HOCL Programming Guide

Chen Wang, Thierry Priol

HOCL Programming Guide

Author
Chen Wang and Thierry Priol

Contact
chen.wang@inria.fr, thierry.priol@inria.fr

MYRIADS Project-team
INRIA Rennes, France

08/2009

Abstract

HOCL(Higher Order Chemical Language) is a chemical programming language. Computations can be seen as chemical reactions which are controlled by a set of chemical rules. An HOCL program is composed of two parts: *chemical rule definitions* and *element organization*. This article aims at introducing how to define chemical rules in HOCL programming. These rules together with the data for computation are called "*elements*". All the elements have to be organized in multi-sets. As for how to organize these elements, it is an advanced topic and it depends much on the personal experience. This article only addresses the part of "*chemical rule definitions*" but will not cover "*element organization*".

Chapter 1: We start with an brief introduction of chemical programming, a short history of its development is addressed; Readers have to pay attention to the definitions of all terminologies. Make clear of their meaning so that you will not be confused when meet them in the following sections. This part is ended with a simple example which gives the first picture of HOCL to readers.

Chapter 2: This part discusses how to define a chemical rule. All basic syntaxes are introduced here and after this part, readers are capable in writing a few lines of HOCL program to solve simple problems. An example is introduced and several exercises are left to readers.

Chapter 3: This section addresses how to define *one-shot rule* and implement "higher-order" property in HOCL programming. A small example will be illustrated in detail, and readers are asked to produce your own codes to solve several problems.

Chapter 4: We then talks about some advanced topics in this section, such as how to use Java class in HOCL programming. A "library application" project is set up step by step; readers are recommended to follow these steps and make this practice by hand.

Readers will find large amount of examples in this document; these examples cover from some "ABC" programs which are easy enough for any beginner to start, to some complex and well designed ones that covers all the characteristics of HOCL. There is a list concerning all the source codes of these examples in the next page. Readers can also find these codes in the **SVN repository** provided along with this paper.

After each chapter, we have some problems left to readers. These exercises help a lot in mastering HOCL programming. Readers could review what you have learned in each chapter and try to produce your own code. This practical experience will improve and enhance your comprehension and coding skills. Readers could also find all source codes for these exercises in the **SVN repository**.

If you still have some problems, or you have the errors to report, feel free to contact us by email. We are looking forward to your feedbacks.

Table of Contents

List of Programs

1 Introduction

1.1 Chemical computing

Chemical programming is an unconventional programming paradigm which is inspired by the chemical metaphor. Computation can be regarded as a chemical reactions and data can be seen as molecules which participate in these reactions. If a certain condition is held, the reaction will be triggered and continuing until it gets inert: no more data can satisfy any computing conditions. And finally, a solution is obtained.

To realize this program paradigm, a multi-set is implemented to act as a chemical tank, which contains necessary data and rules. When a condition is satisfied, the computation starts. This computation is performed by multi-set rewriting. This process vividly describes the Brownian motion; two molecules meet together, hold a certain chemical condition and then generate a new molecule(s).

1.2 History

- **Gamma**[1]

  Gamma was proposed 20 years ago as a paradigm for parallel computing through a chemical metaphor, computation in Gamma can be seen as chemical reaction;

  A Gamma program consists in 2 parts, *rule definition* and *data organization*. A rule is composed of *reaction condition* and *action*; if some elements satisfy a certain reaction condition of a chemical rule, they will be replaced by other elements specified by the relative "action" part of this rule. These newly produced elements may meet other reaction conditions and trigger other calculation; this process will continue until the reaction become inert. When such a stable state is reached, we get the result of our programs.

- **γ-calculus**[2]

  γ-calculus extends Gamma with higher order. It can be seen as a formal and minimal basis for the chemical paradigm. The γ-calculus is a quite expressive higher-order calculus. However, compared to the original Gamma and other chemical models, it lacks two fundamental features: reaction condition and atomic capture. As a result, it is then enriched with conditional reactions and the possibility of rewriting atomically several molecules.

  These two extensions are orthogonal and enhance greatly the expressivity of the chemical calculi. So from now, γ-abstractions (also called active molecules) can react according to a condition and can extract elements using pattern-matching. Furthermore, we consider the γ-calculus extended with booleans, integers, arithmetic and booleans operators, tuples (written x1: . . . :xn) and the possibility of naming molecules (ident = M).

[1] ref: JP Banâtre et al. *Gamma and the Chemical Reaction Model: Fifteen Years After*. C.S. Calude et al. (Eds.): Multiset Processing, *LNCS 2235*, pp. 17-44, 2001

[2] ref: Banâtre, J.-P., Fradet, P. and Radenac, Y. *Higher-order programming style*. In: Proc. of the workshop on Unconventional Programming Paradigms (UPP'04). Springer-Verlag Lecture Notes in Computer Science 3566.

- **HOCL**

  HOCL stands for *Higher-Order Chemical Language*. It implements γ-calculus and extends the previous presented models with expressions, types, empty solution and naming. These four points will be mentioned in the following chapters. Finally, multisets are further generalized by allowing elements to have infinite and negative multiplicities.

  An HOCL compiler has been developed in JAVA and this paper is to equip the readers with the ability of programming in HOCL masterly and effectively.

1.3 Glossary

Before starting, it is very important and necessary to introduce the glossary. It clarifies the definitions of all basic concepts which are the basis of further understanding. Normally, from different perspectives, different names are used to represent the same entity or process. All these terminologies that you are going to meet in the following sections are defined here; and readers should make clear of their meanings so that you will not be confused when you meet them in reading this document. Once you meet some words that are unclear to you, you could turn back to this section to check its definition.

- **Chemical Concepts**

  A **chemical reaction** is a process that leads to the transformation of a set of chemical substances to another. These substances are always presented in the form of **molecules**. A **chemical equation** can be see as a means of writing out or describing the chemical reaction. The space limitation of a reaction is always restricted by a **chemical tank**. It is a storage container for different chemicals where reactions take place. In chemical computing, this chemical tank is also called "**solution**". A solution can not only contain molecules for chemical reactions, but also, more important, block the entry of other molecules from the space outside of the solution unless the reaction inside reaches inert.

- **Multi-set**

  Multi-set is used to implement **solution**. It is the principal data structure in chemical computing, all the data for computing should be stocked in multi-sets. *Multi-set* extends the concept "*set*" with "multiplicity". An element can present only once in a *set* whereas multiple times in a *multi-set*. The time of its occurrence is defined as multiplicity. A short example, $\{1, 2, 1\}$ is a *multi-set* while not a *set* because the multiplicity of element "1" is 2. In HOCL, it is represented by a pair of brackets("<>").

- **Elements**

  A multi-set is used to contain **elements**. These elements are necessary for computing. There are two kinds of elements: **data** and **rules**. Data can be regarded as molecules in a chemical reaction, which take part in the computation; while rules play the same role as reaction equations, it defines multi-set rewriting to perform the computation.

- **Multi-set Rewriting**

The **chemical reaction** is implemented by **multi-set rewriting**. Rewrite a *multi-set* means to remove some elements and then produce new ones. in this way, the computation can be performed. Similar to chemical reaction, this process is also controlled by chemical rules.

Figure 1 summarize the basic terminologies in chemical computing, it compares the different definitions from two perspectives: chemical concept as well as its implementation.

Chemical Concepts	Chemical Computing	Implementation
Chemical tank	Solution	Multi-set
Molecule	Data	Element
Chemical equation	Chemical rule	Element
Chemical reaction	Computation	Multi-set rewriting

Figure 1: Terminologies

- **Container, Sub-solution, Solution structure**

  A multi-set can contain not only the computing data but also other multi-sets that we call "**sub-solutions**". In particular, the out-most solution is called "**container**". A *container* is never included in other *solutions*. It is also called **tier-1 solution**. The *sub-solutions* of *tier-1 solution* are called **tier-2 solutions**. On the analogy of this, the *sub-solutions* of **tier-n solutions** are named **tier-(n+1) solutions**.

  Solution structure means the composition of a container solution, including all its sub-solutions systems and elements. A *solution structure* is called **N-tiers solution** if this container has *tier-n solution* but no *tier-(n+1) solution*.

1.4 First example

You are going to meet your first example of chemical program, to compute the maximum value of a set of numbers. Using conventional programming languages, there are a couple of algorithms. For example, compare each number with a "max" number which should be initialized as one of the numbers included in the set. If this number is larger than this "max" number, its value will be assigned to "max" number. After the comparison with all numbers, the value of "max" number is our expecting result.

But in HOCL, computation is achieved by multi-set rewriting. As a result, we could compare each pair of numbers and remove the smaller one. In this way, the largest number will be finally left in the multi-set. The HOCL source code is shown in *Program 1*.

- **HOCL Program Structure**

  From this example, we can see that an HOCL program composes of two parts: rule definition (*line 1-5*) and element organization (*Line 6-8*). All the rules have to be defined before you call them in the final solution. If you have more than one rule, you have to make them defined after *line 5*, using the same format: "**let**...**replace**...**by**...**if**...**in**". We will introduce how to define a rule in the next section.

Program 1 Get maximum number: getMaxNumber.hocl

```
1  let getMaxNumbers=
2      replace x::int, y::int
3      by x
4      if x>y
5  in
6  <
7      getMaxNumbers, 16, 5, 9, -6, 0, 3, 5, 0, 5, 9, -3, 1
8  >
```

All the elements have to be put between "<" and ">" characters. It acts as a chemical tank, containing data and chemical rules. A chemical tank can also contain other tanks so that there can be more than one chemical reaction triggered simultaneously and separately.

- **Execution Sequence**

  Figure 1 gives out the first execution step of this program. In the first time, the chemical rule *getMaxNumbers* can be applied. The compiler chooses 2 numbers in the multi-set, and assign their values to two variables x and y. In this example, 16 is assigned to x while 5 is assigned to y. Since the condition is satisfied ($x>y$), then multi-set rewriting can be performed, 5 will be removed from the multi-set.

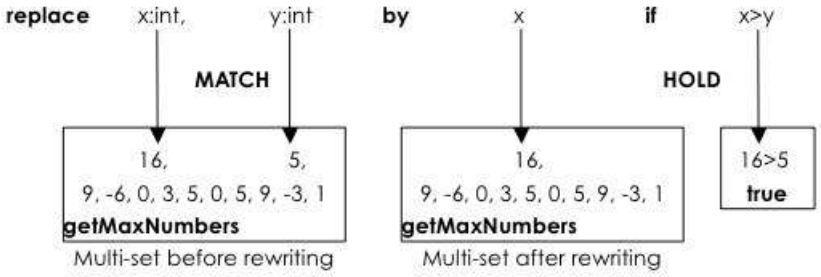

Figure 1: Max Numbers: the first step execution

And then, there are still a lot of numbers left in the multi-set which can trigger this chemical reaction. For example, this time the compiler can take 9 and -6 for the comparison. As shown in *Figure 2*, after the comparison, -6 will be removed from the multi-set.

