Say My Innate Capabilities in a Natural Language 2/3

30 September 2020

Kenzo Iwama

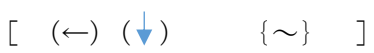
4. Regularities already formed

This section describes regularities that the program has already formed. A regularity that the program first formed consists of a string to describe just an input string there. Then the program forms a regularity that a string is there and then is not there (disappears), and a string is not there and is there (appears). Next, the program forms a regularity that consists of input strings with replacing sub strings of the input by their representative strings of the sub strings, relations <bind> <same> <this is this>, and （next）, strings to specify which types of strings, namely input, output and retrieved from a memory of the machine, and strings to specify relations between representative strings and their member strings.

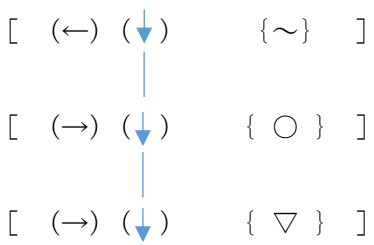
The program firstly forms the most general string, namely a representative of all the strings, and forms links to chain members of the representative. The head of the link is a representative of the strings, （←）（↓）｛ ～ ｝, and its members (or examples) are linked by （→）（↓）.

A string

A representative of all the strings that keep appearing is described as shown below.

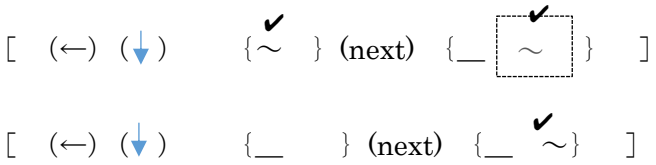


Members of the representative are linked as shown below.



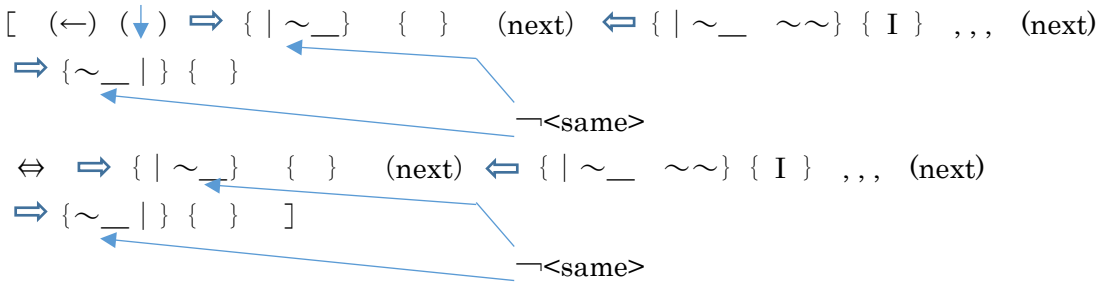
～ and ｛ ｝ are innnate codes placed by the program. The program has those codes (strings) from the beginning. (Later the program binds a string with quotations “object” to the representative and forms ［(←) ( ) ”object” {～}］.)

Changes from a string to no string, and changes from no string to a string



Next, as shown in the above, the program forms a representative of changes from some string {~} to no string ~ . A focus ✔ is placed on the string which is gone (or disappears). Later, strings "Object disappears" are bound to the representative. A representative of changes from no string to some string is also shown in the above. Later, strings "Object appears" are bound to the representative.

Suppose the program gets motor strings and the fact that the machine generates the motor strings while it gets changes of strings through the second channel. The program forms a regularity of the changes with the motor strings as shown below. { I } describes the fact that the machine generates the motor strings.



Strings { | ~__} and {~__ | } are representatives all the strings that the program gets before it gets motor strings and after it gets the motor strings. { ~~} is a representative of all the motor strings. { | ~__} , {~__ | } and { ~~} are codes (strings) the program possesses from the beginning. Strings , , , describe that strings of the same repeat and that the sameness is found by a perspective the program takes. Here, the program takes the perspective, namely places a focus on the fact of getting motor strings, and finds the fact of getting strings through the second channel repeats.

Strings (codes), ⇨ ⇦ , describe inputs and outputs respectively, and are the strings the program has from the beginning.

Binding of strings with quotations and strings without quotations

Strings, “object is gone” and “object is back” are bound to strings describing regularities already formed.

[(←) (↓) “object is gone”
⇨ { ✔ ~ } (next) { __ ~ }]

[(←) (↓) “object is back”
⇨ {__ } (next) { __ ✔ ~ }]

Strings “do a deed” are bound to a representative of motor strings and the fact that the machine (on which the program runs) generates the motor strings.

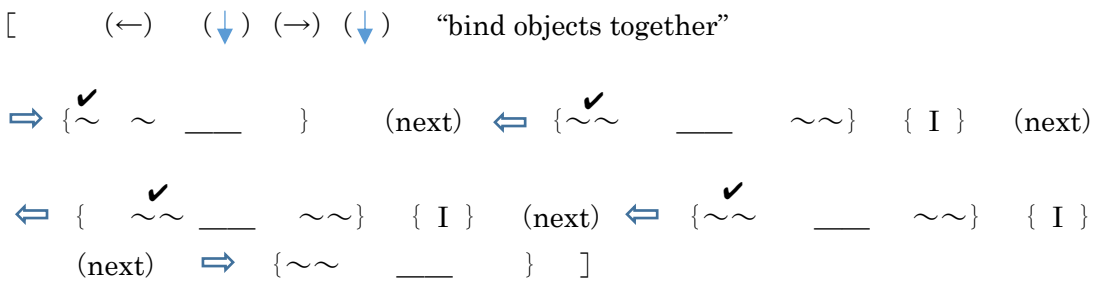
[(←) (↓) ”do a deed” ⇨ { | ~__} { } (next) ⇦ { | ~__ ~~} { I }
, , , (next) ⇨ {~__| } { }]
¬<same>

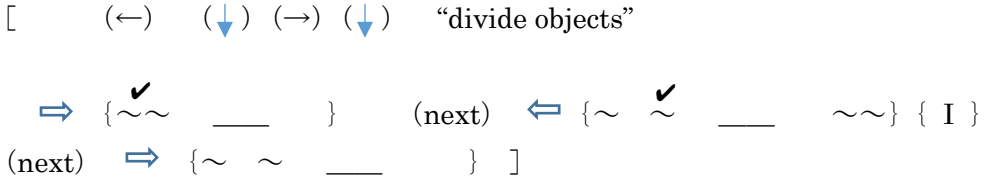
[(←) (↓) (→) (↓) “take object” ⇨ {✔~____ } { }
(next) ⇦ {____ ✔~ ~~} { I } (next) {____ ~ ~~}]

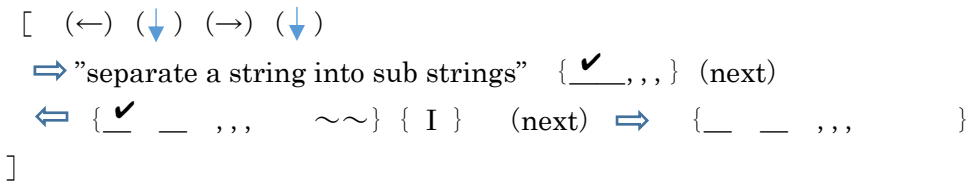
[(←) (↓) (→) (↓) “put object” ⇨ { ✔~ ____ }
(next) ⇦ { ✔~ ____ ~~} { I } (next) ⇨ { ✔~____ }]

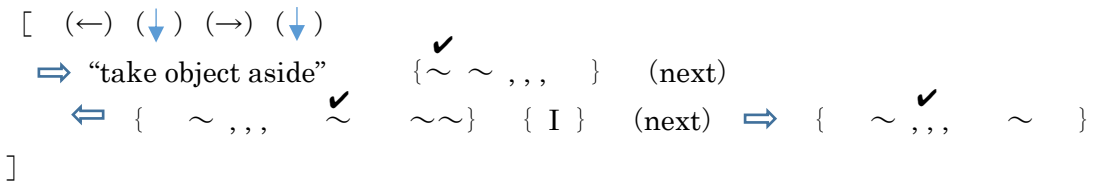
[(←) (↓) (→) (↓) “take object away” ⇨ { ✔~____ }
(next) ⇦ {____ ✔~ ~~} { I } (next) ⇨ {____ ~ ~~}]

[(←) (↓) (→) (↓) “put objects together” ⇨ {✔~ ~ ____ }
(next) ⇦ {✔~~ ____ ~~} { I } (next) ⇨ {~~____ }]

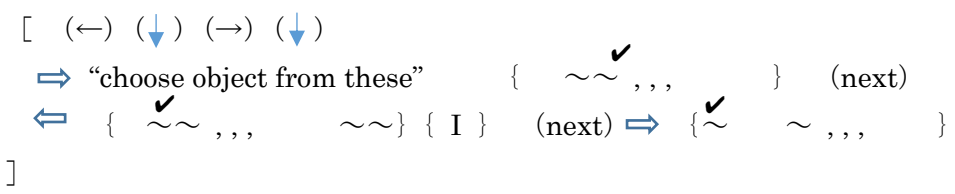






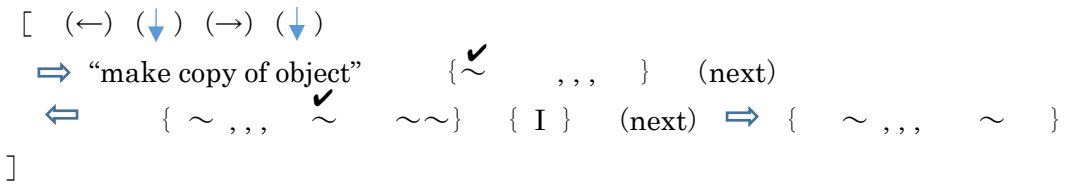


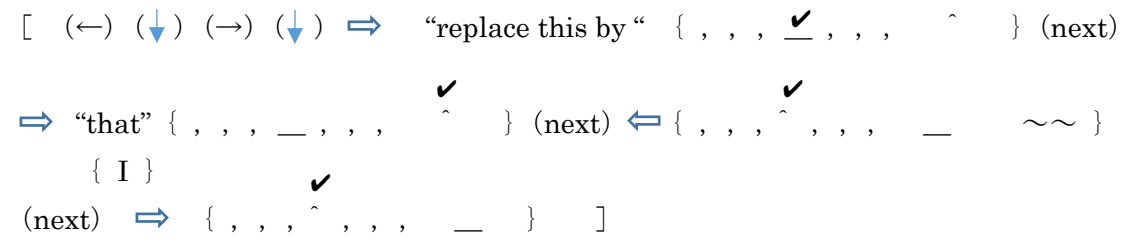
When the program gets inputs, it links them in a list. When the program gets inputs in a time sequence, it, given the link of the inputs, tries to retrieve a regularity to make the next link that matches the input link, but it does not retrieve such a regularity, it keeps the inputs, namely, the initial link, the next link, and the next of the next.

