A program that acquires how to solve problems in mathematics

MACHIKO FUJIWARA, KENZO IWAMA
Engicom Corporation
2-16-6 Higashi Jujo Kita-ku Tokyo 114-0001
JAPAN
m-fujiwara@engicom.co.jp, k-iwama@engicom.co.jp

Abstract:- In mathematics, a sequence of sentences describes how to solve a problem; for instance, sentences, "Calculate the least common multiple of 10 and 15. Firstly divide 10 and 15 by 2. Results are '5' and 'cannot divide'. '5' and 'cannot divide' are not 1 and 1. Divide 5 and 15 by 2. Results are 'cannot divide' and 'cannot divide'." and so on, describe how to calculate the least common multiple of 10 and 15. While example sequences of sentences are given to our program, the program transforms the example sequences to generate a procedure, *pG*, to solve a problem. For instance, a generated procedure, *pGp*, checks if a given number is a prime number, another procedure, *pGc*, calculates the least common multiple of any two numbers, and another procedure, *pGf*, adds given two fractions. This paper explains our program that generates procedures, one after another, each of which solves one mathematical problem. The paper also argues, as a result of generating a procedure, *pG*, the meaning of a sentence (or sentences) is represented in the generated procedure, *pG*.

Key-words:- Acquisition of how to solve problems, Generation of programs from examples, Meaning of a sentence.

1 Introduction

Philosophers have argued that the meaning of natural language construction can be identified with a representation of its denotation (e.g., [2]). In [4], the author states "their thought has been developed in the direction of identifying the meaning of an expression with a procedure for finding its extension". The meaning of a sentence is a procedure of computing the truth or the falsity of the sentence (e.g., [10]).

In psychology, linguistics, and cognitive science, acquiring word meanings inductively has been studied much (e.g., [8]), and inductive methods have been developed and applied for acquiring meanings of words in [13]. Cross situation is a key to acquire word-to-meaning mappings.

Studies in machine learning have developed frameworks for learning and various techniques to make a machine acquire concepts. The learnability in [15] has shown a learning machine that uses a protocol to get inputs, members of a concept, and applies a deduction procedure to have a program that recognizes the concept. The procedure deduces a program (logical expression) from a general program in a way that the truth of the expression is the same as that of a member taken from the concept during a learning period. After the leaning, the truth of the logical expression becomes the same as that of a member taken from the concept. In a decision tree model, a tree node has a logical expression, and the value of the expression on an input decides which branch the input should go. General logical expressions at nodes are set specific so that inputs go down branches as the inputs specify during learning period (e.g., [10]). After the learning, the tree with the logical expressions is used as a decision tree (or a classification tree). These classical studies have been applied in various machine learning programs (see [9]). For instance,

behavioral cloning aims at inducing a model mapping inputs to outputs from pairs of inputs and outputs given by a person (e.g., [1]). Through procedural learning [16] forms a model of percepts and actions, and to make a machine imitate human skills, and [7] describes a method of learning control programs.

Inductive logic programming by [11] integrates a model of inductive processes into a framework of logic programming. Applying the framework, [3] proposes a model of vocabulary acquisition by children. After its formulation of ILP, studies on inductive synthesis of programs have been revived. Within a limited domain, [6] and [12] in Inductive Learning represent meanings conveyed in example pairs of inputs and outputs by synthesizing programs. Although their inputs are limited to data structures, their synthesized programs correctly guess what outputs would be, given inputs new to the programs. Synthesizing, namely combination and serialization, of initial programs is a way of acquiring meanings conveyed in the examples.

To the knowledge of the authors no study has been conducted to develop a program that generates procedures to check the truth or the falsity of expressions and sentences in mathematics. A reason behind this seems that it is simple for a human to write procedures, in a conventional programming language, to check the truth of each sentence such as “7 is a prime number”.

However, it seems worth developing a program that generates procedures to check the truth of a sentence: “7 is a prime number”, “60 is the least common multiple of 15 and 20”, “1 / 3 + 1 / 5 = 8 / 15”, “3 x (5 – 2) – 4 = 5”, “100X + 50 = 350 and X = 3”, and others. Inputs to the program are example sentences, and the sentences are similar to those that a teacher gives students at school. Fig. 1 shows samples of inputs. Techniques for developing such a program may be useful for representing denotation of a mathematical sentence, and for constructing an intelligent machine. Researchers may get hints how students develop denotation of a mathematical sentence. The next section describes our approach to developing the program and assumptions on which the program[1] is developed.

2 Approach

2.1 Example sequences and a program

One writes example sequences of sentences that describe how to solve a mathematical problem, and gives them to our program. A way of writing the example sequences is similar to those a teacher explains to children how to solve the mathematical problem.