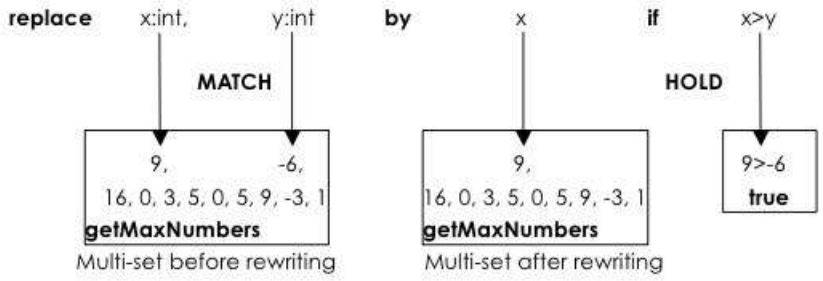

Figure 2: Max Numbers: the second step execution

This process will hold on until there are only two numbers left in the multi-set. At this time, the last round of reaction will be launched. *Figure 3* presents this reaction. Before the reaction, 16 and 1 are left in the multi-set and finally 1 is removed and the largest number 16 is the only one that is left in the multi-set. At this moment, there is only one number, the reaction cannot be triggered anymore. When this solution becomes inert, we obtain the final result.

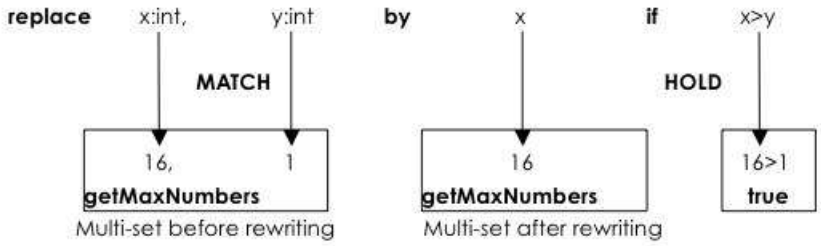

Figure 3: Max Numbers: the third step execution

In following parts, we aim at illustrating how to program in HOCL. *Chapter 2* introduces basic HOCL syntaxes to define chemical rules while *chapter 3* addresses some higher level definition of a chemical rule. *Chapter 4* talks about some advanced topics such as how to use JAVA classes in HOCL.

2 Basic Definition

As we have introduced, an HOCL program contains two parts: *rule definition* and *data organization*. In this section, we are going to introduce basic HOCL syntax to define chemical rules.

In HOCL, rules are defined by "**let...replace...by...if...**" expression. Before passing into details of its syntax, we are firstly going to look at an example.

Considering the following context, a college has founded a scholarship to encourage the best student who gains the highest score; but there is an additional requirement that his average score has to be greater than 18 (18/20). If there is no such student, this scholarship will not be allocated. *Program 2* is created to find the eligible candidate.

Program 2 Search eligible scholar: selectScholar.hocl

```
1   let selectCandidate =
2       replace stu::String:score::double, ?w
3       by w
4       if score < 18.00
5   in
6   let searchScholar=
7       replace stu1::String:score1::double, stu2::String:score2::double
8       by stu1:score1
9       if score1 >= score2
10  in
11  <
12      <
13          selectCandidate, searchScholar,
14          "Thierry":17.96,
15          "Nicola":16.98,
16          "Ameli":17.26,
17          "Christina":12.89
18      >,
19      replace-one <selectCandidate=x, searchScholar=y, ?w> by w
20  >
```

This short program is not difficult to understand. It defines chemical rules from *line 1* to *line 10* and organizes the elements (data+rules) from *line 11* to *line 20*. *selectCandidate* selects all the qualified candidates, whose average scores are greater than 18; *searchScholar* gets the student to allocate this scholarship, whose score is the highest among all qualified candidates.

In this example, elements are organized in *2 tiers*. Data and both above defined rules are put into a *tier-2 solution* (*Line 12-18*) to perform the computation by rewriting multi-set. Once this *sub-solution* become inert, a *one-shot* rule in *tier-1 solution* (*Line 19*) removes both former chemical rules (*selectCandidate*, *searchScholar*) and extracts results.

From this context, it is clear to know that no student can meet all the requirements for obtaining this scholarship. Executing this HOCL program, the multi-set rewriting will be performed firstly

within *tier-2 solution* (*line 12-18*); after this sub-solution become inert, we will get the following intermediate result shown in *Figure 2*:

```
<
   <
      selectCandidate, searchScholar,
   >,
   replace-one <selectCandidate=x, searchScholar=y, ?w> by w
>
```

Figure 2: Intermediate Result after Step 1 of *program 2*

From this intermediate result, we can see that all student *pairs* have been removed since their scores are less than 18.00. The *one-shot* rule[3] is then applied to remove both chemical rules, and finally we get an *empty solution*:

"<>"

In the next section, we will focus on how to define chemical rules by explaining this program.

2.1 Rule definition

In HOCL, "**let...replace...by...if...**" expression is used to define a rule. This expression has 4 keywords. "**let**" gives a rule with a certain name. This keyword is followed by a name and then an equation operator ("="). "**replace...by...**" defines the rewriting process of multi-set, which performs the calculation. "**if**" denotes the condition of the reaction, the rewriting of multi-set is performed only when this condition is satisfied. This keyword is not necessary to define a rule.

2.1.1 Naming

In HOCL, a rule can be named (or tagged) to facilitate its reuse. A developper can give a rule with a certain name, and later in his program he can reuse this rule simply by calling its name. The "**let**" key word is used to assign a name to a rule in the following way:

let *putNameHere* = ...

A rule name can be any legal identifier: an unlimited-length sequence of *Unicode* letters, digits and "_" characters, beginning with a letter or "_" character. A rule defines the action of multi-set rewriting so that it is strongly recommended to start its name with a verb. The name of a rule should be clear to reflect its responsibility. If the name consists of only one word, spell that word

[3]Why a one-shot rule is used here: As you can see, all student *pairs* are removed after the first computation, so there are only two rules left in the final solution. This *one-shot rule* is used to remove both rules and therefore we can get an *empty solution* at the end so that we can introduce this important extension in *section 2.1.5: Empty Solution*

in all lowercase letters. If it consists of more than one word, capitalize the first letter of each subsequent word, for example, *"selectCandidate"*. At last, pay attention that the rule name is case-sensitive.

In *Program 2*, two rules have been defined, *selectCandidate* from *line 1* to *line 5* and *searchScholar* from *line 6* to *line 10*. Both are named by using "**let**" keyword (*cf. line 1 and line 5*).

2.1.2 Variable types

In HOCL, every variable has be defined with a certain type. The type of a variable should be either a basic type such as *Int* or *String*, or some user defined types (*cf. chapter 4.1: Hybrid Model:Using Java class in HOCL*). Every variable is defined within "**replace**" expression in the following form:

$$VariableName :: Type$$

Two colons are used together ("::") to indicate the type of a variable. Take *Program 2* for example, in the definition of *selectCandidate*, a *pair* (*cf. next point*) has been defined in *line 2* ("*stu*::**String**:*score*::**double**"). This *pair* comprises two variables: *stu* which is a **String** type and *score* which is a **float** number.

After the definition of variables in **"replace"** expression, we could use them in the following parts such as **"by"** and **"if"** expressions. The scope of a variable lasts within the definition of a rule. A rule can never use any variables defined in other rules.

2.1.3 Pairs

A pair is denoted by "A_1:A_2", a colon is used to connect two isolated molecules. This is easy to understand using the chemical metaphor. Considering the following chemical equation, two SO_2 molecules meet together and then a S_2O_4 compound molecule is generated. This compound molecule can be seen as a simple combination of both SO_2 molecules. As a result, a pair can be also regarded as a molecule which can participate in the chemical reactions.

$$SO_2 + SO_2 \Longleftrightarrow S_2O_4\ (SO_2{:}SO_2)$$

In *program 2*, a pair is used to store a student's name and his score. As a result, it contains two fields: one indicates the name of a student while the other stores his everage score (**ex.:** *stu1:score1* at *Line 8*).

Much in general, the notion of *pair* is extended by using several colons to connect multiple fields. As a result, a pair could be defined in the following form:

$$field_1{:}field_2{:}...{:}field_n$$

Each field is a variable and has to be defined with a certain type. A *pair* can be regarded as a "struct" in C language, which, like an union, groups multiple variables into a single record.

2.1.4 Universal pattern

In HOCL, there is a universal pattern which can match all kinds of molecules. It is defined by a question mark("?"):

"?*name*"

A "?*name*" can match no matter what type of variable, such as an *integer* variable, a *String* variable, a *pair* or even an "empty molecule". Furthermore, it can match not only one molecule but many of them. This pattern means: "The remaining".

In *Program 2*, we have such a variable defined in *line 2* ("?w"). When the rule *selectCandidate* is applied for the first time, there are four *pairs* in the solution, **stu:score** can match any one *pair* of them and **?w** matches the other three. This process is shown in *Figure 4*. In this case, "The remaining" means "The remaining three pairs".

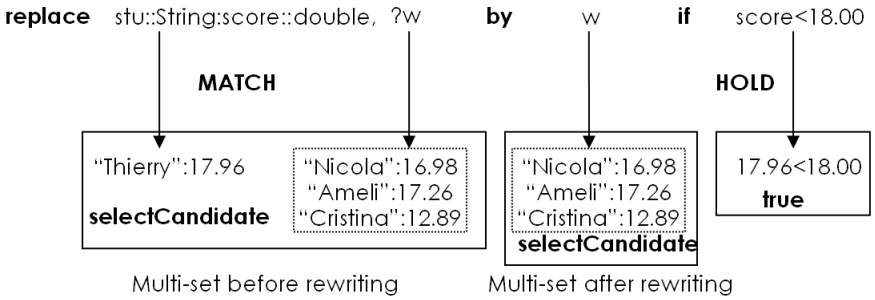

Figure 4: Universal pattern: match multiple molecules

This process will continue until there is only one *pair* left in the multi-set, we assume that "Ameli":17.26 is that pair (because the execution of a chemical program is highly non-deterministic, different executions may have different execution orders. As a result, another *pair* can be left for another execution). Then *selectCandidate* chemical rule is applied. **Stu:Score** *pair* will match "Ameli":17.26 pair while **?w** will match an "empty molecule". This process is shown in *Figure 5*. In this case, "The remaining" means "nothing", there is noting left in the solution.

This universal pattern is meaningful since the multi-set contains a lot of molecules, but rewriting is usually performed just within a certain part of them. Since not all elements need to participate in multi-set rewriting, a universal pattern (**?name**) can be used to match all those elements that will leave unchanged. This pattern enables the developper focus on the sub-set of elements that need to be rewritten.

