Say My Innate Capabilities in a Natural Language 1/3

30 September 2020

Kenzo Iwama

Abstract

A human uses a programming language and writes a program that will say what the program does, in strings that the program will acquire. The program does not see any program code written by the human but it says what the program code conducts while it gets input strings and makes output strings. It finds regularities in the input and output strings, stores the regularities, and later retrieves regularities that match new inputs, for applying them to generate strings that would follow the new inputs.

The program gets input strings through four channels, and it binds them together when it gets them at the same time. While the program finds many regularities and applies regularities to new inputs, it forms a way W_n of making regularities out of inputs as a new regularity. The program gives strings S_r to have another program P_a do S_r by applying a regularity R_r assuming that P_a has formed the regularity R_r. If the program P_a is not able to do it, the program tries to have the program P_a do the way W_n, namely make the regularity R_r so that the program P_a becomes able to apply the regularity R_r for doing S_r. The program binds the way W_n and strings S_w together, and gives S_w to have P_a do the way W_n. Here, the program has formed a regularity of having P_a do W_n to make R_r if the program P_a is not able to do R_r.

Since 1) the innate (or original) capability forms the way W_n, and 2) the program says the way W_n in strings S_w that the program aquires after it runs when the program tries to have the other program P_a do the way W_n, the program says its innate capability in strings S_w that it gets after it runs.

1. Introduction

This paper describes a case where a program becomes to say its innate (or original) capability in strings acquired after it runs. The capability is written in a programming language by a human, and the strings are given in a language other than the programming language. This section describes points of what and how our program says (or describes) its innate capability in English strings.

1.1 Sets and Representatives (or Regularities)

The program gets input strings through four channels and stores them in its memory. It binds strings together when it gets them through the channels at the same time. Strings through one channel (the first channel) are with quotations " ", and strings through the other three are without quotations. The program sequencially gets input strings, and stores them in the sequence as it gets. It places a string (next) between two strings when it gets the two strings, one string and the next string in a sequence. The program finds regularities (or forms sets and their representatives) in sequences of strings stored in the memory, and adds strings [ ] at the start and the end of each regularity.

The program conducts a cycle of three steps; at one step it does as it is written in the programming language, namely, gets input strings, retrieves regularities that match current inputs, and tries to find relations between sub strings of the input strings and find regularities in the input and retrieved regularities, at the next step it does regularities, and outputs strings, at the third step it does strings through the forth channel, namely, either continue to do regularities currently doing or to stop to do the regularities currently doing. Then it conducts the next cycle.

At the first step, the program tries to find a new regularity as follows; it finds (or chooses) two strings such that the two strings become the same if sub strings of each of the two strings are replaced by a representative string of a set that includes the sub strings as its members. For example, "Two apples and three apples equal five apples" and "Two oranges and three oranges equal five oranges" become the same when "apples" and "oranges" are replaced by "objects" respectively. "objects" is a representative of a set that includes "apples" and "oranges. The program tries to find other strings that become the same as the two strings if the sub strings are replaced by the representative string, and then makes a new set. The new set includes the two strings and the other strings as its

members. For example, “Two apples and three apples equal five apples” and “Two oranges and three oranges equal five oranges” become members of the new set. “Two objects and three objects equal five objects” is a representative of the new set. The representative is a new regularity.

To be more precise, a member of a set consists of strings gotten through more than one channel, and the strings are bound together. Its representative is formed by the program, and has no strings with quotations before it is bound to strings with quotations. For example, “apples” apples, and “oranges” oranges, are strings gotten through the channels, and become members of a set whose representative is “objects” objects. The string objects is given at the time of initialization, and “objects” is later bound to the string objects.

This paper assumes that the program has already formed various sets, which are explained later in the paper, and gets them at the time of initialization. Getting them, the program forms new sets consisting of new input strings. Then the program constructs a hierarchical set that includes sets already formed, which is also explained later.

1.2 Relations

The program gets / finds relations, <same> and <bind>. A relation <same> specifies which sub string is the same as which sub string. Here the relation <same> is between sub strings gotten through the same channel or representatives whose members are gotten through the same channel. A relation <bind> specifies which sub string without quotatins appear (or occur) together with which sub string with quotations. Figure 1 illustrates examples of the relation <same> between two sub strings and <bind> between two sub strings.

1.3 Doing Strings

Suppose that the program gets an input string. For example, it gets “Two desks and three desks equals “. Then it tries to retrieve a string that matches the input with replacing input sub strings with their representatives. “Two objects and three objects equal five objects” becomes the same when the program replaces “desks” in “Two desks and three desks equals “ by “objects”. [“objects” objects] is a representative of a set consisting of [“apples” apples], [“oranges” oranges], [“desks” desks] and others. Then the program does the retrieved string after replacing representative strings by input sub

strings and keeping relations <same>; For example, it gets “Two desks and three desks equals five desks“. Finding and keeping the relations is explained in Section 3.

When the input string with replacing input sub strings with their representatives does not match any string kept in its memory, the program keeps the input in its memory. The program tries to see if two strings match each other with replacing sub strings of each by their representatives after it keeps many input strings in the memory.

This paper assumes that the program gets several sets and their representatives as its initial setting, and Section 4 describes the sets. The most basic set is a set of all the sub strings that are supposed to repeatedly appear in its inputs, and its representative string is set to be ～ by the program.

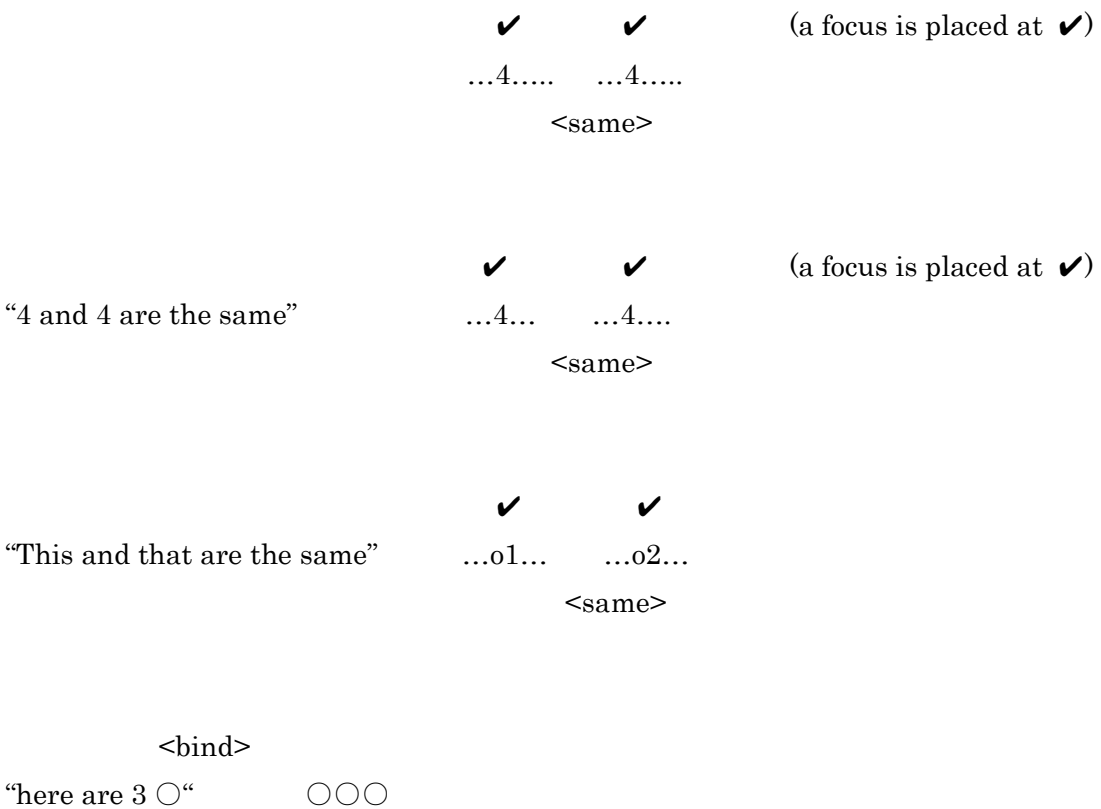


Figure 1. Example relations <same> between two sub strings, and an example relation <bind>. The program gets strings without quotations such as …4….. through channel 2 and 3, and gets ✔ through channel 3, and finds <same> between 4 and 4. The program gets strings “here are 3 ○” and ○○○ at the same time, and then it binds them together.

1.4 Other Programs

Suppose that the program finds the following many times; strings through the second channel and the third channel change at the same time, or changes through the third channel occur first and then changes through the second channel occur. Then the program binds a string “I do something.” to such changes of strings.

Suppose that the program gets strings many times in such a way that strings though the second channel at one moment differ from those at the next moment, but strings through the third channel keep the same. Then a string “another program does something.” is bound to such changes of strings.

1.5 Hierarchical Regularities

The program forms hierarchical regularities in three ways; 1) It first forms a regularity that holds in all the inputs, and then splits it into two regularities; a regularity holds in inputs with sub strings S_s, and the other holds in inputs with sub strings other than S_s. 2) It firstly forms two regularities and then unites the two to form a regularity that integrates the two. 3) It firstly forms a regularity, and finds there are cases where the regularity does not hold. Then the program forms a regularity that the original one holds in these cases and the original one does not hold in the other cases.

The following explains more of the case 3) of the above paragraph. The program forms a regularity R_L and then finds there are cases where the regularity R_L does not hold. The program stores the cases in its memory. Then the program tries to form a new regularity R_H describing when (or why) the regularity R_L holds and when (or why) the regularity R_L does not hold. The new regularity R_H is a hierarchical regularity that includes the cases where R_L holds and the cases where R_L does not hold. The next step the program takes is to form a way W_n either from cases R_L holds to cases R_L does not hold or from cases R_L does not hold to cases R_L holds.

After the program forms several hierarchical regularities, it forms a set of hierarchical regularities, and finds a representative of the set. A string with quotations bound to the representative becomes “there are cases where a certain regularity holds and cases where the regularity does not hold”.

Suppose that the program has formed a regularity W_n that the program, given a string S_r, first does not do the string, then makes a way (or a string) to do the string S_r, and does the string S_r. Now, the program has another program P_a do a string S_r, but it turns out that the other program P_a does not do the string. Then the program tries to make a string S_n bound to W_n. The program gives the string S_n to the other program P_a for the puropose of P_a's doing the string S_n bound to Wn. As a result of doing S_n bound to W_n, the program P_a forms a way (or a string) to do the string S_r, and does the string S_r.

Section 2.4 explains a regularity W_n mensioned in the above paragraph. Forming a regularity W_n is the same as forming a new set and its representative whose member is to make a way to do strings such as a string S_r.

1.6 Find and Bind Strings with Quotations to a Regularity

When the program forms a new regularity, it chooses strings through the second and the third channel that match each orher with replacing sub strings by their representatives already made. Each of the representatives already made consists of strings through the second channel and the third channel and strings made by the program, but may not bound to strings with quotations. The program does not create strings with quotations to bind them to the new regularity. If the new regularity is a hierarchical one, strings to specify a regularity is bound to the new regularity, but all the component regularities may not be bound to strings with quotations.

The program tries to let another program P_a do W_n, and it tries to output strings with quotations bound to W_n. But it finds that W_n is not fully bound to strings with quotations. Then the program tries to fully bind strings with quotations to W_n.

For example, the program has a regularity, […4… (next) …numb…] without bound to strings with quotations. The program retrieves strings (or regularities), ["Replace A by B." …A… (next) …B…], strings ["4" 4], and links implementing 4 is a member of a set with its representaive string ["numb" numb] . Then it binds "Replace 4 by numb." to …4… (next) …numb… .

…4… (next) …numb…

“Replace A by B” …A… (next) …B…

“Replace 4 by numb”. …4…. (next) …numb…..

Figure 2. An example of strings without a binding to a string with quotations and strings with a binding to a string with quotations.

2. Input Strings that become bound to Capabilities of the Program

2.1 Forming Sets

The program gets a sequence of input strings from outside of the program through four channels. The program gets a string a_1, it tries to find the same string in a memory. If it does not find the string in there, it stores the string a_1 in the memory. (A series of strings is cut at a space, a quotation, a comma, and a period.)

The program then gets a sequence of strings a_2, and a_3, and stores them in the memory. It then makes a sequence of strings, A_g, that are common to strings a_1, a_2 and a_3; it applies a function F on a_1, a_2, and a_3, and gets A_g. F is a function composed of its original function (a human writes) F_o and its acquired function F_e.

Next, the program gets a string, a_i^f, and applies F to a_i^f, to get a string, and finds the string is the same as the former part of A_g, A_g^f. The program takes a string A_g^l, from the memory, that follows A_g^f. It then applies the reverse of F to a_i^f and A_g^l and gets a_i^l. It claims (or assumes) that a_i^l follows a_i^f.

Similarly, the program gets strings, b_1, b_2, and b_3, and applies F to the strings to get a string B_g. Then the program gets a string, b_i^f and applies F to b_i^f, to get a string, and finds the string is the same as the former part of B_g, B_g^f. The program takes a string B_g^l, from the memory, that follows B_g^f. It then applies the reverse of F to b_i^f and B_g^l and gets b_i^l. It claims (or assumes) that b_i^l follows b_i^f. The same can be said when the program gets c_1, c_2, and c_3 and applies F to the strings to get a string C_g.

The program forms a set that includes strings, a_1, a_2, a_3, and others, as its members, which become A_g when F is applied to them by the program. The program sets A_g to be the representative of the set. Similarly, the program forms a set consisting of b_1, b_2, b_3 and others, which become B_g when the program applies F to them, and forms a set consisting of c_1, c_2, c_3 and others, which become C_g when the program applies F to them.

The program finds common strings S_c among pairs of (a_1, a_2, a_3, , : A_g), (b_1, b_2, b_3, , : B_g), (c_1, c_2, c_3, , : C_g), and other pairs of input strings and their representative strings. The common strings S_c among the pairs describe approximately the function F of the program, and the function are composed of those a human write, F_o, and those the program has

acquired, F_e. The more pairs the program forms, the more precisely the strings common to the pairs S_c describe the function F.

When the number of pairs such as ($a_1, a_2, a_3, , ,: A_g$), ($b_1, b_2, b_3, , ,: B_g$), ($c_1, c_2, c_3, , , : C_g$) increase, the common part of the pairs becomes F_o since F_e is acquired functions and the common part of F_e becomes the common part of all the acquired functions, namely assumptions about all the functions the program has from the beginning (namely, any function should be such and such).

Suppose a case occurs where the program tries to have the program P_a make d_1^l, after giving d_1^f, d_2^f, and d_2^l to P_a, and suppose the program Pa happens to have not formed yet strings S_c common in pairs ($a_1, a_2, a_3, , ,: A_g$), ($b_1, b_2, b_3, , ,: B_g$), ($c_1, c_2, c_3, , , : C_g$), and other pairs and thus the program P_a is not able to make d_1^l. Then the program binds strings S_I gotten from outside and strings S_c common in pairs ($a_1, a_2, a_3, , ,: A_g$), ($b_1, b_2, b_3, , ,: B_g$), ($c_1, c_2, c_3, , , : C_g$), and other pairs together for the purpose of having another program P_a make d_1^l. The program gives P_a the strings S_I, so that P_a is able to make d_1^l. The program P_a receives the strings S_I as its inputs, retrieves strings S_F, not integrated as strings S_c, bound to parts of function F, and makes S_F become true; program P_a, given $d_2^f d_2^l$ and d_1^f, makes a pair ($d_2^f d_2^l, d_1^f d_1^l: D_g$) since the pair made becomes consistent to strings S_F.

Given S_I, $d_2^f d_2^l$ and d_1^f, to make the pair ($d_2^f d_2^l, d_1^f d_1^l: D_g$), namely to make the pair consistent to S_F, (strings S_F matche S_I) is the same as to apply function F to string d_1 to get D_g and apply the reverse of F to d_2 and d_1^f to get d_1^l. Thus the program describes its own function F in strings S_I gotten from outside.

Section 2.3 explains why and how the program binds strings S_c made by function F, and strings S_I from outside together.

2.2 Inputs, Outputs, Regularities and Consistency

The program tries to achieve consistency between input strings and regularities already formed. When it gets new inputs s_i, the program retrieves strings s_r the former part of which match the inputs s_i with replacing sub strings by their representatives. The retrived strings, as a whole, are a regularity r, and the input strings s_i turn out to be an example of the regularity r. It outputs strings s_o if the retrieved strings s_r and the input strings s_i make the outputs s_o, in other words, the latter part of the retrieved strings s_r with replacing representatives by input sub strings include the output strings s_o.

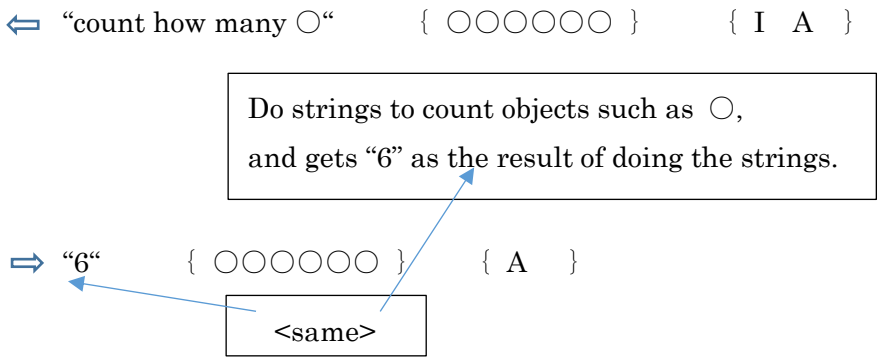


Figure 3. An example of consistency. Our program gives strings “count how may ○“ { ○○○○○○ } to program A, and gets “6” output by A. While the program gets “6” made by A, the program conducts strings followed by “count how many ○“, and gets “6”. Strings made by A match strings made by our program.

If the retrieved strings s_r and the output strings s_o further match strings s_c of a regularity with replacing sub strings by their representatives, the program retrieves the strings s_c and tries to achieve consistency among the strings s_r previously retriecved, the outputs s_o and the strings s_c retrieved now.

The program gets input strings s_i, retrieves strings s_r the former part of which match the inputs s_i with replacing sub strings by their representatives. Strings s_r is a representative of a set S whose member is the strings s_i. The program makes strings s_o as the retrieved strings s_r and the input s_i specify, and outputs the strings s_o. When the program has another program do the same as just described, namely, has the other program get s_i and output s_o, it outputs strings to have the other begin to get the strings

s_i. After the output, the program gets input strings s_{oa} made by the other, and sees if s_{oa} is the same as s_o. If they are the same, the program does nothing. Otherwise, namely inconsistency occurs, the program keeps the occurrances of the inconsistency. The program further has another program get the strings s_i as well as strings that are members of the set S and tries to form a regularity that includes cases where the program and the other do the same with the strings of S and cases where they do not do the same.