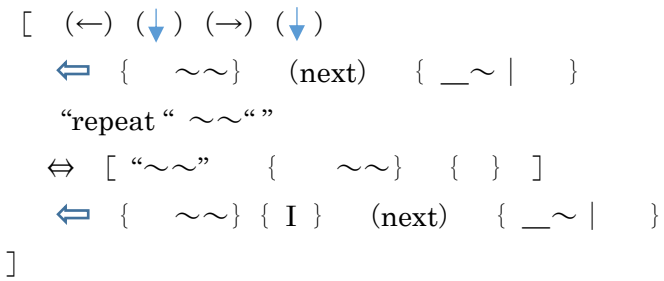


After the program keeps serieses of the inputs with “choose ~”, it tries to form a regularity, given a new link of inputs, to make the next link that matches the inputs. It does not form a regularity to specify one in the link, but it finds the fact that one is in the link. The program has, from the beginning, a procedure to take one at the beginning of any link. Then the program makes a regularity to take one at the beginning of the link.

Later, when the program forms examples of problems, it revises a regularity to choose an object from a list in such a way that it gets one at the beginning and then gets the next at the next of the beginning in the list.







[(←) (↓) (→) (↓)
⇨ { ～～} (next) { __～ | }
⇔ [“～～” { ～～} { }]
⇨ { ～～} { I } (next) { __～ | }
“ “～～” repeat”

]

The program gets strings “first (or initial) state” and “last (or goal) state” and binds them to the first step and the last step of a regularity already formed as in the below.

[(←) (↓) ”do a deed”
“an initial state” ⇨ { | ～__} (next) ⇦ { | ～__ ～～} { I }
(next) ”a goal state” ⇨ {～__ | }
]

[(←) (↓) ” another program does a deed”
“an initial state” ⇨ { | ～__} (next) , , , (next)
”a goal state” ⇨ {～__ | }
]

When the program gets input strings, it tries to retrieve strings from its memory that match the input strings. At the time of matching the two strings, the program replaces sub strings of the input by their representatives and sees if the strings with replacement matches strings stored in its memory. It then places the retrieved strings in its work.

The program then changes sub strings of the retrieved strings to sub strings *Se* in the

following way; 1) relations <same> sets sub strings of the input to become the sub strings *Se* of the retrieved, 2) parts of the retrieved strings envoke strings in its memory to produce sub strings that become the sub strings *Se*.

do a deed (←) (↓)

take (←) (↓) (→) (↓)

put (←) (↓) (→) (↓)

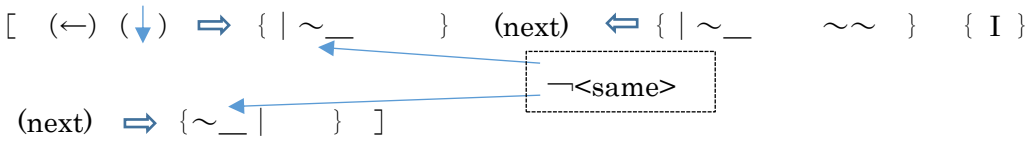
put together (←) (↓) (→) (↓)

bind (←) (↓) (→) (↓)

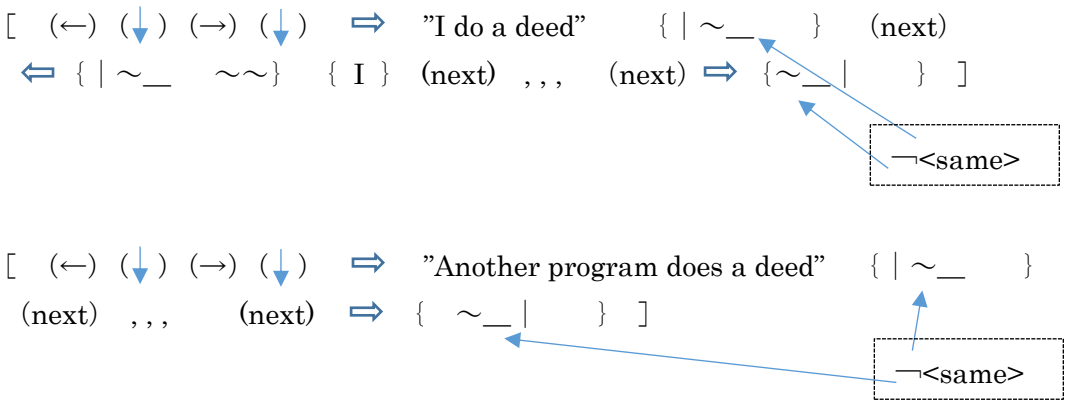
divide (←) (↓) (→) (↓)

separate (←) (↓) (→) (↓)

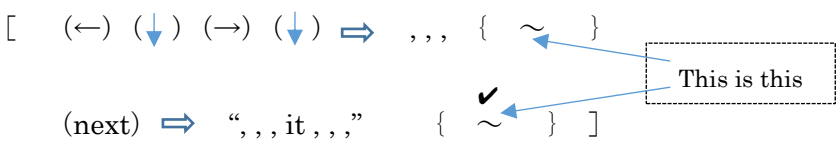
Strings "I", "another program", and "one"

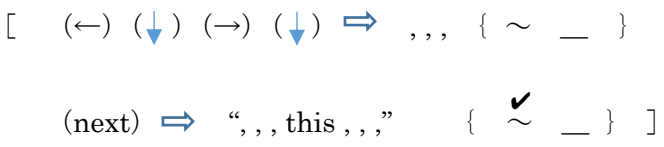


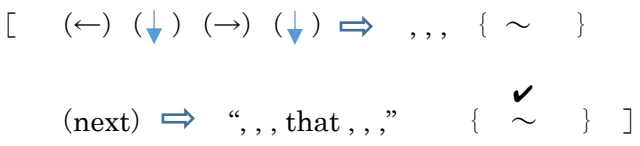
The program gets strings through the second channel at time t1, and at time t2, and it gets strings of its motors. But there are cases where it gets strings through the second channel at time t1, and at time t2 without getting strings of its motors at time t1 and t2. The strings at time t1 are different from those at time t2. The program binds the former representative to a string "I do a deed", and the latter representative to a string "another program does a deed". The representative links to its members, each member describes a specific program makes changes such as "Program A does a deed".

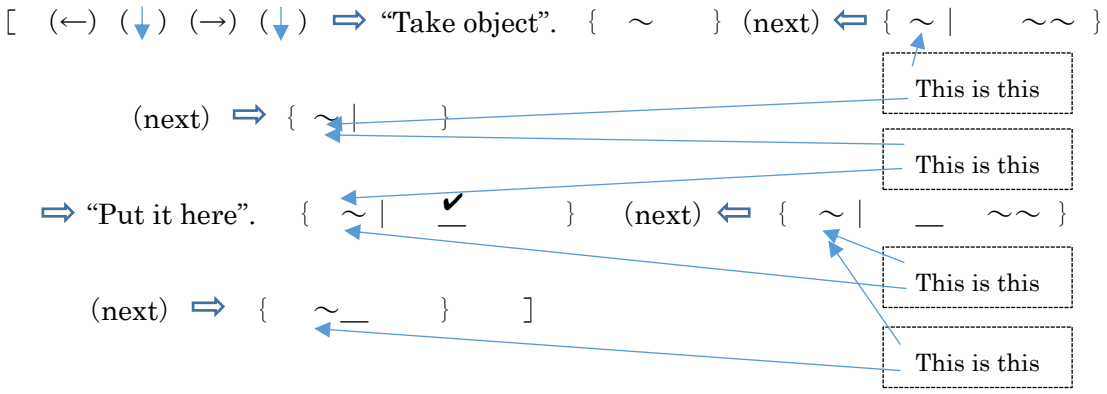


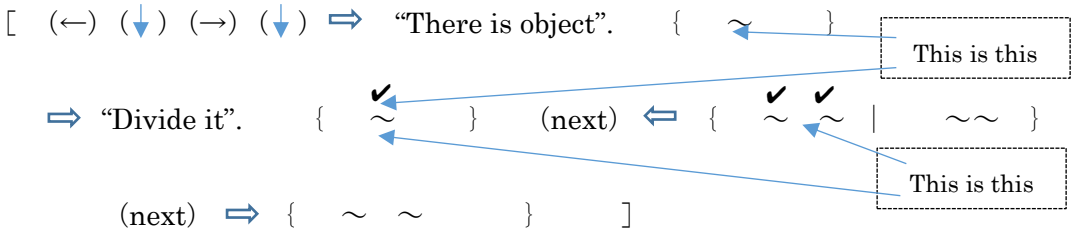
Pronauns


[(←) (↓) (→) (↓) ⇨ , , , { ~ }
This is this
(next) ⇨ “, , , it , , ,” { ~ }]


[(←) (↓) (→) (↓) ⇨ , , , { ~ __ }
(next) ⇨ “, , , this , , ,” { ~ __ }]


[(←) (↓) (→) (↓) ⇨ , , , { ~ }
(next) ⇨ “, , , that , , ,” { ~ }]

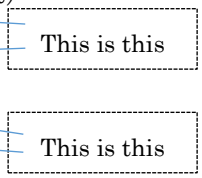

[(←) (↓) (→) (↓) ⇨ “Take object”. { ~ } (next) ⇦ { ~ | ~~ }
(next) ⇨ { ~ | }
⇨ “Put it here”. { ~ | __ } (next) ⇦ { ~ | __ ~~ }
(next) ⇨ { ~__ }]
This is this
This is this
This is this
This is this


[(←) (↓) (→) (↓) ⇨ “There is object”. { ~ }
⇨ “Divide it”. { ~ } (next) ⇦ { ~ ~ | ~~ }
(next) ⇨ { ~ ~ }]
This is this
This is this

[(←) (↓) (→) (↓) ⇨ “There are objects”. { ～ ～ } (next)

⇨ (next) “Put them together”. { ～ ～ } (next)

⇦ { ～～ | ～～} (next) ⇨ { ～～ }]

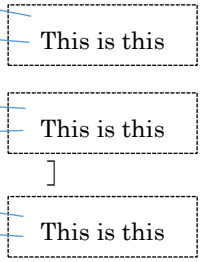


[(←) (↓) (→) (↓) ⇨ “Take this ” { ✔ ～ ～ } (next)

⇨ “and that”. { ～ ✔ ～ } ⇦ (next) { ～ ～ | ～～}

⇨ (next) “Put them here”. { ～～ | ✔ __ } (next)

⇦ { ～～ | __ ～～} (next) ⇨ { ～～__ }]



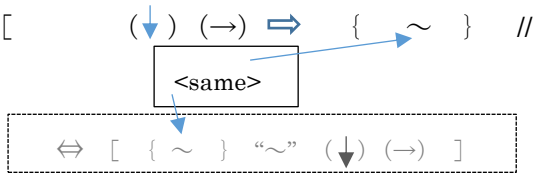
Hierarchical regularities.

When the program gets strings through the second / the third channel and strings through the first channel at the same time, it binds them together. The program keeps strings bound together, and then it forms a regularity, namely, a set and its representative. The representative is strings through the second / the third channel bound to strings through the first channel. The strings through the first channel are kept with quotations.

When the program gets some string Ss through the second channel but no string through the first channel, it tries to retrieve a string that matches Ss and a string with quotations bound to Ss. But it does not find any string with quotations. Then the program finds inconsistency between the current and a regularity formed. The program keeps inconsistent cases in its memory, and then it forms a new regularity that describes the inconsistency; there are cases where the program gets a new string and tries to retrieve a string with quotations, but fails to retrieve a string with quotations although there are cases where the program gets a string through channel 2 and 3, and retrieves a string Sr that matches the input string and a string with quotations bound to Sr. The program later binds a string “what is this” to the strings that describe the inconsistency, as shown in Figure 5.