2.1.5 Empty solution

The notion of empty solution in HOCL is raised since that after reaction, the multi-set might become empty. This is caused by the universe pattern ?*w* which can match any molecules even the

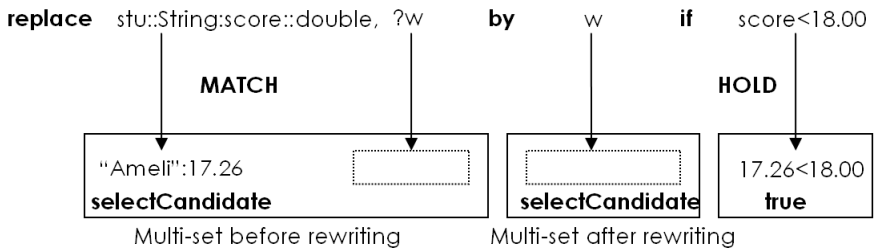

Figure 5: Universal pattern: match "empty molecule"

"empty ones". Let us look at *Program 2*, chemical rule *selectCandidate* will remove a student *pair* if a student's everage score is lower than 18.00. In this case, no student could meet this requirement. As a result, All pairs will be removed in the end. And then a *one-shot* rule will remove all chemical rules (*selectCandidate* and *searchScholar*), consequently an empty solution is finally obtained.

2.2 Practice: Write your own example

After introducing some basic syntax of HOCL, we are now going to set up the first program with readers step by step. Through this practice, readers can be more familiar in writing an HOCL program.

Think about this problem, program in HOCL to calculate all prime numbers that are smaller then 10.

The first thing first is to organize data. In this simple example, it is easy to think that all integer numbers from 2 to 10 can be put in a single multi-set. As a result, we have the following segment of source code shown in *Figure 3*:

```
1    <
2        2, 3, 4, 5, 6, 7, 8, 9, 10
3    >
```

Figure 3: Code segment: Data Organization

And then, an algorithm for the calculation has to be designed. From the multi-set rewriting point of view, all the non-prime numbers have to be removed. If a number is divisible by another number in the multi-set, this number must be removed. To achieve this, the chemical rule *"getPrimeNumbers"* is created. Based on this algorithm, we compare every two numbers, if one is a multiple of the other, we will then remove this greater number. For example, if we have 9 and 3, since 9 is a multiple of 3, 9 will be removed. This strategy can be implemented with HOCL using the following rule displayed in *Figure 4*:

And now, we have rule defined and data organized, place this rule into the *multi-set* shown in *Figure 3*, and we can have the following complete source code shown in *Program 3*:

```
1   let getPrimeNumbers=
2       replace x::int, y::int
3       by x
4       if x%y==0
```

Figure 4: Code segment: Definition of *getPrimeNumbers*

Program 3 Compute prime numbers

```
1   let getPrimenumbers=
2       replace x::int, y::int
3       by x
4       if x%y==0
5   in
6   <
7       getPrimenumbers, 2, 3, 4, 5, 6, 7, 8, 9, 10
8   >
```

Running this program, we will finally get the following results:

$$<getPrimenumbers, 2, 3, 5, 7>$$

2.3 Exercises

Up to now, we believe that readers can write some HOCL programs to solve simple problems. Here is a list of exercises, if you are motivated to make some practices, you can try them and check the results. The source codes and running results can be founded in **the SVN repository**.

- **Compute the sum of a set of numbers**

  To get start, look at this quite simple example. Given a set of numbers, compute the sum of all elements. The numbers are given below:

  $$\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$$

- **Convert a multi-set to a set**

  Try to convert the following multi-set to a set:

  $$\{3,58,4,1,3,8,4,3,1,41,31,5,135,1,5,1,3,15,1,9,1,3,54,684,5,16,84,3,1\}$$

  Tip: the difference between the two is that multi-set can have multiple instances of the same element.

- **Find major element**

In some applications, such as a "vote system", it is wanted to find a candidate whose approval rating is greater than 50%. In such case, given a multi-set containing all ballot tickets, write a small program to search the eligible candidate (whose approval rating is greater than 50%). This multi-set is given bellow:

{"Barack","Barack","Barack","Jack","Bill","Bill","Jack","Barack","Jack", "Barack","Barack","Bill","Barack","

- **Sort by swapping**

  10 numbers are stored in a multi-set, each number is represented by a pair:

  index::**int**:*number*::**int**

  This pair has two fields, the first indicates the number's index while the second indicates its value. In the very beginning, all numbers are mis-indexed. The original data is given below:

  {1:5, 2:2, 3:8, 4:6, 5:7, 6:9, 7:3, 8:1, 9:10, 10:4}

  Write an HOCL program to sort these numbers ensuring that the smaller number finally possesses a smaller index. The final result is quite like the following form:

  {1:1, 2:2, 3:3, 4:4, 5:5, 6:6, 7:7, 8:8, 9:9, 10:10}

- **Shortest path**

  We want to compute the shortest path between every two points in an oriented map. A pair is employed to record the distance between two points:

  startPoint::**String**:*endPoint*::**String**:*distance*::**int**

  This pair is composed of three fields, the first two are *Strings* standing for points, the last field is an integer value meaning the distance between these two points. If it is unreachable from the start point to the end point, we assign it with the maximum integer value ("Integer.MAX_VALUE"). In the beginning, every point is reachable to its one-step neighbors. The initial state of this oriented graph is given below:

  "A":"B":Integer.MAX_VALUE, "A":"C":1,
  "A":"D":Integer.MAX_VALUE, "A":"E":Integer.MAX_VALUE,
  "A":"F":Integer.MAX_VALUE, "B":"A":3,
  "B":"C":5, "B":"D":Integer.MAX_VALUE,
  "B":"E":Integer.MAX_VALUE, "B":"F":Integer.MAX_VALUE,
  "C":"A":Integer.MAX_VALUE, "C":"B":Integer.MAX_VALUE,
  "C":"D":2, "C":"E":4,
  "C":"F":Integer.MAX_VALUE, "D":"A":5,
  "D":"B":Integer.MAX_VALUE, "D":"C":Integer.MAX_VALUE,
  "D":"E":Integer.MAX_VALUE, "D":"F":5,
  "E":"A":Integer.MAX_VALUE, "E":"B":Integer.MAX_VALUE,
  "E":"C":Integer.MAX_VALUE, "E":"D":Integer.MAX_VALUE,
  "E":"F":1, "F":"A":Integer.MAX_VALUE,
  "F":"B":Integer.MAX_VALUE, "F":"C":Integer.MAX_VALUE,
  "F":"D":Integer.MAX_VALUE, "F":"E":Integer.MAX_VALUE

3 Advanced Definition

In this section, we are going to talk about some advanced definitions of chemical rules. If you have no problem in solving all the exercises in previous section, we believe that you could write some basic and simple programs using HOCL. These programs always contain less rules and simple data organization. The most important, these procedures comprise only one step of calculation or several parallel steps.

But most of computation in reality is composed of multiple steps. Readers may find that it is easy to perform parallel computing in HOCL: just put every rule and data in a multi-set, and then the execution will be launched automatically. This process is highly dynamic and non-deterministic. It is hard to bring some controls on the execution, such as sequential control. To solve these problems, we are going to introduce some advanced definitions of chemical rules ensuring that readers could produce a more powerful and flexible program.

3.1 Advanced Definition

3.1.1 *one-shot* rule Versus *n-shot* rule

HOCL has implemented two kinds of rules, *one-shot* rule and *n-shot* rule. From the name, we can easily tell the difference between the two. *One-shot* rule can be applied only once, after its consumption, the rule will not exist anymore. In contrast, *n-shot rule* can be used unlimited times, once there are elements satisfying its condition, the rule will be applied and the computation will be launched. *N-shot* rule will not be consumed after the computation. Almost all the rules that we have introduced in the previous sections are *n-shot rules* (except that *one-shot* rule in *line 19* of *Program 2*, we will talk about that later).

In this section, we are going to introduce how to define *one-shot* rules and in which case we prefer to use this kind of rules. First of all, as in the previous sections, we are firstly going to have a look at an example shown in *Program 4*.

Program 4 Compute prime numbers using *one-shot* rule

```
1  let getPrimeNumbers=
2      replace-one x::int, y::int
3      by x
4      if x%y==0
5  in
6  <
7      getPrimenumbers, 2, 3, 4, 5, 6, 7, 8, 9, 10
8  >
```

In the example, an *one-shot* rule *getPrimeNumbers* has been defined (*Line 1*). The definition of a *one-shot* rule is almost the same as *n-shot* rule, the only difference is to change the **replace** keyword by **replace-one** (*Line 2*). Readers can compare this program with the one shown in *Program 3*.

Running this program, we can get the following result (because its execution is highly non-deterministic, so another execution may get another result. But there is one and only one number removed):

<2, 3, 4, 5, 6, 7, 8, 10>

Compare it with the result of *Program 3*, we can see that after the first comparison, 9 was removed because it is a multiple of 3. And then, this *one-shot* rule *getPrimenumbers* was consumed and the computation became "inert". That is why we can not work out final result. In addition, notice that there is no chemical rule in the final *solution*.

From this example, we can conclude that *one-shot* rule is specially suitable for the tasks that should be performed once and only once. For instance, removing/changing rules (*cf. Chapter 3.1.3: Higher order property*), extracting results (*cf. Program 5*) or performing some one-shot calculation. For most of functional rules (rules for computation such as *getPrimeNumbers*), *n-shot* rule is the best choice.

In fact, *one-shot* rule has a limited life. In this sense, we need not give it a name (remembering that *Naming* is implemented for facilitating the reuse of a rule, *cf. Section 2.1.1: Naming*). Therefore, we can define a *one-shot* rule when we need it, have a look at the following example.

This time we want to compute the maximum numbers contained in a multi-set. But we want to extract the results into a "result" *sub-solution*. In this way, the final result is collected and can be easily passed to other steps for futher computation. *Program 5* shows you the HOCL source code..

Program 5 Max numbers: using one-shot rule to extract final result

```
1   let getMaxNumbers=
2       replace x::int, y::int
3       by x
4       if x>y
5   in
6   <
7       replace-one <i::int, ?w> by "result":<i>
8       <
9           getMaxNumbers, 16, 5, 9, -6, 0, 3, 5, 0, 5, 9, -3, 1
10      >
11  >
```

In this program, readers can easily find that a chemical rule *getMaxNumbers* is defined(*Line 1-4*) to compute the maximum number in a set of integer numbers. And data is organized in two tiers. The reaction in the *inner solution* (*Line 8-10*) will be triggered automatically and the computation will hold on until this *sub-solution* become inert (get the maximum number). At that moment, the state of multi-set is shown in *Figure 5*:

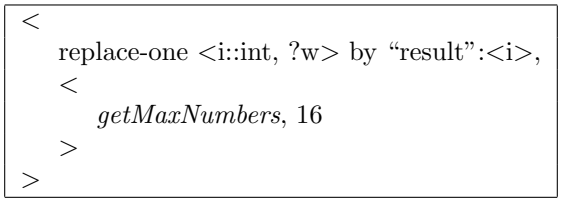

Figure 5: Intermediate Result after Step 1 of *Program 5*

And then, the *one-shot* rule in the *container* will collect the result. This rule is defined in the *container* and can be applied after the inner solution become inert. So, this rule will not disturb the calculation of maximum number carried in the inner solution. Finally, we have the following result shown in Figure 6:

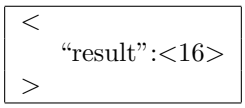

Figure 6: Final Result of *Program 5*

One-shot rule is useful and powerful, it enables your program more concise and professional looked. Readers can find more usage of *one-shot* rule in the following sections.