Is 6 a prime number? Firstly divide 6 by 2. The result is 3. No, 6 is not a prime number.	Is 7 a prime number? Firstly divide 7 by 2. Cannot be divided. The next of 2 is 3. 3 is the same as 7. No. Divide 7 by 3. Cannot be divided. The next of 3 is 4. 4 is the same as 7. No. Divide 7 by 4. Cannot be divided. (omitted.) The next of 6 is 7. 7 is the same as 7. Yes. 7 is a prime number.

Fig. 1: Example sequences of checking if a given number is a prime number.

The one is required to consider orders of giving example sentences to a program. For instance the one gives the program examples of adding two

[1] On request for a program, the authors will send a source program of their trial to the requester.

fractions after the one has let the program acquire how to calculate the least common multiple of two integers since the addition of the two fractions uses the calculation of the least common multiple.

The one develops a program, *pI*, that generates a procedure, *pG*, by generalizing example sequences of sentences. Given an instance of the problem new to the program, *pI*, the program uses the generated procedure, *pG*, and solves the instance of the problem. Fig. 2 illustrates a procedure generated by the program that checks if a given number is a prime number.

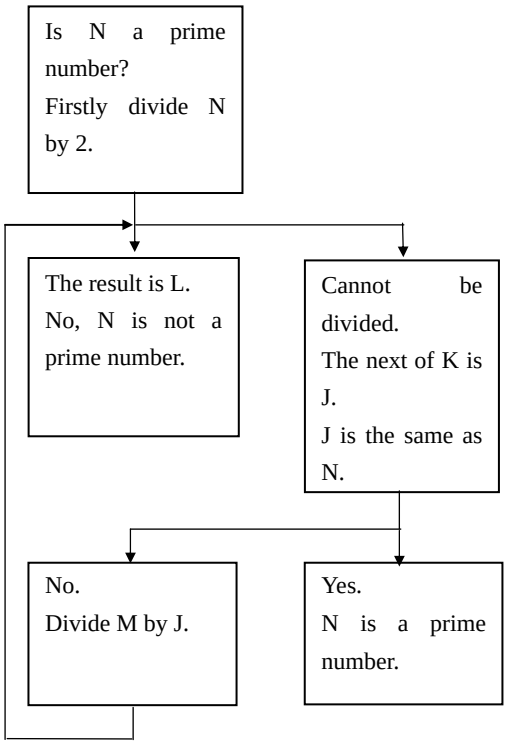


Fig. 2: A procedure generated by our program that checks if a given number is a prime number.

2.2 Assumptions

We assume the following when one gives example sentences to our program:

1) Example sentences use the same words in the same word orders. The sentences are separated by new lines, and the words are separated by blanks when one gives the program example sentences.
2) The program has already acquired how to solve some basic problems for the sake of practicality. For example, the program is assumed to have procedures of counting, basic arithmetic operations such as addition of integers, and of distinguishing numerals from other words[2].
3) The program has acquired how to use memos; storing calculated values temporarily and retrieving them later from the memos.
4) The program has acquired string functions such as string matching, inserting, and deleting.
5) The program has acquired how to detect explicit variables such as "x" in a sentence "100x + 50 = 350".

2.3 Example sequences of sentences

One writes test sets of example sequences and gives the program, one test after another, to see how the program works. The following is a list of problems without using explicit variables:

1) Is 7 a prime number?
2) The next prime number of 3 is.
3) Factorize 12.
4) Least common multiple of 10 and 15 is.
5) 3 / 16 + 7 / 12 is.
6) 2 x (5 – 3) – 3 is.
7) Greatest common factor of 16 and 24 is.
8) Find 2 / 3 of 57.
9) Change 11 / 4 to a mixed number.
10) Cancel down 9 / 12 to its lowest terms.
11) What percent of 39 is 9?
12) 30% of what number is 15?
13) How much 4 apple and 1 basket is when 1 piece of apple is 100 and 1 piece of basket is 150. (Ignore plural s and articles).

The following is a list of problems with using

[2] Iwama [5] explains how a program acquires a concept of counting from examples. Our program may start with acquiring counting instead of assuming that the program has the procedure of counting. But the complexity will grow if we start with acquiring counting.

explicit variables:

1) 100x + 50 = 350. x is.
2) Apply a distributive law to x(x + y).
3) How much N apple and 1 basket is when 1 piece of apple is 100 and 1 piece of basket is 150. (Ignore plural s and articles.)
4) 3X + 4X = 7X.