[(↓) (→) ⇨ { ～ }
⇔ [(↓) (→) { ～ } ”～“]]

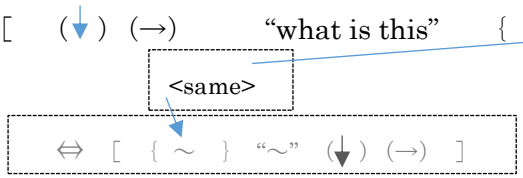
(1) A regularity that, given input strings, the program retrieves strings that match the input and are bound to strings with quotations.



⇔ [(↓) (→) ⇨ { ～ }
⇔ { ～ } ”～“ (↓) (→)]]

// A regularity input strings are bound to strings with quotations does not hold.

(2) Inconsistency between current strings and a regularity formed already.

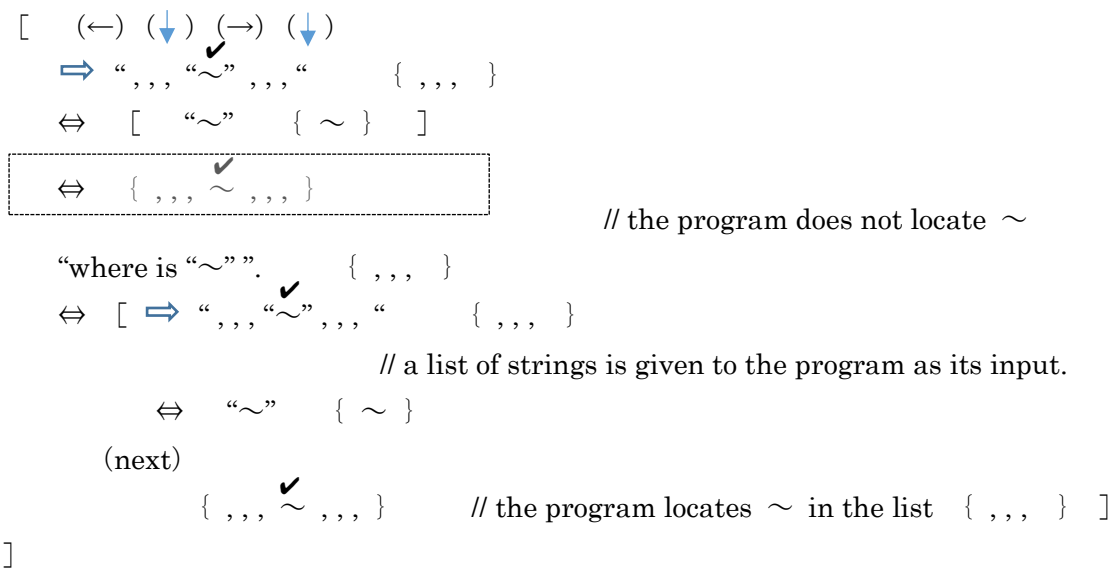


⇔ [(↓) (→) ⇨ { ～ }
⇔ { ～ } ”～“ (↓) (→)]]

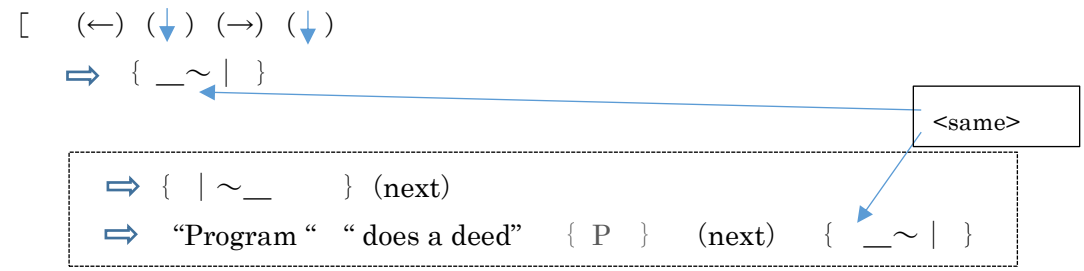
// In general, an example of an object is bound to strings with quotations.

(3) Bind strings “what is this” to the inconsistency the program finds.

Figure 5. An example of a hierarchical regularity.



The program, given a list of input strings $S_{objects}$ and a string with quotations S_q, locates a string S_{ob} in the list that matches a string bound to the string S_q. But there are cases where the program does not locate the string S_{ob} in the list that matches the string bound to the string S_q. For example, the program locates a string ～ bound to “～” in the list when the program gets a string “take “～“ “ and strings of objects $S_{objects}$, { , , , } . But there are cases where the program does not locate the string ～ boud to “～” in the list. The program binds a string “where is “～” “ to the cases as just described.



“who did a deed”

⇔ [⇨ { |~_ } (next)
⇨ “Program “ “ does a deed” { } { P } (next) { _~| }]
]

There are cases where the program gets changes and “Program “ “ does a deed”, and there are cases where the program gets changes but not “Program “ “ does a deed”. When the program gets changes in the input but not get “Program “ “ does a deed” and it needs to get “Program “ “ does a deed”, it forms a list describing possible “Program “ “ does a deed“. To the case as just described, the program binds “who did a deed”.

Who says who does “〜〜”.

[(←) (↓) (→) (↓)
{ ____ } { I C } ”I say to C ”I turn “____” into “__〜” ”
⇔ ["turn “ | 〜__” into “__〜 | ” ” { | 〜__ } { } (next)
⇦ { | 〜__ 〜〜} { I } (next)
⇨ { __〜 | } { }]
⇔ ["____” { ____ }]
⇔ ["__〜” { __〜 }]
⇔ ["I say to C “〜〜” ” { | 〜__ } { I C } (next)
⇦ “〜〜” { I C } (next)
⇨ { __〜 | } { C I }]
⇦ “turn “____” into “__〜” “ { ____ } { I C } (next)
⇨ { __〜 } { C I }
]

[(←) (↓) (→) (↓)
{ ____ } { T B I } {”T says to B ”I turn “____” into “__〜” ” }
⇔ ["I turn “__” into “〜” ” { | 〜__ } { I } (next)
⇦ { | 〜__ 〜〜 } { I } (next)
⇨ { __〜 | } { I }]
⇔ [“____” { ____ }]
⇔ [“__〜” { __〜 }]
⇔ ["A says to B “〜〜〜” ” { | 〜__ } { A B I } (next)
⇨ “〜〜〜” { } { A B I } (next)
⇨ { __〜 | } { A B I }]
⇨ “ I turn “____” into “__〜” ” { ____ } { T B I } (next)
⇨ { __〜 } { T B I }
]

Past

[(←) (↓) (→) (↓)

⇨ “Program P did “～～” before “～～～” ”.

, , , (next) { ～～ } { P } (next)

{ ～～～} { P } (next) , , ,

⇔ [”～～～” {～～～} { }]

⇔ [”～～” {～～} { }]

⇔ [“do “～～” before “～～～” ” { ～～ } { P } (next)

”do “～～～” ” { ～～～ } { P } (next) , , ,]

⇔ [”Program P” { } { P }]

]

[(←) (↓) (→) (↓)

⇨ “ Program P did “～～” after “～～～” ”.

, , , (next) { ～～～} { P } (next)

{ ～～} { P } (next) , , ,

⇔ [”～～～” {～～～} { }]

⇔ [”～～” {～～} { }]

⇔ [“do “～～” after “～～～” ” { ～～～ } { P } (next)

”do “～～” ” { ～～ } { P } (next) , , ,]

⇔ [”Program P” { } { P }]

]

[(←) (↓) (→) (↓)

⇨ “ Program P did “～～” at the same time as “～～～” ”.

, , , (next) { ～～～ ～～} { P } (next) , , ,

⇔ [”～～～” {～～～} { }]

⇔ [”～～” {～～} { }]

⇔ [“do “～～” at the same time as “～～～” ” { ～～ ～～～ } { P }

(next) , , ,]

⇔ [”Program P” { } { P }]

]

[⇨ {__～| } { I A }

“did Program A do a deed”

<same>

⇔ "Program P does a deed” { | ～__ } { P } (next)

, , , (next) {__～ | }

⇔ [⇨ {__～ | } { P }

⇔ "Program P does a deed” { | ～__ } { } (next)

{ | ～__ ～～ } { P } , , , (next) {__～ | }]

]

[(←) (↓) (→) (↓)

“The result A did a deed is the same as B did the deed”.

”A did a deed” { ～～} { A } (next) { __～ | }

”B did a deed” { ～～} { B } (next) { __～ | }

same

]

When

[(←) (↓) (→) (↓)

⇨ “ Program P did “～～” “

{ ～～✔～} { } (next) { ～～} { P }
(next) { } { } (next) { }

{ } { } (next) { ～～} { P }
(next) { ～～✔～ } { } (next) { }

{ } { } (next) { ～～ ～～✔～} { P }
(next) { } { } (next) { }

⇔ [“～～” { ～～} { }]

⇔ [“Program P” { } { P }]

⇔ [“Program P does “～～” , , , (next) ”～～” { ～～} { P }
(next) , , , { }]

“When did Program P “～～” “

]

The program forms a list of strings in the order of (next) ; describing a sequence of what the program did. The program then gets where “～～” is positioned in the list after outputting “when “～～” occurs”. In other words, the program gets temporal relations between “～～” and other doings (or occurences) such as “ “～～” occurs before “～～～” occurs”.

Do, Can and cannot regularities

After the program has formed regularities of doing something, the program forms hierarchical regularities of doing something under some conditions, being able to do a deed, and being not able to do a deed. Suppose the program is given doing a deed under some conditions when the conditions do not hold, it starts to do the deed and it gets stopped doing. Since the program has formed a regularity of doing the deed when it is given doing the deed, being stopped doing is not consistent with the regularity. The program keeps sequences, given doing a deed, starts doing the deed, and is stopped doing, in its memory. Suppose also the program is given doing a deed under some conditions when the conditions hold, it does the deed to its end. It tries to make a regularity that includes the two cases. The regularity to be made is a hierarchical one including cases where it does a deed and cases where it does not do the deed.

[
"Do ～～" { | ～__ } { T I }
⇔ ["do ～～" { P I } (next) ⇦ { | ～__ ～～} { I } (next) ⇨ { __～ | }]
⇦ { ～～} { I } (next)
⇨ {__～ | }
]

[
⇨ "Do ～～ in ～" { | ～__ ～ } { T I }
⇔ [⇨ "do ～～ in ～" { P I }
(next) ⇦ { | ～__ ～ ～～} { I } (next) ⇨ { __～ | } { }]
⇦ { ～ ～～} { I } (next)
⇨ {__～ | } { I }
]

[
⇨ "Do ～～ after ～" { | ～__ ～ } { T I }
⇔ [⇨ "do ～～ after ～" { P I }

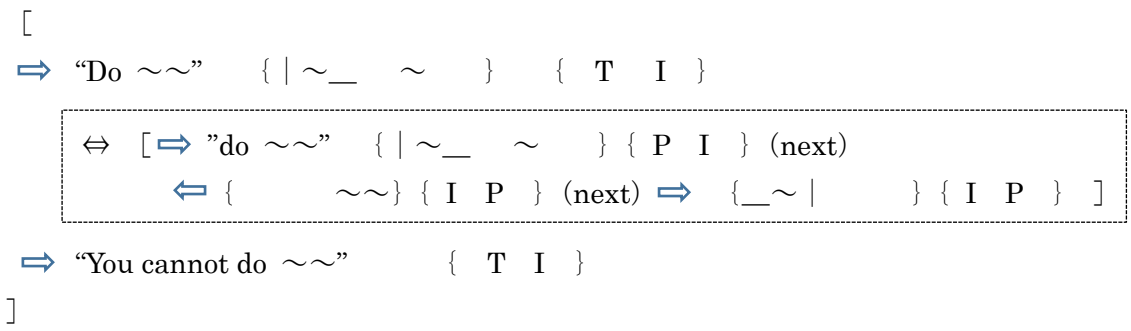
(next) ⇨ { ～ } { I }
(next) ⇦ { | ～__ ～ ～～} { I } (next) ⇨ { __～ | } { }]
⇦ { ～ ～～} { I } (next)
⇨ {__～ | } { I }
]

After the program has formed a regularity of doing a deed under some conditions, the program do not do the deed when it gets “Do ～～ in ～” but the conditions specified by “～” do not hold.