2.3 Forming Hierarchical Regularities

The program forms a regularity R_L and then finds there are cases where the regularity R_L does not hold. The program stores the cases in its memory. The program tries to form a new regularity R_H describing when the regularity R_L holds and when the regularity R_L does not hold. After that, the program tries to find a way from cases where R_L does not hold to cases R_L holds when the program has had cases where R_L does not hold and has a string to have the program do R_L. Or the program tries to find a way from cases where R_L holds to cases where R_L does not hold when the program has had cases where R_L holds and has a string to have the program do R_L does not hold. If it finds the way, the program tries to bind a new string with quotations to the way found.

Strings to specify that Given Strings are not in Any Set Already Created

The program gets a string v and applies F to it (the string turns out to be the former part v^f of the input string v). 1) If the application result is the former part of either A_g, B_g, C_g, or one of other representatives, v becomes a member of one of the sets whose representative is either A_g, B_g, C_g, or one of the others. Then the program applies the reverse of F to one of $A_g{}^l$, $B_g{}^l$, $C_g{}^l$, and the others, and the former part v^f of v, and gets v^l. 2) If the application result is not any of the former part of A_g, B_g, C_g, or other representatives, but the application result happens to be the same as the result of applying F to w^f, then the program applys T to w and gets V_g, and applys the reverse of F to v^f and V_g to get v^l. (Strings v and w become members of a new set although the new set is not formed yet.) 3) Its result is not the same as the result of applying F to any w. There is no way to get v^l.

The program finds three cases described in the previous paragraph occur several times. Then for the case of 1) the program binds a string with quotations "A_g is a way to do "a_1, a_2, a_3, or the others" " to the string A_g. Similarly it binds a string "B_g is a way to do "b_1,

b_2, b_3, or the others“ “ to the string B_g. 2) The program binds a string with quotations “there is a way to do v” to a sequence of strings; the program applies F to v and w^f to find the results are the same. The program applies T to w and gets V_g, and that the program applys the reverse of F to v^f (here, v^f is v) and V_g to get v^l. 3) The program binds a string with quotations “there is no way to do v” to the sequence of strings that the program finds no string that is the same as a result string of appliying F to v.

2.4 Strings to have Another Program make a New Set

The program has formed a regularity that there is a way to do a string that matches an input string after the application of F to the input string. The program has also found cases where the regularity does not hold; namely, there is not a way to do a string that matches an input string after the application of F to the input string. Then the program tries to find a way to go from cases where the regularity does not hold to cases where the regularity holds; namely, the program tries to form a way of making (or generating) a way to do the string, and to form a new set consisting of the input string and its representative.

The way W_n starts with the following: Given v, apply F to v and find v is not a member of any set already formed. Then the way is to choose w, apply F to w^f and to find the application result is the same as the result of applying F to v. The program sets v and w be members of a new set, and make the result of the application F to w be a representative V_g of the new set. V_g is now a way to do v.

When the program has another program P_a do v and the program P_a fails to do so, the program does the following; it binds strings with quotations to the way W_n so that the program gives the strings with quotations to program P_a, and has the other program P_a do the way W_n. Doing the way W_n, the program P_a chooses w so that the application F to w^f is the same as that to v. It applies T to w to get V_g, and applies the reverse of F to v^f (here, v^f is v) and V_g to get v^l. Getting v^l is to do v (v is actually v^f). Then the program P_a forms a new set consisiting of v and w.

As explained in the above paragraph, strings bound to the way W_n are the strings to have another program, given v and w, make a new set (or make its representative) consisting of v and w.

3. Strings to have another make a way of forming a Regularity

Assume the program has formed regularities described in Section 4, then the program will have another program make a way of forming a regularity. To do so, the program binds strings with quotations to steps of making the way. The strings describe what the program does to make rhe way, namely the strings describe its original capabilities. This section explains how the program does just described.

3.1 Strings that the program has from the beginning

The program has three types of strings from the beginning; the first type consists of representatives of sets, representatives of all the strings, temporal changes of all the strings. The second type consists of strings describing activities on strings such as input, output, focusing, partitions (or punctuations), temporal sequencing of strings, relations such as <bind>, <same>, <this is this>. The third type consists of strings describing through which channel the program gets and puts strings.

Strings “” {} {} show through which channel the program gets input stings.
“” first channel
{} second / third channel
{} forth channel

{ I } The program gets changes through channel 3 at the same time it gets changes through channel 2, or after it gets changes through channel 2.
{ another program } The program gets changes through channel 1 and/or 3 whithout changes through channel 2.

{continue / stop} get strings through channel 4 to specify continuing a current doing or stopping the current doing.

✔ shows on which sub string through channel 2 a focus is placed through channel 3.

⇦ output strings using its motors through channel 3.
⇨ get strings through channel 1 and / or channel 2.

A string ⇔ describes that strings following ⇔ are those retrieved from a memory that matche current strings.

Time sequence
A （next） B　　A occurs first and then B occurs (gets string A first and gets string B).

Strings to show two strings are the same, one before next and the other after next.
（ this is this ）

<bind> describes strings are bound together.

<same> describes two strings are the same. (Namely, the two are the same member of a set.)

A representative and its members are linked by pointers:
(→)　a pointer to a representative
(↓)　a pointer to the next member.
(←)　a representative of its members.

A pair of strings 《 》 describe a work space to keep input strings, retrieved strings, strings describing relations among strings, and output strings.

A head of a link has a representative of its members, and the link keeps its members.

Strings of the following type { (←) (↓) ・ } describe a link head, and ・ is a representative of its members, and also the representative of a set consisting of the members. ・ is a string the program has from the beginning.

A member is linked by (→) (↓) as in the below, and → points to its head. Since the member may become a head of other members, it has multiple links.

{ (←) (↓) (→) (↓) ・ }

Example strings that will be formed are described in the below:

“Find the number”. { (→1) (↓) (←2) (↓) ・ } (next)

“ “ { (→1) (↓) (←2) (↓) (→3) (↓) ・ }

A pointer →1 points to the most general string, →3 points to the number, namely strings before （next）, and describes strings after （next） are a member of the strings before （next）. ←2 points to members that are common to ・.

“Add 15 to 8 makes “
{(→) (↓) (←) (↓) (→) (↓) 15 (→) (↓) (←) (↓) (→) (↓) 8} (next)
“23” { (→) (↓) (←) (↓) (→) (↓) 23}

Strings to describe replacing an input specific number by a representative of all the specific numbers.
“Replace “ “ by a number”. (→) (↓) (←) (↓) (→) (↓) ・ ”・” (next)
(→) (↓) (←) (↓) ・ ”number” .

If the same strings repeat, then the program replaces them by strings , , , . Namely, “～” “～” “～” are replaced by , , , .

3.2 Making methods of solving math problems
-- Forming regularities of solving math problems

This section describes cases where the program forms regularities of solving mathematical problems, and two types of general regularities. While forming the regularities of solving problems, the program forms the two; One type consists of strings common in various problems, and the common goal state of seriese of strings is {specific link ・} ((→) (↓) (←) (↓) (→) (↓) ・ is abbreviated by specific link ・), describing the common goal state is a specific number of a set. This regularity is described in the below. The other consists of pairs of strings common in inputs and in regularities, and is described in the next section.

[
"There ～ number." {representative ・}
, , ,
"Find ～ number." {representative ・} (next) {specific link ・}
, , ,
"～ number is " " ." {specific link ・}
]

Strings, representative ・, in the above abbreviates (→) (↓) (←) (↓) ・.

Figure 4. A regularity of various mathematical problems.

When the program gets three or more serieses of inputs, each of which, the program finds, is an example of a mathematical problem, as shown in 《 》 below, it forms a regularity of solving the mathematical problem. To find if two serieses of inputs are examples of a math problem, the program replaces sub strings in the inputs by their representatives and sees if the strings with the replacements are the same. While the program forms various mathematical problems, it forms a general regularity described as in Figure 4.

《　　{ T　I　}

⇨ “There is one number. Multiply it by 3. Add 5 to the multiplication result is 38. Find the number.”

⇔ [{～}　”there is ～”]　　⇔ [　{　・　}　”there is one number”]

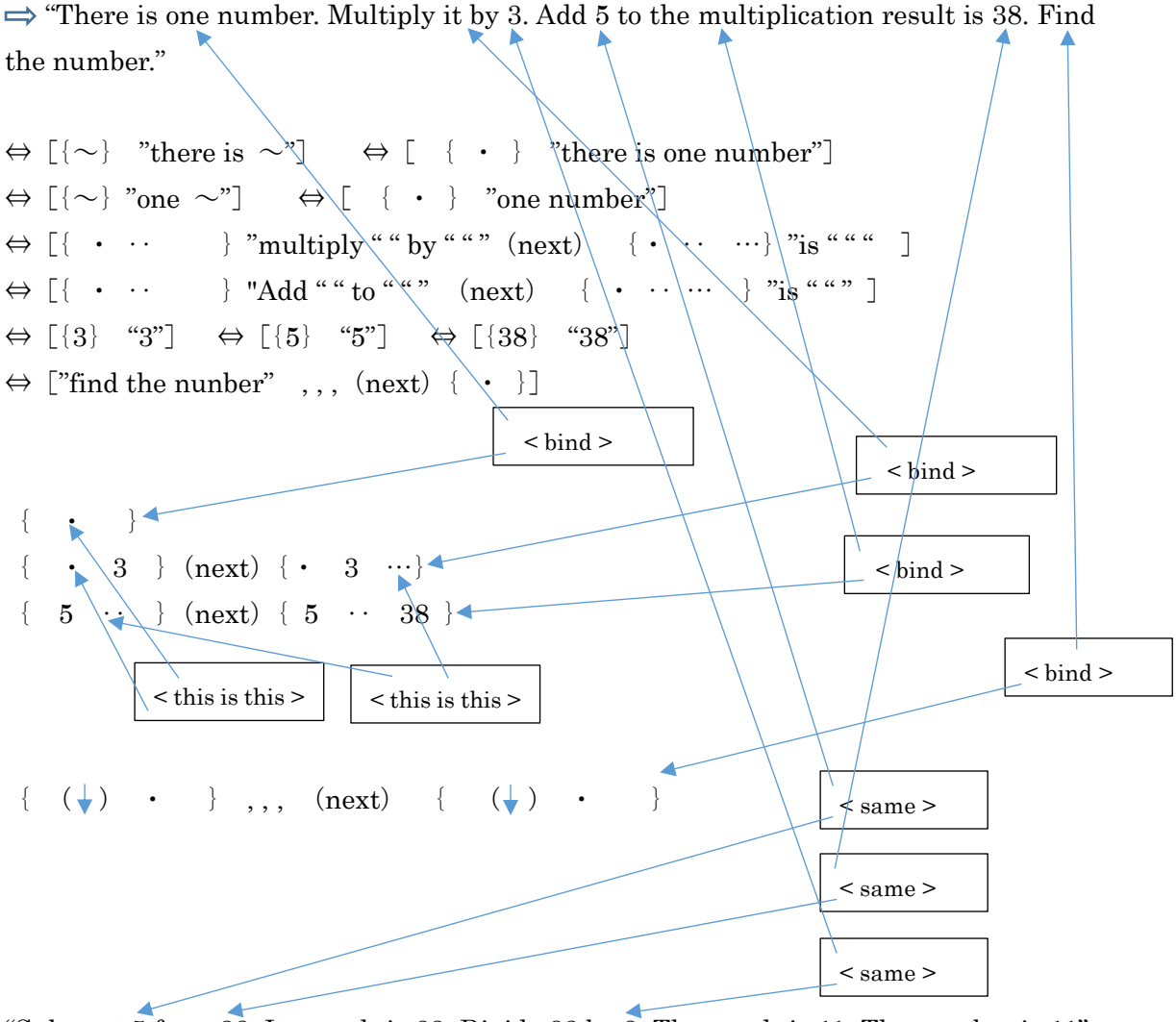
⇔ [{～} ”one ～”]　　⇔ [　{　・　}　”one number”]

⇔ [{　・　‥　　} ”multiply “ “ by “ “ ”　(next)　{ ・　‥　…} ”is “ “ “　]

⇔ [{　・　‥　　} "Add “ “ to “ “ ”　(next)　{　・　‥ …　} ”is “ “ ”　]

⇔ [{3}　“3”]　　⇔ [{5}　“5”]　　⇔ [{38}　“38”]

⇔ [”find the nunber”　, , ,　(next)　{　・　}]



{　・　}

{　・　3　} (next) { ・　3　…}

{　5　‥　} (next) { 5　‥　38 }

{　(↓)　・　}　, , ,　(next)　{　(↓)　・　}

“Subtract 5 from 38. Its result is 33. Divide 33 by 3. The result is 11. The number is 11”.

⇔ [{　・　‥　} ”subtract “ “ from “ “ “　(next) { ・　‥　…} ”is “ “ ”　]

⇔ [“Its result is “ “ “　{　・　}　]

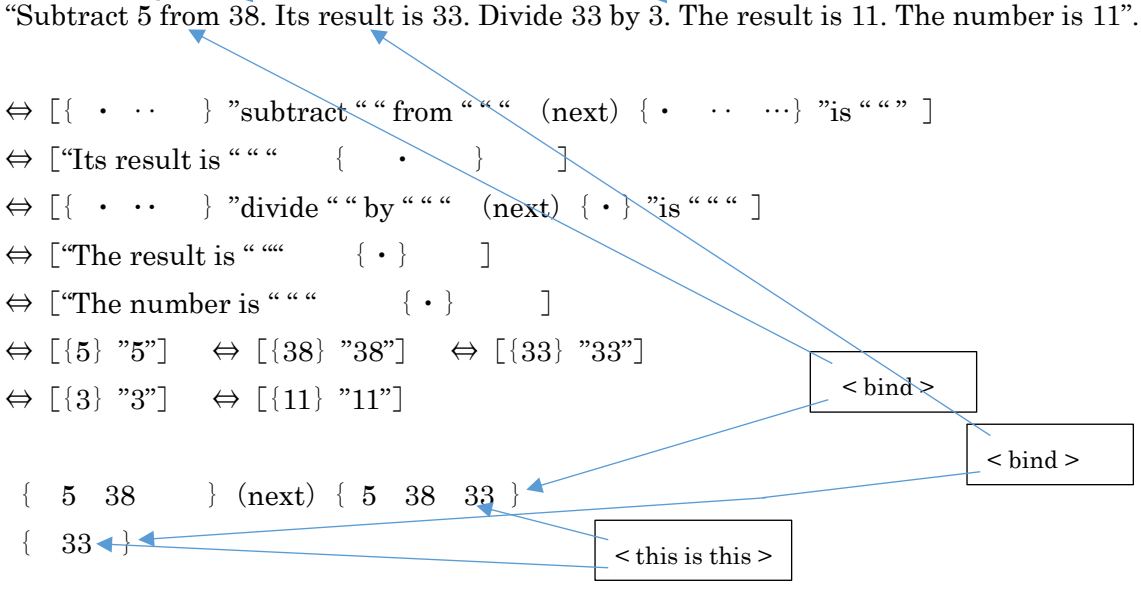
⇔ [{　・　‥　} ”divide “ “ by “ “ “　(next) { ・} ”is “ “ “]

⇔ [“The result is “ ““　{ ・}　]

⇔ [“The number is “ “ “　{ ・}　]

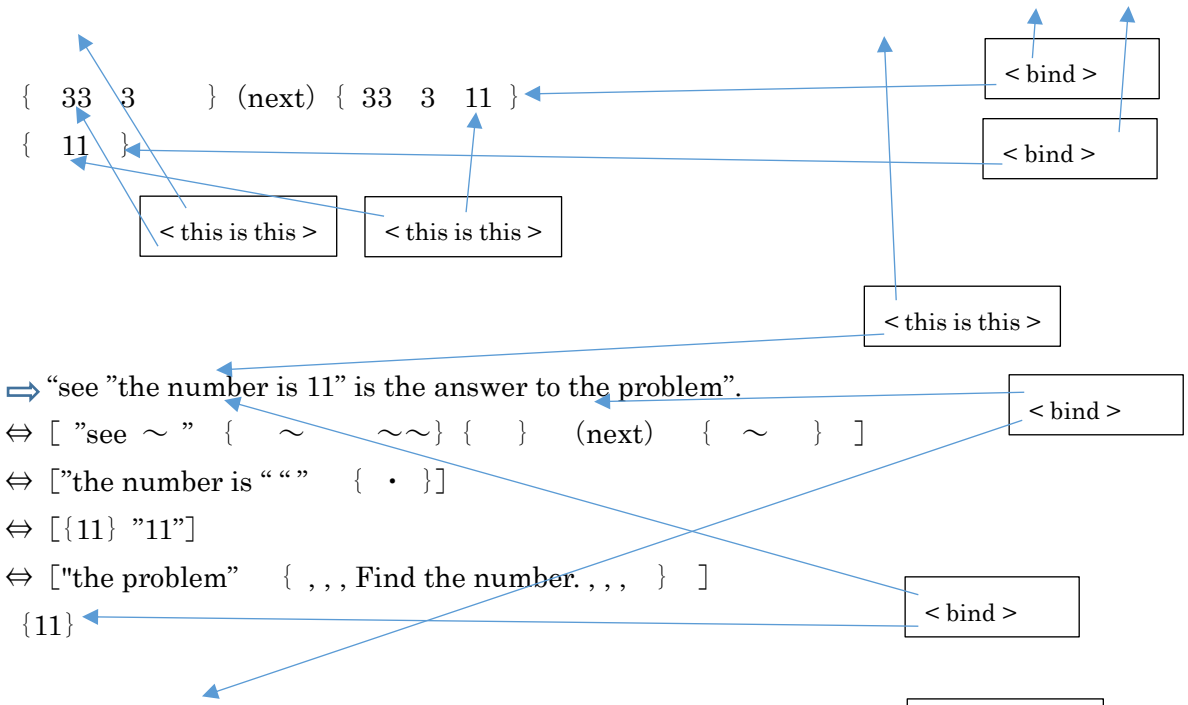
⇔ [{5} ”5”]　⇔ [{38} ”38”]　⇔ [{33} ”33”]

⇔ [{3} ”3”]　⇔ [{11} ”11”]



{　5　38　} (next) { 5　38　33 }

{　33 }



⇨ “see ”the number is 11” is the answer to the problem”.