It is difficult that one explains how to solve a geometric problem in a sequence of sentences without drawing lines or circles. It is also difficult to explain addition and multiplication of two matrices without writing visually arranged rows and columns of numerals and/or variables. Thus the one puts such mathematical problems out of the scope of the current article that require 2D and/or 3D writings and drawings for their explanation by examples.

3 Explanation of our program

3.1 Key functions

Our program has six key functions:

(1) One is finding relations “same” among words in an example sequence (see Fig. 3).
(2) The second is finding sentences that repeat in the example sequence and forming subsequences each of which has sentences repeating in the sequence (see Fig. 4).

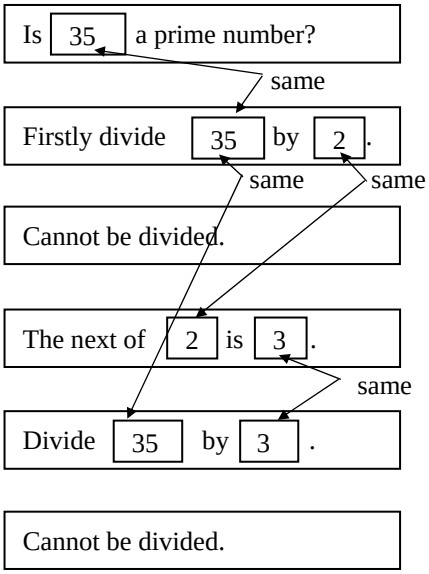


Fig. 3: Relations “same” in a sequence.

(3) The third is finding branches among subsequences (see Fig. 4 and Fig. 5).
(4) The forth is connecting the subsequences by using branch conditions so that a subsequence can be selected and executed when the subsequence holds the branch conditions (see Fig. 8).
(5) The fifth is generalizing sequences, which consist of subsequences with branches, across examples (see Fig. 7).
(6) The sixth is finding a procedure already generated by matching a sentence in a current input sequence and a head sentence of the procedure. If match, execute the procedure. Check if a sentence of the executed procedure matches a head sentence of another procedure. If match, execute it. (See Fig. 8 and Fig. 9).

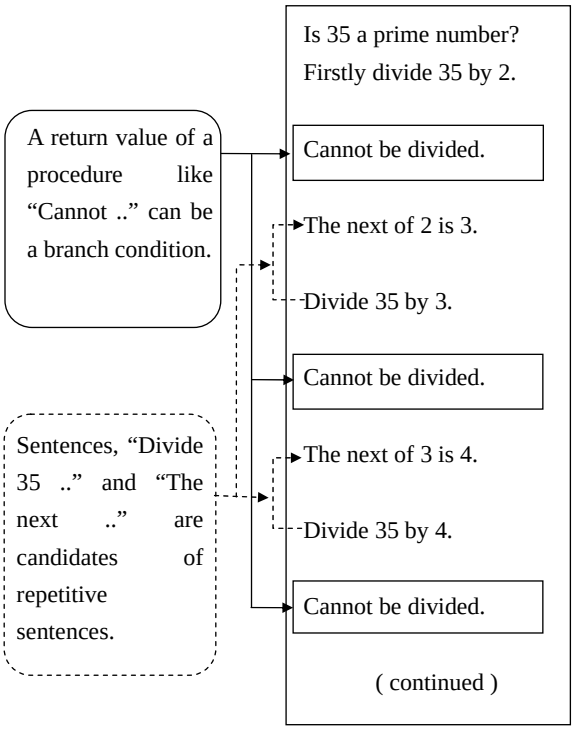


Fig. 4: Finding repetitions and branch conditions.

3.2 Explanation of our program

Our program gets example sequences, one by one, and stores them in its temporary memory. It compares a sentence of words (e.g., "divide 9 by 2") in an input with those of a sentence already stored in the memory. If they are not the same, the program stores the input sequence as a new example. The program also compares the sentence in the input and a head sentence of a procedure already generated. If match, it remembers that the sentence is the head sentence of the procedure and uses it later. When the program stores more than a certain number of example sequences, it generates a procedure out of the examples. See Fig. 6.

Our program compares numerical values in an input sequence of sentences and finds relations "same" among the numerical values (see Fig. 3). The relations "same" have mathematical meaning in the sequence, but some relations happen to hold. Later generalization, across examples, removes the relations "same" that happen to hold.

The program tries to find sentences that repeat in an input sequence by starting with a sentence that can be executed by a procedure already generated. For instance, "Divide 35 by 3" is such a sentence, and the program replaces 35 and 3 by internal variables to find the same sentence, "Divide N by J", appear in the sequence. If it repeats, the program appends sentences preceding the sentence, and checks if the sentences repeat in the sequence. Then the program forms a subsequence out of the repeating sentences (see Fig. 4).