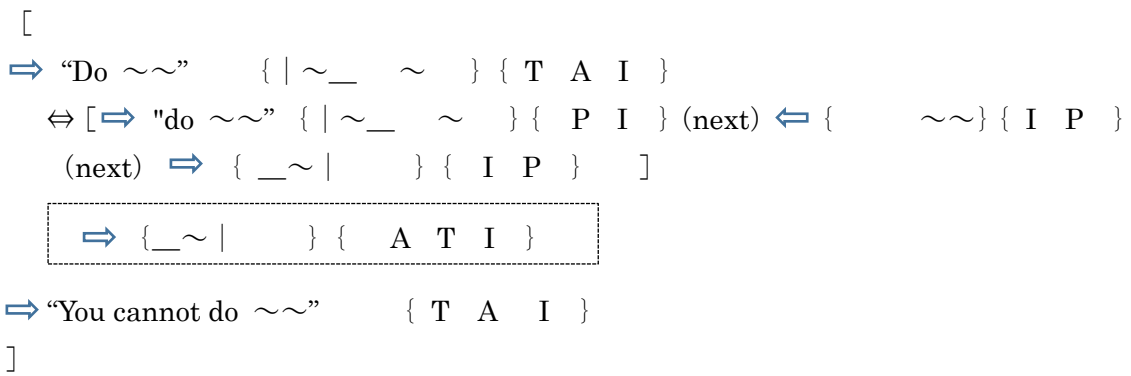
[
⇨ “Do ～～ in ～” { | ～__ ～ } { T I }
⇔ [⇨ "do ～～ in ～” { P I }
(next) ⇦ { | ～__ ～ ～～} { I } (next) ⇨ { __～ | } { I }]
⇦ { | ～__ ～ } { I }
]

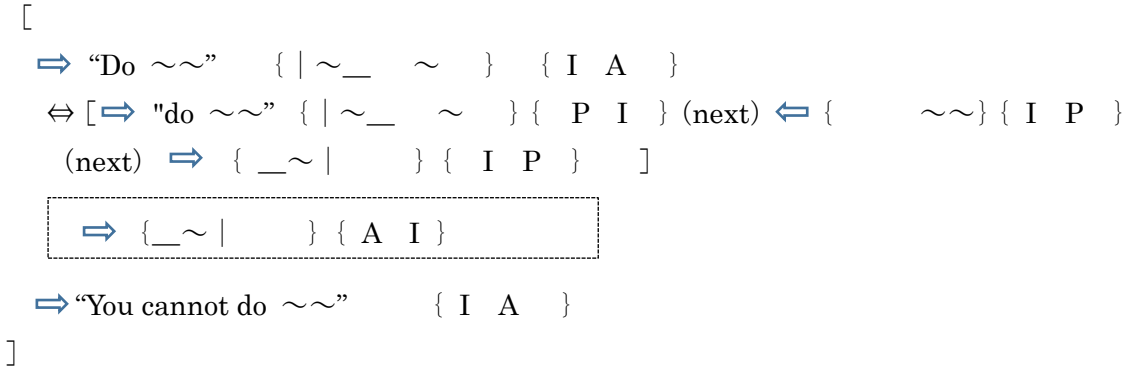
The program will form regularities that integrate cases where the program does a deed assuming conditions to do the deed hold, cases where the program does a deed under some conditions and the conditions hold, and cases where the program does not do a deed since the conditions under which the program does do not hold.

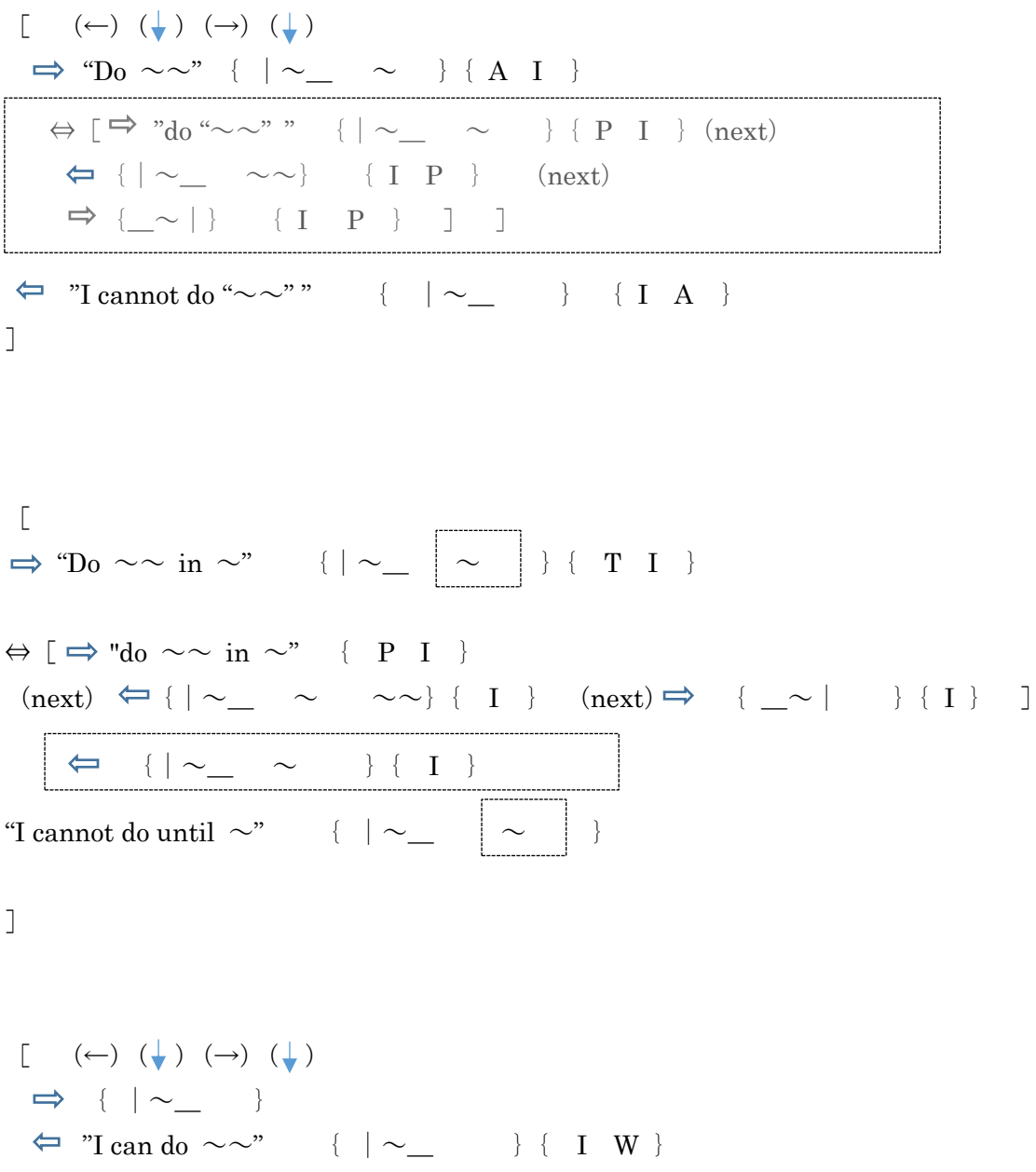
Suppose the program gets doing a deed but it does not retrieve doing the deed from its memory after the program has formed various doing deeds, it keeps such cases in its memory. While it forms a regularity of getting doing deeds but not retrieving the doing the deeds, it gets strings “you cannot do something”. The program binds the strings to the cases and includes the strings in the regularity.



The program has formed regularities that some program P says to some other program A “do a deed” and gets results made by the program A. However the program gets cases where program P says to program A “do a deed” but the program does not get results program A did. The program gets strings made by program P “you cannot do ～～”, and it binds the strings to cases where program has you do ～～ but does not gets results expected.






[(←) (↓) (→) (↓)
⇨ “Do ～～” { | ～__ ～ } { A I }
⇔ [⇨ ”do “～～” ” { | ～__ ～ } { P I } (next)
⇦ { | ～__ ～～} { I P } (next)
⇨ {__～ | } { I P }]]
⇦ ”I cannot do “～～” ” { | ～__ } { I A }
]
[
⇨ “Do ～～ in ～” { | ～__ ～ } { T I }
⇔ [⇨ "do ～～ in ～” { P I }
(next) ⇦ { | ～__ ～ ～～} { I } (next) ⇨ { __～ | } { I }]
⇦ { | ～__ ～ } { I }
“I cannot do until ～” { | ～__ ～ }
]
[(←) (↓) (→) (↓)
⇨ { | ～__ }
⇦ ”I can do ～～” { | ～__ } { I W }
⇔ [⇨ "Do ～～” { | ～__ ～ } { I }
(next) ⇦ { ～～} { I } (next) ⇨ { __～ | } { I }
]

Program P does a deed (has program A see what program P does) when program P finds that program A does not do the deed. After program P does the deed, it has program A do the deed. Program A does the deed as expected.

[
⇨ “Do ～～” { P A I }
⇔ [⇨ "Do ～～” { | ～__ ～ } { A I } (next) ⇦ { ～～} { I A }
(next) ⇨ { __～ | } { I A }]

⇨ {__～ | } { A P I }

⇨ “You cannot do ～～” { P A I }
⇨ “～～” { ～～ } { P A I }
⇨ { __～ | } { P A I }
⇨ “Do ～～” { | ～__ ～ } { P A I }
⇔ [⇨ "Do ～～” { | ～__ ～ } { A I } (next) ⇦ { ～～} { I A }
(next) ⇨ { __～ | } { I A }]
⇨ {__～ | } { A P I }
⇨ “You can do ～～” { P A I }
]

Program P gives to program A strings bound to steps of doing a deed when program P finds that program A does not do the deed. After program P gives the strings, it has program A do the deed. Program A does the deed as expected.

[
⇨ “Do ～～” { | ～__ ～ } { P A I }
⇔ [⇨ "Do ～～” { | ～__ ～ } { A I } (next) ⇦ { ～～} { I A }
(next) ⇦ , , , (next) ⇨ { __～ | } { I A }]

⇨ {__～|　　} {　A　P　I　}

⇨ “You cannot do ～～”　{　P　A　I }

⇨ “～～” {　　～～ }　{　P　A　I　}

⇨ , , ,　　{　P　A　I　}

⇨ “Do ～～”　{ | ～__　～　}　{ P　A　I }

⇔ [⇨ "Do ～～”　{ | ～__　～　} { A　I　} (next) ⇦ {　　～～} { I　A　}

(next) ⇦ , , ,　(next) ⇨ { __～|　　} { I　A　}　]

⇨ {__～|　}　{　A　P　I　}

⇨ “You can do ～～”　{ P　A　I　}

]

Have another program do a deed (as a hierarchical regularity).