3.1.2 Catch a rule

During the definition of a rule, some patterns are defined to catch certain kinds of data in order to rewrite the multi-set (**ex. x::int** is defined to catch an integer number while **?w** catches all the “*remaining molecules*”). And from the former HOCL programs, we can see that in a multi-set, there are just two kinds of elements: data and rules. Since it is possible to catch an element and then change it, we also prefer to define a pattern catching rules so that we can delete them or change them with other rules.

In fact, readers have already seen this pattern in the very beginning of this chapter. In *Program 2*, we have an *one-shot* rule defined, this rule could catch chemical rules *selectCandidate* and *searchScholar*, and then delete them. To catch a certain rule, it is possible to use the following expression:

putRuleNameHere=x

In this way, a rule could be caught and further actions could be performed. it is possible to delete this rule, or change it with another rule. This is very practical and useful since it implements the *higher order* property of HOCL. We will introduce this property in the next section.

3.1.3 Higher order property

The capability of catching a chemical rule enables a developper to delete a rule or change a rule with another one. This point ensures the higher order property of HOCL.

Consider the following context: given a set of numbers (**ex.:**1, 6, 9, 11, 15, 18, 16, 24), we want to compute the included maximum even number which are between 10 and 20 (in this case, it is 18). This calculation could be divided into three steps: first of all, it selects the numbers between 10 and 20 (**ex.:** 11, 15, 18, 16); and then, it should calculate the even numbers included (**ex.:** 18, 16); and finally, a rule is created to get the maximun number in the results of step 2 (*ex.:* 18).

Accordingly, three rules have to be created in order to perform these three steps of rewriting. The definition of these rules is shown in *Program 7*:

```
1   let selectNumbers=
2       replace x::int, ?y
3       by y
4       if x<10 || x>20
5   in
6   let getEvenNumbers=
7       replace x::int,?y
8       by y
9       if x%2!=0
10  in
11  let getMaxNumber=
12      replace x::int,y::int
13      by x
14      if x>y
15  in
```

Figure 7: Code segment: Rule Definitions

Readers now can easily read these rule definitions. *selectNumbers* gives out all numbers from 10 to 20 included in the multi-set; *getEvenNumbers* removes all odd numbers and *getMaxNumber* calculates the maximum number. However, these three steps have to be performed sequentially, three rules can not be placed in a single *solution* (If we put them in a sole solution, they will be triggered and the calculation for these three steps will be performed simultaneously. They will disturb each other and it will lead the final result inaccurate. For example, *getMaxNumber* rule might compare 18 and 24 and finally 18 will be removed.).

Notice that chemical computing has *higher order* property that only the inner solution has been inert, its elements can participate in the reaction of outer solutions. This property give us a way to perform sequential executing. The earlier executed steps are placed in the inner solutions while the later executed steps are placed in the outer solutions. Therefore, *selectNumbers* should be placed in the inner solution. Then, it is necessary to create rules in outer solution to extract results and change the old rule with the new ones for next steps. For this one-shot works, we always use *one-shot* rules. And then, all computing data has to be placed in the inner-most *solution* in order to launch the computation from the first step. Finally, the element organization is shown in *Figure*

8. And *Program 6* gives out the complete HOCL source code.

```
<
   replace <getEvenNumbers=x,?w> by getMaxNumber,w,
   <
      replace <selectNumbers=x,?w> by getEvenNumbers,w,
      <
         selectNumbers,1,6,9,11,15,18,16,24
      >
   >
>
```

Figure 8: Data Organization

Executing this program, the computation is launched from the *tier-3 solution* (*line 20-22*) to compute numbers between 10 and 20. After this first step, the intermediate result is presented in *Figure 9*:

```
<
   replace <getEvenNumbers=x,?w> by getMaxNumber,w,
   <
      replace <selectNumbers=x,?w> by getEvenNumbers,w,
      <
         selectNumbers,11,15,18,16
      >
   >
>
```

Figure 9: Intermediate Result after Step 1 of *Program 6*

And then, the *one-shot* rule in *tier-2 solution* (line 19) extracts the results from *tier-3 solution* and change rule *selectNumbers* with *getEvenNumbers*. In this way, the second round of computation is triggered and the follwing result shown in *Figure 10* is obtained:

```
<
   replace <getEvenNumbers=x,?w> by getMaxNumber,w,
   <
      getEvenNumbers,18,16
   >
>
```

Figure 10: Intermediate Result after Step 2 of *Program 6*

Finally, the *one-shot* rule in *tier-1 solution* (*line 17*) extracts the results from *tier-2 solution* and change rule *getEvenNumbers* with *getMaxNumber*. At the end, we get the following result:

Program 6 Higher Order Property

```
1   let selectNumbers=
2       replace x::int, ?y
3       by y
4       if x<10 || x>20
5   in
6   let getEvenNumbers=
7       replace x::int,?y
8       by y
9       if x%2!=0
10  in
11  let getMaxNumber=
12      replace x::int,y::int
13      by x
14      if x>y
15  in
16  <
17      replace <getEvenNumbers=x,?w> by getMaxNumber,w,
18      <
19          replace <selectNumbers=x,?w> by getEvenNumbers,w,
20          <
21              selectNumbers,1,6,9,11,15,18,16,24
22          >
23      >
24  >
```

<*getMaxNumber*, 18>

3.2 Practice: Write your own example

At present, we suggest that you could write some more powerful programs to solve relatively complex problems. Besides defining simple rules, readers are now familiar with *one-shot* rules to perform sequential computation. Considering the follwing context. We want to write a small program for a library. A library possesses some books, and the registered students can borrow books from the library. The library keeps the borrowing record. We want to develop this "library administration" application.

3.2.1 Element organization

To write an HOCL program, the first step is to clarify the type of elements and how they can be organized. In this case, there are three entities: book, student and borrowing record. As a result, three *tier-2* sub-solutions have to be created, the "BOOKS" *tier-2 sub-solution* is created to store all the books, the "STUDENTS" *tier-2 sub-solution* is created to store all the students, and the

```
1  <
2      “BOOKS”:<>,
3      “STUDENTS”:<>,
4      “RECORD”:<>
5  >
```

Figure 11: Code segment: Data Organization

“RECORD” *tier-2 sub-solution* is designed for storing borrowing records. We have the following segment of code shown in *Figure 11*:

3.2.2 Basic rule definition

In this context, there are three actions. Firstly, both “BOOKS” and “STUDENTS” sub-solutions need to be initialized; and then, students can borrow books from the library. Accordingly, in this section, we create the *addBook* and the *addStu* rules for initializing the information of books and students. Furthermore, the *borrowBook* is created to perform the borrowing books.

“BOOKS” *sub-solution* is to contain books. We use the following *pair* to represent a book:

“BOOK”:*book_id*:*book_name*

This *pair* is composed of three fields. “BOOK” is an identifier indicating this is a *pair* for storing information of a book. It is followed by the *id* of a book, which is the identifier of a book. Two books can have the same name but different *id*s. The last field is the book’s name. Students search and borrow books by the name.

In the first time, there are a lot of strings in the “BOOKS” sub-solution (such as “Thinking in Java”). These strings are the names of books used for constructing a “BOOK” *pair* (in the form of “BOOK”:1:“Thinking in Java”). For each *pair*, the first field (“BOOK”) is the same and the last field comes from these strings. Now it is necessary to find a method for assigning the identical *id* number to the second field.

To achieve this, the following *pair* should be created in “BOOKS” *sub-solution*:

“*next_id*”:**int**

Whenever we construct a book *pair*, its second field has to be increased by one. This pair is initialized as “*next_id*”:1.

And now, the rule *“addBook*” can be defined. For each string in the multi-set, we combine it with a “BOOK” string as a head, and the integer number indicated by the *next_id pair*. As a result, the definition of this rule is given out in *Figure 12*:

Similarly, we have the same definition for the “*addStu*” rule. To represent a student, the following *pair* is used:

```
let addBook =
    replace bookname::String, "next_id":i::int
    by "BOOK":i:bookname, "next_id":i+1
```

Figure 12: *L*ibrary project: Definition of addBook chemical rule

"STUDENT":*stu_id*:*stu_name*

It also contains three fields, and each of them has the same meaning as a book pair. At the very beginning, there are some strings in the multi-set and these strings are used to construct student pair. Finally, we have *addStu* rule defined in *Figure 13*:

```
let addStu =
    replace stuname::String, "next_id":i::int
    by "STUDENT":i:stuname, "next_id":i+1
```

Figure 13: Library project: Definition of addStu chemical rule

These two rules are used for initializing the "BOOKS" and "STUDENTS" *tier-2 solutions*. And then, students can borrow books from the library. When there is a "Borrow" *pair* in the final solution, it means a borrow activity has to be performed. A "Borrow" pair is in the following form:

"Borrow":*stu_name*:*book_name*[4]

This pair means the student "*stu_name*" wants to borrow the book "*book_name*". A borrow action is performed by moving this "Borrow" pair into "RECORD" *sub-solution* and deleting the relative book pair in "BOOKS" *sub-solution*, because if a student has borrowed a book, this book is not visible to other students from the bookshelves. We represent this process in HOCL with the rule shown in *Figure 14*:

```
let bowrrowBook =
    replace "BOOKS":<"BOOK":book_id:book_name, ?w>,
            "Borrow":stu_name::String:book_name::String, "RECORD":<?x>
    by "BOOKS":<w>, "RECORD"<x,"Borrow":stu_name:book_name>
```

Figure 14: Library project: Definition of bowrrowBook chemical rule

3.2.3 Advanced rule definition

Three rules are defined to perform three actions, since borrowing books has to be performed after the initialization of "BOOKS" and "STUDENTS" *sub-solutions*, we could use a *one-shot* rule to

[4]Here we are assuming that any two books have the different names so that we can use *book_name* as a key (instead of *book_id*). But actually, a library holds multiple copies of a certain book so that the book name cannot be used as an identifier of a book; But the student usually search and borrow a book by its title (name).We want to make our example simple so that here in "Borrow" *pair*, we still use *book_name* field.

perform sequential execution. This *one-shot* rule removes both *addBook* and *addStu* rules and then brings in the new rule *borrowBook* for the second round computation. Its definition is shown in *Figure 15*:

replace-one "BOOKS":<*addBook*=x,?w>,"STUDENTS":<*addStu*=y,?k>
by *borrowBook*, 'BOOKS":<w>,"STUDENTS":<k>

Figure 15: Definition of One-shot Rule

Put this rule in the final solution. And then, we put some strings as the name of students and books for initializing the "BOOKS" and "STUDENTS" sub-solutions. Finally, we have the complete program shown in *Program 7*.