After finding repeating sentences in the input sequence, the program folds the repeating sentences to make a subsequence of the sentences. For instance, sentences, "Cannot be divided", "The next of J is K", and "Divide N by K", repeat in the sequence, and a subsequence of these sentences is formed. See Fig. 5. Here "J", "K", and "N" are internal variables introduced by the program.

Sentences that appear only once are also formed as a subsequence. For instance, sentences "The result is 7" and "No, 35 is not a prime number", do not repeat in a sequence, and the program makes a subsequence out of the sentences.

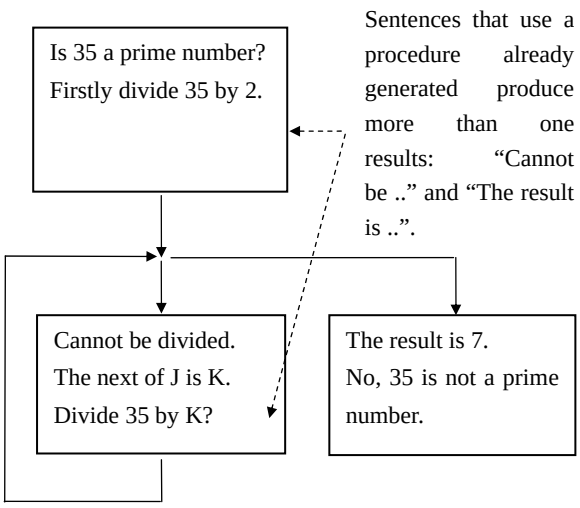


Fig. 5: Folding of repeating sentences with branches.

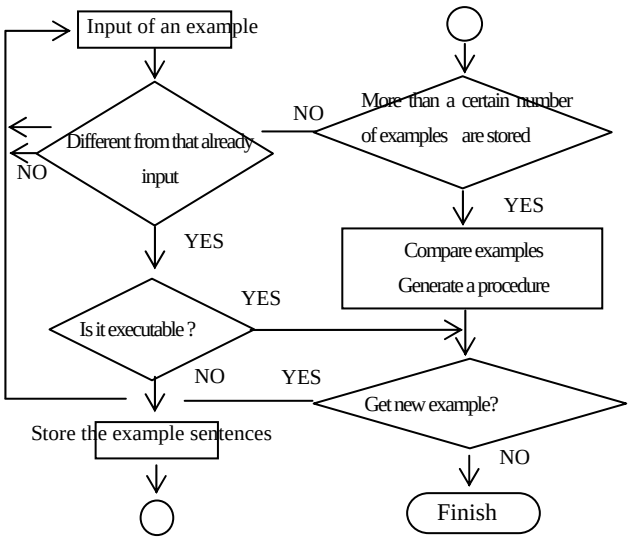


Fig. 6: A flow of generating a procedure.

A subsequence can be a branch of another subsequence, and return values from a procedure already generated can be branch conditions of subsequences. For instance, return values, “Cannot be divided” and “The result is 7” can be branch conditions of a subsequence ending with a sentence “Divide N by J”. Our program gets the possible return values of a procedure, puts subsequences in parallel that have different return values, and connects them to the subsequence ending with the sentence producing branch conditions. See Fig. 5.

After forming subsequences with branches, the program compares subsequences of one example and those of another example. The program forms a general procedure by replacing different numerical values by internal variables. See Fig. 7. Values “7”, “8” and others are replaced by internal variables. The program removes relations “same” if they do not appear across examples since the relations that appear only in one example happen to occur. The program keeps them with the procedure when they appear across the examples.

