

A Program that acquires how to execute sentences

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:- One writes example sequences of sentences so that one sequence solves an instance of a problem, and writes how each example runs on a computer. For instance, the one writes a sentence “From 1 to 10, repeat Body”, and also writes how the sentence Body repeats its execution on the computer. Then the one gives them to a program, *pI*, and lets the initial program generalize how the example sequences run and generate a procedure, *pg*. When the program, *pI*, gets a new example sequence to solve a new instance of the problem, the program, *pI*, executes the procedure, *pg*. For instance, the one writes a sentence “From 5 to 8, repeat Body”, and then the procedure, *pg*, repeats the sentence Body four times. As a result of generating a procedure, *pg*, the program, *pI*, acquires implicitly rules of a grammar that produce sentences. Since the generated procedures, *pg*’s, describe how to execute sentences of conditional branches, varying number of repetitions, and varying depth of recursive calls, this paper argues our program, *pI*, acquires a grammar of a language that is equivalent to that used in a conventional programming language.

Key-Words:- Generation of procedures from examples, Acquiring how to execute sentences from examples, Acquisition of a language grammar.

1 Introduction

Philosophers have argued that the meaning of a sentence is a procedure of computing the truth or the falsity of the sentence (e.g., [11], [6]. In [4] Fujiwara and Iwama argue that meaning of a sentence “Is 7 a prime number” is a procedure to check if 7 is a prime number, and explains a program that acquires how to check it. Their program gets example sequences of sentences to solve instances of a mathematical problem and generates a procedure to solve the instances of the mathematical problem. In our case, meaning of a sentence is a procedure of how the sentence runs on a computer. Our problem is to develop a program that conducts the following: one writes example sentences to solve a problem (e.g., Fig. 1) and sentences describing how the sentences run on a computer (e.g., Fig. 2). The one gives them to our program and lets the program transform the sentences to form a procedure (e.g., Fig. 3). Later

the program, given new sentences, executes the generated procedure (e.g., Fig. 3) to get results of the new sentences.

Acquiring meanings of words and sentences have been a focus in various fields including linguistics, psychology, cognitive science, and artificial intelligence. Researchers have studied inductive methods (e.g., [10], [15]). Cross situation is a key to acquire word-to-meaning mappings.

Researchers in artificial intelligence studied much of inductive methods. Logic programming and a model of inductive processes are integrated into a framework of inductive logic programming ([12]). It is applied to modeling acquisition of vocabulary by children (e.g., [5]). After its formulation of ILP, studies on inductive synthesis of programs have been revived ([3]). Within a limited domain, [9] and [14] 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.

A work by Gold, [7], shows that no algorithm correctly guesses a grammar of a language by only positive examples if the language has infinite members. A positive example is a member of the language. If positive and negative examples are given to the algorithm, it identifies the grammar of the language. This result and observation on language acquisition by children support a hypothesis that a human has a universal grammar, [2], from the time of his birth.

Researchers have developed algorithms to learn a regular grammar and/or a context free grammar either by giving the algorithm positive and negative examples or by giving more information (e.g., [1], [13]). Competition has been held to show how fast a new algorithm acquires a language grammar.

From 1 to 4, repeat Body starts.	<i>L</i>
Body of From 1 to 4, repeat Body.	
From 1 to 4, repeat Body ends.	

Fig. 1: Example sentences of “From to”. Besides an example shown in the figure, one may give “From 1 to 10, ...”, “From 3 to 8, ...”, and others to our program. Here, it is assumed that the one has already given the program how “Body of From 1 to 4, ...” run on a computer.

The next section describes a new approach to acquiring how to execute sentences as well as learning a language grammar. Taking the approach, we develop our program, which Section 3 explains.

2 Approach

This section first explains when one can say that a program has acquired how to execute sentences.

Then it explains an approach to constructing such a program.

Execute “From 1 to 4, repeat Body”.	<i>EL</i>
From 1 to 4, repeat Body starts. Body of From 1 to 4, repeat Body. From 1 to 4, repeat Body ends.	<i>L</i>
The first of From 1 to 4 is 1. Execute Body of From 1 to 4, repeat Body. Is 1 4? No. The next of 1 From 1 to 4 is 2. Execute Body of From 1 to 4, repeat Body. Is 2 4? No. The next of 2 From 1 to 4 is 3. Execute Body of From 1 to 4, repeat Body. Is 3 4? No. The next of 3 From 1 to 4 is 4. Execute Body of From 1 to 4, repeat Body. Is 4 4? Yes.	<i>EL</i>

Fig. 2: An example of sentences. The sentences in (*EL*) describe how the sentences, “From to” in Fig. 1 run on a computer.