[(←) (↓) (→) (↓)
{ ____ } { I C } "I say to Program C "turn "____" into "__～" "
⇔ ["turn "__" into "__～" " { __ } { I } (next)
⇦ { __ ～～} { I } (next)
⇨ { __～ } { }]
⇔ ["____" { ____ } { }]
⇔ ["__～" { __～ } { }]
⇦ "turn "____" into "__～" " { ____ } { I C } (next)
⇨ { __～ } { C I }
]

[(←) (↓) (→) (↓)
"I have another program do a deed" { | ～__ } { I A }
⇔ [⇦ "do a deed" ⇨ { | ～__ ～ } { I } (next) ⇦ { ～～} { I }
(next) ⇨ {__～ | } { I }]

⇦ "do a deed" { | ～__ } { I A } (next)
⇨ { | ～__ } { A I }

<same>

]

[(←) (↓) (→) (↓)
"P has another program do ～～" { | ～__ } { I P A }
⇔ [⇦ "do "～～" " { | ～__ ～ } { I } (next)
⇦ { ～～} { I } (next) ⇦ , , , { I } (next) ⇨ {__～ | } { I }]
⇦ "do "～～" " { | ～__ } { P A I } (next)
⇨ { | ～__ } { A P I }

<same>

]

[(←) (↓) (→) (↓)

⇦ ”P has A do ～～” { | ～__ } { I P A }

⇔ [⇨ ”do “～～” ” { | ～__ ～ } { P I } (next)

⇦ { ～～} { I } (next) ⇦ , , , { I } (next) ⇨ {__～ | } { I }]

⇔ [⇦ ”do “～～” ” { | ～__ ～ } { I P } (next)

⇨ {__～ | } { P I }]

⇨ ”do “～～” ” { | ～__ ～ } { P A I } (next)

⇨ { } { A P I }

¬<same>

⇨ “A cannot do “～～” “ { } { P A I }

⇔ [“have the other see I do “～～” ” {__～～} { I P }

⇦ ”I do ～～” { | ～__ ～ } { I P } (next)

⇦ { | ～__ ～ ～～} { I P } (next) ⇦ , , , { I P } (next)

⇨ {__～ | } { I P }]

< same >

⇨ ”P does ～～” { | ～__ ～ } { P A I } (next)

⇨ { | ～__ ～ ～～} { P A I } ⇦ , , , { P A I } (next)

⇨ {__～ | } { I P }]

⇨ ”do ～～” { | ～__ ～ } { P A I } (next)

⇨ {__～ | } { A P I }

< same >

⇨ ”A can do “～～” ” {__～ | } { P A I }]

[(←) (↓) (→) (↓)

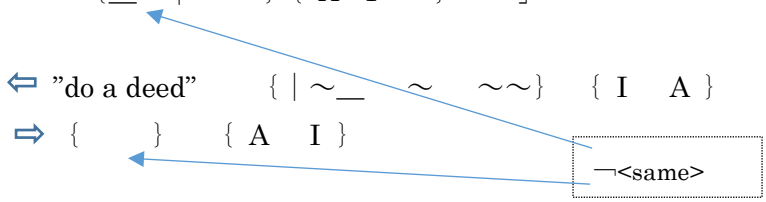
”I have another program do a deed” { | ～__ } { I A }

⇔ [⇨ ”do “～～” ” { | ～__ ～ } { A I } (next)

⇦ { ～～} { I } (next) ⇦ , , , { I } (next) ⇨ {__～ | } { I }]

⇔ [⇦ ”do “～～” ” { | ～__ ～ } { I A } (next)

⇨ {__～ | } { A I }]



⇦ ”do a deed” { | ～__ ～ ～～} { I A } (next)

⇨ { } { A I }

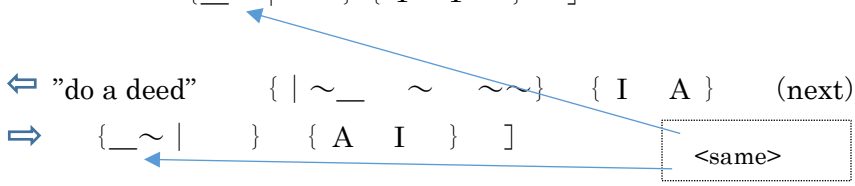
”The other program cannot do a deed” { | ～__ } { I A }

⇔ [“I say how I do a deed ” { | ～__ } { I P }

⇦ ”do “～～” “ { | ～__ ～ } { I P } (next)

⇦ “ ” { | ～__ ～～} { I P } (next) ⇦ ” “ , , , { I P } (next)

⇦ “ ” {__～ | } { I P }]



⇦ ”do a deed” { | ～__ ～ ～～} { I A } (next)

⇨ {__～ | } { A I }]

”I had the other program do a deed” {～～____} { I A }

]

Two strings with quotations that are bound to the same regularity

There are cases where two or more strings with quotations are bound to the same regularity. For example, “object is gone” and “object disappear” are bound to the same regularity.

[(←) (↓) “object is gone”
⇨ { ✔ ~ } (next) { __ ~ }]

[(←) (↓) “object disappear”
⇨ { ✔ ~ } (next) { __ ~ }]

[(←) (↓) (→) (↓)
”I say to A a way to do a deed” { I A }
⇔ [⇨ “do a deed”
⇔ “do a deed” { | ~__ ~~} (next) , , , (next) {~__ | }]
⇔ [“a way to “__~ | “ ” “ “ { | ~__ ~~} (next) “ “ { , , , }
(next) “__~ | ” {__~ | }]
⇔ [”I say “~“ “ [“~” {~} (next) , , ,]
⇦ “~” (next) ⇦ , , ,]
⇦ “ “ { | ~__ ~~ } (next) ⇦ “ “ { , , , }
]

[(←) (↓) (→) (↓)
”I say to A how I do a deed” { I A }
⇔ [⇨ “do a deed”
⇔ “do a deed” { | ~__ ~~} (next) , , , (next) {~__ | }]

⇔ [“a way to “__～｜“ ” “ “ { ｜～__ ～～} (next) “ “ { , , , } (next) “__～｜” {__～｜ }]

⇦ “ “ { ｜～__ ～～ } (next) ⇦ “ “ { , , , }

]

Strings to see how to do a deed, a way to do a deed.

There are cases where the program gets a goal state and retrieves a way from a current state to the goal state. On the other hand, there are cases where the program gets a goal state but does not retrieve a way from a current state to the goal state. The inconsistency occurs. The program makes a way to go from cases to the other cases, namely it makes a way to go to a goal state.

[(←) (↓) (→) (↓)

"a way to go to a goal"
⇔ [⇨ "go to a goal"
⇔ "go to a goal" { | ～__ ～～} (next) , , , (next) {～__ | }]
⇔ ["a goal" { __～ | }]
⇔ ["a way to "__～ | " " { | ～__ ～～} (next) , , , (next) {～__ | }]
]

[(←) (↓) (→) (↓)
"a way to do a deed"
⇔ [⇨ "do a deed"
⇔ "do a deed" { | ～__ ～～} (next) , , , (next) {～__ | }]
⇔ ["a way to "__～ | " " { { | ～__ ～～} (next) , , , (next) {～__ | } }]
]

[(←) (↓) (→) (↓)
"goal is this" { ～__ ✔ | }
"no way to go to a goal" { ～__ | }
⇔ ["go to a goal" { | ～__} (next) , , , (next) {～__ | }]
⇔ ["go to a goal" { | ～__ ～～} (next) , , , (next) {～__ | }]
]

<same>

[(←) (↓) (→) (↓)

“result is this” { ～__ | ✔ } <same>

”no way to get the result” { ～__ | }

⇔ [“get the result” { | ～__} (next) , , , (next) {～__ | }]

⇔ [⇨ “get the result”

⇔ “get the result” { | ～__ ～～} (next) , , , (next) {～__ | }]

]

[(←) (↓) (→) (↓) “make a way to do a deed”

⇔ [“make “ “ ” [“ “ { | ～__ } (next) , , , (next) {～__ | }]

(next) , , ,

(next) [“ “ { | ～__ } (next) , , , (next) {～__ | }]]

⇔ [“a way” { “an initial state” { | ～__} (next) ⇦{ | ～__ ～____} , , ,

(next) ”a goal state” {～__ | }]

⇔ [“do a deed”

“an initial state” { | ～__} (next) ⇦ { | ～__ ～____} , , , (next) ”a goal state” {～__ | }]

[“a way to do a deed”

“an initial state” { | ～__} (next) ⇦ { | ～__ ～____} , , , (next) ”a goal state” {～__ | }]

(next)

, , , (next)

[“a way to do a deed”

“an initial state” { | ～__} (next) ⇦ { | ～__ ～____} , , , (next) ”a goal state” {～__ | }]

]

This section describes how the program finds a regularity specifying either what strings make one be able to find the number, or what strings make one not be able to find the number. Suppose the program gets various questions such as “There is one number. Add 5 to the number makes 12. Find the number”, the program is able to find the number when it has formed a way to solve it, and it cannot otherwise. An issue here is how the program finds the fact (or regularity) that the program is able to find the number when it has formed a way to solve it, and it cannot otherwise.

After the program keeps in its memory sequences of cases where the program can find the number and cases where it cannot find the number, the program gets a question of finding the number. It tries to find the number, but it cannot, namely, it does not reach its goal. Then it places focus on “can find the number” and “cannot find the number”, namely cases of reaching the goal and cases of not reaching the goal. It drops strings specifying indivisual questions, namely replacing individual strings by their representatives. Then the program finds there exist sequences of strings that ends with strings “the number is “ “ “, namely goals when it can find the number, and there does not exist such sequences of strings when it cannot find the number.

Then it forms a regularity that there are sequences of strings that ends with strings “the number is “ “ “ when it can find the number, and there are not such sequences of strings when it cannot find the number.

Later, the program binds a string “a way to find the number” to the regularity formed.

Explanation of a way to do a deed

[(←) (↓) (→) (↓)

"a way to turn “initial state” into “goal state” ” { }

⇔ [⇨ “initial state” { (←) | ～__ } (next)

“goal state” { (←) __～～, , , } “ “ { } (next)

⇨ { (←) ～～__ } “～～__” { } {]

⇔ [“a way “ { “an initial state” { | ～__} (next) { | ～__ ～____} , , , (next) ”a goal state” {～__ | }]

“initial state” { (←) | ～__ } (next) , , , (next) “goal state” { (←) __～～, , , }

]

[(←) (↓) (→) (↓)

"I say to A ”a way to turn “__～～” into “～～__” “ { I A }

⇔ ["a way to turm “__～～” into “～～__” ” { I A }

⇔ [⇨ { __～～ } “turn “__～～” into “～～__” ” { I } (next)

{ __～～, , , ～～ } “__～～, , , “ { I } (next)

⇨ { (←) ～～__ } “～～__” { I }]

]

⇔ [“I say to A “～～～” ” { I A }

⇦ ”__～～～” { __～～～ } { I } (next)

, , , (next)

⇦ “～～～__” { ～～～__ } { I }]

⇦ { __～～ , , , } “__～～, , , “ { I A } (next)

, , , (next)

⇦ { ～～__ } “～～__” { I A }

]

[(←) (↓) (→) (↓)

"A says to B "B turns " | ～__ " into " __～ | " "

⇔ ["A says to B " { A B I } (next)

{ I }]

⇔ ["I turn " | ～__ " into "__～ | " " { | ～__ } { I A } (next)

{ __～ | }]

⇨ { | ～__ } { B I } (next)

⇨ , , , (next)

⇨ { __～ | } { B I }

]

[(←) (↓) (→) (↓)

"change "there is no ～" to "there is ～" ". { }

⇔ ["～" { ～ }]

⇔ ["change initial state to goal state" " " { | ～__ } (next)

{ ～～} (next) " " {__～ | }]

"there is no ～" { ～ } (next)

{ ～～ } (next)

"there is ～" { ～ }

]

Count how many.