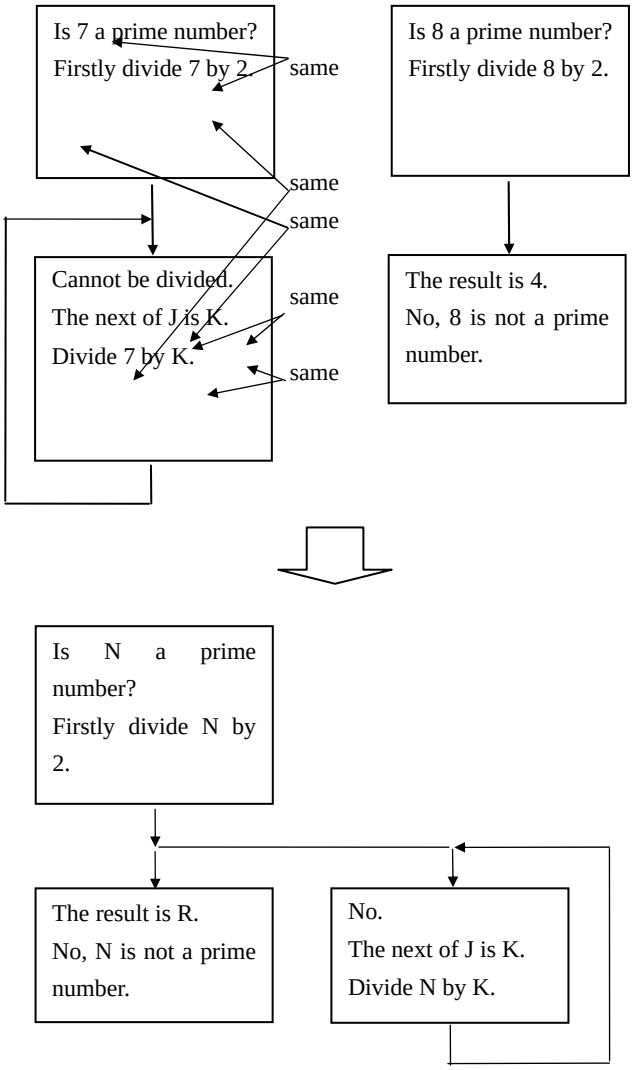


Fig. 7: Generalizing folded subsequences across examples.

Then the program locates words different across examples but the other words and the relations are the same in the sequences. Replace the words by internal word variables with relations “same” among the words. The program also finds relations “same” among explicit variables.

When the program gets a new sentence, it takes a procedure, one after another, out of its long term memory to compare the new sentence with a head sentence of the procedure. Trying to match, the program replaces numerical values in the sentence by any where the head sentence has internal variables. The program also replaces words by any where the head sentence has internal word variables.

If the two match, the program executes the procedure.

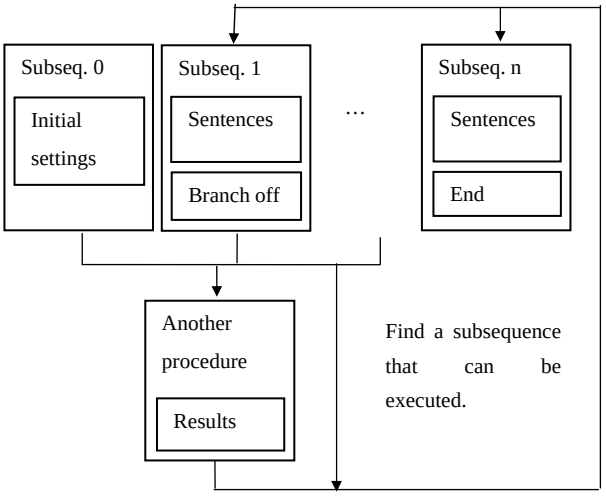


Fig. 8: A dynamic structure when a procedure is executed.

A generated procedure has usually more than one subsequence, and one subsequence may start its execution after another subsequence. The procedure may include sentences that use some other procedures already generated (see Fig. 8). To find the procedure in its memory, the program replaces numerical values in the sentence by any where the head sentence of the procedure have internal variables. The program also replaces words by any where the head sentence has internal word variables. If it finds the procedure, the program executes the procedure. See Fig. 8 and Fig. 9.

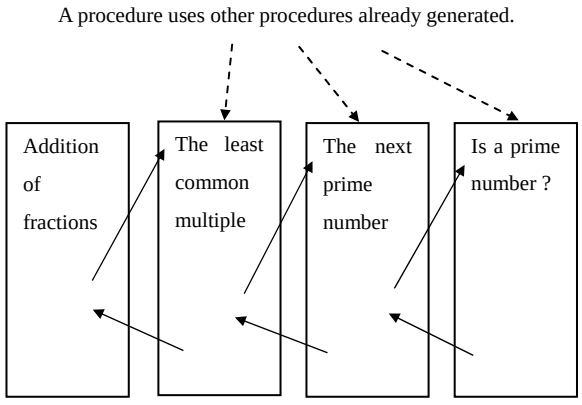


Fig. 9: An example of executing programs. When our program gets “4/15 + 7/35“, it uses procedures, such as the least common multiple of 15 and 35, that the program has already generated.

At the time of executing a procedure, our program sets specific values to internal variables by applying one of the following:

1) An input value determines the value of an internal variable.
2) A relation “same” determines the value of an internal variable.
3) A return value from a procedure determines the value of an internal variable.

For instance, one gives a sentence of adding two fractions, 4 / 15 + 7 / 35, to the program. It retrieves a procedure of adding two fractions, and puts input values, 4, 15, 7, and 35 in internal variables of the first sentence of the procedure. Relations “same” determine internal variables of the next sentence, which becomes “the least common multiple of 15 and 35”. A procedure, the least common multiple, is retrieved from its memory, and is executed to produce 105, the least common multiple of 15 and 35. The value, 105, determines the value of an internal variable of the third sentence. See Fig. 14.