2.1 When one can say a program acquires how to execute sentences

At the beginning, a program does not know how to execute sentences; for instance, it does not know how to execute a sentence “From 1 to 4, repeat Body”. One writes example sentences as in (*L*) of Fig. 1. The one also writes how the example sentences run on a computer as in (*EL*) of Fig. 2. After getting examples as shown in Fig. 1 and 2, the program transforms the example sentences to generate a procedure (e.g., see Fig. 3). The program, given a new example of sentences, puts specific values in the generated procedure, and executes the generated procedure. For instance, given “From 5 to 8, ...”, the program puts specific values 5 and 8 in

C1 and C2 of the procedure, and repeats its execution from 5 till 8 (see Fig. 4). Then one can say the program acquires how to execute sentences “From to” followed by “Body”. Fig. 5 is another example of sentences, “while”, and shows an explanation of how “while” sentences run on a computer.

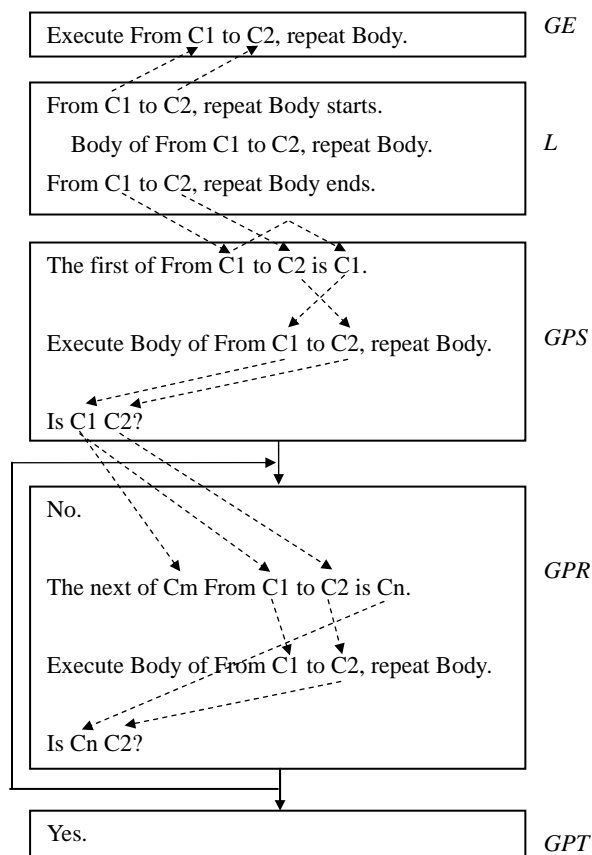


Fig 3: A generated procedure. *GPS*, *GPR*, and *GPT* are subsequences generated by our program. “C1”, “C2”, “Cm”, and “Cn” are internal variables introduced by the program. Arrows in the figure indicate relationships “same” among numerals (some are omitted).

2.2 Construction of a program

One writes example sequences of sentences that solve a problem (e.g., Fig. 1), and inputs them to a program. A way of writing examples is how a computer executes sentences (e.g., Fig. 2), and is similar to explanation of how statements run on the

computer in a class of programming language for beginners. The one constructs the program that memorizes example sequences of sentences and transforms them to generate a procedure of executing a new example as shown in Fig.3 and 4.

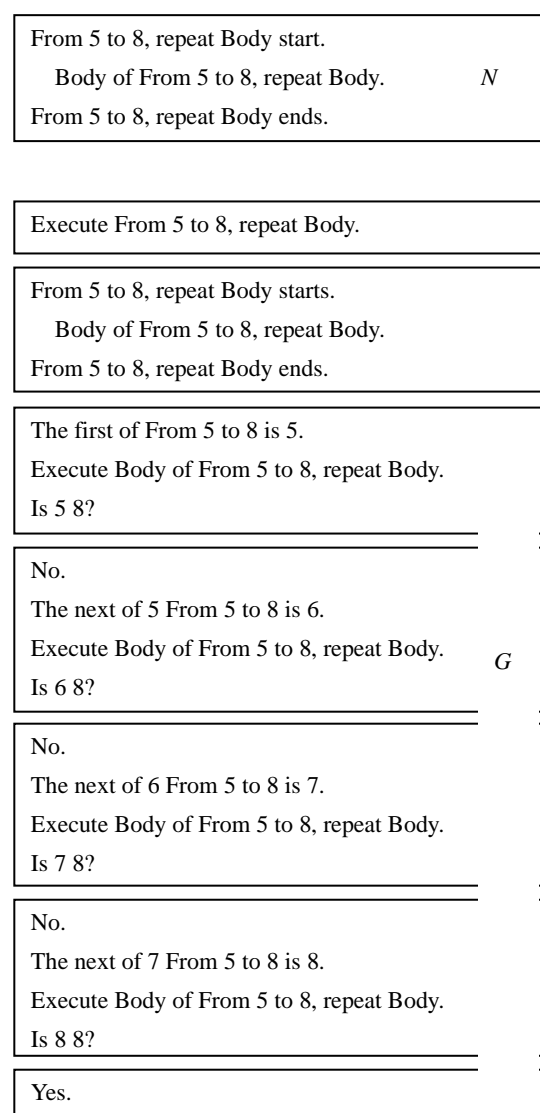


Fig. 4: Execution of a procedure. Our program, given sentences as in (N) of this figure, determines values of internal variables, Ci's of the procedure as shown in Fig. 3, and executes sentences of the procedure as described in (G) of this figure.