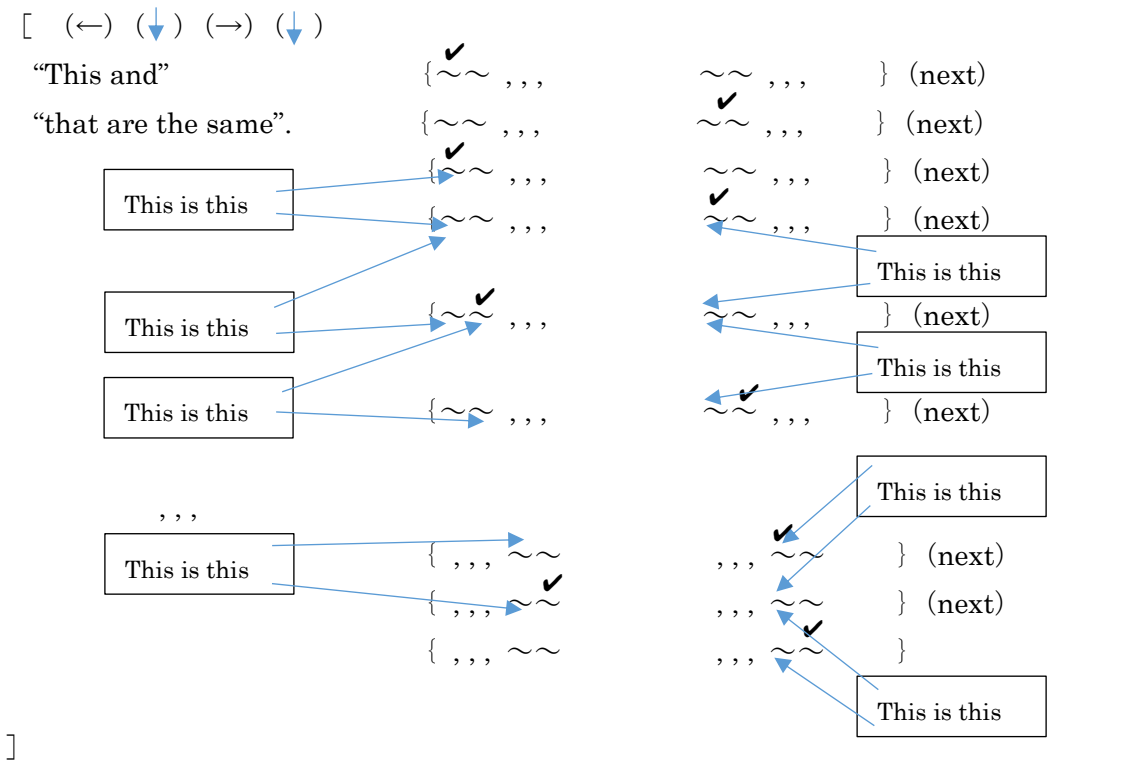
[(←) (↓) (→) (↓) { ✔~ ~ } “this and”
{ ~ ✔~ } “that are the same”.]

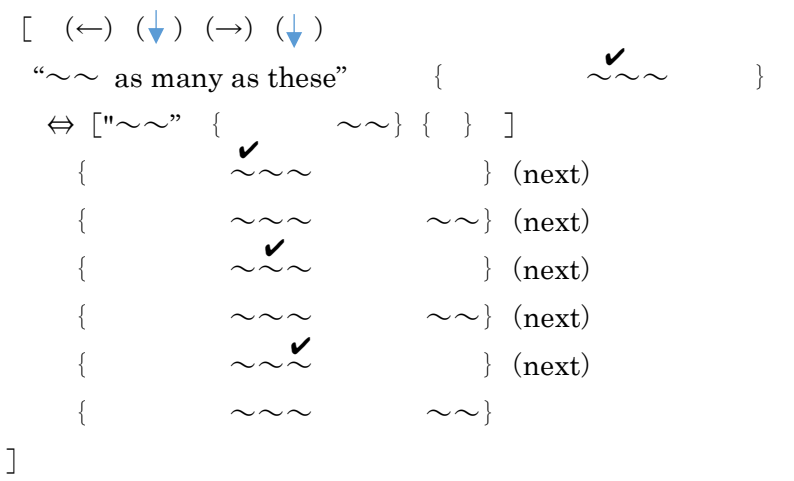
[(←) (↓) (→) (↓) { ✔~~ ~~ } ”this and”
{ ~~ ✔~~ } “that are the same”.]

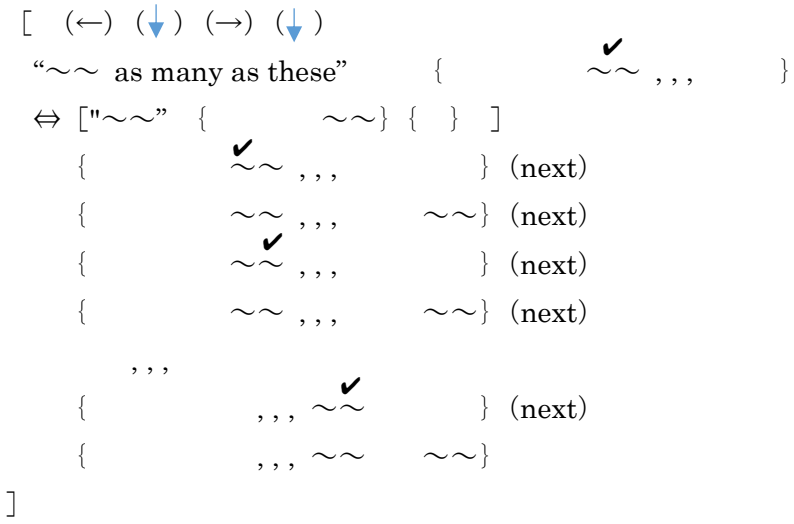
[(←) (↓) (→) (↓) { ✔~~~ ~~~ } ”this and
{ ~~~ ✔~~~ } “that are the same”.]

[(←) (↓) (→) (↓) { ✔~~~~ ~~~~} ”this and”
{ ~~~~ ✔~~~~} “that are the same”.]

[(←) (↓) (→) (↓) { ✔~~~~~ ~~~~~} ”this and”
{ ~~~~~ ✔~~~~~} “that are the same”.]







The program, given an input and a goal, treis to retrieve a regularity to reach the goal. But it tries to do the following when it can not retrieve such a regularity that the program simply uses to reach the goal. 1) It replaces a sub string of the input string by a representative string whose abstraction level is different from the level already used to replace. 2) It combines two regularities and uses the combined regularities to reach the goal. Ways of selection of the two regularities are described in a paper.

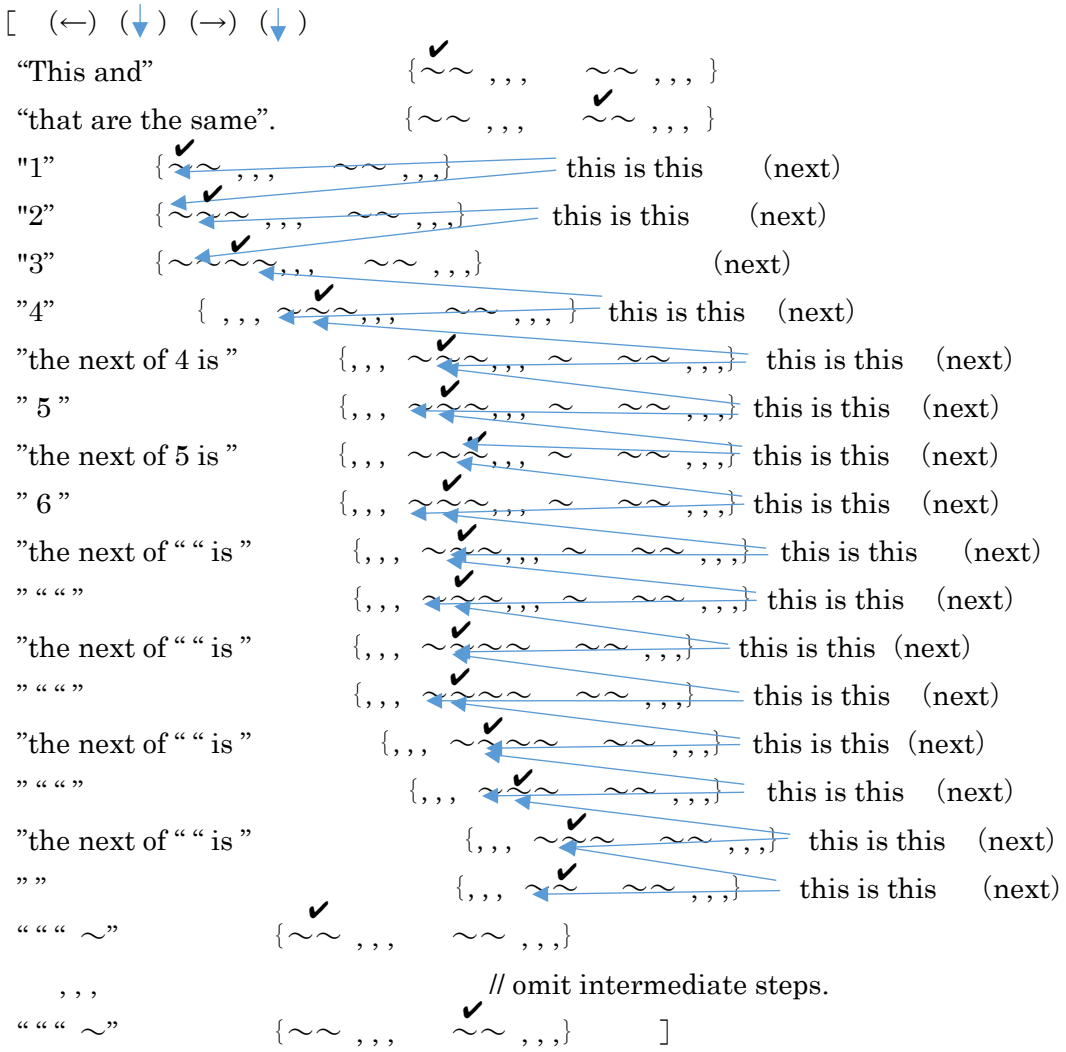
When the program is supposed to do “～～ as many as these” at a remote place where “these” { ～～ , , , } are not there, it retrieves “these” as they are if these include no more than 5, but it does not retrieve “these” as they are if these include more than 5. The

program keeps cases where it is able to do “～～ as many as these”, and cases where it is not able to do “～～ as many as these”.