3.3 Example procedures generated by our program

As described in Subsection 2.4, one prepares example sequences of sentences to test our program. Appendix illustrates example sentences of prime factorization, calculating the least common multiple, and addition of two fractions. The appendix also shows a procedure of prime factorization, and a procedure of adding two fractions. These procedures are generated by our program. We have confirmed that the program uses procedures and makes correct outputs to given instances of a problem listed in Subsection 2.3.

4 Discussion

Our program, *pI*, has generated a procedure of checking the truth of a sentence in mathematics (or solving a mathematical problem), and has represented denotation of the sentence. The sentence may be "the least common multiple of 10 and 15 is 30". The program, *pI*, has no procedure of checking it (or solving it) at the beginning, and one has not written a computer program to check it (or solve it). The program, *pI*, generates a procedure to check the truth of a given sentence after the program has received example sequences to check it, and the example sequences are similar to those used for explaining to students at school. Therefore the program, *pI*, has formed by itself a representation of meaning of a mathematical sentence, e.g., "the least common multiple of N1 and N2 is N3". Here N1, N2, and N3 are any integers.

Future works include investigation of what types of problems a current program is able to acquire how to solve. We are currently writing a series of example sequences of sentences to let the program acquire methods of mathematical induction to prove formula from examples. To use the program at school, an intelligent interface needs to be developed so that one can give example sentences to the program through the interface as a teacher writes examples on a blackboard.

It is difficult that one explains how to solve a geometric problem in a sequence of sentences without drawing lines and circles. The one usually relies on spatial relations between the lines and/or circles to explain the geometric problem. It is also difficult to explain addition and multiplication of two matrices without writing visually arranged rows and columns of numerals and/or variables. Thus a serious issue is to introduce methods of extracting and generalizing spatial relations among geometric elements such as lines and circles.

References:

[1] M. Bain, and C. Sammute, A framework for behavioral cloning, In K. Furukawa, D. Michie, and S. Muggleton, (Eds.), *Machine Intelligence 15, Intelligent Agents*, pp. 103-129. Oxford Univ. Press, 1999.

[2] G. Frege, Über Sinn und Bedeutung. *Zeitschrift fur Philosophie und Philosophishe Kritik*, Vol. 100, 1892, pp. 25-50.

[3] K. Furukawa, I. Kobayashi, T, Ozaki, and M. Imai, A Model of Children's Vocabulary, Acquisition Using Inductive Logic Programming. *LNCS* Vol. 1721. Springer, 1999.

[4] N. Gierasimczuk, The problem of learning the semantics of quantifiers, *LNAI*, Vol. 4363/2007, pp. 117-126. Springer, 2007.

[5] K. Iwama, A robotic program that acquires concepts and begins introspection, *NueroQuantology*, Vol. 4, No. 4, 2006, pp. 321-328.

[6] E. Kitzelmann, and U. Schmid, Inductive synthesis of functional programs: an explanation based generalization approach, *J. Machine learning research*, Vol. 7, 2006, pp. 429-454.

[7] P. Langley, and D. Choi, Learning recursive control programs from problem solving, *J. Machine Learning Research*, Vol. 7. 2006, pp. 493-518.

[8] E. Margolis, and S. Laurence, How to learn natural numbers: inductive inference and the acquisition of number concepts, *Cognition*, Vol. 106, Issue 2, 2008, pp. 924-939.

[9] R. S. Michalski, J. G. Carbonell, and M. T. Mitchell, *Machine Learning: An artificial intelligence approach, Vol. II*. Morgan Kaufmann, 1986.

[10] Y. Moschovakis, Sense and denotation as algorithm and value, In J. Oikkonen, and J. Väänänen, (Eds.), *Lecture notes in logic*, Vol. 2, pp. 210-249, Springer, 1990.

[11] S. Muggleton, and L. De Raedt, L. Inductive logic programming: Theory and methods, *J. Logic programming*, Vol. 19, No. 20, 1994, pp. 629-679.

[12] U. Schmit, Inductive systhesis of functional programs, *LNAI*, Vol. 2654, Springer, 2003.

[13] J. M. Siskind, A computational study of

cross-situational techniques for learning word-to-meaning mappings, *Cognition*, Vol. 61, 1996, pp. 39-91.
[14] J. R. Quinlan, *Programs for machine learning*, Morgan Kaufman, 1993.
[15] L. G. Valiant, L. G. A theory of the learnable, *CACM*, Vol 27, No. 11, 1984, pp. 1134-1142.
[16] M. van Lent, and J. E. Laird, Learning procedural knowledge through observation, First Int'l conf. on Knowledge Capture, 2001, pp. 179-186.