One also takes partiality of examples into consideration when the one constructs the program since the examples selected do not often cover all

the cases of a problem. The one makes the program to, given new examples, reorganize a generated procedure so that the reorganized procedure can execute all the examples given to the program.

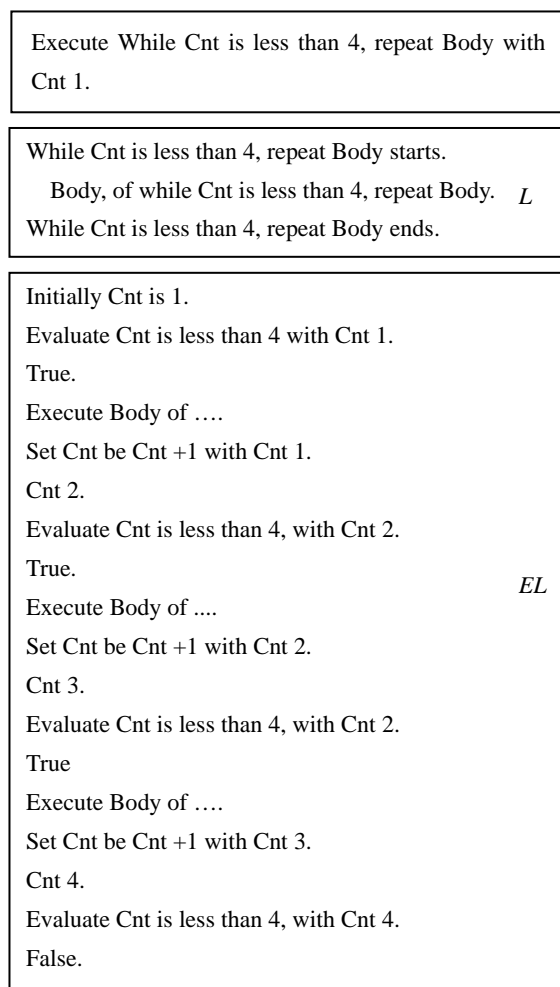


Fig. 5: An example of sentences, *EL*, to explain how sentences of “while”, *L*, run on a computer.

3 Explanation of our program

3.1 Six key functions

The program has six key functions: 1) Finding relations “same” among words in a sequence of sentences. See Fig. 6. 2) Finding repeating sentences and forming a subsequence out of the repeating sentences. See Fig. 7. 3) Finding branches among subsequences and connect them. See Fig. 7. 4) Folding repeating subsequences and generalize them. See Fig. 8. 5) Generalizing sequences of

sentences across examples, generate a procedure, and store it in a long term memory. See Fig. 9, and 10. 6) Given a new instance, retrieving a procedure that matches the new instance, putting specific values of the new instance into sentences of the procedure, and execute the procedure. See Fig. 4.

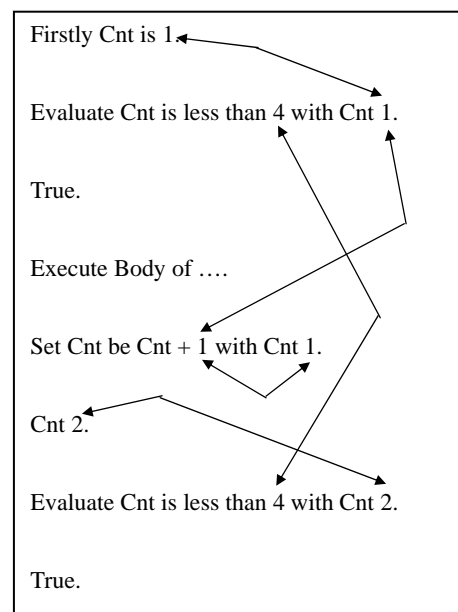


Fig. 6: Finding relations “same” among numerals.

3.2 Function of reorganizing a procedure

When one gives examples to a program, the examples may not cover all the cases of solving a problem. Later the one may give the program new examples that cover the remaining cases of the problem. The program then reorganizes the generated procedure so that the program solves the cases both the old and the new examples cover.

Our program finds repetitions within new examples and replaces specific values by internal variables. It locates a place till that the new example sentences are the same as those already generated as a procedure. Remove the sentences that match the generated already, and make the rest sentences a branch from the place where the sentences differ. For instance, if one gives sentences, “From 9 to 5, ..”, “The previous of 9 is 8”, and so on, the

program forms new subsequences, and adds a branch of the new subsequences to the procedure shown in Fig. 3 to make a new procedure shown as in Fig. 11.