⇔ [”see ～ ” { ～ ～～} { } (next) { ～ }]

⇔ [”the number is “ “ ” { ・ }]

⇔ [{11} ”11”]

⇔ ["the problem” { , , , Find the number. , , , }]

{11}

{There is one number.} { ・ }

{Multiply it by 3.} { ・ 3 } (next) { ・ 3 …}

{Add 5 to the multiplication result is 38.} {5 ・・ } (next) {5 ・・ 38 }

{Find the number.} , , , (next) { ・ }

⇨ {There is 11.} { 11 }

⇔ ["There is “ “.” { ・ }]

{There is one number.} { ・ }

{ Multiply it by 3.}

{ ・ 3 } (next) { ・ 3 …}

{Add 5 to the multiplication result is 38.} {5 ・・ } (next) {5 ・・ 38 }

{Find the number.} , , , (next) { ・ }

{There is 11.} { 11 }

⇨ { Multiply 11 by 3. }

{11 3 } (next) {11 3 33}

⇔ [”multiply by “ { ・・・} (next) { ・ ・・ … }]

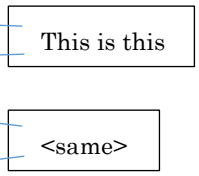
“multiply 11 by 3 ”

{ Multiply it by 3.}

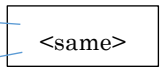
{Add 5 to the multiplication result is 38} {5 ・・ } (next) {5 ・・ 38 }

{Find the number } , , , (next) { ・ }

{There is 11.} { 11 }
{Multiply 11 by 3. } {11 3 } (next) {11 3 33}
⇨ {Add 5 to 33 is 38.}
{5 33 } (next) {5 33 38 }
⇔ [”Add “ “ to “ “ is “ “.” { ・ ‥ } (next) { ・ ‥ …}]
“Add 5 to 33 is 38”
{Add 5 to the multiplication result is 38}
{Find the number} , , , (next) { ・ }



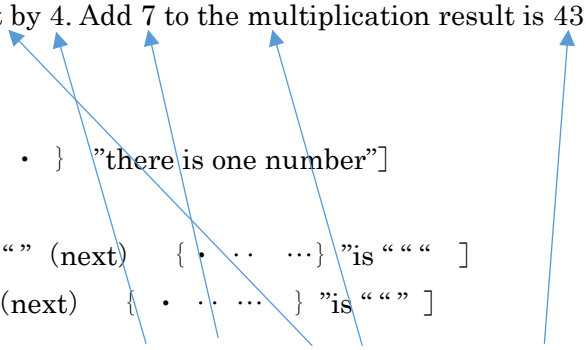
{There is 11.} { 11 }
{ Multiply 11 by 3. } {11 3 } (next) {11 3 33}
{Add 5 to 33 is 38.} {5 33 } (next) {5 33 38 }
⇨ {Find 11.}
{Find the number} , , , (next) { 11 }
⇔ ["Find “ “ “ { ・ } (next) { ・ }]
“Find 11.”
“ “the number is 11” is the answer to the problem.”
// In the above, some relations <same>, <bind>, <this is this> are omitted.
》

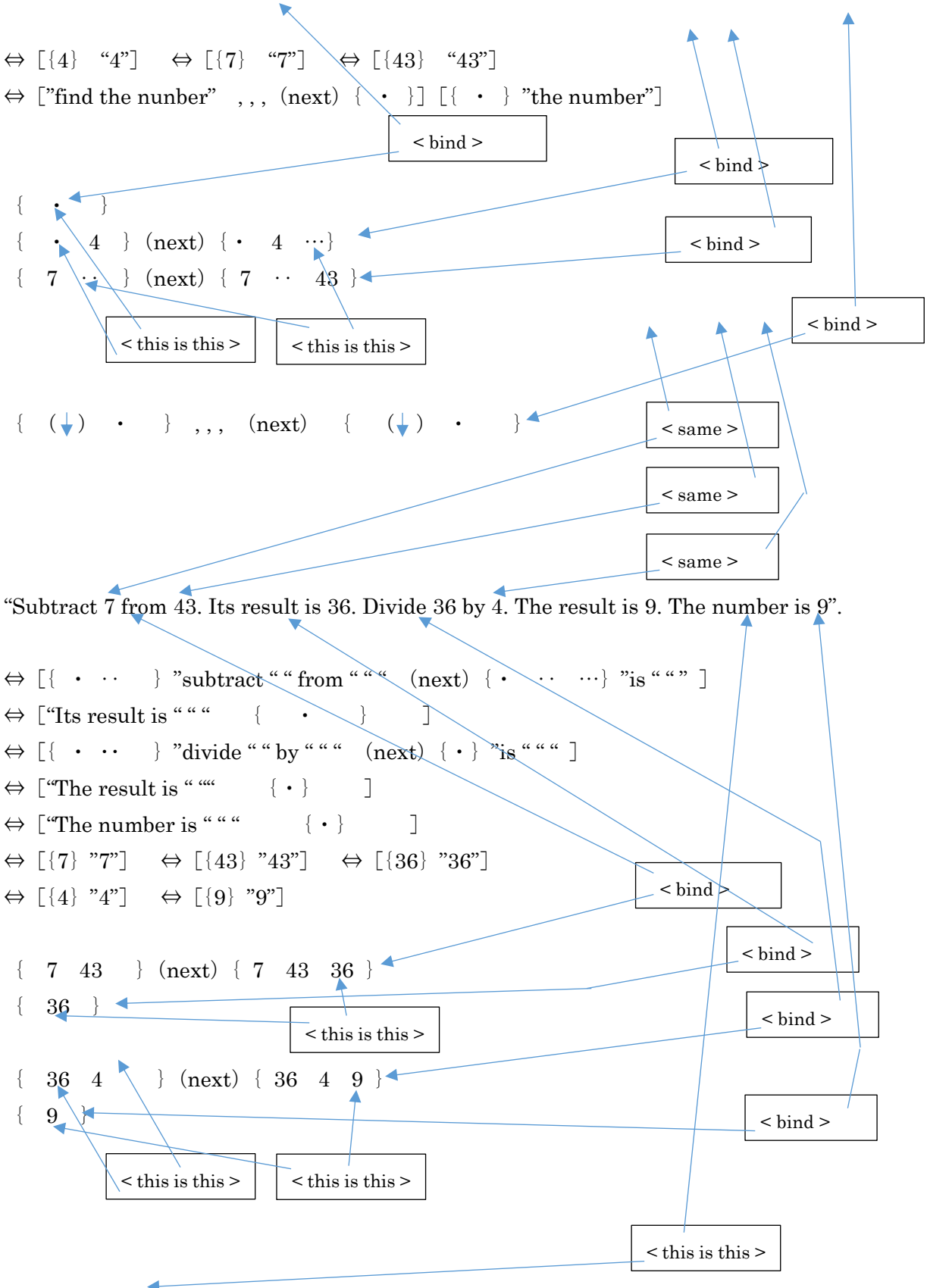


The program gets input strings, and tries to retrieve strings of regularities already formed that match the input with replacing sub strings by their representative strings. It tries to place (write in its work) strings in such a way that the retrieved strings become parallel to the inputs and the strings placed. In the above, strings { ・} are placed which are bound to strings “there is one number.”, and are parallel to the strings retrieved.

《 { T I }
⇨ “There is one number. Multiply it by 4. Add 7 to the multiplication result is 43. Find the number.”

⇔ [{～} ”there is ～”] ⇔ [{ ・ } ”there is one number”]
⇔ [{ ・ } ”one number”]
⇔ [{ ・ ‥ } ”multiply “ “ by “ “ ” (next) { ・ ‥ …} ”is “ “ “]
⇔ [{ ・ ‥ } "Add “ “ to “ “ ” (next) { ・ ‥ … } ”is “ “ ”]




⇔ [{4} “4”] ⇔ [{7} “7”] ⇔ [{43} “43”]
⇔ [”find the nunber” , , , (next) { ・ }] [{ ・ } ”the number”]
< bind >
< bind >
< bind >
{ ・ }
{ ・ 4 } (next) { ・ 4 …}
{ 7 ‥ } (next) { 7 ‥ 43 }
< this is this >
< this is this >
< bind >
{ (↓) ・ } , , , (next) { (↓) ・ }
< same >
< same >
< same >
“Subtract 7 from 43. Its result is 36. Divide 36 by 4. The result is 9. The number is 9”.
⇔ [{ ・ ‥ } ”subtract “ “ from “ “ “ (next) { ・ ‥ …} ”is “ “ ”]
⇔ [“Its result is “ “ “ { ・ }]
⇔ [{ ・ ‥ } ”divide “ “ by “ “ “ (next) { ・} ”is “ “ “]
⇔ [“The result is “ ““ { ・}]
⇔ [“The number is “ “ “ { ・}]
⇔ [{7} ”7”] ⇔ [{43} ”43”] ⇔ [{36} ”36”]
⇔ [{4} ”4”] ⇔ [{9} ”9”]
< bind >
< bind >
{ 7 43 } (next) { 7 43 36 }
{ 36 }
< this is this >
< bind >
{ 36 4 } (next) { 36 4 9 }
{ 9 }
< bind >
< this is this >
< this is this >
< this is this >
“see ”the number is 9” is the answer to the problem”.

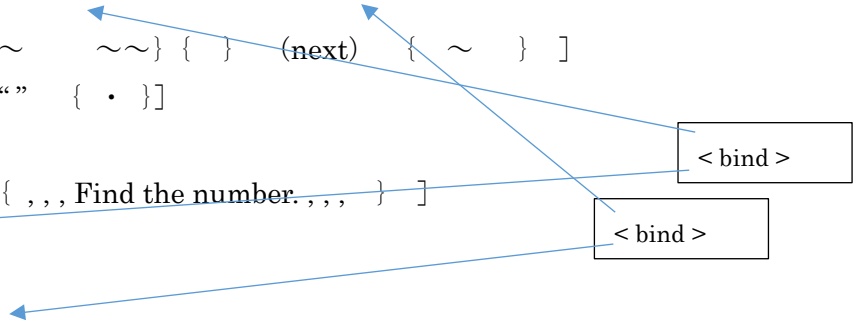
⇔ [”see ～ ” { ～ ～～} { } (next) { ～ }]

⇔ [”the number is “ “ ” { ・ }]

⇔ [{9} ”9”]

⇔ ["the problem” { , , , Find the number. , , , }]

{9}



{There is one number.} { ・ }

{Multiply it by 4.} { ・ 4 } (next) { ・ 4 …}

{Add 5 to the multiplication result is 38.} {7 ‥ } (next) {7 ‥ 43 }

{Find the number.} , , , (next) { ・ }

⇨ {There is 9.} { 9 }

⇔ ["There is “ “.” { ・ }]

{There is one number.} { ・ }

{ Multiply it by 4.}

{ ・ 4 } (next) { ・ 4 …}

{Add 5 to the multiplication result is 38.} {7 ‥ } (next) {7 ‥ 43 }

{Find the number.} , , , (next) { 9 }

{There is 9.} { 9 }

⇨ { Multiply 9 by 4. }

{9 4 } (next) {9 4 36}

⇔ [”multiply by “ { ・‥} (next) { ・ ‥ … }]

“multiply 9 by 4 ”

{ Multiply it by 4.}

{Add 7 to the multiplication result is 43} {7 ‥ } (next) {7 ‥ 43 }

{Find the number } , , , (next) { 9 }

{There is 9.} { 9 }

{Multiply 9 by 4. } {9 4 } (next) {9 4 36}

⇨ {Add 7 to 36 is 43.}

{7 36 } (next) {7 36 43 }

⇔ [”Add “ “ to “ “ is “ “.” { ・ ‥ } (next) { ・ ‥ …}]

“Add 7 to 36 is 43”

{Add 7 to the multiplication result is 43}

{Find the number} , , , (next) { 9 }

{There is 9.} { 9 }
{ Multiply 9 by 4. } {9 4 } (next) {9 4 36}
{Add 7 to 36 is 43.} {7 36 } (next) {7 36 43 }
⇨ {Find 9.}
{Find the number} , , , (next) { 9 }
⇔ ["Find " " " { • } (next) { • }]
"Find 9."
" "the number is 9" is the answer to the problem."

》

The program finds relations <same> among inputs that are replaced by their representatives and <bind> to specify that strings through different channels are bound together. The program gets as inputs the facts that strings at one time t_1 are the same as those at the time following t_1 and adds <this is this> to point to the strings at t_1 and those at t_2.

《　　{ T　I　}

⇨ “There is one number. Multiply it by 6. Add 2 to the multiplication result is 80. Find the number.”

⇔ [{～}　”there is ～”]　　⇔ [　{　・　}　”there is one number”]

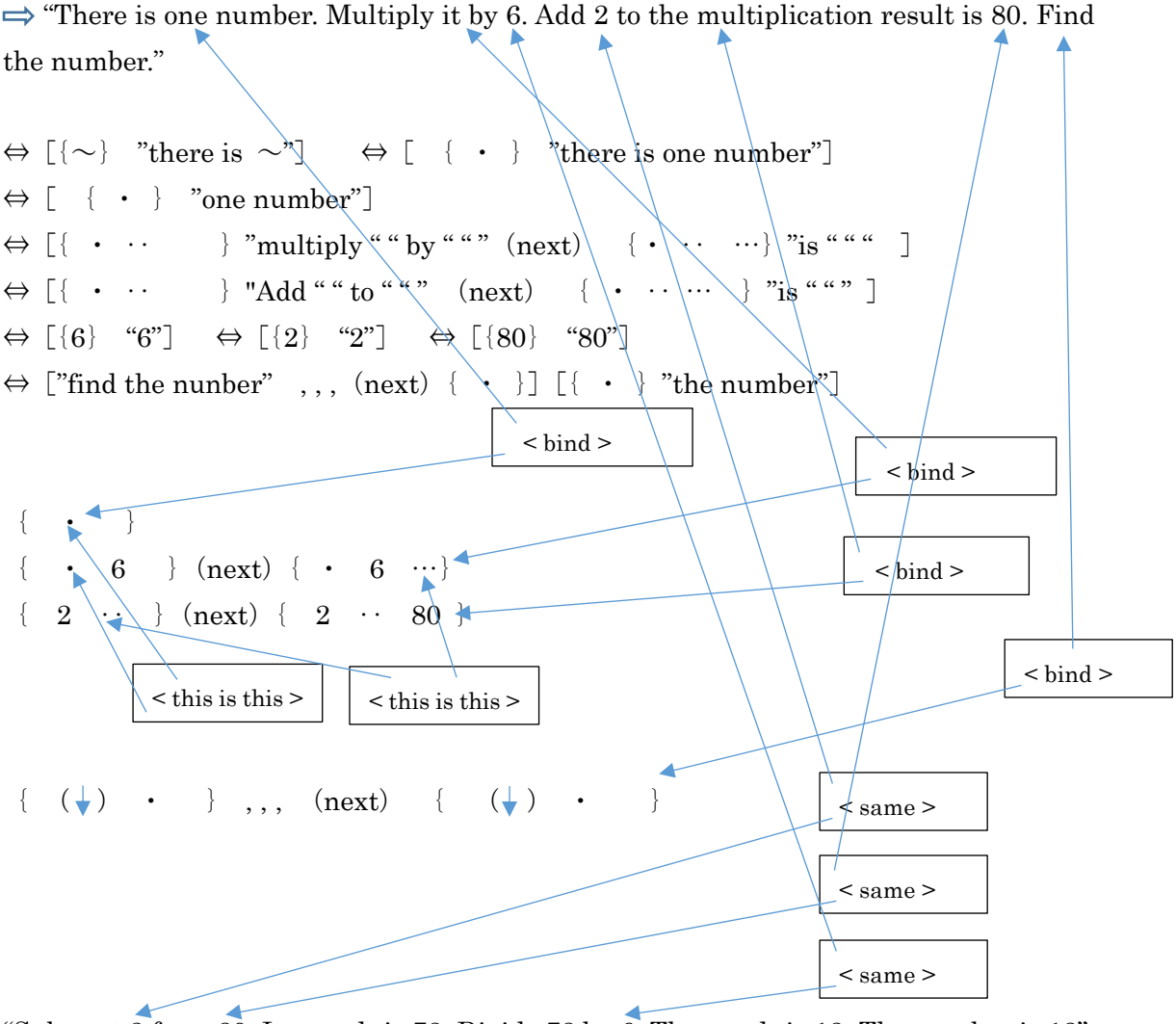
⇔ [　{　・　}　”one number”]

⇔ [{　・　‥　　} ”multiply “ “ by “ “ ”　(next)　{ ・　‥　…} ”is “ “ “　]

⇔ [{　・　‥　　} "Add “ “ to “ “ ”　(next)　{　・　‥ …　} ”is “ “ ”　]

⇔ [{6}　“6”]　　⇔ [{2}　“2”]　　⇔ [{80}　“80”]

⇔ [”find the nunber”　, , ,　(next)　{　・　}] [{　・　} ”the number”]



“Subtract 2 from 80. Its result is 78. Divide 78 by 6. The result is 13. The number is 13”.