Appendix: Example sentences and procedures generated

Fig. 10 shows example sequences of sentences that describe prime factorization, and Fig. 11 illustrates a procedure generated by our program to factorize a given integer.

Fig. 12 illustrates an example sequence that calculates the least common multiple of two integers, and Fig. 13 shows an example sequence of adding two fractions. Fig. 14 shows a procedure of adding two fractions, the figure also illustrates how internal variables have specific values to conduct its calculation.

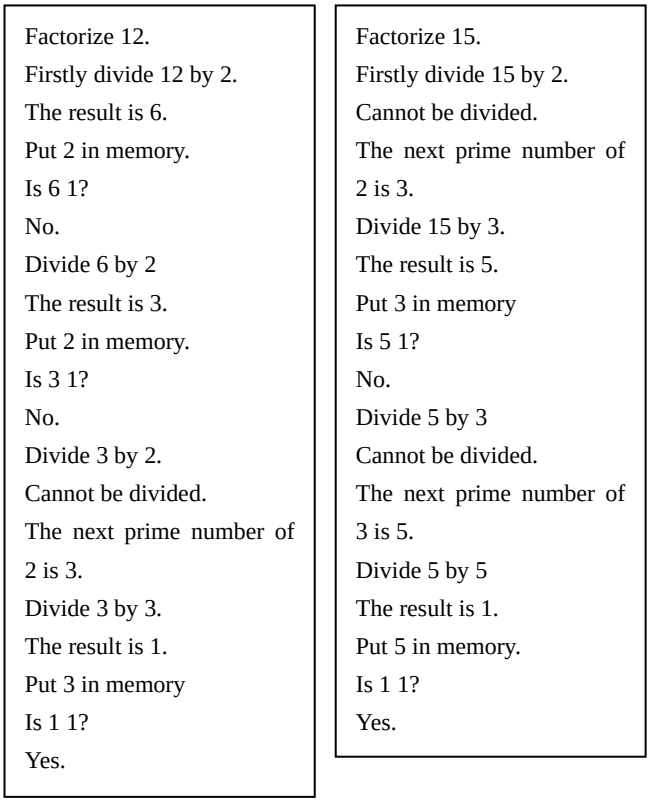


Fig. 10: Example sequences of factorizing an integer.

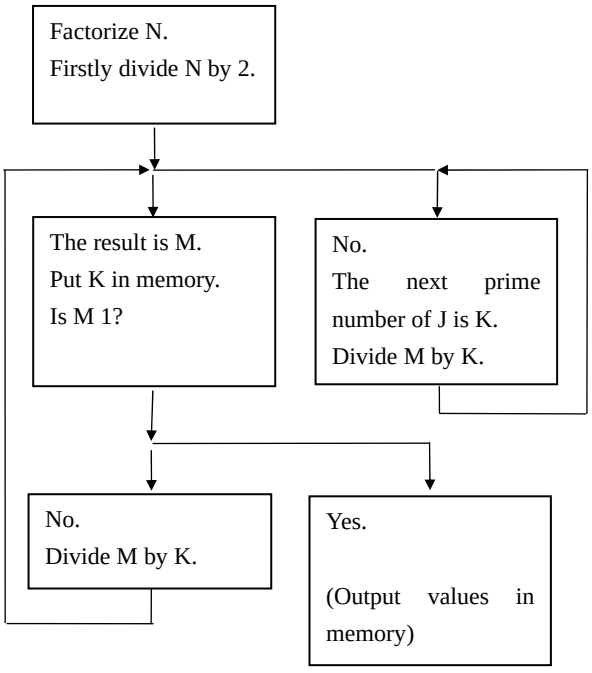


Fig. 11: A procedure of factorizing an integer.

Least common multiple of 10 and 15.
The first to divide 10 and 15 is 2.
Divide 10 and 15 by 2.
Results are 5 and No.
Put 2 in memory.
5 and No are 1 and 1.
No.
Divide 5 and 15 by 2.
Results are No and No.
The next prime number of 2 is 3.
Divide 5 and 15 by 3
Results are No and 5.
Put 3 in memory
No and 5 are 1 and 1.
No.