3.2.4 Running results

Running this program, the initializations of "BOOKS" and "STUDENTS" *sub-solutions* are performed at the very beginning. "*addBook*" and "*addStu*" generate book pairs and student pairs until there is no string in both *sub-solutions*. After this step, the state of the solution is shown in *Figure 16*:

```
<
  replace-one "BOOKS":<addBook=x,?w>,"STUDENTS":<addStu=y,?k>
       by borrowBook, 'BOOKS":<w>,"STUDENTS":<k>,
  "BOOKS":<
    addBook,
    "next_id":4,
    "BOOK":1:"Thinking in JAVA",
    "BOOK":2:"Code Complete",
    "BOOK":3:"Code Complete"
  >,
  "STUDENTS":<
    addStu,
    "next_id":4,
    "STUDENT":1:"Thierry",
    "STUDENT":2:"Nicola",
    "STUDENT":3:"Valentine"
  >,
  "RECORD":<>,
  "Borrow":"Thierry":"Thinking in JAVA",
  "Borrow":"Nicola":"Code Complete",
  "Borrow":"Thierry":"Code Complete"
>
```

Figure 16: Intermediate Result after Step 1 of *Program 7*

Program 7 Library application

```
1   let addBook =
2       replace bookname::String, “next_id”:i::int
3       by “BOOK”:i:bookname, “next_id”:i+1
4   in
5   let addStu =
6       replace stuname::String, “next_id”:i::int
7       by “STUDENT”:i:stuname, “next_id”:i+1
8   in
9   let borrowBook =
10      replace “BOOKS”:<“BOOK”:book_id::int:book_name::String, ?w>,
11              “Borrow”:stu_name::String:book_name::String, “RECORD”:<?x>
12      by “BOOKS”:<w>, “RECORD”:<x,“Borrow”:stu_name:book_name>
13  in
14  <
15      replace-one “BOOKS”:<addBook=x,?w>,“STUDENTS”:<addStu=y,?k>
16              by borrowBook, ‘BOOKS”:<w>,“STUDENTS”:<k>,
17      “BOOKS”:<
18          addBook,
19          “next_id”:1,
20          “Thinking in JAVA”,
21          “Code Complete”,
22          “Code Complete”
23      >,
24      “STUDENTS”:<
25          addStu,
26          “next_id”:1,
27          “Thierry”,
28          “Nicola”,
29          “Valentine”
30      >,
31      “RECORD”:<>,
32      “Borrow”:“Thierry”:“Thinking in JAVA”,
33      “Borrow”:“Nicola”:“Code Complete”,
34      “Borrow”:“Thierry”:“Code Complete”
35  >
```

And then, both tier-2 solutions become inert. The *one-shot* rule defined in the container could be applied. It will remove *addBook* and *addStu* rules in both tier-2 solutions and put *borrowBook* rule in the container. After its consumption, the solution state is given out by *Figure 17*:

```
<
    "BOOKS":<
        "next_id":4,
        "BOOK":1:"Thinking in JAVA",
        "BOOK":2:"Code Complete",
        "BOOK":3:"Code Complete"
    >,
    "STUDENTS":<
        "next_id":4,
        "STUDENT":1:"Thierry",
        "STUDENT":2:"Nicola",
        "STUDENT":3:"Valentine"
    >,
    "RECORD":<>,
    borrowBook,
    "Borrow":"Thierry":"Thinking in JAVA",
    "Borrow":"Nicola":"Code Complete",
    "Borrow":"Thierry":"Code Complete"
>
```

Figure 17: Intermediate Result after Step 2 of *Program 7*

At this moment, as there are some "Borrow" pairs in the container, *borrowBook* can be applied to perform borrowing activity. It will transfer all "Borrow" *pairs* from the container to "RECORD" sub-solution and remove the relative book *pair* in "BOOKS" *sub-solution*. Finally, we will obtain the following result shown in *Figure 18*:

From the result, we can see all the books have been borrowed so that we can see no "BOOK" pair in "BOOKS" sub-solution. At the same time, all "Borrow" pairs have been moved into "RECORDS" sub-solution.

3.3 Exercises

Up to now, we believe that our readers can write a more powerful program. *One-shot* rules is applied to ensure the higher order property which gives a possible solution to perform sequential execution. In this section, we list a couple of exercises. Interested readers can try to solve these problems. All the results and source codes can be found in the **SVN Repository**.

- **Factorial**

  Given an integer number, readers are required to compute its factorial. For example, if 4 is given, to compute its factorial, we need to calculate $4\times3\times2\times1=24$. Here, we ask readers to

```
<
    “BOOKS”:<
        “next_id”:4,
    >,
    “STUDENTS”:<
        “next_id”:4,
        “STUDENT”:1:“Thierry”,
        “STUDENT”:2:“Nicola”,
        “STUDENT”:3:“Valentine”
    >,
    “RECORD”:<
        “Borrow”:“Thierry”:“Thinking in JAVA”,
        “Borrow”:“Nicola”:“Code Complete”,
        “Borrow”:“Thierry”:“Code Complete”
    >,
    borrowBook
>
```

Figure 18: Final result of *Program 7*

calculate 10!.

- **Factorization**

  Every positive integer number can be expressed by a multiple of several prime numbers, for example, 24=2×2×2×3. Given a positive integer number, readers are asked this time to compute its factorization. Here, we can take 840 for example.

- **Extend “Library” application**

  Extend the “Library” application by creating a “*returnBook*” rule. When a student returns a book, it will delete the relative “Borrow” *pair* in the “RECORDS” *sub-solution* and re-generate the relative “BOOK” *pair* in “BOOKS” sub-solution.

  To re-generate a “BOOK” *pair*, it has to know its id number, but this information is deleted while performing “*borrowbook*” rule; as a result, some modifications also have to be made to “*borrowBook*”rule.

4 Advanced Topics

At this point, we have introduced almost all syntax of HOCL and their usage. We are assuming that our readers now have good understanding of HOCL. In this section, we are going to address some advanced topics.

4.1 Hybrid model: Using Java class in HOCL

The HOCL compiler is developed in JAVA. It works on the top of Java compiler. Once there is an HOCL source file created (*filename.hocl*), the HOCL compiler translates it into Java source files. These Java files are then compiled by Java compiler to produce the executable binary files. Accordingly, HOCL can be greatly enriched by mixing with Java. In this section, we will introduce a "hybrid programming model": how to use Java statements and classes in HOCL programming.

As a way of getting comfortable with this "hybrid" model, let us work together through all details needed with an example of "library application". This project extends the "*library application*" that we have developed in *section 3.2*. With this scenario, we will show you how to create Java objects and operate on these objects to perform certain tasks such as registration of a student, borrowing or returning a book etc. Experienced Java developer can pass quickly the Java development part and focus on HOCL source code.

This project will be constructed step by step; We strongly recommend the readers to follow these steps in your machines. This practical experience will help a lot in mastering "hybrid programming model". Since the source codes of latter steps have a lot of redundancy with the previous ones, we are not going to list the complete source codes for each step in this manual, but only give out the lines that extends the former steps; As a result, readers have to pay attention on passing from one step to the next and carefully produce your own codes. If you find you are lost in a certain step, you can check your codes with the ones that we provide in the **SVN repository**, it contains the complete source codes for each step. After following this project, readers can make further extensions by your interests.

In this project, instead of using pairs, we employ Java objects to store the information of students, books and borrowing records. In this way, three Java classes are created for representing respectively student, book and record. And then, in HOCL program, new Java objects are created. These Java objects can be regarded as molecules which can take part in the chemical reactions: some Java objects are removed while some new ones are created. Besides creating new objects, the state of a Java object can also be changed during the reactions; for example, if a student wants to borrow a book, the state of this book has to be changed from "Available" to "Not_available". We have several chemical rules created to perform these operations such as borrowing or returning a book. These functionality will be enriched step by step.

4.1.1 Step 1: Construct *Student* and *Book* JAVA classes

First and foremost, before performing "borrowing" and "returning" operations, some variables of certain types have to be created to represent books and students. In *section 3.2*, pairs are used to fulfill this task; but this time, we prefer to use Java objects. Therefore, a **Student** class is designed

for representing student while a **Book** class is created to store all information of a book. This is the first step of this project so that we need only to create the class frames and then, we will enrich their contents in the following steps.

- **Class: Book**

  We assume that readers have some experience on *object-oriented programming* and Java development. *Program 8* shows the definition of **Book** class.

Program 8 Library Project: Book.java

```
1   package example.libraryJava;
2
3   public class Book {
4
5       Book(String aName){
6           this.book_name=aName;
7           this.book_id=next_bookid;
8           next_bookid++;
9       }
10
11      public String getBookName(){
12          return this.book_name;
13      }
14
15      public int getBookId(){
16          return this.book_id;
17      }
18
19      public String toString(){
20          return "\n This is a book.ID:"+this.book_id+", Name:"+
21              this.book_name+".";
22      }
23
24      private static int next_bookid;
25      private String book_name;
26      private int book_id;
27  }
```

A book contains an *id* number as well as a name. As a result, two *instance fields* have been created: *book_id* indicates the *id* number of a book and *book_name* tells its name (*Line 24-25*). "*next_bookid*" field is defined as **statics**(*Line 23*) saying that it is the variable of class. Every "**Book**" object has its own *book_id* field, but there is only one *next_bookid* field that is shared among all instances of the "**Book**" class. This field is used to initialize the identical *book_id* field.

Having all fields defined, the next step is to define the *constructor* (*Line 5-9*). When constructing a "**Book**" object, a **String** value should be provided as parameter for initializing

the *book_name* field; but for the *book_id* field, it has to be identical without doubt. *book_id* is the unique identification for each book. To ensure its uniqueness, its value cannot be provided by users but should be assigned automatically. In this case, "*variable of class*" becomes the best choice. That is why we define *next_bookid* field as **static**. In this way, *book_id* will be allocated as the value indicated by the *next_bookid* field. And then, *next_bookid* field will be increased by one ensuring that the next time, no duplicated *id* number will be assigned to another **Book** object.

It is then necessary to define some methods to perform certain operations. The most basic ones are *accessor* and *mutator* methods. These methods are used for visiting those instance fields, getting or modifying their values. For a **Book** object, it is necessary to get its name, id number and description. In consequence, three *accessor* methods have to be created to perform these tasks. *getBookName* (*Line 11-13*) returns a String indicating the name of a book; *getBookId* method (*Line 15-17*) returns its identical *id* number; and *toString* method (*Line 19-21*) prints a short description of a **Book** object in the following form: "This is a book. ID: *book_id*, Name: *book_name*".