⇔ [{　・　‥　} ”subtract “ “ from “ “ “　(next) { ・　‥　…} ”is “ “ ”　]

⇔ [“Its result is “ “ “　{　・　}　]

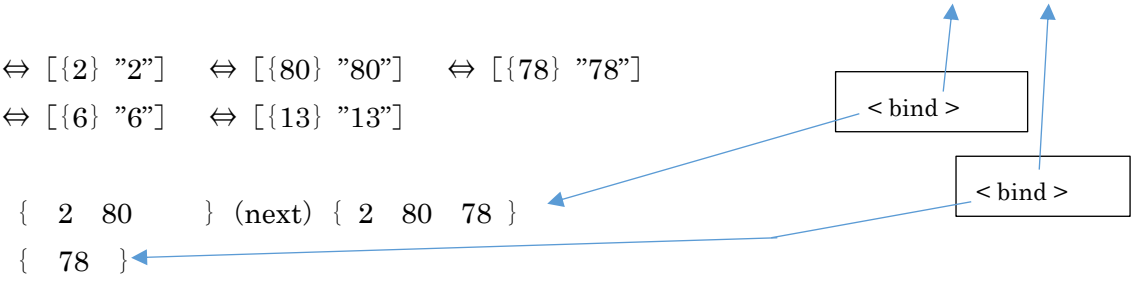
⇔ [{　・　‥　} ”divide “ “ by “ “ “　(next) { ・} ”is “ “ “　]

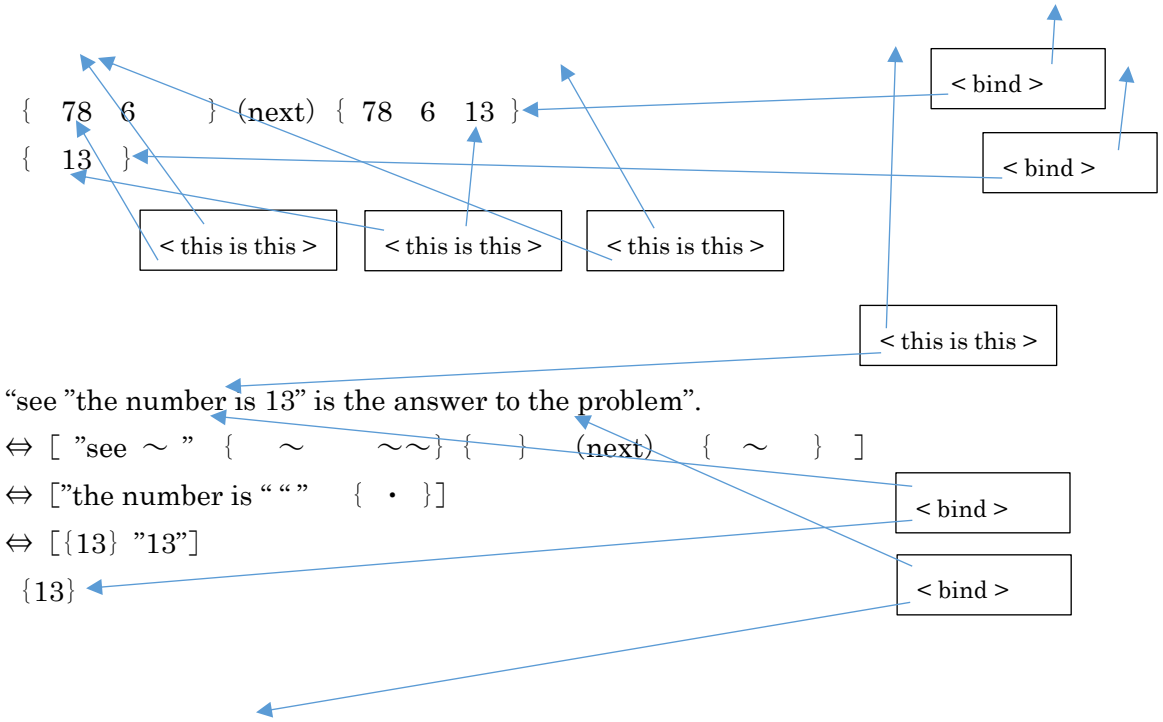
⇔ [“The result is “ ““　{ ・}　]

⇔ [“The number is “ “ “　{ ・}　]

⇔ [{2} ”2”]　　⇔ [{80} ”80”]　　⇔ [{78} ”78”]

⇔ [{6} ”6”]　　⇔ [{13} ”13”]





“see ”the number is 13” is the answer to the problem”.

⇔ [”see ～ ” { ～ ～～} { } (next) { ～ }]

⇔ [”the number is “ “ ” { ・ }]

⇔ [{13} ”13”]

{13}

{There is one number.} { ・ }

{Multiply it by 6.} { ・ 6 } (next) { ・ 6 …}

{Add 2 to the multiplication result is 80.} {2 ‥ } (next) {2 ‥ 80 }

{Find the number.} , , , (next) { ・ }

⇨ {There is 13.} { 13 }

⇔ ["There is “ “.” { ・ }]

{There is one number.} { ・ }

{ Multiply it by 6.}

{ ・ 6 } (next) { ・ 6 …}

{Add 2 to the multiplication result is 80.} {2 ‥ } (next) {2 ‥ 80 }

{Find the number.} , , , (next) { 13 }

{There is 13.} { 13 }

⇨ { Multiply 13 by 6. }

{13 6 } (next) {13 6 78}

⇔ [”multiply by “ { ・‥} (next) { ・ ‥ … }]

“multiply 13 by 6 ”

{ Multiply it by 6.}

{Add 2 to the multiplication result is 80} {2 ‥ } (next) {2 ‥ 80 }

{Find the number } , , , (next) { 13 }

{There is 13.} { 13 }

{Multiply 13 by 6. } {13 6 } (next) {13 6 78}

⇨ {Add 2 to 78 is 80.}

{2 78 } (next) {2 78 80 }

⇔ ["Add " " to " " is " "." { • •• } (next) { • •• …}]

"Add 2 to 78 is 80"

{Add 2 to the multiplication result is 80}

{Find the number} , , , (next) { 13 }

{There is 13.} { 13 }

{ Multiply 13 by 6. } {13 6 } (next) {13 6 78}

{Add 2 to 78 is 80.} {2 78 } (next) {2 78 80 }

⇨ {Find 13.}

{Find the number} , , , (next) { 13 }

⇔ ["Find " " " { • } (next) { • }]

"Find 13."

" "the number is 13" is the answer to the problem."

》

The program, given examples, forms a regularity, namely a problem and a way to solve it out of the examples. The below shows the regularity just formed.

(←) (↓) (→) (↓)
[{ I }
⇨ “There is one number. Multiply it by “ “. Add “ “ to the multiplication result is “ “. Find the number.”

⇔ [{～} ”there is ～”] ⇔ [{ • } ”there is one number”]
⇔ [{ • } ”one number”]
⇔ [{ • ‥ } ”multiply “ “ by “ “ ” (next) { • ‥ …} ”is “ “ “]
⇔ [{ • ‥ } "Add “ “ to “ “ ” (next) { • ‥ … } ”is “ “ ”]
⇔ [{ • } “ ”] ⇔ [{ • } “ ”] ⇔ [{ • } “ ”]
⇔ [”find the nunber” , , , (next) { • }] [{ • } ”the number”]

< bind >
< bind >
< bind >
< bind >
{ • }
{ • ‥ } (next) { • ‥ …}
{ • ‥ } (next) { • ‥ … }
< this is this >
< this is this >
{ (↓) • } , , , (next) { (↓) • }
< same >
< same >
< same >

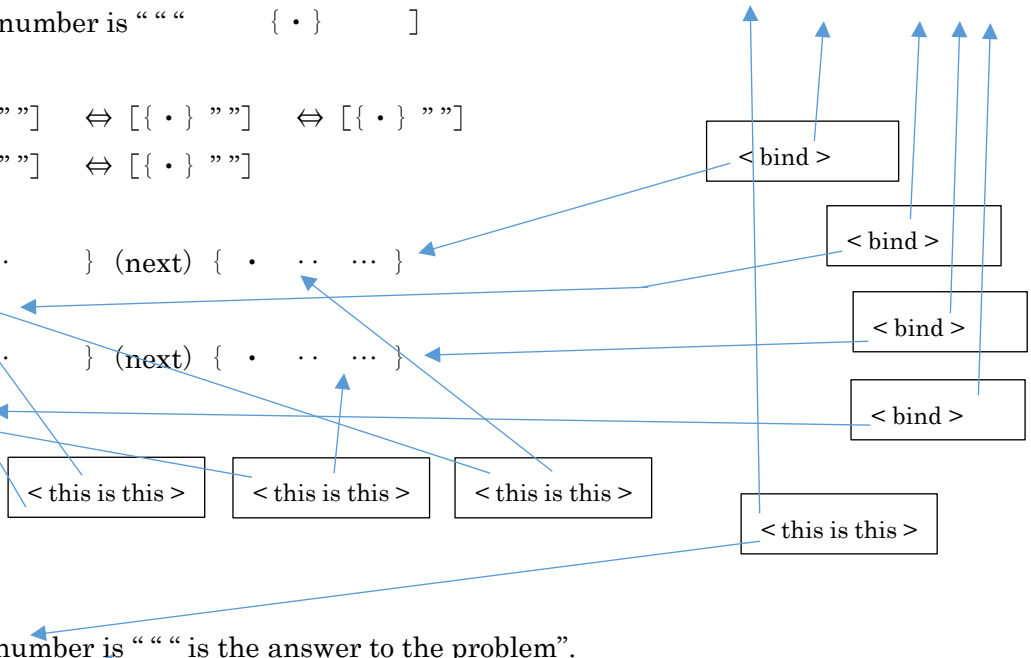
“Subtract “ “ from “ “. Its result is “ “. Divide “ “ by “ “. The result is “ “. The number is “ “ ”.

⇔ [{ • ‥ } ”subtract “ “ from “ “ “ (next) { • ‥ …} ”is “ “ ”]
⇔ [“Its result is “ “ “ { • }]
⇔ [{ • ‥ } ”divide “ “ by “ “ “ (next) { • } ”is “ “ “]
⇔ [“The result is “ ““ { • }]

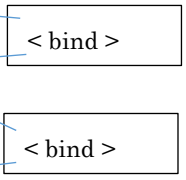
⇔ [“The number is “ “ “　　{ ・ }　　]

⇔ [{ ・ } ” ”]　⇔ [{ ・ } ” ”]　⇔ [{ ・ } ” ”]
⇔ [{ ・ } ” ”]　⇔ [{ ・ } ” ”]

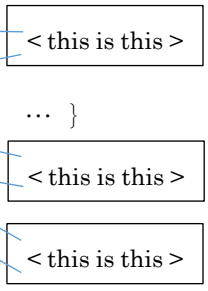
{　・　‥　　} (next) { ・　‥　… }
{　・ }
{　・　‥　　} (next) { ・　‥　… }
{　・ }



“see ”the number is “ “ “ is the answer to the problem”.
⇔ [”see ～ ”　{　～　～～} {　}　(next)　{　～　}　]
⇔ [”the number is “ “ ”　{ ・ }]
⇔ [{ ・ } ” ”]
{ ・ }



{There is one number.} { ・ }
{Multiply it by ‥.} { ・ ‥ } (next) { ・ ‥ …}
{Add ・ to the multiplication result is ….} { ・ ‥ } (next) { ・ ‥ … }
{Find the number.}　, , , (next) { ・ }



⇨ {There is ・.} { ・ }
⇔ ["There is “ “.”　{ ・ }　]
{There is one number.} { ・ }
{ Multiply it by ‥.}
{ ・　‥　} (next) { ・　‥　…}
{Add ・ to the multiplication result is ….} { ・　‥　} (next) { ・　‥　… }
{Find the number.}　, , ,　(next) { ・ }

{There is ・.} { ・ }
⇨ { Multiply ・ by ‥. }
{ ・　‥　} (next) { ・　‥　…}

⇔ [”multiply by “ { • ··} (next) { • ·· ··· }]

“multiply “ “ by “ “ ”

{ Multiply it by ··.}

{Add • to the multiplication result is ···} { • ·· } (next) { • ·· ··· }

{Find the number } , , , (next) { • }

{There is •.} { • }

{Multiply • by ··. } { • ·· } (next) { • ·· ···}

⇨ {Add • to ·· is ···.}

{ • ·· } (next) { • ·· ··· }

⇔ [”Add “ “ to “ “ is “ “.” { • ·· } (next) { • ·· ···}]

“Add “ “ to “ “ is “ “ ”

{Add • to the multiplication result is ···}

{Find the number} , , , (next) { • }

{There is •.} { • }

{ Multiply • by ··. } { • ·· } (next) { • ·· ···}

{Add • to ·· is ···.} { • ·· } (next) { • ·· ··· }

⇨ {Find •.}

{Find the number} , , , (next) { • }

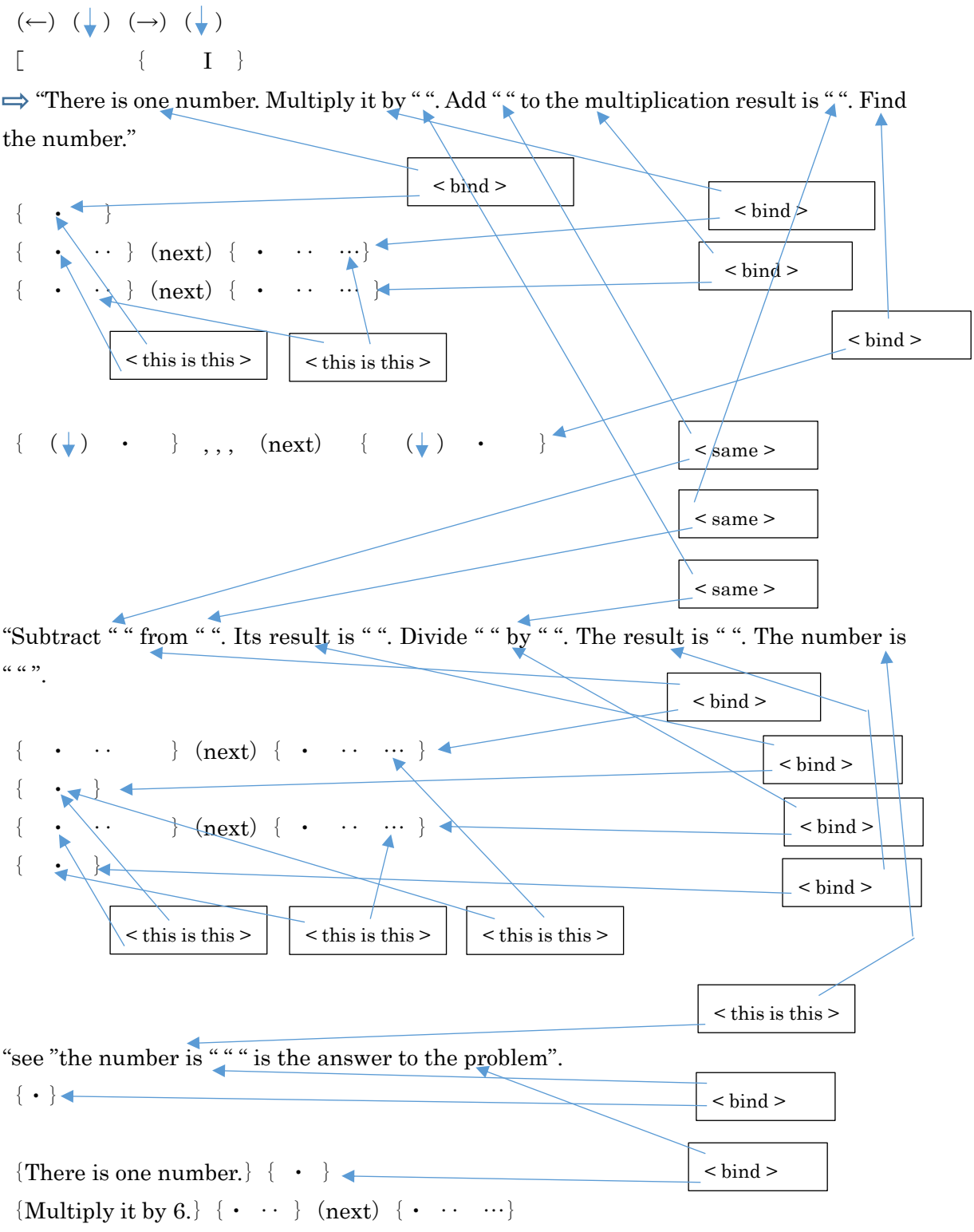
⇔ ["Find “ “ “ { • } (next) { • }]

“Find “ “.”

“ “the number is “ “ ” is the answer to the problem.”

]

It applies the regularity just formed when it gets strings describing a new problem. Then it simplifies the regularity as in the following.



{Add ・ to the multiplication result is ….} { ・ ‥ } (next) { ・ ‥ … }
{Find the number.} , , , (next) { ・ }

⇨ {There is ・} { ・ }
{There is one number.} { ・ }
{ Multiply it by ‥.}
{ ・ ‥ } (next) { ・ ‥ …}
{Add ・ to the multiplication result is ….} { ・ ‥ } (next) { ・ ‥ … }
{Find the number.} , , , (next) { ・ }

{There is ・.} { ・ }
⇨ { Multiply ・ by ‥ }
{ ・ ‥ } (next) { ・ ‥ …}
{ Multiply it by ・.}
{Add ・ to the multiplication result is …} { ・ ‥ } (next) { ・ ‥ … }
{Find the number } , , , (next) { ・ }

{There is ・.} { ・ }
{Multiply ・ by ‥. } { ・ ‥ } (next) { ・ ‥ …}
⇨ {Add ・ to ‥ is …. }
{ ・ ‥ } (next) { ・ ‥ … }
{Add ・ to the multiplication result is …}
{Find the number} , , , (next) { ・ }

{There is ・.} { ・ }
{ Multiply ・ by ‥. } { ・ ‥ } (next) { ・ ‥ …}
{Add ・ to ‥ is ….} { ・ ‥ } (next) { ・ ‥ … }
⇨ {Find ・.}
{Find the number} , , , (next) { ・ }
“ “the number is “ “ ” is the answer to the problem.”

]

<same>

<same>

The program, given a new input, retrieves a regularity that matches the input, and applies the regularity. It replaces representative strings in the retrieved by example strings in the input, and makes strings so that relations <same>, <bind> and <this is this> hold among the strings retrieved with replacements and the strings just made.