3.3 Outline of our initial program

Our program gets example sequences of sentences, one by one, and stores them in a temporary memory. It compares a sentence (e.g., “if 10 is larger than 5”) in a sequence with that in another sequence, and stores the sequence if they are different. After the program stores more than certain number of example sequences, it generalizes them to form a procedure and stores the procedure in its long term memory. Given a sequence of sentences, it takes one procedure after another from the long term memory to compare the sentence with a head sentence of the procedure. If match, it executes the procedure on values in the given sentence. Fig. 12 illustrates an outline of our program.

The program compares numerical values in one input sequence and finds relations “same” among the numerical values. For example, see Fig. 6. The program tries to find consecutive sentences that repeat in the sequence; firstly it finds a sentence that can be executed by a procedure already generated, and it appends sentences preceding the sentence. See Fig. 7. If the program finds repetitions of the sentences, the program forms a subsequence out of the sentences and tries to find if the subsequence appears repeatedly. If so, the program folds the subsequences and conducts generalization on the subsequences. See Fig. 8.

One subsequence formed can become a branch from other subsequences. For example, a sentence “Cnt is less than 4” produces “Yes.” or “No.” depending on the value of Cnt. Sentences following “Yes.” become a subsequence, and those following “No.” become another sub step.

After forming subsequences, the program compares sentences of a subsequence of one example and those of another example. The

program introduces internal variables and replaces numerals different in the subsequences by the internal variables. The program also compares relations “same” among words of the sentences, and removes a relation “same” when only one example has it. If only one example has the relation “same”, that must happen to hold accidentally in the example.

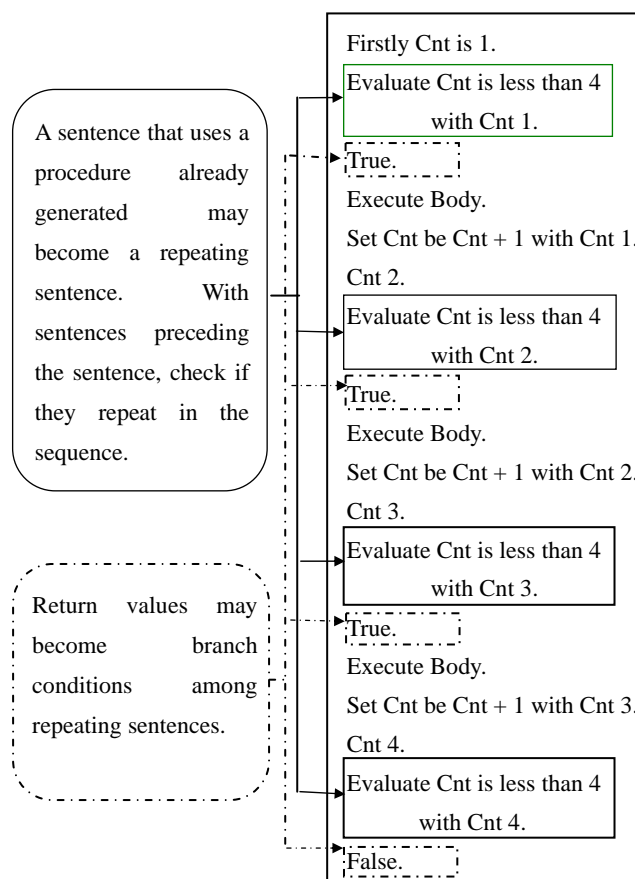


Fig. 7: Finding sentences that repeat in a sequence, and finding branch conditions.

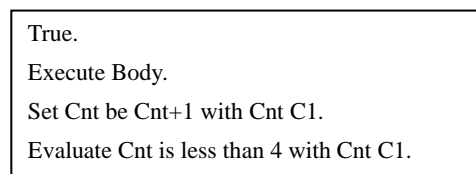


Fig. 8: Subsequences folded. Repeating sentences in a sequence, found as in Fig. 7, are formed as a subsequence. If subsequences appear more than once, they are folded and generalized.