At this moment, we have constructed the framework of a **Book** class. It is now possible to create Java objects in HOCL source file; and then we can get its *id* number and name as well as a short description using its methods. As this project goes on, further extensions will be added to this class to perform more complex operations.

- **Class: Student**

  Student class is defined in the similar way. There are three instance fields: *stu_name*, *stu_id* and *next_stuid*, where the last one is declared as **static**. It has the same functionality as the *next_bookid* field in **Book** class. There are also three methods defined, *getStuName* will return a student's name; *getStuId* will return a student's id number and *toString* will print the information of a student in the following manner: "This is a student. ID: *stu_id*+, Name: *stu_name*"; Sharing much similarities with **Book** class, **Student** class is defined in *Program 9*.

4.1.2 Step 2: Construct HOCL file

Having Java classes defined, in this step, we are going to construct an HOCL source file to realize this "library application" mixing chemical and Java programming. The above defined Java classes have to be used in this HOCL program. In this step, we will call the constructors to build new Java objects. These Java objects are regarded as molecules which can later participate in the chemical reactions.

To use Java classes, an HOCL source file has to be placed in the same package with those Java classes; simply say, they have to be stored in the same repository. If you want to use the classes in other packages, you must use "include" expression to include them. *Program 10* shows the HOCL source code.

In *line 1*, it is declared that this file is included in **example.libraryJava** package. This is the same package where **Book** and **Student** classes are defined (cf: *Line 1* of *Program 8* and *9*). In a nutshell, *libraryJava.hocl* (*Program 10*), *Book.java* (*Program 8*) and *Student.java* (*Program 9*) have to be placed in the same folder.

Program 9 Library Project: Student.java

```
1   package example.libraryJava;
2
3   public class Student {
4
5       Student(String aName){
6           this.stu_name=aName;
7           this.stu_id=next_stuid;
8           next_stuid++;
9       }
10
11      public String getStuName(){
12          return this.stu_name;
13      }
14
15      public int getStuId(){
16          return this.stu_id;
17      }
18
19      public String toString(){
20          return "\n This is a student. ID:"+this.stu_id+", Name:"+
21              this.stu_name+".";
22      }
23
24      private static int next_stuid;
25      private String stu_name;
26      private int stu_id;
27  }
```

Note that this is a very particular program in which no chemical rule is defined. Here, a *2-tiers solution structure* is used to organize data. The *tier-1 solution* (*Line 3-15*) contains two *tier-2 solutions*. One named as "BOOKS" (*Line 4-8*) can be seen as a container to place all the objects of type **Book**; the other named as "STUDENTS"(*Line 10-14*) contains all the **Student** Java objects. And then, we need to have them initialized.

To create an object of a certain class, the following Java statement is used:

$$\text{new } \textbf{className} \ (parameter_1, \ parameter_1, \ ...)$$

In this example, three **Book** objects (*line 5-7*) and three **Student** objects (*Line 11-13*) are created. Executing *Program 10*, we can get the following results shown in *Figure 19*:

Program 10 Library Project: libraryJava.hocl

```
1   package example.libraryJava;
2
3   <
4       "BOOKS":<
5           new Book("Thinking in JAVA"),
6           new Book("Code Complete"),
7           new Book("Code Complete")
8       >
9
10      "STUDENTS":<
11          new Student("Thierry"),
12          new Student("Nicola"),
13          new Student("Valentine")
14      >
15  >
```

```
1   <
2       "BOOKS":<
3           This is a book. ID:1, Name: "Thinking in JAVA",
4           This is a book. ID:2, Name: "Code Complete",
5           This is a book. ID:3, Name: "Code Complete"
6       >,
7
8       "STUDENTS":<
9           "This is a student. ID: 1, Name: "Thierry",
10          "This is a student. ID: 2, Name: "Nicola",
11          "This is a student. ID: 3, Name: "Valentine"
12      >
13  >
```

Figure 19: Library Application: Running result for step 2

When printing a Java object to the console or a command window, the *toString* method is called automatically to display a string indicating the information of this class. From this result, we can see that by calling the constructors of Java classes, some Java objects have already been created. And the contents of these objects are printed out by their *toString* methods (*Line 3-5* and *line 9-11* in *Figure 19*). The *id* numbers of students and books are assigned automatically and identically.

In this section, we demonstrated how to construct a Java object in HOCL program; in the next section, a chemical rule will be added to perform borrowing operation. We are later going to show how to operate on Java objects in the following sections.

4.1.3 Step 3: Create a chemical rule: *borrowBook*

We assume that all **Book** objects and **Student** objects have been well created. Now we can define a chemical rule *borrowBook* to perform borrowing operation. In this step, we stay only at the chemical programming level, that is to say, a borrowing operation is only performed by rewriting molecules. Apart from that, the object state leave unchanged. In the next section, we will show you how to call Java methods to change the state of an object.

To borrow a book, a "Borrow" *pair* has to be added in the final solution. This *pair* tells the compiler that there is a borrowing operation to be performed. We have the same definition of this "Borrow" pair as in *Program 7*. *borrowBook* chemical rule will move the "Borrow" pair from *container* into a "RECORDS" *tier-2 solution*. First of all, we extend *Program 10* with a 'RECORDS" *tier-2 solution*. The following line has to be added after *line 14* of *Program 10*:

"RECORDS":<>

It is very **IMPORTANT** that **DO NOT** forget to add a comma at the end of *line 14*. And then, the following definition (shown in *Figure 20*) of rule *borrowBook* has to be placed before *line 3*:

```
let borrowBook =
    replace "Borrow":stu_name::String:book_name::String, "RECORDS":<?x>
    by "RECORDS":<x,"Borrow":stu_name:book_name>
in
```

Figure 20: Library Project: Definition of *borrowBook* Chemical Rule

This rule is identical to the one defined in *Program 7*. As we stated before, in the next section, readers will find that we extend its definition by calling some Java methods in order to perform borrowing activity at both chemical computing level as well as Java object level.

At this stage, it is necessary to put the following testing pairs (shown in *Figure 21*) in the final solution:

```
"Borrow":"Thierry":"Thinking in JAVA",
"Borrow":"Nicola":"Code Complete",
"Borrow":"Thierry":"C++ Primer"
```

Figure 21: Library Project: Testing Pairs

Running this program, the following result (show in *Figure 22*) is obtained:

```
<
    "BOOKS":<
        This is a book. ID:1, Name: "Thinking in JAVA",
        This is a book. ID:2, Name: "Code Complete",
        This is a book. ID:3, Name: "Code Complete"
    >

    "STUDENTS":<
        "This is a student. ID: 1, Name: "Thierry",
        "This is a student. ID: 2, Name: "Nicola",
        "This is a student. ID: 3, Name: "Valentine"
    >

    "RECORDS":<
        "Borrow":"Thierry":"Thinking in JAVA",
        "Borrow":"Nicola":"Code Complete",
        "Borrow":"Thierry":"C++ Primer"
    >
>
```

Figure 22: Library Project: Running Result for step 3

We can see that all "Borrow" *pairs* are moved into "RECORDS" sub-solution. Because this rule performs only at chemical computing level, we cannot find any modification on the Java object (From the strings that are printed out by *toString* method, we can see that the states of these object leave unchanged).

4.1.4 Step 4: Add a method "*borrowBook*" to class "Book"

From the former development, we can see that the chemical rule "*borrowBook*" has nothing to do with the Java objects. It stays only at chemical programming level. In this section, we will firstly add a method "*borrowBook*" to **Book** class which can perform borrowing activity at Java object level (change the state of an object); and then some modifications are made to the "*borrowBook*" chemical rule so that it can call Java method to operate on Java objects.

An "available" field has to be added to **Book** class indicating whether this book is available. As a result, the following line is added after *line 25* of *Program 8*:

private String *available*;

If this field equals to "YES", that means this book is available; in contrast, if it equals to "NO", this means the book is borrowed or reserved by another student. After construction of a **Book** object, every book is initialized as "available". As a new field is added, the following line has to be added to its *constructor*, after *line 8* of *Program 8*:

this.available="YES";

For the same reason, *toString* method returns the information of a book; this information also has to tell its availability. In consequence, replace *line 19-21* of *program 8* with the following lines in *Figure 23*:

```
public String toString(){
    return "\n This is a book.ID:"+this.book_id+", Name:"+this.book_name+
            ",Available:"+this.available+".";
}
```

Figure 23: Definition of *toString* Method

And now, we can define a new method *borrowBook* in **Book** class. The return type of this method is defined as **boolean**. We will explain the reason of this definition later on. It checks the "available" field of a **Book** object; if it is marked as "YES", that is to say this book is available, we can then perform the borrowing; then the value of this field is set to "NO" and return true; On the other hand, if it is marked as "NO", meaning this book is not available, it returns false which blocks moving "Borrow" *pair* into "RECORDS" *tier-2 solution*. The following code listed in *Figure 24* has to be added in *Program 8* after *line 22*:

```
public boolean borrowBook(){
    if (this.available=="YES"){
        this.available="NO";
        return true;
    }
    else
        return false;
}
```

Figure 24: Definition of *borrowBook* Method

Having this method defined in Java class, it is needed to re-define the *borrowBook* chemical rule ensuring that HOCL program can call this Java method to operate on **Book** objects. This chemical rule will firstly find a **Book** object that has the same name with the *book_name* field of a "Borrow" pair (In the **if** expression). Next, it will call *borrowBook* method of this **Book** object to check its availability. Because this operation only change the state of an object but no new object is created, we can only call this method in **if** expression (in **by** expression, only new molecules could be presented, but this method returns a **boolean** type while not a **Book** type). That is why we prefer to define *borrowBook* Java method as **boolean** for its *return type*.

If it returns **true**, this "Borrow" pair will be moved into the "RECORDS" *tier-2 solution*. The following code (shown in *Figure 25*) replaces the former definition of the *borrowBook* chemical rule in *Figure 20*:

Compile this HOCL program and run it, you will get the following result shown in *Figure 26*:

From the result, we can see that all books have been borrowed("Borrow" pairs have been moved to "RECORDS" *tier-2 solution*) and their state are marked as "non-available". Up to this point,

```
let borrowBook=
    replace "BOOKS":<?w, book::Book>,"RECORDS":<?l>,
            "Borrow":stu_name::String:book_name::String
    by "RECORDS":<l, "Borrow":stu_name:book_name>, "BOOKS":<w, book>
    if book.getBookName()== book_name && book.borrowBook()
in
```

Figure 25: Definition of *borrowBook* Chemical Rule

```
<
    "BOOKS":<
        This is a book. ID:1, Name: "Thinking in JAVA", "Available": "NO",
        This is a book. ID:2, Name: "Code Complete", "Available": "NO",
        This is a book. ID:3, Name: "C++ Primer", "Available": "NO"
    >

    "STUDENTS":<
        "This is a student. ID: 1, Name: "Thierry",
        "This is a student. ID: 2, Name: "Nicola",
        "This is a student. ID: 3, Name: "Valentine"
    >

    "RECORDS":<
        "Borrow":"Thierry":"Thinking in JAVA",
        "Borrow":"Nicola":"Code Complete",
        "Borrow":"Thierry":"C++ Primer"
    >
>
```

Figure 26: Library Project: Running result for step 4

readers can create Java object and call its methods to perform some operations in a HOCL program. We will make further extensions in the following sections.

