grammar detections. The order of the parameters for each variable definition is specified first, and then checked one by one. In the future, if other types of container definitions are to be extended, the same method can be used to expand the detection function.

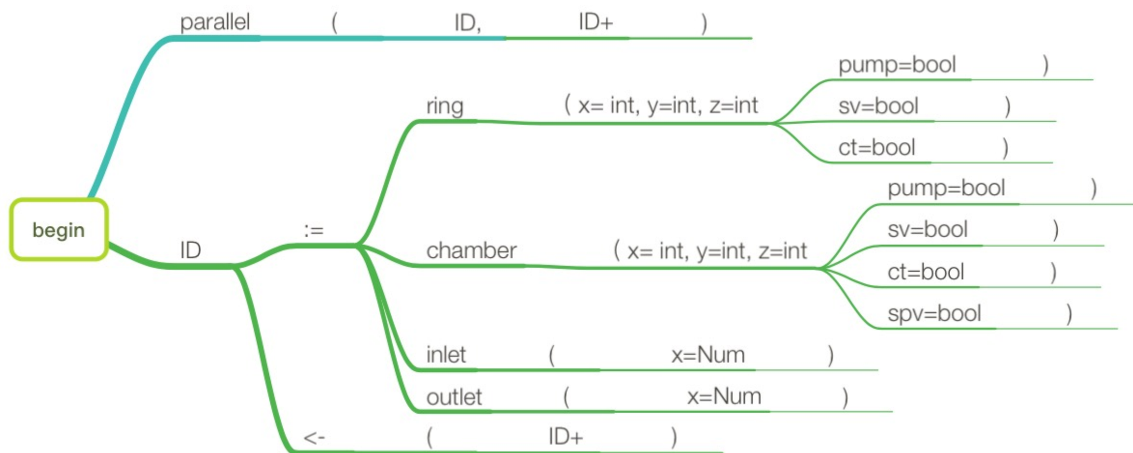

Figure 16: Syntax diagram

5.3 Semantic analysis

The traditional semantic analysis involves linking the definitions of variables to their respective uses, examining whether each expression has the correct type, and translating the abstract syntax into a simpler representation that is suitable for generating machine code. But in this section, it is mainly to discuss whether the logic in the code written by the user is true.

All the items that will be detected at this stage are discussed below.

5.3.1 Detection of module

At least an Inlet, an Outlet, and a Container should be defined Then a basic connected graph can be constructed.

All variable names that appear in the script must be defined It means that the physical attribute values and the connections to other variables should be assigned.

A variable name can be only once defined It is not possible to repeatedly define modules with a same variable name.

The range of values of all modules on the x, y, and z-coordinates will be detected

- Ring's Definition: x>=4800, y>=2400, z<=150
- Chamber's Definition: x>=1200, y>=1200, z<=150
- Inlet's Definition: 500<= x <= 2000
- Outlet's Definition: 500<= x <= 2000

5.3.2 Detection of Parallel's Definition

One Container can be only once defined Because parallel relationships are transitive, when a container is defined to be parallel to another container, the container cannot appear in other parallel definitions. The following shows an example should be avoided.

```
parallel(m1,m2)
parallel(m1,m3)
```

According to this code, it is actually defined that the three containers m1, m2, and m3 are parallel to each other at the same time. In Co, separate definitions are not allowed and must be written in the following form

```
parallel(m1,m2,m3)
```

Module type detection Any module in the parallel Statement cannot be an inlet or an outlet, it can only be a container.

Parameter detection All parallel containers can only have different variable names. All other parameters must be the same.

5.3.3 Detection of Netlist's Definition

Inlet and outlet can not be directly connected At least one container should be defined between any pair of an inlet and an outlet.

A module can not be connected to itself The following shows an example should be avoided.

```
1 m1<-Netlist(m1)
```

5.3.4 Detection of Connected Graph

Only when the above tests pass, Co's compiler will detect whether the design described by this script is a connected graph. Columba can only generate connected graphs. A graph is said to be connected if every pair of vertices in the graph is connected, which means that for every pair of vertices in the graph contains a path from one vertice to another.