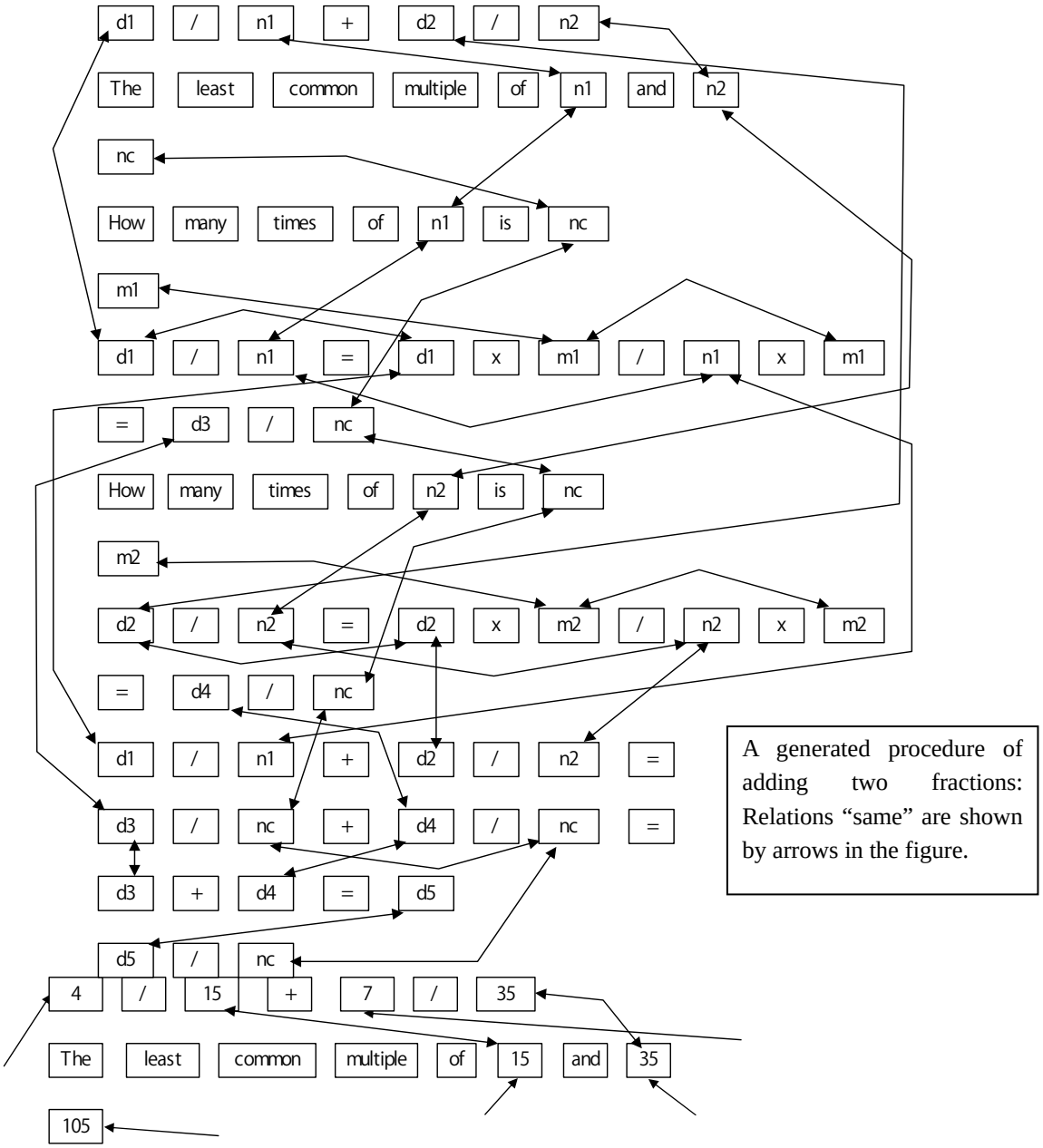
Divide 5 and 5 by 3.
Results are No and No.
The next prime number of 3 is 5.
Divide 5 and 5 by 5.
Results are 1 and 1
Put 5 in memory.
1 and 1 are 1 and 1.
Yes.

(Output values in memory)

Fig. 12: An example sequence to calculate the least common multiple of two integers.

5 / 12 + 8 / 27
The least common multiple of 12 and 27 is
108
How many times of 12 is 108
9
5 / 12 = 5 x 9 / 12 x 9
= 45 / 108
How many times of 27 is 108
4
8 / 27 = 8 x 4 / 27 x 4
= 32 / 108
5 / 12 + 8 / 27
= 45 / 108 + 32 / 108
45 + 32 = 77
77 / 108

Fig. 13: An example of adding two fractions.



At a time of execution, values in inputs determine the values of d1, d2, n1, and n2. In the figure, 4, 15, 7, and 35 are the input values. Then values of the next sentence are determined by relations “same”. Procedure, “the least common multiple”, is used, and a value 105 is returned from the procedure, which determines the value of the third sentence.

Fig. 14: A procedure of adding two fractions.