4.1.5 Step 5: Add a method "*borrowBook*" to class "Student"

In the previous section, the chemical rule "*borrowBook*" changes the state of a **Book** object; in this section, we will do more. It is nice to maintain a list for each student indicating all the books that he has not yet returned. When a student borrows a book from the library, an item concerning the information of this book (such as its name and ID number) will be added to the list; on the other side, if a student returns back a book, the relative item in the list will be removed.

To achieve this, we need to extend "**Student**" class. A list is created to store all books that a student keeps. We use a "**LinkedList**" because it can allocate the memory units dynamically according the length of the list. The following line is added before *line 23* of the *Program 9*:

private List<Book> *borrowedBooks*;

DO NOT forget to import the associated packages that define "**List**" and "**LinkedList**" classes. Place the following lines after the first line of the program listed in *Figure 9*:

import java.util.LinkedList;
import java.util.List;

As this new instance field has been added, we need to initialize it in the *constructor*. So the following line has to be added after *line 8* of **Student** class (*Program 9*):

this.*borrowedBooks*=**new** LinkedList();

At this point, when constructing a "**Student**" object, a list will be created. This list is empty after its creation. Accordingly, it is needed to create a *borrowBook* method for **Student** class which will add an item of **Book** object to this list when this student borrows that book. The same as *borrowBook* method of "**Book**" class, this method is also declared as "**boolean**". The following definition shown in *Figure 27* has to be added in "**Student**" class after *line 21*:

```
public boolean borrowBook(Book aBook){
    this.borrowedBooks.add(aBook);
    return true;
}
```

Figure 27: Library Project: The definition of *borrowBook* method

This method needs a **Book** object as parameter. Getting a **Book** object, it adds it to the end of the list. It is also necessary to re-define *toString* method because we want also to print out the list of books that a student kept. A loop is implemented to print out all items in the list. The lines listed in *Figure 28* replaces *line 19-21* of *Program 9*:

```
public String toString(){
    String info= "\nThis is a student. ID:"+this.stu_id+
          ", Name:"+this.stu_name+"BorowedBooks:";
    for(int i=0;i<this.borrowedBooks.size();i++){
        info+=(i+1)+": "+this.borrowedBooks.get(i).getBookName()+";";
    }
    info+="\n";
    return info;
}
```

Figure 28: The Definition of *toString* Method

At this moment, all extensions to "**Student**" class have been done, what is left to us is just to redefine the *borrowBook* chemical rule in HOCL program so that it can call both *borrowBook*

methods in **Book** class and **Student** class to perform borrowing operation. The re-definition of *borrowBook* chemical rule is given out in *Figure 29* and the following lines will replace the former definition in *Figure 25*:

```
let borrowBook=
    replace "BOOKS":<?w, book::Book>, "STUDENTS":<?k,stu::Student>,
        "RECORDS":<?l>,"Borrow":stu_name::String:book_name::String
    by "RECORDS":<l, "Borrow":stu_name:book_name>,
        "BOOKS":<w, book>, "STUDENTS":<k, stu>
    if book.getBookName()== book_name && stu.getStuName()==stu_name &&
        book.borrowBook() && stu.borrowBook(book)
in
```

Figure 29: The Definition of *borrowBook* Chemical Rule

Compare to the former definition, this rule has more functionality. Apart from setting the availability of the borrowed book to "NO", it will call *borrowBook* method of a **Student** object to add this **Book** object to its *borrowedBooks* list. Executing this program, we have the following result shown in *Figure 30*:

```
<
    borrowBook,
    "RECORDS":<
        "Borrow":"Thierry":"Thinking in JAVA",
        "Borrow":"Nicola":"Code Complete",
        "Borrow":"Thierry":"C++ Primer"
    >,
    "BOOKS":<
        This is a book.ID:0, Name:Thinking in JAVA, Available:NO.,
        This is a book.ID:1, Name:Code Complete, Available:NO.,
        This is a book.ID:2, Name:C++ Primer, Available:NO.
    >,
    "STUDENTS":<
        This is a student. ID:2, Name:Valentine
        BorowedBooks:,
        This is a student. ID:1, Name:Nicola
        BorowedBooks:1: Code Complete;,
        This is a student. ID:0, Name:Thierry
        BorowedBooks:1: Thinking in JAVA;2: C++ Primer;
    >
>
```

Figure 30: Library Project: Running result for step 5

From this result, we can see that the *borrowBook* chemical rule has done the following three tasks:

- In chemical programming level, all "Borrow" *pairs* have been moved into "RECORDS" *tier-2 solution*; This is performed by multi-set rewriting defined by "**replace...by...**" expression of the *borrowBook* rule;

- All **Book** objects have been marked as "non-available". It is performed by *borrowBook* method of **Book** class; the HOCL program calls this method in the "**if**" expression of the *borrowBook* rule;

- Each student maintains a list indicating the books that he has not yet returned. This is achieved by *borrowBook* method of **Student** class; the HOCL program calls this method in "**if**" expression of the *borrowbook* rule;

At this point, we believe that readers have mastered how to operate on Java classes in HOCL programming. In the following parts, we will still take a step forward to make further extensions on this project. A chemical rule "*returnBook*" will be created perform the reverser operation to "*borrowBook*" chemical rule; furthermore, another Java class "Record" will be designed to replace "Borrow" pair in "RECORDS" *tier-2 solution* (As stated before, we prefer to use Java objects to represent entities than *pairs*). These extensions will be introduced rapidly since they share a lot of similarities with the former ones. There will be nothing new but readers can improve your comprehension by reading the following sections.

4.1.6 Step 6: Add *returnBook* chemical rule

In this part, we are going to add *returnBook* chemical rule which performs the reverse operations to "*borrowBook*" chemical rule:

- Firstly, it adds a "Returned" string field at the end of "Borrow" pair in "RECORDS" tier-2 solution, meaning that this book has been returned. For example, if student "Thierry" has returned "C++ Primer", the following *pair*:

  "Borrow":"Thierry":"C++ Primer"

  will be replaced by:

  "Borrow":"Thierry":"C++ Primer":"Returned"

  pair. We do not prefer to delete these "Borrow" pairs because we can check the borrowing history of a student by leaving them in the "RECORDS" solution.

- If a student returns a book, this book has to be re-marked as "available" in order that other students can borrow it. This requires the creation of a new method "*returnBook*" in the **Book** Java class. And "*returnBook*" chemical rule will call this method in its **if** expression.

- Apart from setting the **Book** object as "available", it is also necessary to remove the relative item from the *borrowedBook* list of a **Student** object. A new method "*returnBook*" is defined in **Student** class and then it will be called by "*returnBook*" chemical rule in its **if** expression.

As a result, the following extension has to be added in **Book** class, *Figure 31* gives out the definition of *returnBook* method which will re-set a book as "available".

```
public boolean returnbook(){
    if (this.available=="NO"){
        this.available="YES";
        return true;
    }
    else
        return false;
}
```

Figure 31: Library project: The definition of *returnbook* chemical rule

Program 11 gives out the complete definition of the **Book** class.

Equally, the *returnBook* method of **Student** class is defined in *Figure 32*, the relative item will be removed from the *borrowedBook* list.

```
public boolean returnBook(Book aBook){
    this.borrowedBooks.remove(aBook);
    return true;
}
```

Figure 32: The Definition of *returnBook* Method

And *Program 12* shows you the complete source code of the **Student** class.

And then, *Figure 33* shows you the definition of "*returnBook*" chemical rule. Since it performs almost the reverse operation to "*borrowBook*" rule, it is not difficult to understand. So we will not explain it in detail here.

```
let returnBook=
    replace "STUDENTS":<?k,stu::Student>, "BOOKS":<?w, book::Book>,
            "Return":stuname::String:bookname::String,
             "RECORDS":<?l,"Borrow":stuname::String:bookname::String>
    by "RECORDS":<l,"Borrow":stuname:bookname:"Returned">,
            "BOOKS":<w, book>, "STUDENTS":<k, stu>
    if book.getBookName()== bookname && stu.getStuName()==stuname &&
            book.returnbook()&&stu.returnBook(book)
in
```

Figure 33: The Definition of *returnBook* Chemical Rule

Program 11 Project "libraryJava": Book.java (Final version)

```
1   public class Book {
2       Book(String aName){
3           this.book_name=aName;
4           this.book_id=id;
5           this.available="YES";
6           id++;
7       }
8
9       public String getBookName(){
10          return this.book_name;
11      }
12
13      public int getBookId(){
14          return this.book_id;
15      }
16
17      public void setBookName(String aName){
18          this.book_name=aName;
19      }
20
21      public String toString(){
22          return "\n This is a book.ID:"+this.book_id+", Name:"+
23              this.book_name+", Available:"+this.available+".";
24      }
25
26      public boolean borrowBook(){
27          if (this.available=="YES"){
28              this.available="NO";
29              return true;
30          }
31          else
32              return false;
33      }
34
35      public boolean returnbook(){
36          if (this.available=="NO"){
37              this.available="YES";
38              return true;
39          }
40          else
41              return false;
42      }
43
44      private static int id;
45      private String book_name;
46      private int book_id;
47      private String available;
48  }
```

Program 12 Project "libraryJava": Student.java (Final version)

```
1   import java.util.LinkedList;
2   import java.util.List;
3
4   public class Student {
5       Student(String aName){
6           this.stu_name=aName;
7           this.stu_id=next_stuid;
8           this.borrowedBooks=new LinkedList<Book>();
9           next_stuid++;
10      }
11
12      public String getStuName(){
13          return this.stu_name;
14      }
15
16      public int getStuId(){
17          return this.stu_id;
18      }
19
20      public void setStuName(String aName){
21          this.stu_name=aName;
22      }
23
24      public String toString(){
25          String info= "\n This is a student. ID:"+this.stu_id+",
26                  Name:"+this.stu_name
27                  +"\n BorowedBooks:";
28          for(int i=0;i¡this.borrowedBooks.size();i++){
29              info+=(i+1)+": "+this.borrowedBooks.get(i).getBookName()+";";
30          }
31          info+="\n";
32          return info;
33      }
34
35      public boolean borrowBook(Book aBook){
36          this.borrowedBooks.add(aBook);
37          return true;
38      }
39
40      public boolean returnBook(Book aBook){
41          this.borrowedBooks.remove(aBook);
42          return true;
43      }
44
45      private List<Book> borrowedBooks;
46      private static int next_stuid;
47      private String stu_name;
48      private int stu_id;
49  }
```

Finally, the following "Return" *pair* has to be put in the HOCL program just after the lines of those "Borrow" *pairs*:

"Return":"Thierry":"C++ Primer"

After compiling this HOCL program and its execution, the final result is shown in *Figure 34*:

```
<
    borrowBook,
    "RECORDS":<
        "Borrow":"Thierry":"Thinking in JAVA",
        "Borrow":"Nicola":"Code Complete",
        "Borrow":"Thierry":"C++ Primer":"Returned"
    >,
    "BOOKS":<
        This is a book.ID:0, Name:Thinking in JAVA, Available:NO.,
        This is a book.ID:1, Name:Code Complete, Available:NO.,
        This is a book.ID:2, Name:C++ Primer, Available:YES.
    >,
    "STUDENTS":<
        This is a student. ID:2, Name:Valentine
        BorowedBooks:,
        This is a student. ID:1, Name:Nicola
        BorowedBooks:1: Code Complete;,
        This is a student. ID:0, Name:Thierry
        BorowedBooks:1: Thinking in JAVA;
    >
>
```

Figure 34: Library Project: Final result for step 6

Compare this result with the one in the previous section (*Figure 30*), we can see that the student "Thierry" has returned the book "C++ Primer"; this book is re-marked as "available" and the relative item is removed from the borrowed list of this student.