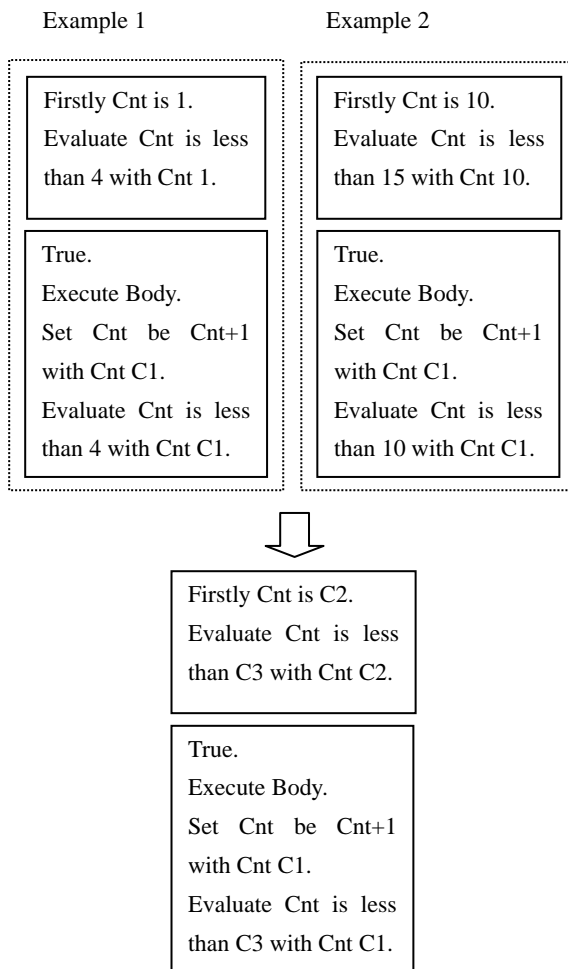


Fig. 9: Generalization of sentences across examples.

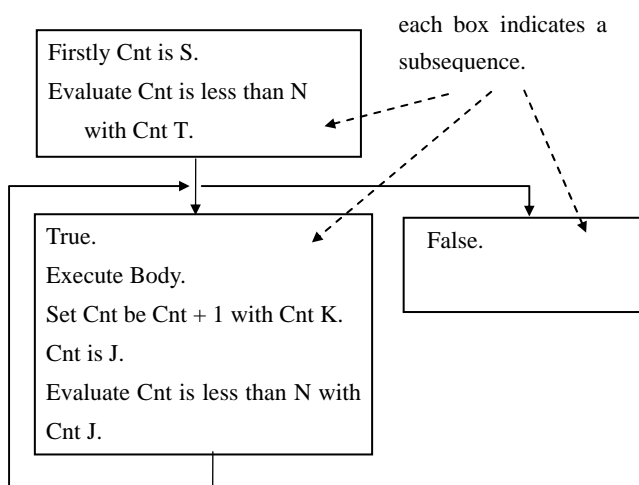


Fig. 10: A procedure generated to run “while” sentences.

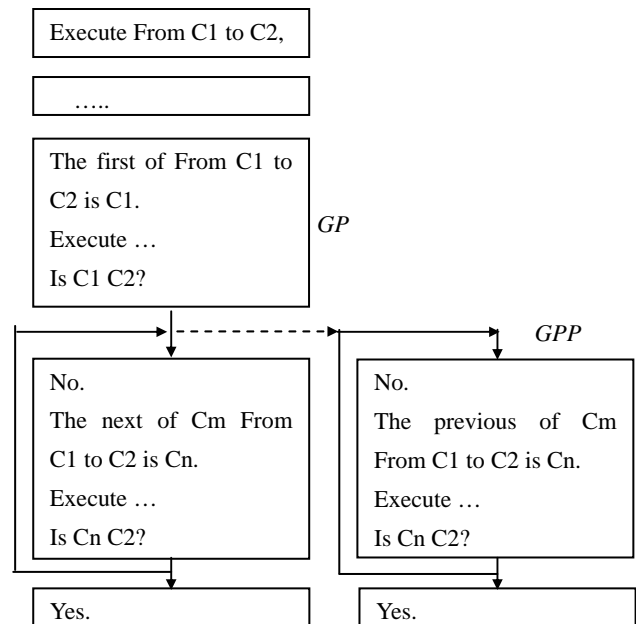


Fig. 11: A branch of “From to”. Generated subsequences, *GPP*, become a branch of a procedure already generated, *GP*.

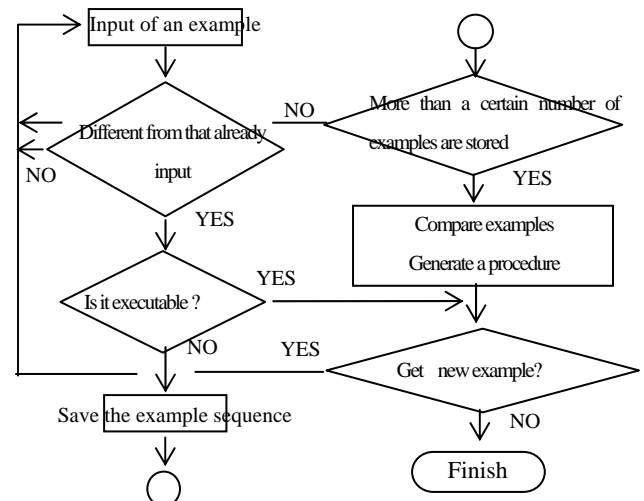


Fig. 12: A flow of generating a procedure and its execution.