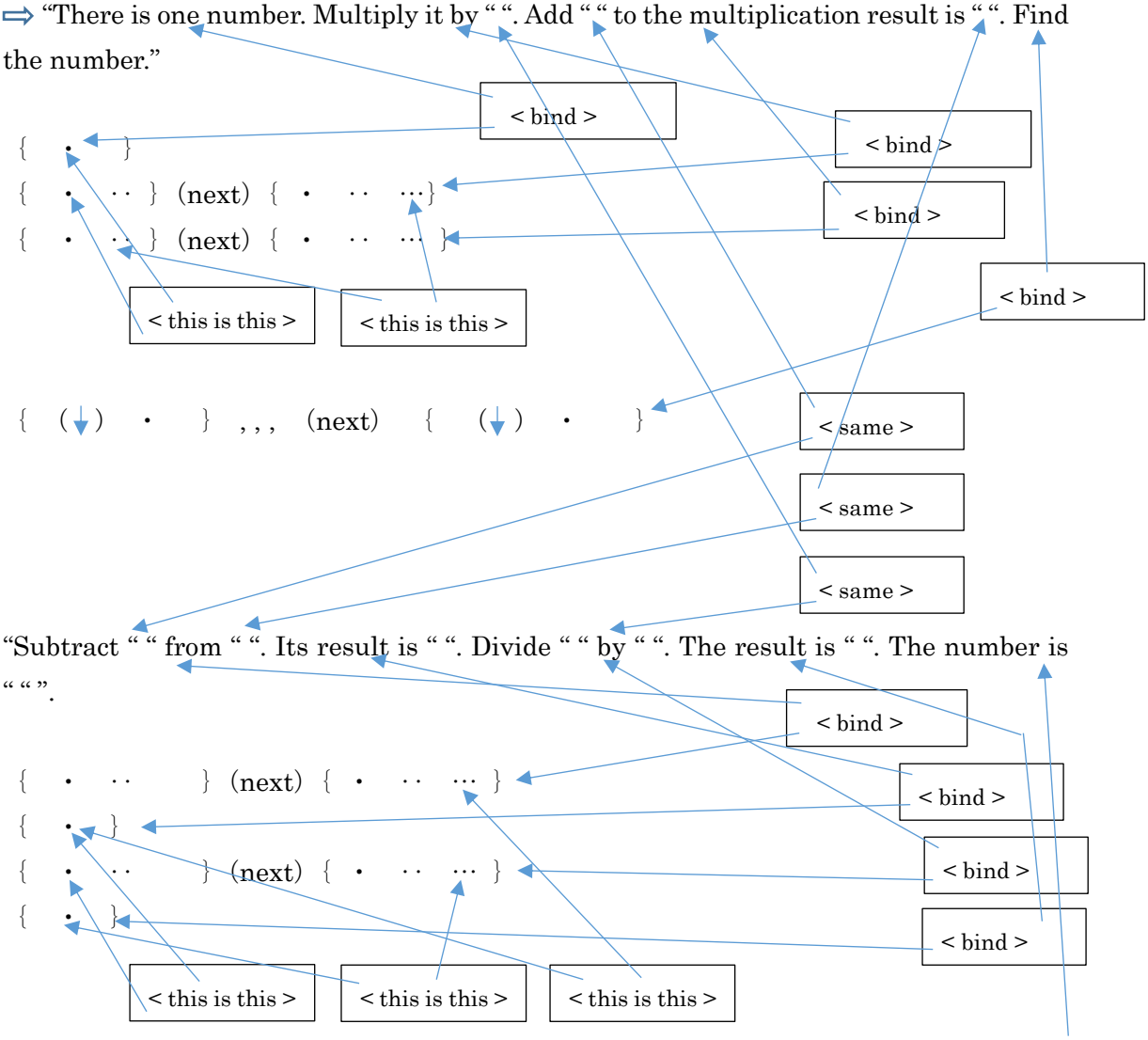
《 { T I }

⇨ “There is one number. Multiply it by 4. Add 11 to the multiplication result is 91. Find the number”

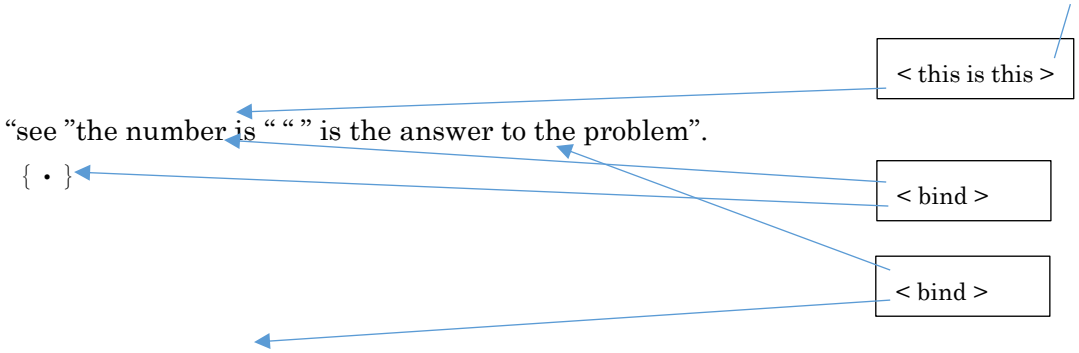
// gets input strings (an example of a math problem).

⇔ [{ I }

⇨ “There is one number. Multiply it by “ “. Add “ “ to the multiplication result is “ “. Find the number.”



“Subtract “ “ from “ “. Its result is “ “. Divide “ “ by “ “. The result is “ “. The number is “ “ ”.



{There is one number.} { • }
{Multiply it by 6.} { • ·· } (next) { • ·· ···}
{Add • to the multiplication result is ···.} { • ·· } (next) { • ·· ··· }
{Find the number.} , , , (next) { • }

⇨ {There is •} { • }
{There is one number.} { • }
{ Multiply it by ··.}
{ • ·· } (next) { • ·· ···}
{Add • to the multiplication result is ···.} { • ·· } (next) { • ·· ··· }
{Find the number.} , , , (next) { • }

{There is •.} { • }
⇨ { Multiply • by ·· }
{ • ·· } (next) { • ·· ···}
<same>
{ Multiply it by •.}
{Add • to the multiplication result is ···} { • ·· } (next) { • ·· ··· }
{Find the number } , , , (next) { • }

{There is •.} { • }
{Multiply • by ··. } { • ·· } (next) { • ·· ···}
⇨ {Add • to ·· is ···. }
<same>
{ • ·· } (next) { • ·· ··· }
{Add • to the multiplication result is ···}
{Find the number} , , , (next) { • }

{There is •.} { • }
{ Multiply • by ··. } { • ·· } (next) { • ·· ···}
{Add • to ·· is ···.} { • ·· } (next) { • ·· ··· }

⇨ {Find ・.}
{Find the number} , , , (next) { ・ }
“ “the number is “ “ ” is the answer to the problem.”

]

// retrieve regularities that match the inputs.
// make strings so that relations <same> <bind> and <this is this>
// hold among the inputs, the strings made, and the retrieved.

{・}
{・ 4 } (next) { ・ 4 …}
{11 ‥ } (next) {11 ‥ 91}

“Subtract 11 from 91. Its result is 80. Divide 80 by 4. The result is 20. The number is 20.”
{11 91 } (next) { 11 91 80}
{80}
{80 4 } (next) {80 4 20}
{20}

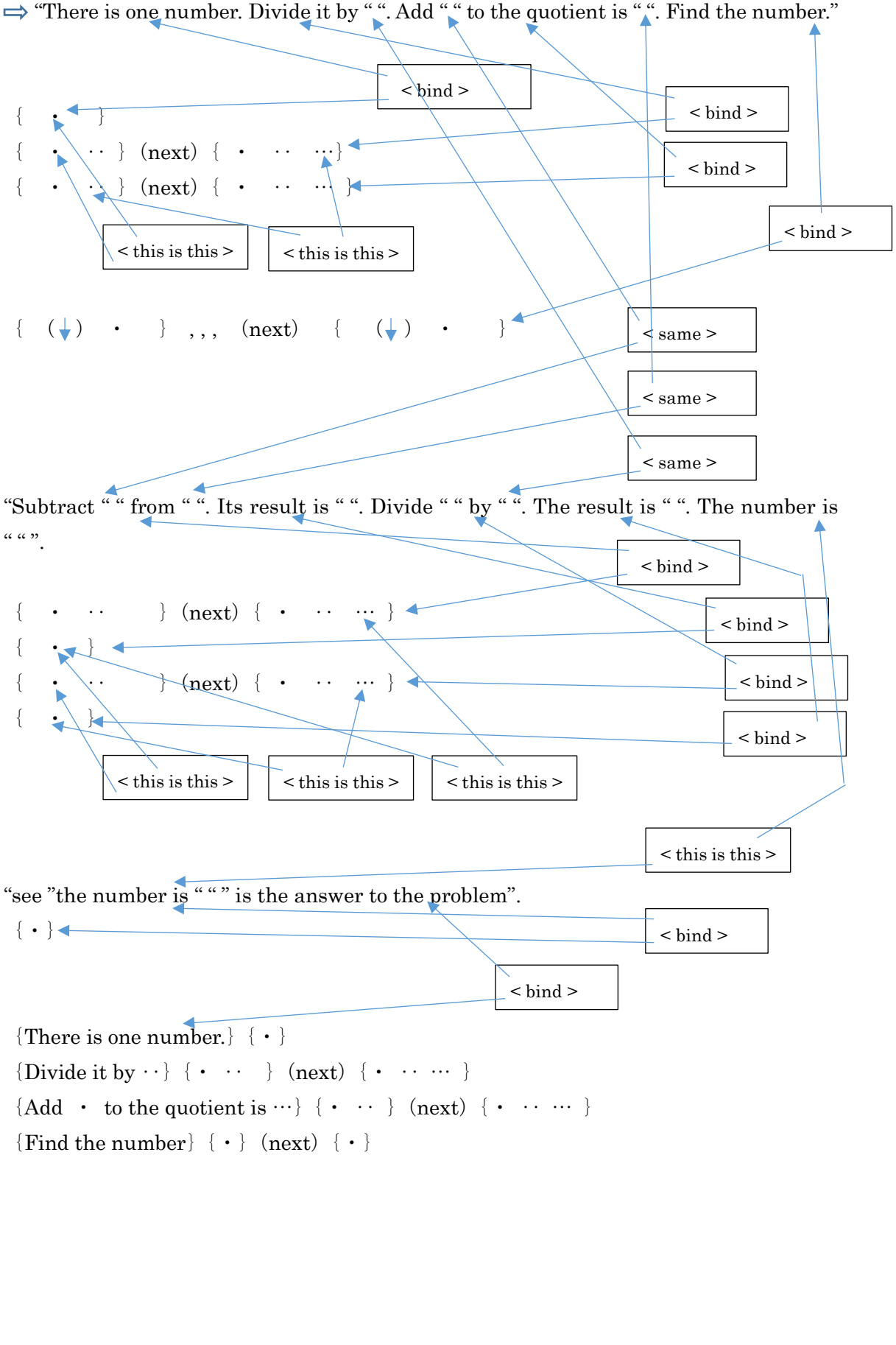
“see ”the number is 20” is the answer to the problem”
{20}

{20}
{20 4 } (next) {20 4 80}
{11 80 } (next) {11 80 91}
{ ・ } (next) {20}
》

The program forms regularities each of which solves each math problem. Example regularities are described in the below.

[(←) (↓) (→) (↓) { I }

⇨ “There is one number. Divide it by “ “. Add “ “ to the quotient is “ “. Find the number.”



{There is ・.} { ・ }
{There is one number.} { ・ }
{Divide it by ‥} { ・ ‥ } (next) { ・ ‥ … }
{Add ・ to the quotient is …} { ・ ‥ } (next) { ・ ‥ … }
{Find the number} { ・ } (next) { ・ }

{There is ・.} { ・ }
{Divide ・ by ‥} { ・ ‥ } (next) { ・ ‥ … }
{Divide it by ‥} { ・ ‥ } (next) { ・ ‥ … }
{Add ・ to the quotient is …} { ・ ‥ } (next) { ・ ‥ … }
{Find the number} { ・ } (next) { ・ }

{There is ・.} { ・ }
{Divide ・ by ‥} { ・ ‥ } (next) { ・ ‥ … }
{Add ・ to ‥ is …} { ・ ‥ } (next) { ・ ‥ … }
{Add ・ to the quotient is …} { ・ ‥ } (next) { ・ ‥ … }
{Find the number} { ・ } (next) { ・ }

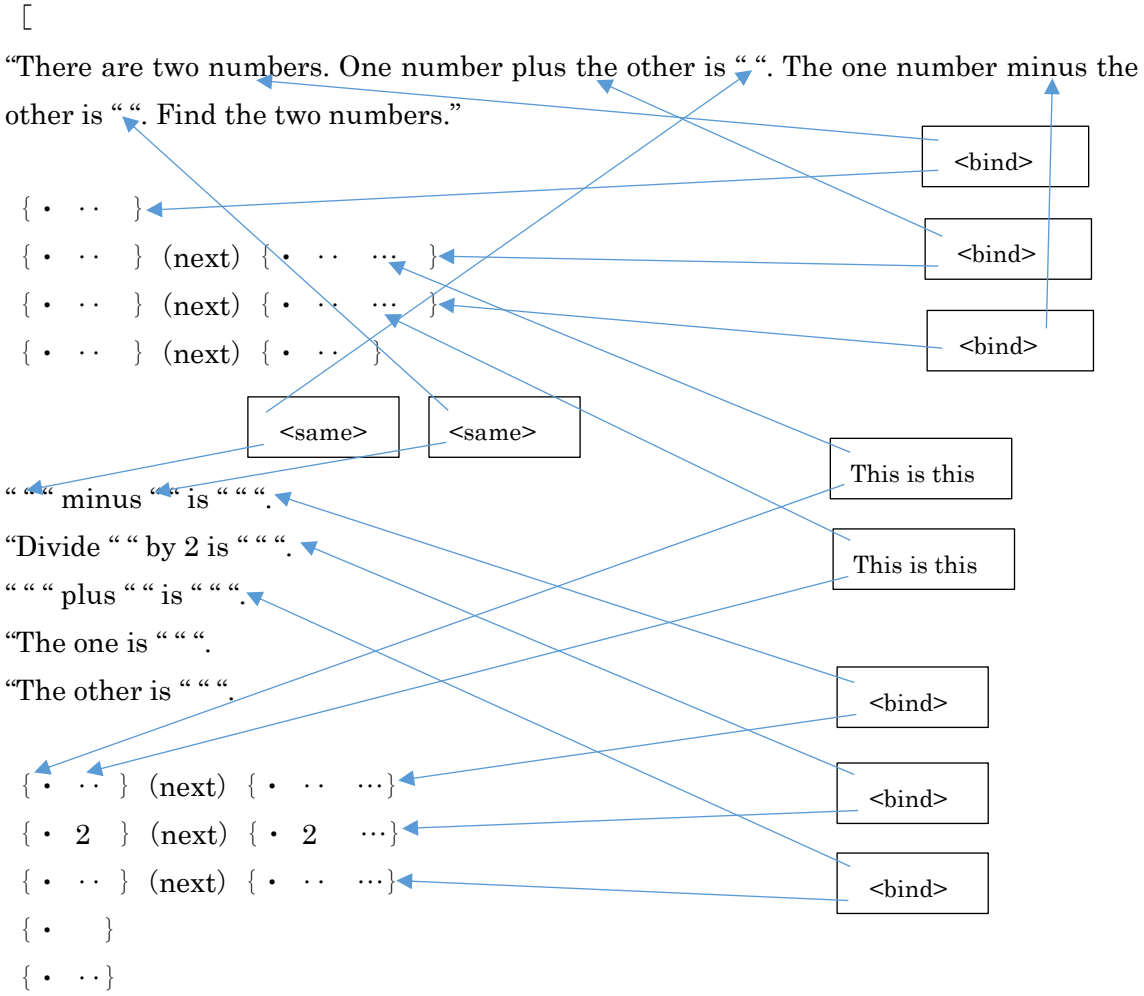
{There is ・.} { ・ }
{Divide ・ by ‥} { ・ ‥ } (next) { ・ ‥ … }
{Add ・ to ‥ is …} { ・ ‥ } (next) { ・ ‥ … }
{Find ・} { ・ } (next) { ・ }
{Find the number} { ・ } (next) { ・ }

< bind >

“ “the number is “ “ “ is the answer to the problem.”

]

The program forms regularities, each one solves one mathematics problem (makes representatives be specific numbers) as described in the following.



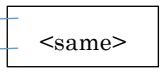
“see “the one is “ “. The other is “ “ ” is the answer to the problem”

{there are two numbers} { • ·· }
{one number plus the other is …} { • ·· } (next) { • ·· …}
{the one number minus the other is …} { • ·· } (next) { • ·· …}
{find the two numbers} { • ·· } (next) { • ·· }

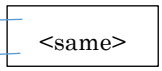
{there are two numbers} { • ·· }
{there are two numbers} { • ·· }
{one number plus the other is …} { • ·· } (next) { • ·· …}

{the one number minus the other is …} { • ‥ } (next) { • ‥ …}
{find the two numbers} { • ‥ } (next) { • ‥ }

{there are two numbers} { • ‥ }
{ • plus ‥ is …}
{one number plus the other is …} { • ‥ } (next) { • ‥ …}
{the one number minus the other is …} { • ‥ } (next) { • ‥ …}
{find the two numbers} { • ‥ } (next) { • ‥ }



{there are two numbers} { • ‥ }
{one number plus the other is …} { • ‥ } (next) { • ‥ …}
{ • minus ‥ is …}
{the one number minus the other is …} { • ‥ } (next) { • ‥ …}
{find the two numbers} { • ‥ } (next) { • ‥ }

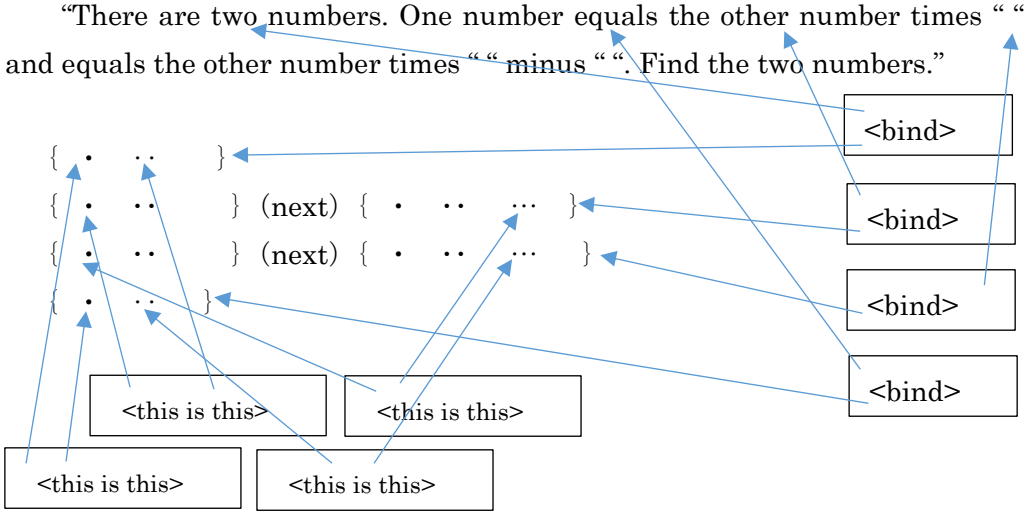


“ “the one is “ “. The other is “ “ ” is the answer to the problem”
]

In the case above, “Divide “ “ by 2” is formed when the program is given three or more examples. Or it is formed after “Divide “ “ by “ “” is formed first and then revised to become “Divide “ “ by 2”. Or it is formed when inputs are given to the program in such a way that the one number and the other become the same, the two become the same, when the difference between the two is subtracted from the one number.

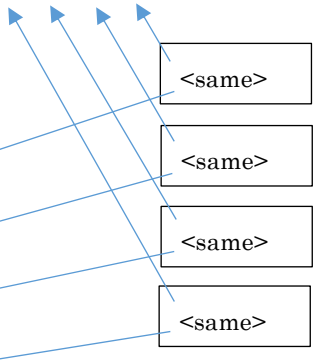
[

“There are two numbers. One number equals the other number times “ “ plus “ “, and equals the other number times “ “ minus “ “. Find the two numbers.”