4.1.7 Step 7: Construct Record class

As we have stated before, we prefer to use Java object to organize data. But in the former version of the project, the borrowing records are still represented by the following *pairs*:

"Borrow":*stu_name*:*book_name*:*returned_or_not*

In this section, we are going to construct a Java class "Record" replacing those *pairs* to represent borrowing records. *Program 13* shows the source code of its definition.

Program 13 Project "libraryJava": Record.java

```
1   public class Record {
2       Record(int stuId, int bookId, String bookName, String stuName){
3           this.book_id=bookId;
4           this.book_name=bookName;
5           this.stu_id=stuId;
6           this.stu_name=stuName;
7           this.record_state="NOT YET";
8       }
9
10      Record(String bookName, String stuName){
11          this.book_name=bookName;
12          this.stu_name=stuName;
13          this.record_state="NOT YET";
14      }
15
16      public int getStuId(){return this.stu_id;}
17
18      public int getBookId(){return this.book_id;}
19
20      public String getStuName(){return this.stu_name;}
21
22      public String getBookName(){return this.book_name;}
23
24      public String getRecordStata(){
25          return this.record_state;
26      }
27
28      public boolean returnBook(){
29          this.record_state="RETURNED";
30          return true;
31      }
32
33      public String toString(){
34          String s= "\nStudent: "+this.stu_name+
35                  " has borrowed a book: "  +this.book_name+".";
36          if (this.record_state=="RETURNED"){
37              s+="Now this book has been returned.\n";
38          }
39          return s;
40      }
41
42      private int stu_id;
43      private int book_id;
44      private String stu_name;
45      private String book_name;
46      private String record_state;
47  }
```

A borrowing record contains a student *id* number as well as his name; furthermore, the *id* number and the name of a book are also necessary; finally, a field "*record_state*" is defined indicating whether this student has returned that book or not. When a student borrows a book from library, a borrowing record is created; the state of this record is initialized as "NOT YET" meaning that the student has not yet returned that book. In contrast, when a student returns a book, the state of relative record is changed to "RETURNED". In consequence, five instance fields are defined from *line 42* to *line 46* in *Program 13*.

As a result, we need to define five accessor methods to get the values of those five fields. These five methods are defined from *lines 16* to *line 26*. There are two constructors defined, one is a complete one (*Line 2-8*) and the other is a compact one (Line 10-14). The complete one initializes each field of a **Record** object but the compact one only assigns the *book_name* and *stu_name* fields. In this simple example, as we have mentioned before, every two books have different names, we could use the both. Every record is initialized with the state: "NOT YET" (*line7,13*).

When a student returns a book, it is needed to change this state to "RETURNED". As a result, a "*returnBook*" method is created. Its definition is shown from *line 28* to *line 31*. As it changes only the state of an object but no new object is created, it has to be declared as "**boolean**" that we can call it in HOCL program.

Moreover, a "*toString*" method is also wanted to print out the state of a **Record** object. Its definition is given from *line 33* to *line 40*. It will print out the student name and the book name; if the state of this record equals to "RETURNED", it will further print the following information "Now this book has been returned".

The only task left to us is to modify the HOCL source code in order to use the **Record** class in the program. A **Record** object is created in "RECORDS" *tier-2 solution* when *borrowBook* chemical rule is applied. For *returnBook* chemical rule, firstly it is needed to find the relative **Record** object which has the same *book_name* and *stu_name* with the ones indicated in the "Return" pair; and then the *returnBook* method is called in order to change the state of this **Record** object to "RETURNED". The final complete source code for this HOCL program is given out in *Program 14*.

Executing this program, we will get the following result shown in *Figure 35*. We can see that three **Record** objects are created and the last one has been marked as "Returned".

Up to now, all the steps of our "library project" have been introduced. From this small project, we believe that the readers now have mastered how to use Java classes in HOCL programming. The interested readers can extend this project by your interests. Here we list some points that could help:

- Book *id* number is the only identifier of a book. That is to say, when perform borrowing and returning operations, it **MUST NOT** use names, neither book name nor student name. Each book has several copies which have the same name, authors, publisher and etc; the only difference between these copies is the id number. This will be the same case for student. Two students can have the same name, sex, birthdate, but they have the different id number. As a result, all the chemical rules as well as Java methods need to use the id number to select the relative molecules or Java objects for the computation.

Program 14 Project "libraryJava": libraryJava.hocl

```
1   let borrowBook=
2       replace "BOOKS":<?w, book::Book>, "STUDENTS":<?k, stu::Student>,
3           "RECORDS":<?l>,"Borrow":stu_name::String:book_name::String
4       by "BOOKS":<w, book>,"STUDENTS":<k, stu>,
5           "RECORDS":<l, new Record(stu.getStuId(),book.getBookId(),
6               book.getBookName(),stu.getStuName())>,
7       if book.getBookName()== book_name &&book.borrowBook()&&
8           stu.getStuName()==stu_name&& stu.borrowBook(book)
9   in
10  let returnBook=
11      replace "BOOKS":<?w, book::Book>, "STUDENTS":<?k, stu::Student>,
12          "RECORDS":<?l,record::Record>,"Return":stuname::String:bookname::String
13      by "RECORDS":<l,record>, "BOOKS":<w, book>, "STUDENTS":<k, stu>
14      if book.getBookName()== bookname && record.getBookName()==bookname &&
15          stu.getStuName()==stuname && record.getStuName()==stuname &&
16          book.returnbook()&& stu.returnBook(book)&&record.returnBook()
17  in
18  <
19      borrowBook,returnBook,
20      "BOOKS":<
21          new Book("Thinking in JAVA"),
22          new Book("Code Complete"),
23          new Book("C++ Primer")
24      >,
25      "STUDENTS":<
26          new Student("Thierry"),
27          new Student("Nicola"),
28          new Student("Valentine")
29      >,
30      "RECORDS":<>,
31
32          "Borrow":"Thierry":"Thinking in JAVA",
33          "Borrow":"Nicola":"Code Complete",
34          "Borrow":"Thierry":"C++ Primer",
35          "Return":"Thierry":"C++ Primer"
36  >
```

```
<
    borrowBook,returnBook,
    “RECORDS”:<
        Student: “Thierry” has borrowed a book: “Thinking in JAVA”.,
        Student: “Nicola” has borrowed a book: “Code Complete”.,
        Student: “Thierry” has borrowed a book: “C++ Primer”.
            Now this book has been returned.
    >,
    “BOOKS”:<
        This is a book.ID:0, Name:Thinking in JAVA, Available:NO.,
        This is a book.ID:1, Name:Code Complete, Available:NO.,
        This is a book.ID:2, Name:C++ Primer, Available:YES.
    >,
    “STUDENTS”:<
        This is a student. ID:2, Name:Valentine
        BorowedBooks:
        ,
        This is a student. ID:1, Name:Nicola
        BorowedBooks:1: Code Complete;
        ,
        His is a student. ID:0, Name:Thierry
        BorowedBooks:1: Thinking in JAVA;
    >
>
```

Figure 35: Library Project: Final Results

- Add new function: “inquire”. This might compose of some sub-functions, for example, a student can search his wanted books and see whether there is an available copy. Or a student can inquire his borrowing history and all the books that he has to return.

- Add new function: “reserve”. If there is no available book, a student can reserve it. When any copy of this book is returned, the system will automatically marked this book reserved by this student while other student could not borrow it. Each book has to maintain a “waiting list” that stores all the student reserving this book.

- Other functions concerning some other limitations, such as a student could borrow only 4 books at a moment, but a professor can borrow more. This depends on the interests of readers.

4.2 Nested Definition

In former chapters, readers have seen that a chemical rule can be defined either in the “*rule definition*” part of an HOCL program, or in the *final solution* (remember some *one-shot rules* to extract results or replace rules). But in this section, we are going to introduce an unconventional style

which we called "nested definition": That is to say, a rule can be defined within the definition of another rule. Reading the following example shown in *Program 15*:

Program 15 Nested definition: factorial.hocl

```
1  let getStart=
2      replace-one num::int
3      by "nextValue":num,
4          replace "nextValue":i::int by i, "nextValue":i-1 if i>0,
5          replace x::int, y::int by x*y
6  in
7  <getStart, 10>
```

In this example, at the starting point, there is a chemical rule *getStart* and an integer number (10) in the container (*Line 7*). And then, *getStart* replaces this number with two other chemical rules and a *pair*. These two rules are defined within the definition of *getStart*. One is to generate all positive numbers smaller than 10 (*Line 4*); while the other calculates the multiple of any two numbers (*Line 5*). In this way, the factorial of 10 can be worked out. The "nextValue":i *pair* is defined for generating integer numbers.

This program is equal to *Program 16*.

Program 16 Regular definition: factorial.hocl

```
1   let generatePair=
2       replace-one x::int
3       by "nextValue":x
4   in
5   let generateNumbers=
6       replace "nextValue":i::int
7       by "nextValue":i-1
8       if i>0
9   in
10  let multiple=
11      replace x::int, y::int
12      by x*y
13  in
14  <generatePair, generateNumbers, multiple, 10>
```

From this comparison, we can see that nested definition makes a program more concise and compact. As every coin has two sides, abuse of using this style to define rules makes your code difficult to understand. It is confused when a reader firstly meet <*getStart*, 10>. It has to follow its execution to know how it works. In theory, the degree of this nest definition can be infinitive but we do not recommend to do that. It makes programs hard to understand (more or less the same to "goto" statement in assemble language, which is easy used but make program hard to understand).