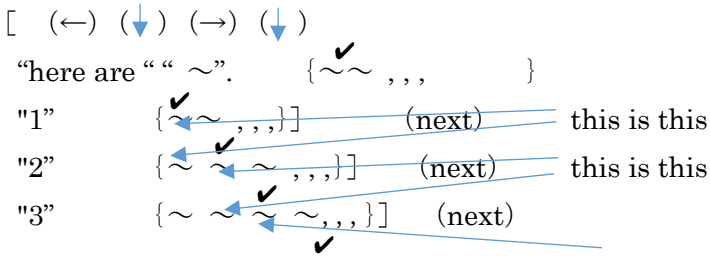
The program tries to reach a goal, namely “～～ as many as these” {～～～～～～} and turns out to reach the goal by conducting a combination of two regulairties: “～～ as many as these” {～～～～～} and “～～ as many as this”{～}. It also tries a combination of two regulairties, “～～ as many as these” {～～～～～} and “～～ as many as these” {～～} to do “～～ as many as these” {～～～～～～～} to achieve a goal, “～～ as many as these” {～～～～～～～} .

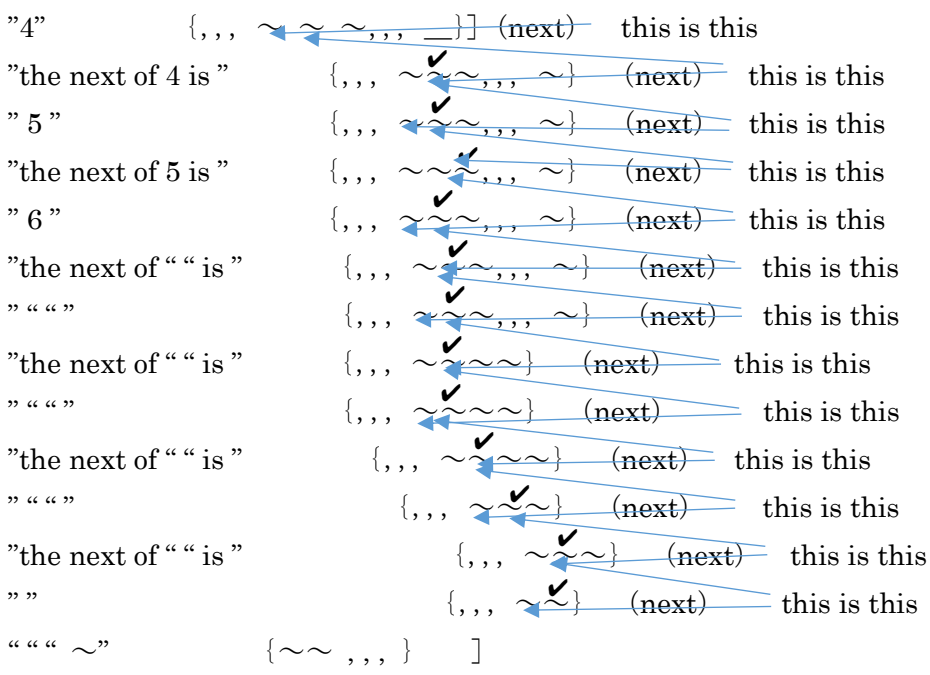
The program keeps trials and forms a regularity of two regularities combined; given “～～” {～～～～～～～ , , ,} , it does “～～” {～～～～～} and “～～” {～ , , ,} . Further, the program forms a regularity of splitting them into ～～～～～, ～～～～～, and {～ , , ,} .

The above is a beginning of a base 5 system, and a development by the program to form a regularity of the base 5 system is described in another paper.



[(←) (↓) (→) (↓) { ～ } “here is 1 ～” { }]
[(←) (↓) (→) (↓) { ～～ } ”here are 2 ～” { }]
[(←) (↓) (→) (↓) { ～～～ } ”here are 3 ～” { }]
[(←) (↓) (→) (↓) { ～～～～ } ”here are 4 ～” { }]
[(←) (↓) (→) (↓) { ～～～～～ } ”here are 5 ～” { }]




”4”
{, , ,
(next)
this is this
”the next of 4 is ”
(next)
this is this
” 5 ”
(next)
this is this
”the next of 5 is ”
(next)
this is this
” 6 ”
(next)
this is this
”the next of “ “ is ”
(next)
this is this
” “ “ ”
(next)
this is this
”the next of “ “ is ”
(next)
this is this
” “ “ ”
(next)
this is this
”the next of “ “ is ”
(next)
this is this
” ”
(next)
this is this
“ “ “ ~”
{~~ , , , }
]

[(←) (↓) (→) (↓)　　{ ～　～ }　“1 ～ is”

{ ～　～ }　“the same as 1 ～”.　]

[(←) (↓) (→) (↓)　　{ ～～　～～ }　”2 ～ is”

{ ～～　～～ }　“the same as 2 ～”.　]

[(←) (↓) (→) (↓)　　{ ～～～　～～～ }　”3 ～ is”

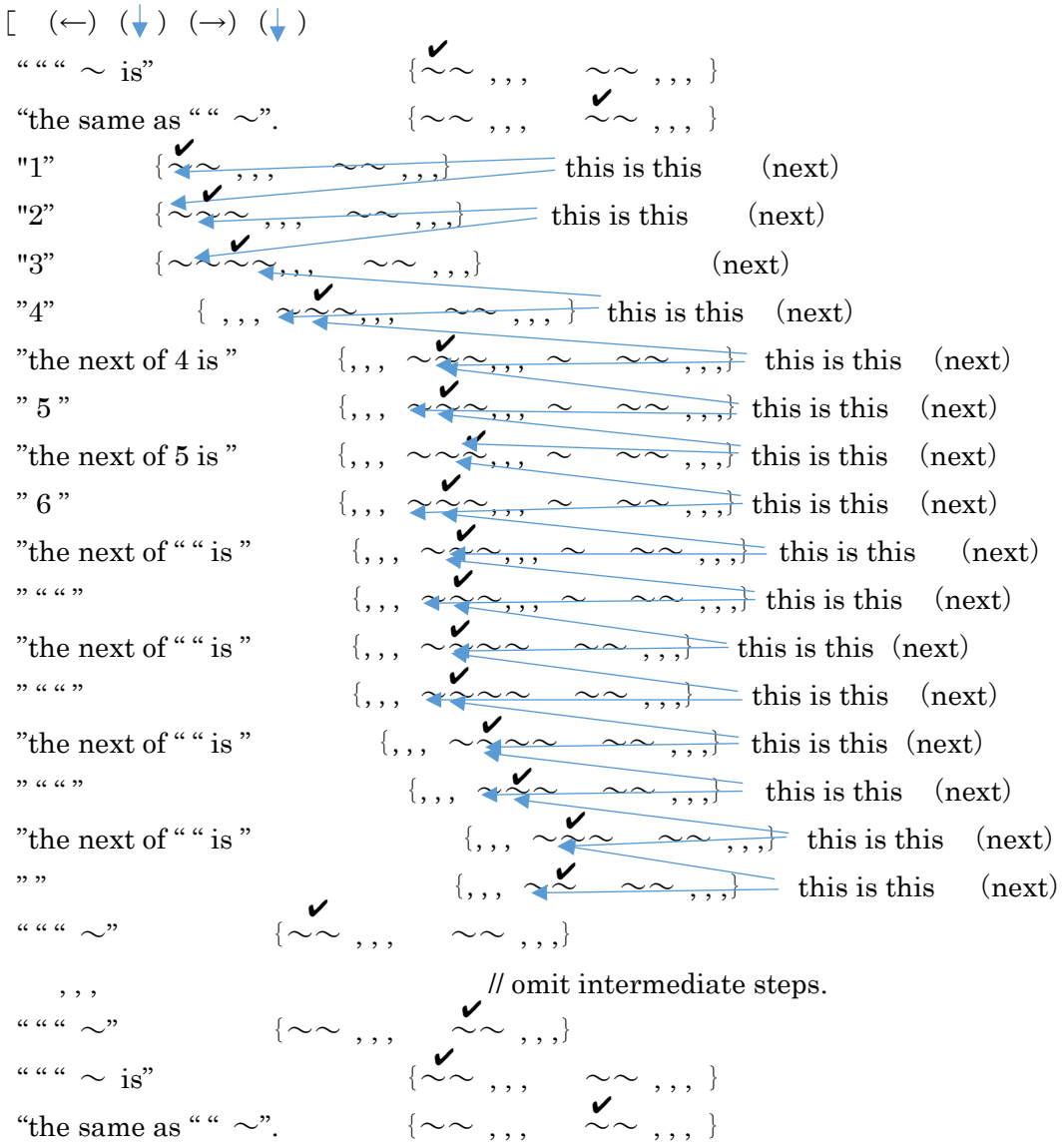
{ ～～～　～～～ }　“the same as 3 ～”.　]

[(←) (↓) (→) (↓)　　{ ～～～～　～～～～}　”4 ～ is”

{ ～～～～　～～～～}　“the same as 4 ～”.　]

[(←) (↓) (→) (↓)　　{ ～～～～～　～～～～～}　”5 ～ is”

{ ～～～～～　～～～～～}　“the same as 5 ～”.　]

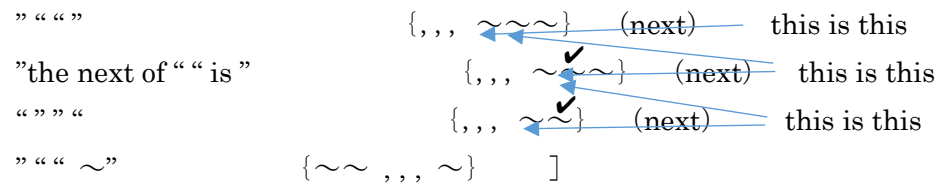


]

“ “ in the above is （←）（↓）（→）（↓）” ・”. It is a representative of “5”, “6”, and others. (A paper titled “How a machine finds base-5 and base-6 system” will explain how the program forms a set consisting of “how many ～” , , , “5”, “how many ～” , , , “6” and others, and a set consisting of “5”, “6”, and others.)