Then the program locates words different across examples but the other words and the relations are the same in the sentences. Replace the words by internal variables with relations “same” among the words, actually explicit variables.

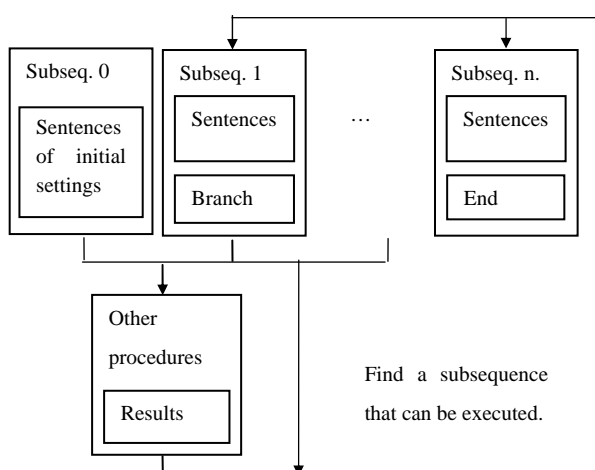


Fig. 13: A dynamic structure of executing a procedure generated by our program.

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.

During the execution of a procedure, another procedure may be invoked and be executed. If a sentence of the procedure matches that of a head sentence of the procedure, the procedure is invoked. Fig. 13 illustrates how the execution of subsequences of the procedure and another procedure proceed. If the procedure invoked uses itself, namely recursively use it, the depth of recursive usages is determined dynamically. Fig. 14 shows a recursive usage of Body sentences.

3.4 Assumptions

One is required to keep two rules: the first is the following: Suppose sentences, Se , use a procedure that can be generated by example sequences of sentences, Sa . Then one gives our program the set of the sentences, Sa , first, and let the program acquire how to execute them, Sa and then the one

gives the sentences, Se . As a result of this, the program forms hierarchical structures of using procedures (e.g., see Fig. 14). The second is that the one uses same words with the same word orders to write sentences of the same meaning.

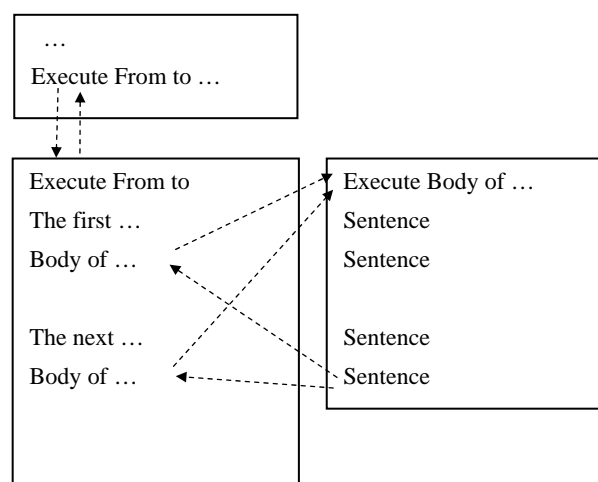


Fig. 14: A hierarchical structure of using procedures. Sentences “Execute From to ..” use “Execute Body of ...”, and sentences “Execute Body ...” use another sentences “Execute Body ...”.

From the beginning, our program is assumed to have functions of inputs, outputs, and some calculation using the inputs as well as using calculation results. 1) The program has functions of counting, and basic arithmetic operations such as addition of two numerical values. It is possible to construct procedures from examples to do arithmetic operations such as addition¹. Complexity of generated procedures grows much unless the program is assumed to have functionality of a certain level. 2) The program has string functions: matching strings, adding sub strings to make a string, and deleting a sub string from a string to make a string that does not have the sub string. Sentences are given as strings and are stored in memory, and are picked up one by one. 3) The program has functions of using memory space.

¹ Iwama [8] explains a program that generates a program of counting numbers from examples.

There are a large number of memory locations that keep values in them. It is assumed the program has functions of getting memory space, labeling the space, putting a value in a labeled space, getting the value from the labeled space, and releasing the memory space when the space are not used any longer.

4. Examples

4.1 Evaluation of expressions

This section describes example sentences that one uses to let our program acquire how to evaluate expressions. Such an example is “5 is smaller than 7”. Suppose that the program has acquired how to count and how to get “the next of a given number”, or has functions of counting as its innate functions. Then, the one gives the program sentences as follows: “Execute Evaluate 5 is smaller than 7. Is 5 8? No. The next of 5 is 6. Is 6 8? No. The next of 6 is 7. Is 7 7? Yes. Repetitions of the next reach 7. True”. When the one gives the program a sentence evaluating an expression, “Evaluate 6 is larger than 4”, example sentences are the following: “Execute Evaluate 6 is larger than 4. Is 6 4? No. The previous of 6 is 5. Is 5 4? The previous of 5 is 4. Is 4 4? Yes. Repetitions of the previous reach 4. True”.