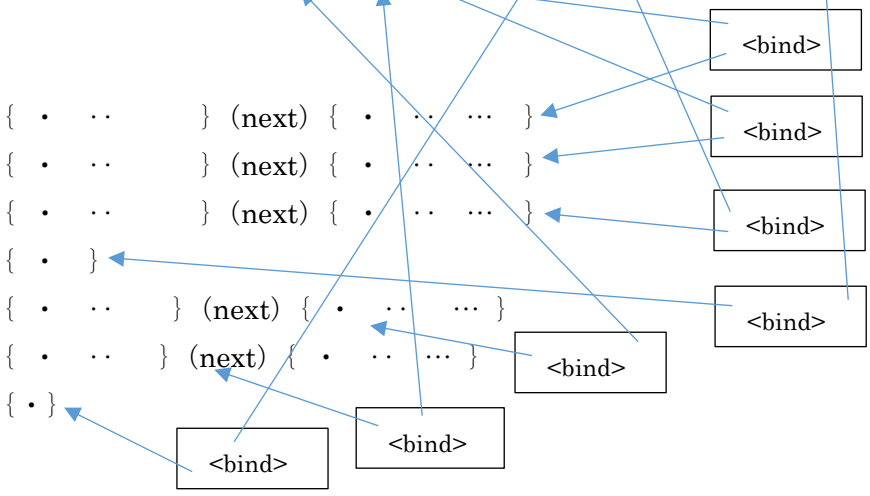


{ • ·· } (next) { • ·· ··· }
{ • ·· } (next) { • ·· ··· }
{ • ·· }
{ • ·· } (next) { • ·· }

// descriptions of relations
// are omitted here.

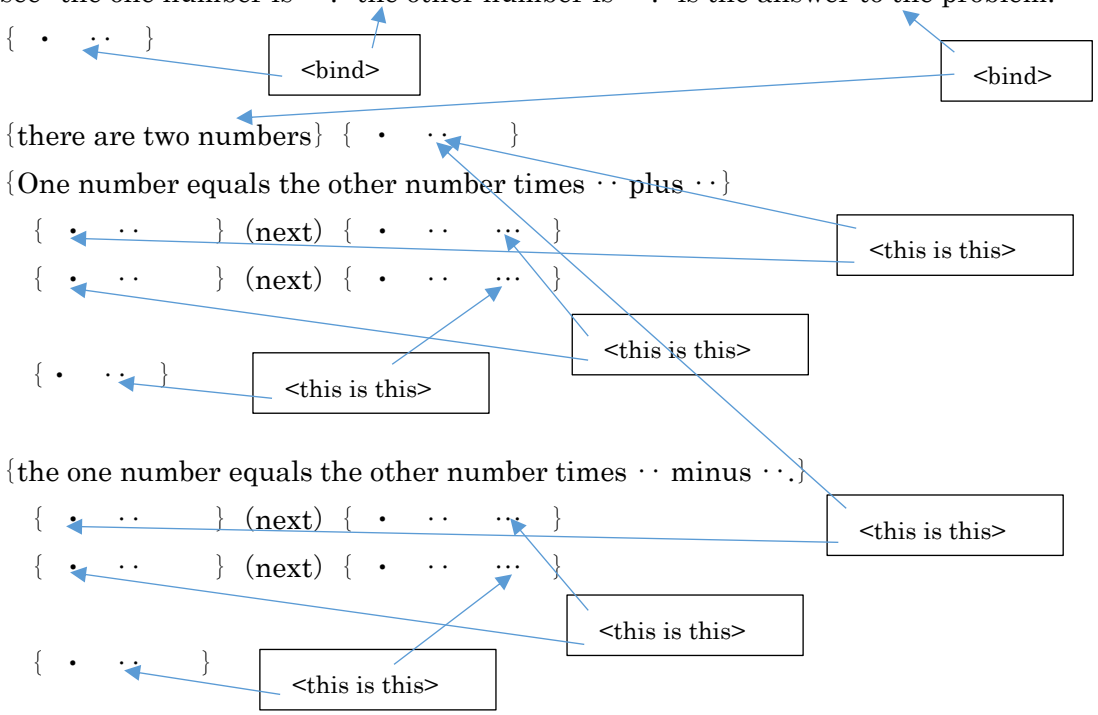


“ “ “ plus “ “ is “ “. “ “ is “ “ greater than “ “. Divide “ “ by “ “ is “ “. The other number is “ “. The one number is “ “ times “ “ plus “ “. The one number is “ “.”



{ • ·· } (next) { • ·· ··· }
{ • ·· } (next) { • ·· ··· }
{ • ·· } (next) { • ·· ··· }
{ • }
{ • ·· } (next) { • ·· ··· }
{ • ·· } (next) { • ·· ··· }
{ • }

“see ”the one number is “ “. ”the other number is “ “.” is the answer to the problem.”



{Find the two numbers.}

{ • ·· } (next) { • ·· }

{there are • ··} { • ·· }

{there are two numbers} { • ·· }

{One number equals the other number times ·· plus ··}

{the one number equals the other number times ·· minus ··.}

{Find the two numbers.}

{there are • ··} { • ·· }

{ • equals ···· times ·· plus ··}

{One number equals the other number times ·· plus ··}

{the one number equals the other number times ·· minus ··.}

{Find the two numbers.}

{there are • ··} { • ·· }

{ • equals ···· times ·· plus ··}

{ • equals ···· times ·· minus ··.}

{the one number equals the other number times ·· minus ··.}

{Find the two numbers.}

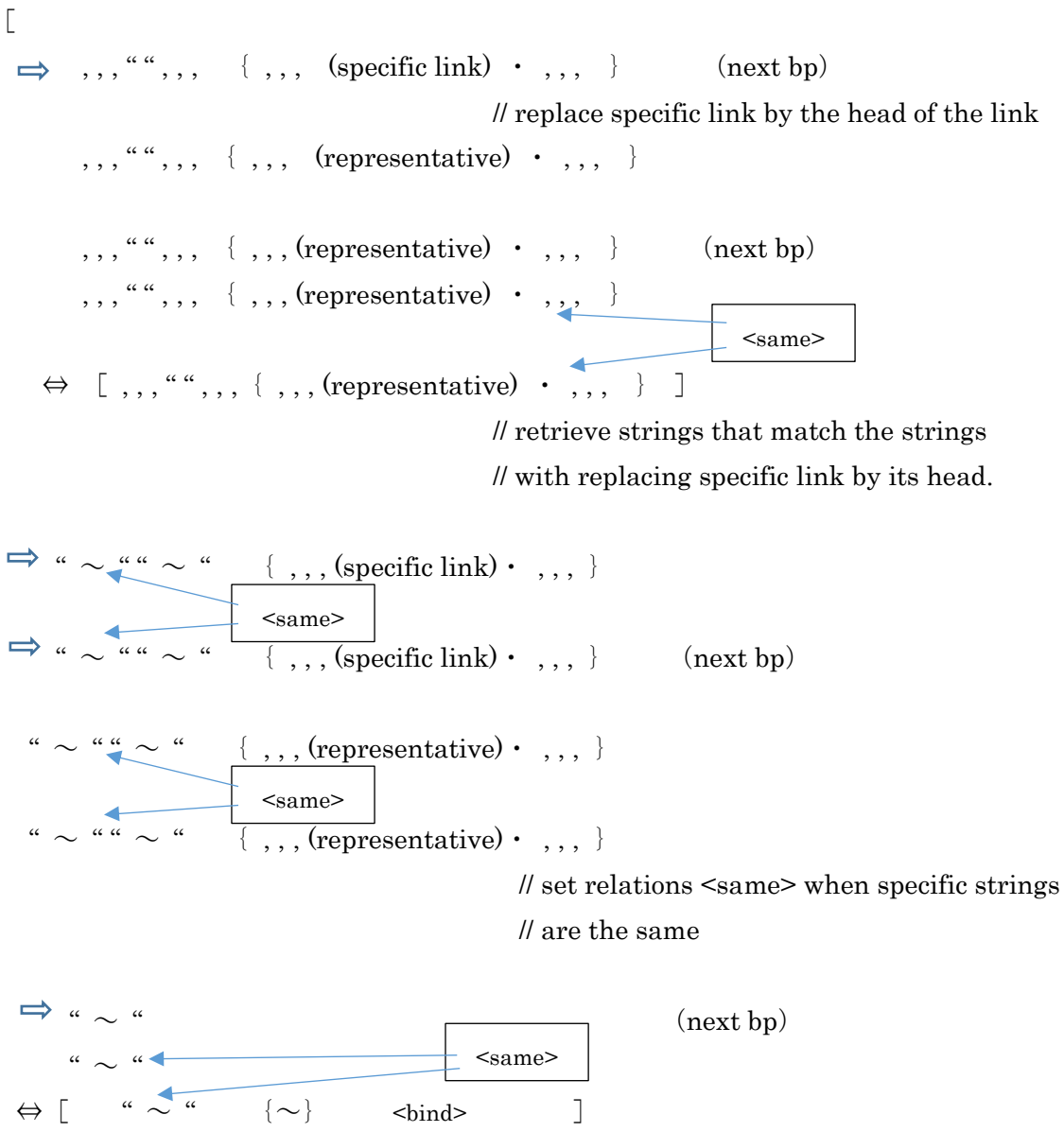
{there are ・ ‥} { ・ ‥ }
{ ・ equals ‥‥ times ‥ plus ‥ }
{ ・ equals ‥‥ times ‥ minus ‥.}
{Find ・ ‥}
{Find the two numbers.}

“ “the one number is “ “. ”the other number is “ “ “ is the answer to the problem“.

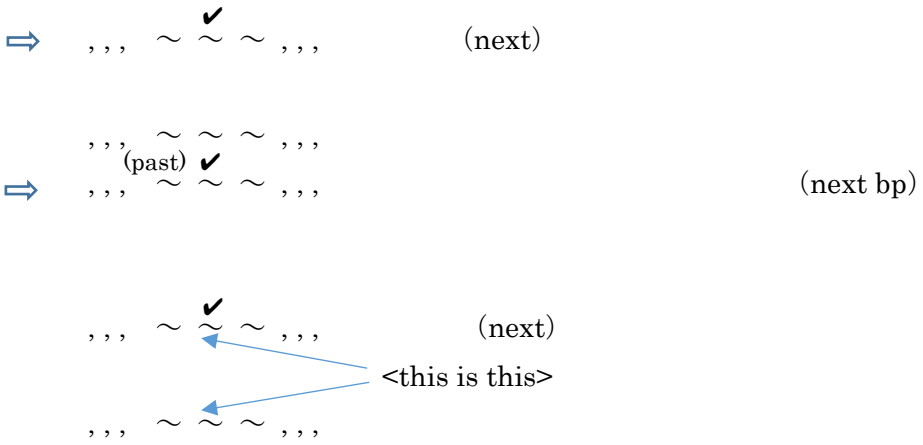
]

3.3 A way of making methods of solving math problems
–A regularity of forming regularities of solving math problems

While it forms more than three regularities of solving math problems, it extracts common strings from pairs of inputs and the regularities formed, or finds common deeds to form the regularities. It can be said that pairs of inputs and the regularities formed are pairs of inputs and outputs and that the common strings describe what the program does to make the outputs. The common strings become a regularity of making a way to solve math problems. The regularity is described in the below.



// find strings with quotations in its memory
// that match the inputs, and retrieve strings
// that are bound to the strings with the quotations



]

While the program applies regularities to solve given math problems, each regularity for each problem, it extracts common strings in a work space. The below is the common strings.

[

⇨ , , , “ “ , , , { , , , (specific link) ・ , , , } (next bp)

// replace specific link by

, , , “ “ , , , { , , , (specific link) ・ , , , }

<same> <same>

⇔ [“ “ { (specific) ・ }]

⇔ [, , , “ “ , , , { , , , (representative) ・ , , , }] (next bp)

, , , “ “ , , , { , , , (representative) ・ , , , }

// the head of the link

, , , “ “ , , , { , , , (representative) ・ , , , } (next bp)

, , , “ “ , , , { , , , (representative) ・ , , , }

<same>

⇔ [, , , “ “ , , , { , , , (representative) ・ , , , } , , ,]

// retrieve strings of a regularity whose first part match

// the strings with replacing a specific link by its head.

⇔ [, , , “ “ , , , { , , , (representative) ・ , , , } (next)

, , , “ “ , , , { , , , (representative) ・ , , , }] (next bp)

<same>

, , , “ “ , , , { , , , (representative) ・ , , , }

// place, in a work space, retrieved strings that follow the first part.

⇨ , , , “ “ , , , (next bp)

, , , “ “ , , ,

<same> <same>

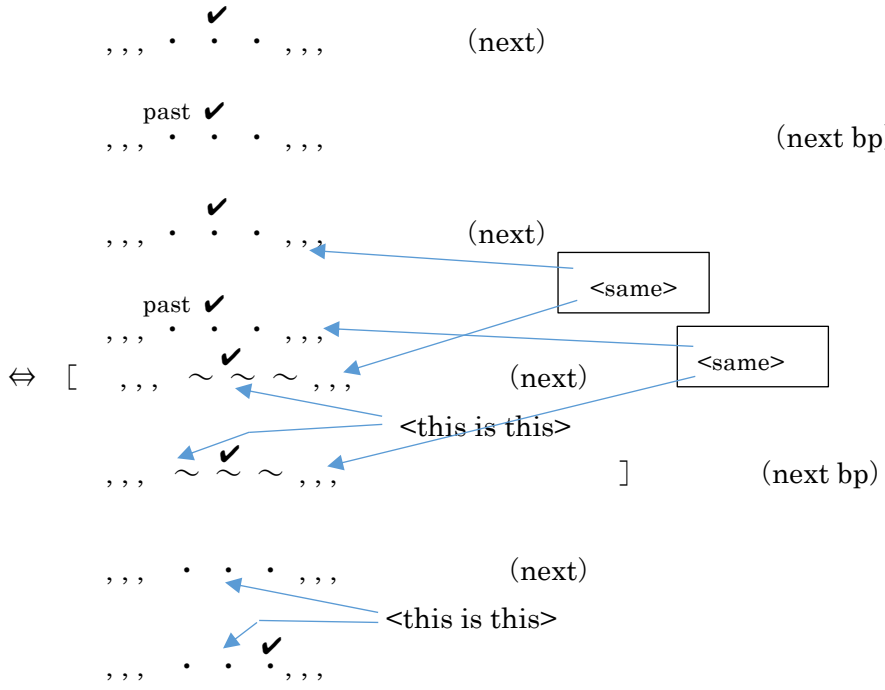
⇔ [“ “ { (specific) ・ }]

⇔ [, , , “ “ , , , { , , , (representative) ・ , , , }] (next bp)

, , , “ “ , , , { , , , (specific) ・ , , , }

// make strings so that strings made are bound to input strings

// so that the binding is consisitent with regularities.



“ ～ “ “ ～ “ { , , , (specific link) ・ , , , }

<same>

“ ～ “ “ ～ “ { , , , (representative) ・ , , , } (next bp)

“ ～ “ “ ～ “ { , , , (specific link) ・ , , , }

<same>

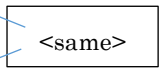
“ ～ “ “ ～ “ { , , , (specific link) ・ , , , }

// make a representative become a specific one

// in such a way that <same> holds.

“ ～ “ “ ～ “ “ “　　{　(specific link) ・　　(specific link) ・　　}
“ “　　{　(representative) ・ }　　(next bp)

“ ～ “ “ ～ “ “ “　　{　(specific link) ・　　(specific link) ・　　}
“ “　　{　(representative) ・ }　　(next bp)



⇔ [“ ～ “ “ ～ “ “ “　　{　(representative) ・　　(representative) ・　　}　　(next)
, , ,　(representative) ・ , , ,　　(next)
“ “　　{　(representative) ・ }
]　　(next bp)

⇔ [“ ～ “ “ ～ “ “ “　　{　(representative) ・　　(representative) ・　　}　　(next)
, , ,　(representative) ・ , , ,　　(next)
“ “　　{　(representative) ・ }
]
, , ,　(representative) ・ , , ,　　(next)
“ “　　{　(representative) ・ }　　(next bp)
// place, in a work space, retrieved strings that follow the first part.

, , ,　(specific link) ・ , , ,　　(next)
“ “　　{　(representative) ・ }　　(next bp)
// conduct strings (do strings) and
// make a representative become a specific one.

, , ,　(specific link) ・ , , ,　　(next)
“ “　　{　(specific link) ・ }

“ ～ “ “ ～ “ “ ～ “ “ “

{ (specific link) ・ (specific link) ・ (representative) ・ }

(next bp)

“ ～ “ “ ～ “ “ ～ “ “ “

{ (specific link) ・ (specific link) ・ (representative) ・ }

<same>

⇔ [“ ～ “ “ ～ “ “ ～ “ “ “

{ (representative) ・ (representative) ・ (representative) ・ } (next)

, , , (representative) ・ , , , (next)

“ ～ “ “ ～ “ “ ～ “ “ “

{ (representative) ・ (representative) ・ (representative) ・ }

] (next bp)

⇔ [“ ～ “ “ ～ “ “ ～ “ “ “

{ (representative) ・ (representative) ・ (representative) ・ } (next)

, , , (representative) ・ , , , (next)

“ ～ “ “ ～ “ “ ～ “ “ “

{ (representative) ・ (representative) ・ (representative) ・ }

]

, , , (representative) ・ , , , (next)

“ ～ “ “ ～ “ “ ～ “ “ “

{ (representative) ・ (representative) ・ (representative) ・ }

// place, in a work space, retrieved strings that follow the first part.

(next bp)

, , , (specific link) ・ , , , (next)

“ ～ “ “ ～ “ “ ～ “ “ “

{ (representative) ・ (representative) ・ (representative) ・ }

// conduct strings (do strings) and

// make a representative become a specific one.

(next bp)

, , , (specific link) ・ , , , (next)

“ ～ “ “ ～ “ “ ～ “ “ “

{ (specific link) ・ (specific link) ・ (specific link) ・ }

```
                    // conduct strings (do strings) and
                    // make a representative become a specific one.

]
```

3.4 Trials of forming a hierarchical regularity
-- A regularity consisting of cases one can do and cases one cannot do

This section describes a case where the program links a regularity describrd in the previous section to ［make a way to do a deed］. When program T gives program I a new math problem and program I does not solve it (cannot reach its goal), program I retrieves regularities already formed and applies them to try reaching the goal. Among its trials of reaching the goal, it retrieves a general regularity that matches the strings of the math problem with replacing ［, , , find , , , number］ by ［a deed］; the regularity includes cases where it can do a deed, where it cannot do the deed, and ［make a way to do a deed］, namely, generates a way to change cases from ［cannot do］ to ［can do］. It applies the regularity to the current case; puts ［, , , find , , , number］ into ［a deed］ but it cannot put any into ［make a way to do a deed］ because a regularity formed in the previous section is not linked to ［make a way to do a deed］.

The program replaces sub strings ｛specific link ・｝ by their representatives ｛representative ・｝ and tries to find strings in its memory that match the inputs with the replaced sub strings. Although the program does not find the strings of a regularity to reach a goal, it finds that the former part of another math problem already got match the inputs with the replaced sub strings. The program then finds that replacing sub strings ｛specific link ・｝ with their representatives ｛representative ・｝ matches replacing sub strings in the inputs by their representatives that are kept in a regularity described in the previous section. It retrieves the regularity and applies it to the example strings that follow and to make the applied strings consistent to the current input strings (namely, make ｛representative ・｝ be ｛specific ・｝ so that ｛specific ・｝ satisfies relations kept in the example. It then finds the goal of the current inputs (the current math problem).