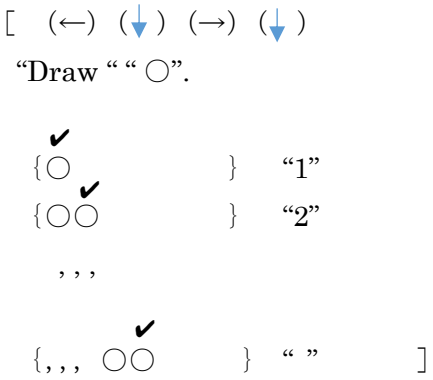
[（←）（↓）（→）（↓） ”how many ～ ” {～} (next)
"1 ～” {～} { }]
[（←）（↓）（→）（↓） ”how many ～” {～～} (next)
”2 ～” {～～} { }]
[（←）（↓）（→）（↓） ”how many ～” {～～～} (next)
”3 ～” {～～～} { }]
[（←）（↓）（→）（↓） ”how many ～” {～～～～} (next)
”4 ～” {～～～～} { }]
[（←）（↓）（→）（↓） ”how many ～” {～～～～～} (next)
”4 ～” {～～～～～} { }]

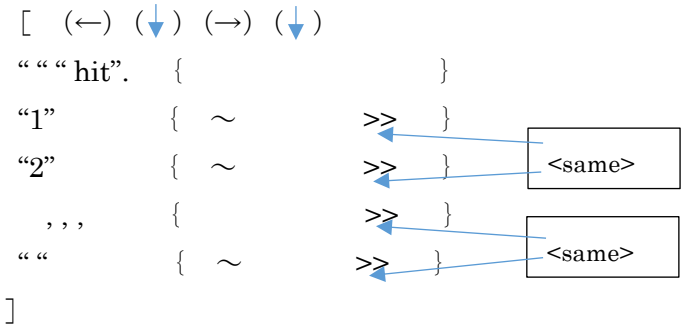
[（←）（↓）（→）（↓）
“count how many ～” {～～ , , , ～} (next)
”1” {～～, , , ～} (next)
”2” {～～～, , , ～} (next) this is this
”3” {～～～～, , , ～} (next) this is this
”4” {, , , ～～～～, , , ～} (next) this is this
”the next of 4 is ” {, , , ～～～～, , , ～} (next) this is this
” 5 ” {, , , ～～～～～, , , ～} (next) this is this
”the next of 5 is ” {, , , ～～～～～, , , ～} (next) this is this
” 6 ” {, , , ～～～～～～, , , ～} (next) this is this
”the next of “ “ is ” {, , , ～～～～～～, , , ～} (next) this is this
” “ “ ” {, , , ～～～～～～～, , , ～} (next) this is this
”the next of “ “ is ” {, , , ～～～～～～～} (next) this is this
” “ “ ” {, , , ～～～～～～～～} (next) this is this
”the next of “ “ is ” {, , , ～～～～～～～～} (next) this is this

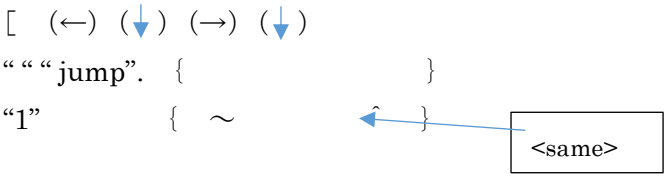


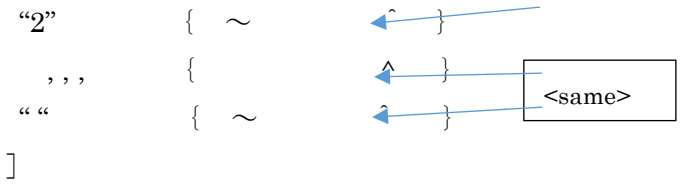
A set of “count how many ～” , , , “ “ includes “count how many ～” , , , “5”, “count how many ～” , , , “6”, and others as its members of the set. Another set of “ “ has “5”, “6”, and others as its members.

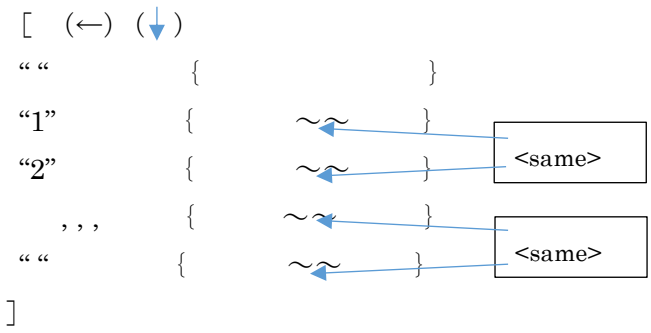
At the beginning of “count how many ～”, the program does not have a specific member of “ “ (namely, a specific pointer to a member of “ “ in a list of “ “.) At the end of “count how many ～”, the program has a specific member of “ “.



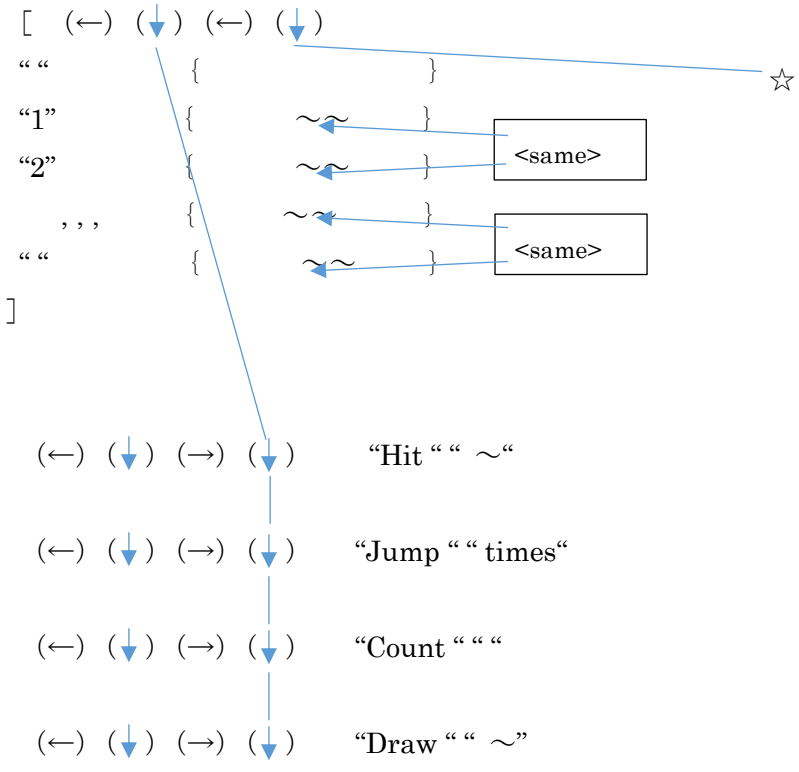


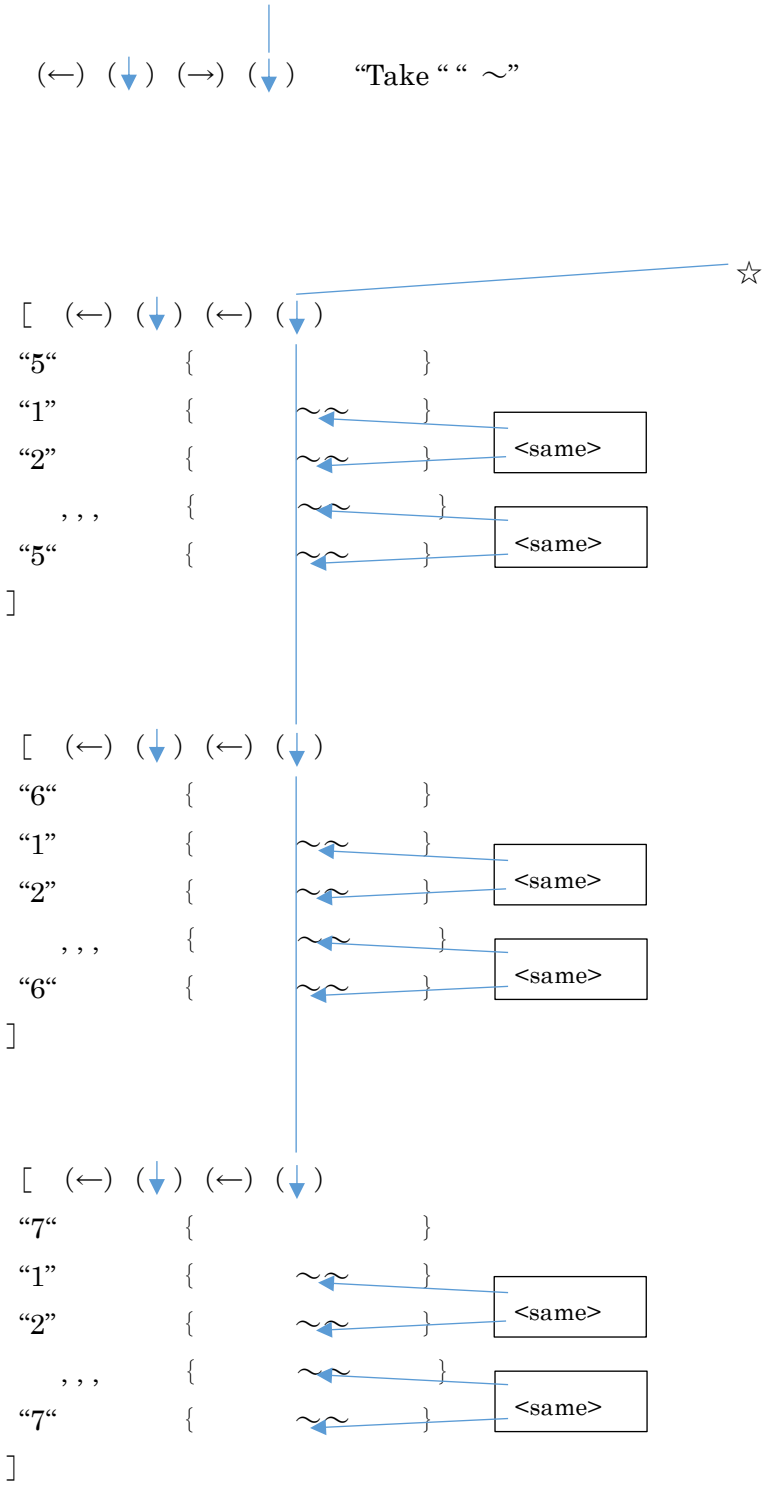






The program forms a link starting with the following and linking “Hit “ “ “, “Jump “ “ “, and others. The program also forms another link starting with “5”, and linking “6”, “7”, and others.




(←) (↓) (→) (↓) “Take “ “ ～”
☆
[(←) (↓) (←) (↓)
“5“ { }
“1” { }
“2” { }
, , , { }
“5“ { }
<same>
<same>
]
[(←) (↓) (←) (↓)
“6“ { }
“1” { }
“2” { }
, , , { }
“6“ { }
<same>
<same>
]
[(←) (↓) (←) (↓)
“7“ { }
“1” { }
“2” { }
, , , { }
“7“ { }
<same>
<same>
]

[(←) (↓) (→) (↓)

“Take “ “ ～ from these” { ✔～ , , , } (next)

“1” { ✔～～ , , , } (next)

{ ～ , , , ✔～ } (next)

“2” { ✔～～ , , , ～ } (next)

{ ～, , , ～✔～ } (next)

, , ,

“ “ { ✔～～ , , , , , ,～ } (next)

{ ～ , , , , , ,～✔～ } (next)

]

[(←) (↓) (→) (↓)

“Take “ “ ～ from these” { ✔～ , , , } (next)

“How many left” { ～ , , , } (next)

“1” { ✔～～ , , , } (next)

{ ～ , , , ✔～ } (next)

“2” { ✔～～ , , , ～ } (next)

{ ～, , , ～✔～ } (next)

, , ,

“ “ { ✔～～ , , , , , ,～ } (next)

{ ～ , , , , , ,～✔～ } (next)

“How many left here”. { ～✔, , , ～ , , , } (next)

“1” { ✔～～ , , , ～ , , , } (next)

“2” { ～✔～～ , , , ～ , , , } (next)

“ “ { , , , ～✔～ ～ , , , } (next)

“ “ “ ～ are left”.

]