One may need to give the program another concept before the one gives the program examples such as “8 is smaller than 5”. The other concept is the number of digits; for instance, “12 has 2 digits”, and “400 has 3 digits”. Suppose the one has given the program the concept of the number of the digits. Then the one can give the following to the program: “8 is smaller than 5. Is 8 5? No. 5 has 1 digit and 8 has 1 digit? Yes. The next of 8 is 9. Is 9 5? No. 5 has 1 digit and 9 has 1 digit? Yes. The next of 9 is 10. Is 10 5? No. 5 has 1 digit 10 has 1 digit? No. False”².

² For a practical purpose, it is required to introduce a mechanism to get interruption by people when it takes too much time to evaluate an expression; for example, “10000 is smaller 9000”, “the next of

4.2 Conditional branches

One writes example sequences of sentences of how conditional sentences run on a computer as follows: “Execute if 8 is smaller than 10, Body of if 8 is smaller than 10, else Body of else. Execute Evaluate 8 is smaller than 10. True. Execute Body of if 8 is smaller than 10.” “Execute if 7 is smaller than 4, Body of if 7 is smaller than 4, else Body of else. Execute Evaluate 7 is smaller than 4. False. Execute Body of else.” Then our program generates a procedure as shown in Fig. 15.

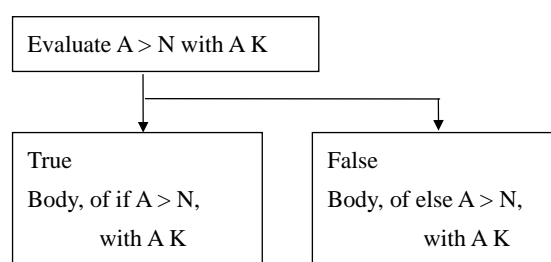


Fig. 15: A procedure generated to run “if else” sentences.

4.3 Recursive usage of sentences

After one lets our initial program acquire how to execute a set of sentences, the one can write recursive usages of the set. For instance, the one writes sentences “Body of if ...” and a sentence in them uses “Body of if ...” as shown in Fig. 16. Then the sentences “Body of if ...” use themselves recursively. Note that the number of recursion is determined at the time of execution, depending on a way of writing sentences.

4.4 Character strings

Although it is assumed that our program has acquired string functions from the beginning, it is possible to let the program acquire how to deal with character strings. Fig. 18 shows example sentences of saving a string in a table space. Using a procedure of saving strings in tables, one can write

10000 is 10001”, “the next of 10001 is 10002”, and so on. While “the next of ... is ...” continues, people may make an interruption.

sentences of adding two strings.

4.5 Simple applications

After one has let the program acquire how to execute sentences of repetitions, conditional branches, and using tables, the one is able to give the program example sequences of sentences used for simple business applications. A simple application may be “Output names of products that sell more than 100”. The application uses sentences as shown in Fig. 17. The one becomes able to write a new application, “Output names of products that sell less than 200”, without changing sentences that the application uses.

5. Discussion

This paper explains a program, pI , that transforms example sequences of sentences to generate a procedure, pG . Since the program, pI , with the procedure, pG becomes able to execute a new example, one can say it has acquired knowledge of how to execute the new example that follows the same rules as the examples given before from the point of their execution. The program, pI , has also acquired a language grammar in a sense that it can execute the new example. The language grammar includes orders of words that specify conditional branches, varying number of repetitions, and varying depth of using other sentences or themselves. Therefore the program, pI , implicitly acquires a grammar equivalent to that of a conventional programming language by getting positive examples only. But the one is required to give the program much more information than the case of learning only a grammar. The one is also required to give the program sentences, S_a , before sentences, S_e , if some of the sentences, S_e , use a procedure exemplified by the sentences, S_a .

Future work includes the following: One is to develop a set of better techniques than those explained in this paper. Another is to explore its feasibility of our approach to developing a library of procedures for business applications.