The program then takes the general regularity that has states of ［cannot do］, states ［can do］, and ［make a way to do a deed］, and tries to see links from the states and the change of the states of the general regularity to current states and current change of the states. Link ［cannot solve a current math problem］ to ［cannot do a deed］, ［can solve a current math problem］ to ［can do a deed］, and the regularity formed in the previous section to ［make a way to do a deed］.

《 { T I }

⇨ “There is one number. Multiply it by 3. Add 7 to the multiplication result is 100. Find the number.”

⇔ [“There is one number. Multiply “ “ by “ “. Add “ “ to “ “ is “ “ ”. Find the number.”

, , ,

“The number is “ “.“

“see “the number is “ “ “ is the answer to the problem.”

, , ,

]

{ • }

, , ,

“The number is 31.” { 31 }

, , ,

“see “the number is 31“ is the answer to the problem.”

, , ,

》

《 { T A I }

⇨ “There is one number. Multiply it by 3. Add 7 to the multiplication result is 100. Find the number.”

⇔ [“There is one number. Multiply “ “ by “ “. Add “ “ to “ “ is “ “ ”. Find the number.”

, , ,

]

{ • } { I }

, , , { I }

“The number is 31.” { 31 } { I }

“see “the number is 31“ is the answer to the problem.” { I }
, , , { I }

⇨ { } { A T I }

⇔ [”You cannot do ”～～”. {～～}]

⇨ ”You cannot do “There is one number. Multiply it by 3. Add 7 to the multiplication result is 100. Find the number.” .“ { T A I }

{ ・ } { T A I }
, , , { T A I }

“The number is 31.” { 31 } { T A I }
“see “the number is 31“ is the answer to the problem.” { T A I }
, , , { T A I }

⇨ “There is one number. Multiply it by 3. Add 7 to the multiplication result is 100. Find the number.” { A T I }

⇨ { 31 } { A T I }

⇔ [”You can do ”～～”. {～～}]

⇨ ”You can do “There is one number. Multiply it by 3. Add 7 to the multiplication result is 100. Find the number.” .“ { T A I }

》

[

“Do a deed”

⇔ “deed” { ～～} { I } // initial state

{__～ | } // no way to a goal

“Cannot do a deed“

“say a way to do a deed”

“ “

“ “

“Do a deed”

“deed” { ～～} { I } // goal state

{ ～～} // a way to a goal

{__～ | }

“Can do a deed“

]

《　　{ T　I }

“There are two numbers. Add one number and the other number is 13. Add the one number times 2 and the other number times 4 is 36. Find the two numbers.”

⇔ [{・　‥} ”two numbers”] [{～　～　　} ”there are ～ ～”]
⇔ [{・} ”one number”] [{・　‥} ”the other number”]
⇔ [”Add “ “ and “ “ is “ “.”　{・　‥　} (next) {・　‥　…}]
⇔ [” “ “ times “ “ is “ “.”　{・　‥　} (next) {・　‥　…}]
⇔ ["Find　number .”　{representative　・} (next) {specific link　・}]
⇔ ["There , , , number , , , Find number .”　{representative　・} (next) , , ,　(next) {specific link　・}]

“13 times 2 is 26. Add the one number times 2 and the other number times 2 is 26. 36 minus 26 is 10. Subtract 2 from 4 is 2. Divide 10 by 2 is 5. 13 minus 5 is 8. The one number is 8. The other number is 5.”

⇔ [” “ “ times “ “ is “ “.”　{・　‥　} (next) {・　‥　…}]
⇔ [{・} ”one number”] [{・　‥} ”the other number”]
⇔ [”Add “ “ and “ “ is “ “.”　{・　‥　} (next) {・　‥　…}]
⇔ [" “ “ minus “ “ is “ “.”　{・　‥} (next) {・　‥ …　}]
⇔ ["Divide “ “ by “ “ is “ “.”　{・　‥} (next) {・　‥　…　}]
⇔ [{・} ”one number is “ “.”] [{・　‥} ”the other number is “ “.”]

// replace sub strings by their representatives to see if strings that match are in its
// memory.

// but, strings to reach a goal are not there. Keep the input strings in its memory.

》

《 { T I }

“There are two numbers. Add one number and the other number is 18. Add the one number times 5 and the other number times 7 is 98. Find the two numbers.”

⇔ [{ ・ ‥} ”two numbers”] [{～ ～ } ”there are ～ ～”]

⇔ [{ ・} ”one number”] [{ ・ ‥} ”the other number”]

⇔ [”Add “ “ and “ “ is “ “.” { ・ ‥ } (next) { ・ ‥ … }]

⇔ [” “ “ times “ “ is “ “.” { ・ ‥ } (next) { ・ ‥ … }]

⇔ ["Find , , , number .” {representative ・} (next) {specific link ・}]

⇔ ["There , , , number , , , Find number .” {representative ・} (next) , , , (next) {specific link ・}]

{ ・ ‥ }

{ ・ ‥ } (next) { ・ ‥ 18}

{ ・ 5 } (next) { ・ 5 ‥}

{ ・ 7 } (next) { ・ 7 ‥}

{ ・ ‥ } (next) { ・ ‥ 98}

{ ・ ‥} (next) { ・ ‥}

// replaces sub strings by their representatives to see if strings that match are in its
// memory
// strings are there; strings of a regularity do not reach a goal of current strings, strings
// of an example reach a goal of the example, not a goal of current strings.
// no strings of regularities lead to the end state, “ “ { specific link ・ } .

// tries to find a way to the end state.
// finds input and output pairs match strings of a regularity
// it tries to apply the regularity to the example and to get strings that lead
// to the end state.

[

⇨ , , , “ “ , , , { , , , (specific link) ・ , , , } (next bp)

// replace specific link by

, , , “ “ , , , { , , , (specific link) ・ , , , }

<same> <same>

⇔ [“ “ { (specific) ・ }]

⇔ [, , , “ “ , , , { , , , (representative) ・ , , , }] (next bp)

, , , “ “ , , , { , , , (representative) ・ , , , }

// the head of the link

, , , “ “ , , , { , , , (representative) ・ , , , } (next bp)

, , , “ “ , , , { , , , (representative) ・ , , , }

<same>

⇔ [, , , “ “ , , , { , , , (representative) ・ , , , } , , ,]

// retrieve strings of a regularity whose first part match

// the input strings with replacing a specific link by its head.

⇔ [, , , “ “ , , , { , , , (representative) ・ , , , } (next)

, , , “ “ , , , { , , , (representative) ・ , , , }] (next bp)

<same>

, , , “ “ , , , { , , , (representative) ・ , , , }

// place, in a work space, retrieved strings that follow the first part.

⇨ , , , “ “ , , , (next bp)

, , , “ “ , , ,

<same> <same>

⇔ [“ “ { (specific) ・ }]

⇔ [, , , “ “ , , , { , , , (representative) ・ , , , }] (next bp)

, , , “ “ , , , { , , , (specific) ・ , , , }

// make strings so that strings made are bound to input strings

// so that the binding is consisitent with a regularity retrieved.

✔

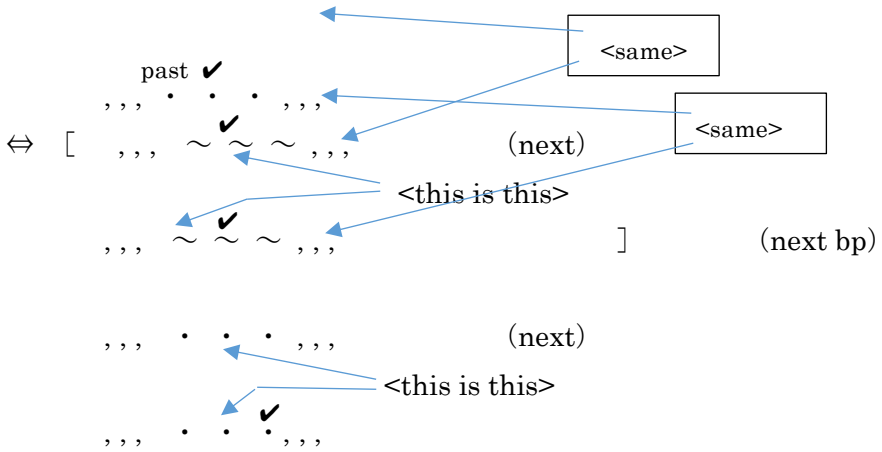
, , , ・ ・ ・ , , , (next)

past ✔

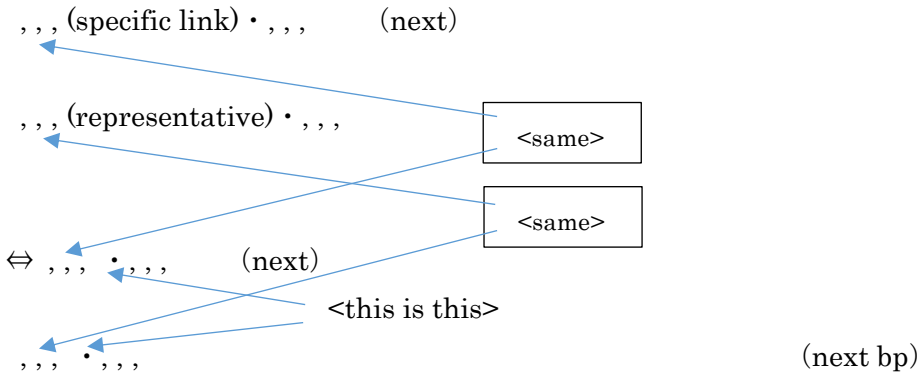
, , , ・ ・ ・ , , , (next bp)

✔

, , , ・ ・ ・ , , , (next)



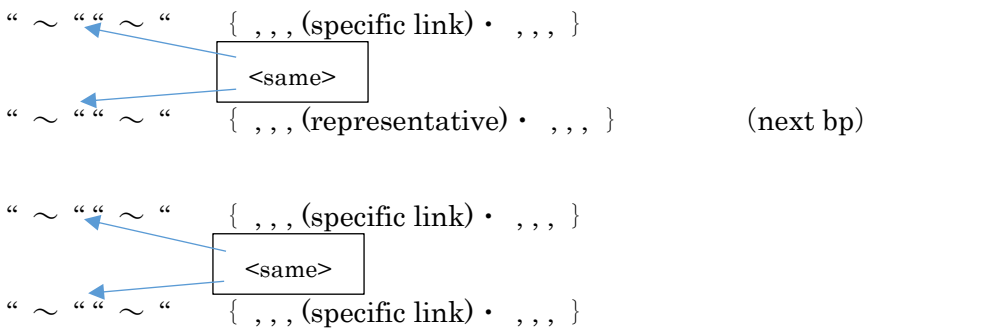
// place a focus in strings so that <this is this> points
// to the place where a focus is placed before (next)
// in input strings; in other words, a place in strings
// pointed by <this is this> before (next) is the place
// where a focus is placed, and the place in strings after
// (next) is pointed by <this is this>. Then a new focus
// is placed as a position relative to the place pointed is
// described in a regularity.



, , , (specific link) ・ , , , (next)

, , , (specific link) ・ , , ,

// make a representative become a specific one
// in such a way that <this is this> holds

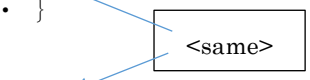


// make a representative become a specific one

// in such a way that <same> holds.

“ ～ “ “ ～ “ “ “　{ (specific link) ・ (specific link) ・ }
“ “　{ (representative) ・ }　(next bp)

“ ～ “ “ ～ “ “ “　{ (specific link) ・ (specific link) ・ }
“ “　{ (representative) ・ }　(next bp)



⇔ [“ ～ “ “ ～ “ “ “　{ (representative) ・ (representative) ・ } (next)
, , , (representative) ・ , , , (next)
“ “　{ (representative) ・ }
]　(next bp)

⇔ [“ ～ “ “ ～ “ “ “　{ (representative) ・ (representative) ・ } (next)
, , , (representative) ・ , , , (next)
“ “　{ (representative) ・ }
]
, , , (representative) ・ , , , (next)
“ “　{ (representative) ・ }　(next bp)
// place, in a work space, retrieved strings that follow the first part.

, , , (specific link) ・ , , , (next)
“ “　{ (representative) ・ }　(next bp)
// conduct strings (do strings) and
// make a representative become a specific one.

, , , (specific link) ・ , , , (next)
“ “　{ (specific link) ・ }

“ ～ “ “ ～ “ “ ～ “ “ “

{ (specific link) ・ (specific link) ・ (representative) ・ }

(next bp)

“ ～ “ “ ～ “ “ ～ “ “ “

{ (specific link) ・ (specific link) ・ (representative) ・ }

<same>

⇔ [“ ～ “ “ ～ “ “ ～ “ “ “

{ (representative) ・ (representative) ・ (representative) ・ } (next)

, , , (representative) ・ , , , (next)

“ ～ “ “ ～ “ “ ～ “ “ “

{ (representative) ・ (representative) ・ (representative) ・ }

] (next bp)

⇔ [“ ～ “ “ ～ “ “ ～ “ “ “

{ (representative) ・ (representative) ・ (representative) ・ } (next)

, , , (representative) ・ , , , (next)

“ ～ “ “ ～ “ “ ～ “ “ “

{ (representative) ・ (representative) ・ (representative) ・ }

]

, , , (representative) ・ , , , (next)

“ ～ “ “ ～ “ “ ～ “ “ “

{ (representative) ・ (representative) ・ (representative) ・ }

// place, in a work space, retrieved strings that follow the first part.

(next bp)

, , , (specific link) ・ , , , (next)

“ ～ “ “ ～ “ “ ～ “ “ “

{ (representative) ・ (representative) ・ (representative) ・ }

// conduct strings (do strings) and

// make a representative become a specific one.

(next bp)

, , , (specific link) ・ , , , (next)

“ ～ “ “ ～ “ “ ～ “ “ “

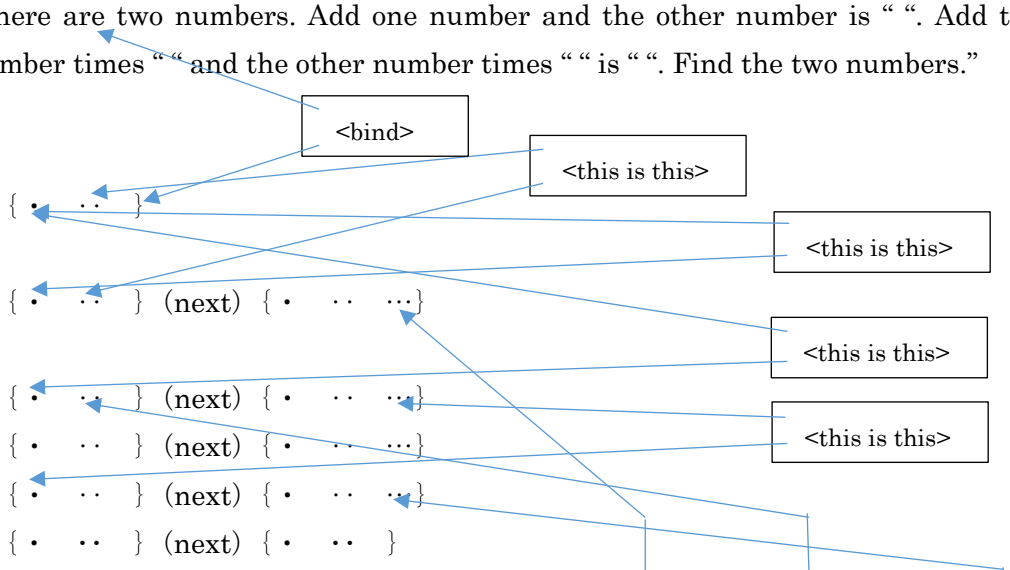
{ (specific link) ・ (specific link) ・ (specific link) ・ }

// conduct strings (do strings) and
// make a representative become a specific one.

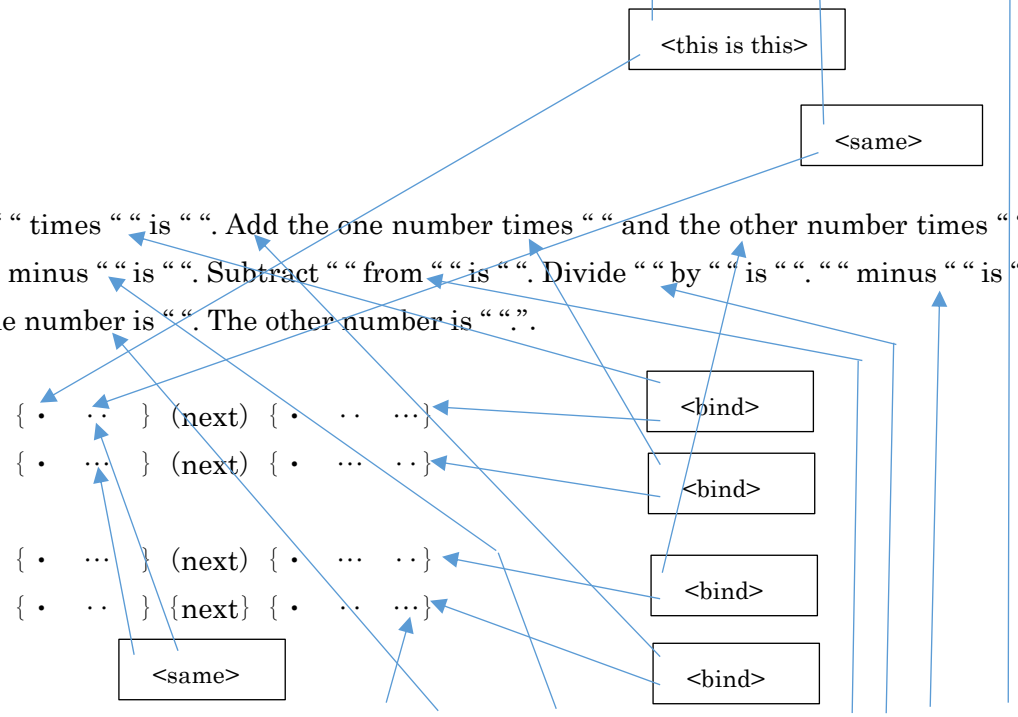
]

// apply the above to the current problem, and get the following.