According to Section 2, for each defined module, two adjacency tables are generated, one for storing all the modules that flow out of the liquid to the module, and one for storing all the modules that have received the liquid from the module.

First, the Co compiler will check if the two adjacency lists of each module are empty.

In particular, the Inlet will only have an adjacency list in the outflow direction, and the outlet will only have an adjacency list in the inflow direction.

In order to detect whether this design is a connected graph, the DFS(Depth-First-Search) algorithm is used to detect if all modules are connected to each other.

Only when it is confirmed that the design is a connected graph and all the previous tests have passed, the source code required by Columba will be generated.

The following code has been all tested to generate an effective design.

```
M1:=Ring(x=7600,y=3800,z=50,pump=true,sv=true,ct=false)
M2:=Ring(x=7600,y=3800,z=50,pump=true,sv=true,ct=false)

RC1:=Chamber(x=3000,y=1200,z=100,pump=true,sv=false,ct=true,spv=false)
RC2:=Chamber(x=3000,y=1200,z=100,pump=true,sv=false,ct=true,spv=false)

i_anti:=Inlet(x=1500)
i_prob:=Inlet(x=1500)
i_kin:=Inlet(x=1500)
o_wf:=Outlet(x=1500)
o_wb:=Outlet(x=1500)
o_wf2:=Outlet(x=1500)
o_wb2:=Outlet(x=1500)

M1<-(i_anti,i_prob,i_kin)
M2<-(i_anti,i_prob,i_kin)
o_wf<-(M1,M2)
o_wb<-(M1,M2)
RC1<-(M1,M2)
RC2<-(M1,M2)
o_wf2<-(RC1,RC2)
o_wb2<-(RC1,RC2)
parallel(M1,M2)
parallel(RC1,RC2)
```

5.4 Code generation

Co's compiler will also generate the source code needed by Columba. The following code is the code generated by the compiler.

```
module:
M1 d r 1 1 0 7600 3800 50
M2 d r 1 1 0 7600 3800 50
RC1 d c 1 0 1 0 3000 1200 100
RC2 d c 1 0 1 0 3000 1200 100
i_anti p 1500 1500
i_prob p 1500 1500
i_kin p 1500 1500
o_wf p 1500 1500
o_wb p 1500 1500
```

```
11 o_wf2 p 1500 1500
12 o_wb2 p 1500 1500
13 fin
14
15 netlist:
16 i_anti M1 1
17 i_prob M1 1
18 i_kin M1 1
19 i_anti M2 1
20 i_prob M2 1
21 i_kin M2 1
22 M1 o_wf 1
23 M2 o_wf 1
24 M1 o_wb 1
25 M2 o_wb 1
26 M1 RC1 1
27 M2 RC1 1
28 M1 RC2 1
29 M2 RC2 1
30 RC1 o_wf2 1
31 RC2 o_wf2 1
32 RC1 o_wb2 1
33 RC2 o_wb2 1
34 fin
35
36 conflict:
37 fin
38
39 parallel:
40 2 M1 M2
41 2 RC1 RC2
42 fin
43
44 group:
45 fin
```

The generated code is passed to the server and the design is generated in the Server. The server returns the generated resulting design. The final design will be displayed on the Cloud Columba.

Appendix shows the design generated from the above code.

5.5 How to develop Co

As Columba continues to evolve, Co also needs to be constantly updated. Co can be extended according to the above compilation process.

5.5.1 New module type

Firstly, the corresponding type of token should be added.

Secondly, determine its parameters' type and its order.

Thirdly, its class should be implemented.

Fourthly, the detection function for syntactic analysis should be implemented.

Fifthly, the corresponding detection function for semantic analysis should be implemented.

5.5.2 New type of relationship between modules

Similar to the above process of extending the module type, the extension of the type of the relationship can also be achieved by five similar steps.