References:

- [1] D. Angluin, Learning regular sets from queries and counterexamples, *Information and computation*, vol. 75, 1987, pp. 87-106.
- [2] N. Chomsky and H. Lasnik, The theory of Principles and Parameters, In J. Jacobs, A. von Stechow, W. Sternfeld, and T. Vennemann (Eds.), *Syntax: An International Handbook of Contemporary Research*, Berlin: de Gruyter, 1993.
- [3] P. Flener and S. Yilmaz, Inductive synthesis of recursive logic programs: Achievements and prospects, *Journal of Logic Programming*, Vol. 41, No. 2-3, 1999, pp.141-195.
- [4] M. Fujiwara and K. Iwama, A program that acquires meanings of sentences in mathematics, Submitted for publication, 2009.
- [5] K. Furukawa, I. Kobayashi, T. Ozaki, and M. Imai, A Model of Children's Vocabulary Acquisition Using Inductive Logic Programming, *LNCs* Vol. 1721. Springer Berlin, 1999.
- [6] N. Gierasimczuk, The problem of learning the semantics of quantifiers, *LNAI*, Vol. 4363/2007, 2007, pp. 117-126. Springer.
- [7] E. M. Gold, Language identification in the limit, *Information and control*, Vol. 10, 1967, pp. 447-474.
- [8] K. Iwama, A robotic program that acquires concepts and begins introspection, *NueroQuantology*, Vol. 4, No. 4, 2006, pp. 321-328.
- [9] 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.
- [10] E. Margolis and S. Laurence, How to learn the natural numbers: Inductive inference and the acquisition of number concepts, *Cognition*, Vol. 106, 2008, pp. 924-939.
- [11] Y. Moschovakis, Sense and denotation as algorithm and value, In J. Oikkonen, and J. Väänänen, (Eds.), *Lecture notes in logic*, Vol. 2, 1990, pp. 210-249, Springer.
- [12] S. Muggleton and L. De Raedt, Inductive logic programming: Theory and methods, *J. Logic*

programming, Vol. 19, No. 20, 1994, pp. 629-679.

- [13] Y. Sakakibara, Learning context-free grammars from structural data in polynomial time. *Theoretical computer science*, vol. 75, 1990, pp. 223-242.
- [14] U. Schmit, Inductive synthesis of functional programs. *LNAI*, Vol. 2654, Springer, 2003.
- [15] J. M. Siskind, A computational study of cross-situational techniques for learning word-to-meaning mapping. In M. R. Brent, (Ed.), *Computational approaches to language acquisition*, MIT Press, 1997.

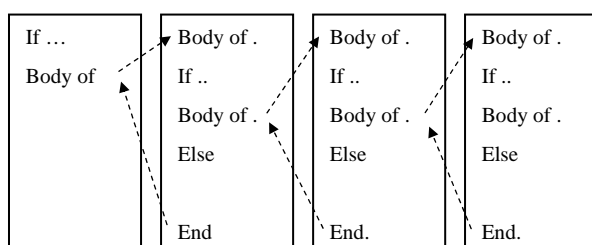


Fig. 16: An example of recursive usage of sentences.

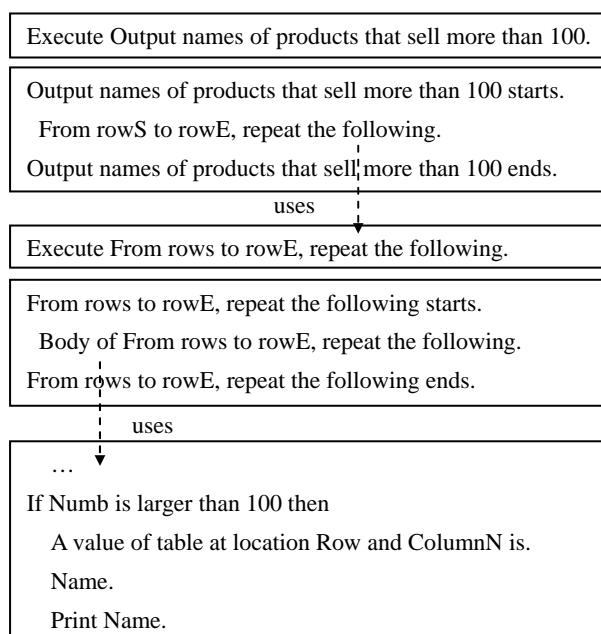


Fig. 17: A simple application (details are omitted in the figure). A sentence "Output names ..." uses sentences of "Body" to output product names.

Execute Save a string, "abc", in table, T1.

Save a string, "abc", in table, T1 starts.
Save a string, "abc", in table, T1 ends.

Location of table, T1, is.
500.
Set llocation be 1.
Letter in a string, "abc", at llocation 1 is.
a.
Is a NIL?
No.
Store a in table, T1, at Location 500.
Set Location be Location+1.
Location 501.
Set llocation be llocation+1.
llocation 2.
Letter in a string, "abc", at llocation 2 is.
b.
Is b NIL?
No.
Store b in table, T1, at Location 501.
Set Location be Location+1.
Location 502.
Set llocation be llocation+1.
llocation 3.
Letter in a string, "abc", at llocation 3 is.
c.
Is c NIL?
No.
Store c in table, T1, at Location 502.
Set Location be Location+1.
Location 503.
Set llocation be llocation+1.
llocation 4.
Letter in a string, "abc", at llocation 4 is.
NIL.
Is NIL NIL?
Yes.

Fig. 18: Example sentences of storing a string in a table.