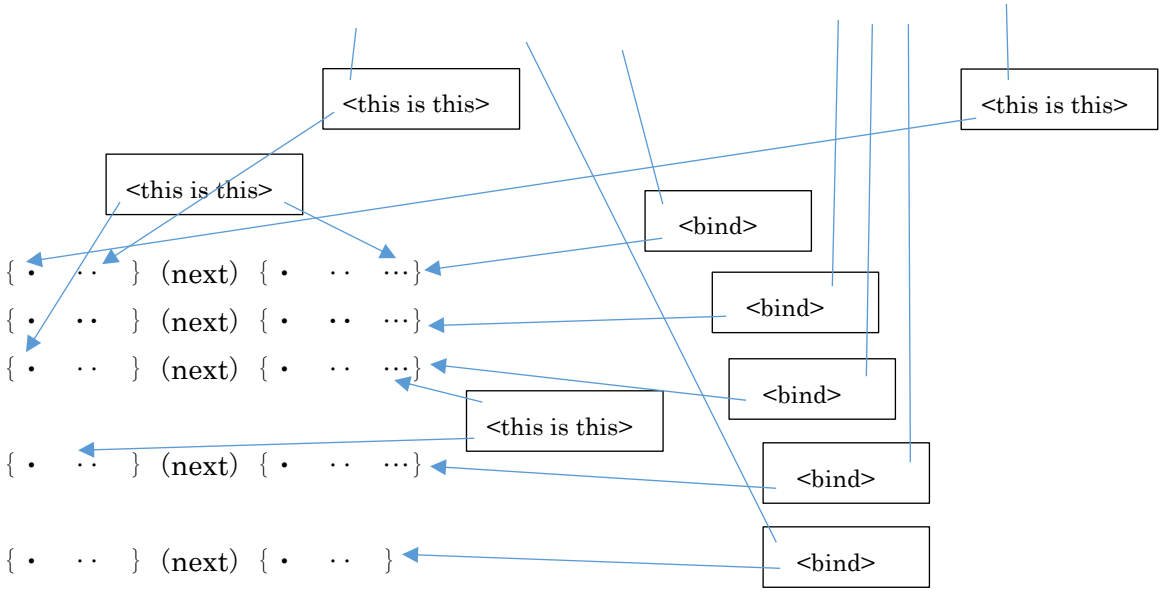
“There are two numbers. Add one number and the other number is “ “. Add the one number times “ “ and the other number times “ “ is “ “. Find the two numbers.”



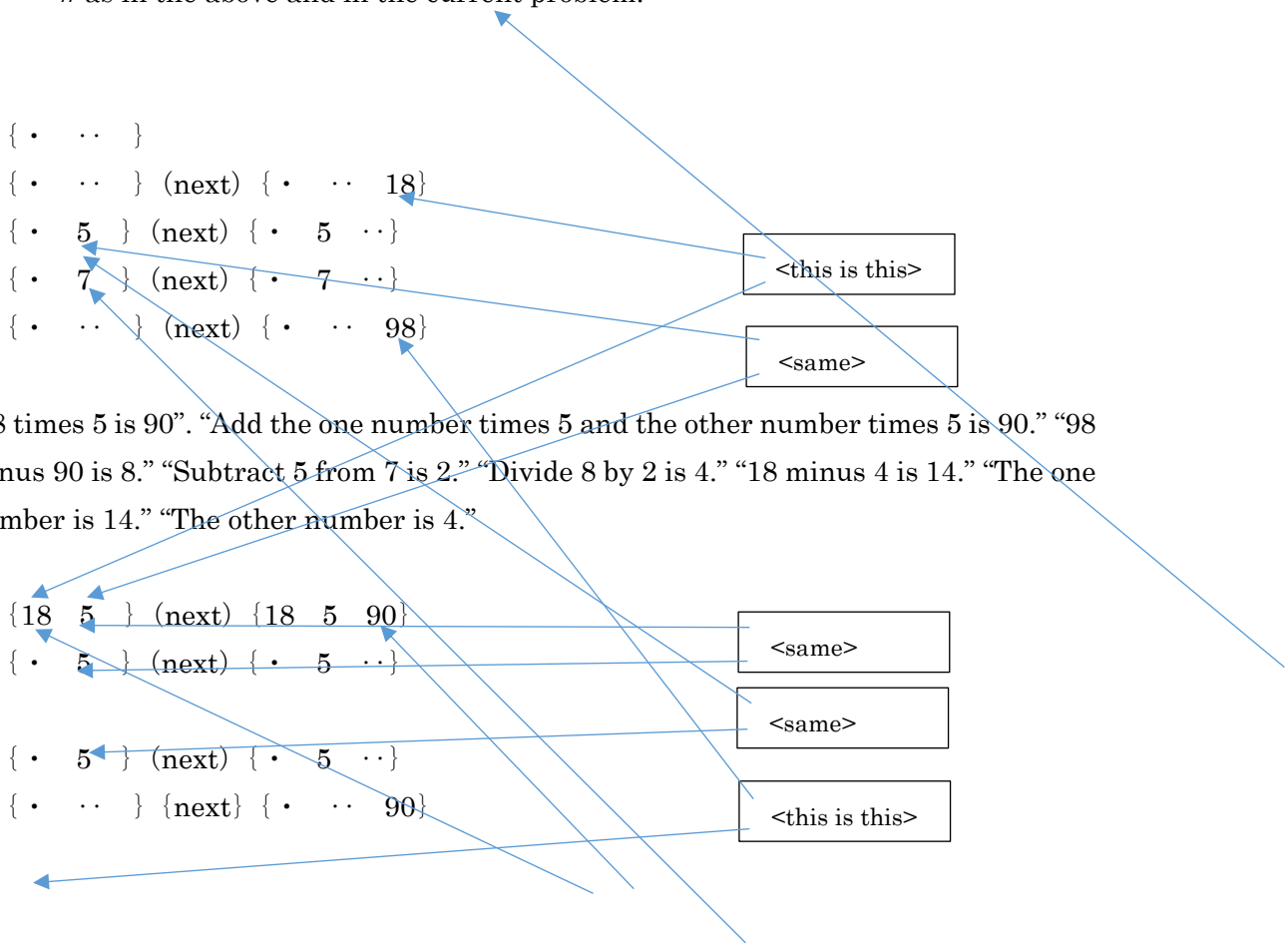
// retrieves strings that match the above with sub strings replaced
// by their representatives. Namely another example with repleace sub strings.
// and strings that follow.



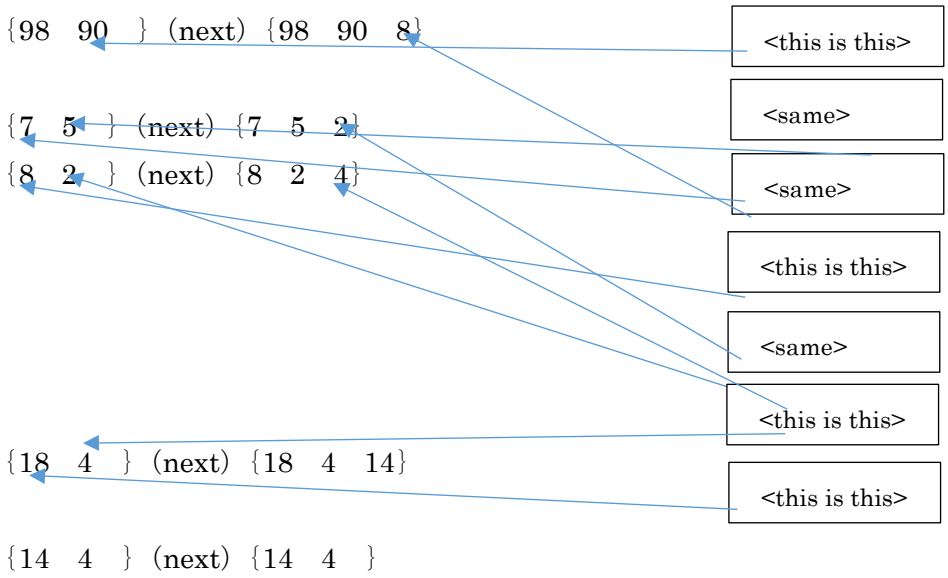
“ “ “ times “ “ is “ “. Add the one number times “ “ and the other number times “ “ is “ “. “ “ minus “ “ is “ “. Subtract “ “ from “ “ is “ “. Divide “ “ by “ “ is “ “. “ “ minus “ “ is “ “. The one number is “ “. The other number is “ “.”.



// apply the regularity to the above so that
// 1) representatives in the above become specific ones,
// 2) the specific ones satisfy <this is this>, <same>, and <bind>
// as in the above and in the current problem.



"18 times 5 is 90". "Add the one number times 5 and the other number times 5 is 90." "98 minus 90 is 8." "Subtract 5 from 7 is 2." "Divide 8 by 2 is 4." "18 minus 4 is 14." "The one number is 14." "The other number is 4."


{98 90 } (next) {98 90 8}
{7 5 } (next) {7 5 2}
{8 2 } (next) {8 2 4}
{18 4 } (next) {18 4 14}
{14 4 } (next) {14 4 }
<this is this>
<same>
<same>
<this is this>
<same>
<this is this>
<this is this>

》

3.5 A regularity of having another program make methods of solving problems

The program tries to have a regularity hold for cases where a sub string of the regularity strings is replaced by another sub string. Here a sub string and another sub string are found to have the same representative.

The program already has formed that I, program Pa, and other program are specific programs and their representative is one. Suppose that the program has formed a regularity of I solve a math problem given by program T. Suppose also that the program is given to have another program Pa solve the math problem. Then it replaces its sub string program T by I and I by program Pa, and it tries to have the regularity hold for sub strings I and program Pa; Program T gives I “do ～～”. I give program Pa “do ～～”.

When it finds the regularity does not hold, namely program Pa does not reach a goal (does not output a solution to the math problem), it tries to make the regularity hold, namely has program Pa reach the goal (output the solution). It retrieves a regularity to have program Pa make a way to reach the goal (output the solution), namely make program Pa change states from it has no way to the goal to it has a way to the goal.program Pa.

The program tries to give program Pa strings with quotations each is bound to a step of reaching the goal. It tries to find strings with quotations when a step has not been bound to strings with quotations.

[

[⇨ ”do ” , , , Find the number” ”

⇔ , , , (next) “the number is “ “ “ { ・ }

// does not retrieve a regularity to reach the goal:

// “the number is “ “ {specific link ・ }

⇔ [”cannot do ” , , , Find the number.” ”

“～～”

⇔ “～～“ { ～～}

]

]

”cannot do ” , , , Find the number.” ”

⇔ [

⇨ , , , “ “ , , , { , , , (specific link) ・ , , , } (next bp)

// replace specific link by

, , , “ “ , , , { , , , (specific link) ・ , , , }

<same>

<same>

⇔ [“ “ { (specific) ・ }]

⇔ [, , , “ “ , , , { , , , (representative) ・ , , , }] (next bp)

, , , “ “ , , , { , , , (representative) ・ , , , }

// the head of the link

“given inputs, replace an example number by a number in the inputs.”

, , , “ “ , , , { , , , (representative) ・ , , , } (next bp)

, , , “ “ , , , { , , , (representative) ・ , , , }

<same>

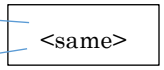
⇔ [, , , “ “ , , , { , , , (representative) ・ , , , } , , ,]

// retrieve strings of a regularity whose first part match

// the strings with replacing a specific link by its head.

“retrieve strings that match the inputs.”

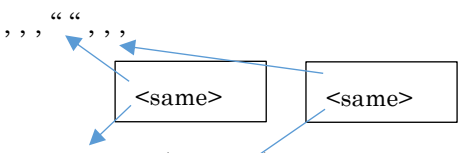
⇔ [, , , “ “ , , , { , , , (representative) ・ , , , } (next)
, , , “ “ , , , { , , , (representative) ・ , , , }] (next bp)



, , , “ “ , , , { , , , (representative) ・ , , , }

// place, in a work space, retrieved strings that follow the first part.

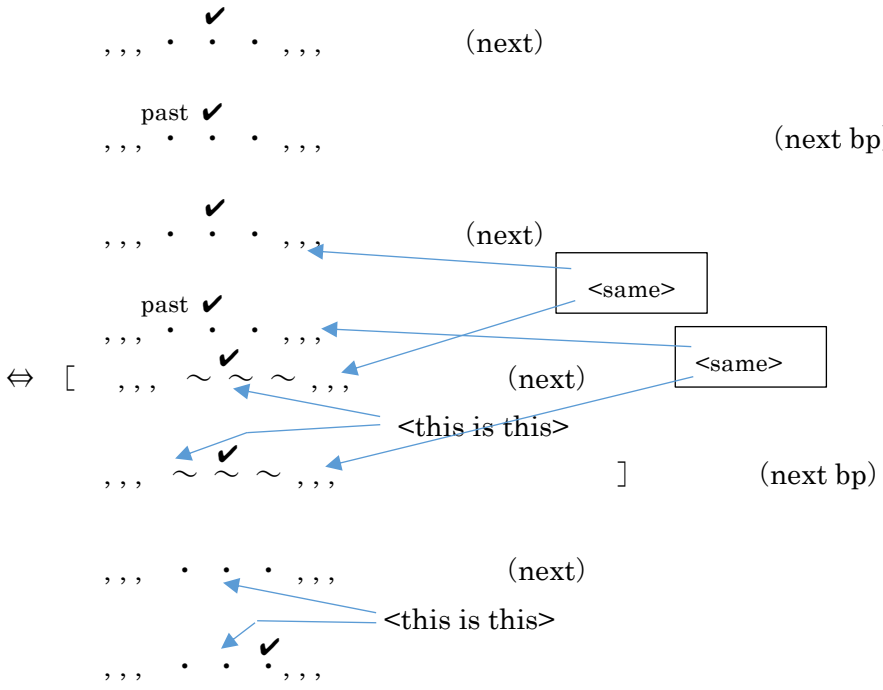
⇨ , , , “ “ , , , (next bp)



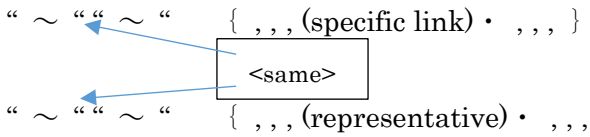
⇔ [“ “ { (specific) ・ }]
⇔ [, , , “ “ , , , { , , , (representative) ・ , , , }] (next bp)
, , , “ “ , , , { , , , (specific) ・ , , , }

// make strings so that strings made are bound to input strings
// so that the binding is consisitent with regularities retrieved.

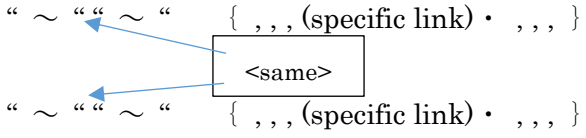
“write the input number”
“make this as the same as this”



“the next is this”



(next bp)



// make a representative become a specific one
// in such a way that <same> holds.

“make this be the same as this.”

“ ～ “ “ ～ “ “ “　　{　(specific link) ・　　(specific link) ・　　}
“ “　　{　(representative) ・ }　　(next bp)

“ ～ “ “ ～ “ “ “　　{　(specific link) ・　　(specific link) ・　　}
“ “　　{　(representative) ・ }　　(next bp)

<same>

⇔ [“ ～ “ “ ～ “ “ “　　{　(representative) ・　　(representative) ・　　}　　(next)
, , ,　(representative) ・ , , ,　　(next)
“ “　　{　(representative) ・ }
]　　(next bp)

⇔ [“ ～ “ “ ～ “ “ “　　{　(representative) ・　　(representative) ・　　}　　(next)
, , ,　(representative) ・ , , ,　　(next)
“ “　　{　(representative) ・ }
]
, , ,　(representative) ・ , , ,　　(next)
“ “　　{　(representative) ・ }　　(next bp)
// place, in a work space, retrieved strings that follow the first part.

, , ,　(specific link) ・ , , ,　　(next)
“ “　　{　(representative) ・ }　　(next bp)
// conduct strings (do strings) and
// make a representative be a specific one.

, , ,　(specific link) ・ , , ,　　(next)
“ “　　{　(specific link) ・ }

“retrieve strings that match the inputs”

“ ～ “ “ ～ “ “ ～ “ “ “

{ (specific link) ・ (specific link) ・ (representative) ・ }

(next bp)

“ ～ “ “ ～ “ “ ～ “ “ “

{ (specific link) ・ (specific link) ・ (representative) ・ }

<same>

⇔ [“ ～ “ “ ～ “ “ ～ “ “ “

{ (representative) ・ (representative) ・ (representative) ・ } (next)

, , , (representative) ・ , , , (next)

“ ～ “ “ ～ “ “ ～ “ “ “

{ (representative) ・ (representative) ・ (representative) ・ }

] (next bp)

⇔ [“ ～ “ “ ～ “ “ ～ “ “ “

{ (representative) ・ (representative) ・ (representative) ・ } (next)

, , , (representative) ・ , , , (next)

“ ～ “ “ ～ “ “ ～ “ “ “

{ (representative) ・ (representative) ・ (representative) ・ }

]

, , , (representative) ・ , , , (next)

“ ～ “ “ ～ “ “ ～ “ “ “

{ (representative) ・ (representative) ・ (representative) ・ }

// place, in a work space, retrieved strings that follow the first part.

(next bp)

, , , (specific link) ・ , , , (next)

“ ～ “ “ ～ “ “ ～ “ “ “

{ (representative) ・ (representative) ・ (representative) ・ }

// conduct strings (do strings) and

// make a representative become a specific one.

(next bp)

, , , (specific link) ・ , , , (next)

“ ～ “ “ ～ “ “ ～ “ “ “

{ (specific link) ・ (specific link) ・ (specific link) ・ }

// conduct strings (do strings) and
// make a representative become a specific one.

“do strings as I did before”

]

[⇨ ”do ” , , , Find the number” ”
⇔ , , , (next) “the number is “ “ “ { ・ }
// retrieve a regularity to reach the goal:
// “the number is “ “ { ・ }
”can do ” , , , Find the number.” ”]
]

[”see “the number is “ “ “ is the answer to “ , , , Find the number.” “ (next)
{ (specific link) ・ }]

]

The above is a regularity the program has formed, and the regularity has strings with quotations bound to capabilities that the program has from the beginning and outputs the strings. One of the original capabilities is to replace sub strings of the inputs by their representatives. For example, it replaces “1”, “2”, or “10” by “a number”. The program binds strings “replace an example number by a number” to the original capability, and outputs the strings.

At this point, the program does not differentiate an original capability from an ability acquired after it starts running. For example, a human gives the program addition of two specific numbers such as add 2 and 3, and an ability that enables the program conduct add 2 and 3 is the acquired ability with an original capability. But extending it to numbers that are not given by the human is the original capability.

The program has not yet formed three regularities; one describes a capability including both original and acquired capabilities, one describes original capabilities, and the third describes acquired capabilities.