Firstly, the corresponding type of token should be added. For example, the name of the new defined relationship like Parallel should be added in the TOKEN set.

Secondly, determining the character of the relationship. It should define which modules can be included in this relationship set.

Thirdly, its class should be implemented.

Fourthly, the detection function for syntactic analysis should be implemented.

Fifthly, the corresponding detection function for semantic analysis should be implemented.

6 Conclusion

In this work, a programming language was developed to detect whether the user-defined design script is syntactically and logically correct, helping users to find out where corrections are needed, and reducing many unnecessary problems. Users can more easily generate the design of the microfluidic chip through the Cloud Columba platform. In the future, with the development of the Cloud Columba platform, this compiler can continue to expand, helping more biologists to use automated technology to more easily conduct biological experiments.

References

[1] Mengchu Li, Tsun-Ming Tseng, Bing Li, Tsung-Yi Ho, and Ulf Schlichtmann. Component-oriented high-level synthesis for continuous-flow microfluidics considering hybrid-scheduling. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2017.

[2] Tsun-Ming Tseng, Mengchu Li, Daniel Nestor Freitas, Amy Mongersun, Ismail Emre Araci, Tsung-Yi Ho, and Ulf Schlichtmann. Columba s: a scalable co-layout design automation tool for microfluidic large-scale integration. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2018.

[3] colum.ga.

[4] Stanford Foundry. Basic design rules, 2017.

[5] Tsun-Ming Tseng, Mengchu Li, Daniel Nestor Freitas, Travis McAuley, Bing Li, Tsung-Yi Ho, Ismail Emre Araci, and Ulf Schlichtmann. Columba 2.0: A co-layout synthesis tool for continuous-flow microfluidic biochips. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(8):1588–1601, 2018.

[6] Jessica Melin and Stephen R Quake. Microfluidic large-scale integration: the evolution of design rules for biological automation. *Annu. Rev. Biophys. Biomol. Struct.*, 36:213–231, 2007.

[7] Mengchu Li, Tsun-Ming Tseng, Bing Li, Tsung-Yi Ho, and Ulf Schlichtmann. Sieve-valve-aware synthesis of flow-based microfluidic biochips considering specific biological execution limitations. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 624–629. IEEE, 2016.

[8] Timothy A Budd, Timothy P Justice, and Rajeev K Pandey. General-purpose multi-paradigm programming languages: an enabling technology for constructing complex systems. In *Proceedings of First IEEE International Conference on Engineering of Complex Computer Systems. ICECCS'95*, pages 334–337. IEEE, 1995.

[9] Adel Smeda, Mourad Oussalah, and Tahar Khammaci. A multi-paradigm approach to describe software systems. *WSEAS Transactions on Computers*, 3(4):936–941, 2004.

[10] Timothy A Budd. An introduction to object-oriented programming 3rd ed. 2001.

[11] Alan C Kay. The early history of smalltalk. In *History of programming languages—II*, pages 511–598. ACM, 1996.

[12] Michael Feathers. *Working Effectively with Legacy Code: WORK EFFECT LEG CODE _p1*. Prentice Hall Professional, 2004.

[13] Michael Fogus. *Functional JavaScript: Introducing Functional Programming with Underscore. js*. " O'Reilly Media, Inc.", 2013.

[14] Michael Feathers. *Working Effectively with Legacy Code: WORK EFFECT LEG CODE _p1*. Prentice Hall Professional, 2004.

[15] Douglas Crockford. *JavaScript: The Good Parts: The Good Parts*. " O'Reilly Media, Inc.", 2008.

[16] Clive Maxfield. *FPGAs: world class designs*. Newnes, 2009.

[17] Andrew W Appel. *Modern compiler implementation in C*. Cambridge university press, 2004.

[18] KJlmfe. Kjcompiler. https://github.com/KJlmfe/KJCompiler, 2012.

A Design from Columba S