This is an English translation of paper /18/ (see below)

**Text Example Generalization:**

**Synthesis of Sampletalk**

**Programs**

**Andrew Gleibman**

### ***Abstract***

*In this paper, a technique for automatic and semi-automatic synthesis of text processing programs is introduced. Processing rules are defined by demo samples of text transformations. A synthesizer discovers and generalizes regularities, presented by the user in form of text examples, and applies them in order to create an algorithm for processing of a wider class of texts.*

*Sampletalk Language serves for implementation of the synthesized algorithms. As a programming language, Sampletalk provides programming technology based on text abstraction.*

*The following program examples are considered: transformation of arithmetical and logical expressions, description of a technical assembly, natural language reasoning, word and list inversion, replacing word occurrences in expressions etc. Some program examples demonstrate reasoning, hardly attainable using other programming technologies.*

*In a theoretical consideration, it is proven that Sampletalk is Turing-complete language and that for any Sampletalk program some equivalent program can be automatically constructed by applying the proposed technique of synthesis to a suitable set of text samples. This technique provides an alternative way of programming, in which the user, instead of writing a program code, trains a program synthesizer by successive text examples of incrementing complexity and detalization.*

*Using the generalization, structurisation and composition of text examples, we can build any algorithm. So, we speak of programming by having a computer to define object combinations and to combine existing objects into new ones automatically. The question of how to lead this process to useful combinations is considered.*

# 1. **Introduction**

The main paradigm, which is described in this work, can be formulated as ***synthesis of algorithms based on human or other object behavior examples,*** without any a-priori knowledge but some universal one. This paradigm has been investigated and represented in literature for various kinds of algorithms, intelligent structures, representations of behavior of investigated objects, and basic knowledge. Table 1.1 contains examples of such research; the last line in the table represents the present work.

| ***Object behavior information*** | ***Kind of synthesized structures*** | ***Universal knowledge*** | ***Reference*** |
| --- | --- | --- | --- |
| Chemical descriptions | Chemical compound | Logical calculus, properties of numbers | Langley et al. /30/ |
| Physical Observations | Physical law | Heuristics about numbers, programming language | Falkenhainer, Michalsky /12/ |
| Mathematical definitions | Mathematical theory | Mathematical heuristics | Lenat /31/ |
| Sample calculations, algorithm history | Program | Arithmetic properties, programming language | Barsdin et al.  /4/,/5/,/28,29/ |
| Physical laws, observations | Simultaneous equations | Logical calculus and model | Gleibman /20,21/; |
| Text transformations | Program | Text matching, Logic programming language | Gleibman /15,16/;  *this paper* |

**Table 1.1. Universal Knowledge and the Paradigm of Synthesis**

Constructions related to the second column of this table represent generalization of constructions from the first column. Rules of generalization are represented in the third column.

Generation of programs sometimes is made using combination and transformation of descriptions of an algorithm or an environment (program language source code, formal program specification, description of an artificial world etc). Attempts to recognize regularities in algorithm history or to build formal description of algorithm performance are made (see our review in Section 4). The source of the formal description is usually assumed to be a human. Sometimes sophisticated syntax and semantic rules are justly or unjustly involved in them.

Note that a-priori syntax and semantic rules of a sophisticated programming language usually are not similar to thinking of an individual programmer and often hinder him from a natural (for him) way of formulating algorithms. On the other hand, he usually may easily formulate his own rules or informal hypotheses about the algorithm being devised based on examples of data being processed, or, what is more important for us, formulate data samples based on hypotheses about the algorithm. Examples of such samples and hypotheses are given in Table 1.2.

| ***Sample*** | ***Possible hypotheses*** |
| --- | --- |
| **ab => ba** | Inversion of all word preceding "**=>**"  Swap two adjacent sub-words  Swap something with letter **b**  ... (other hypotheses) |
| **abc => cba** | Inversion of all word preceding "**=>**"  ... |
| **(y/x)axb = ayb** | Replace all occurrences of **x** by **y**  Replace only the first such occurrence  Replace something with word **ayb**  ... |
| **(y/x)axbxc = aybyc** | Replace all occurrences of **x** by **y**  ... |
| **axbcd -> abxcd** | Shift **x** one position right  Swap **x** with its left or right neighbor  ... |
| **axbcd -> xabcd** | Shift **x** to the beginning of a word  Swap **x** with its left or right neighbor  ... |
| **Polish(a+b) is +ab** | Swap **a** and **+** and eliminate parentheses  Translate an expression into Polish notation  … |
| **Where is a tree? In the forest.**  **Where is a fish? In the sea.** | A definition for location of some object  and reasoning about this location |
| **New York is situated in America.**  **Where is New York? In America.** | The similar definition and reasoning  about location of another object |
| **x0 is variable** | Something is a notation of a variable  Something includes number 0 |
| **x10 is variable** | Something is a notation of a variable  ... |
| **(forall x0)a or (forall x0)b-->**  **(forall x0)(forall x0)(a or b)** | Shift quantifiers into the formula prefix  Move the second **(forall x0)** to the left  ... |
| **forall x0)a(x0,y) or (forall x0)b(x0,t)-->**  **(forall x0)(forall x10)(a(x0,y) or b(x10,t))** | Shift quantifiers into the formula prefix  … |
|  | |

**Table 1.2. Text examples and informal hypotheses about them**

Our question is *how can such hypotheses be automatically formulated in a more formal way*, so that a useful text-processing program could be automatically constructed from them.

This problem is typical for Machine Learning: we can consider it as an attempt to automatically improve performance or another characteristic of an artificial system via discovery of useful information from the world, which is here represented in texts.

Consider the following characteristics of Machine Learning systems: 1) kind of the external world for the system (physical motions, chemical reactions, historic events, arbitrary data expressed in texts as in this chapter, etc); 2) discovery component, which forms hypotheses about the observed events; 3) evaluating function for filtering useful hypotheses from other hypotheses.

Assume that a human can obtain texts, which represent the external world. He wants to create a program and is able to provide the discovery system with desired data processing examples. Assume also that he has a possibility to evaluate newly created programs against the initial examples and against other examples. So, we are concentrating on a discovery component for program synthesis.

Our preceding note about thinking of an individual programmer prevents us from considering any sophisticated computer programming language as a universal knowledge for text processing. More likely, an ability to recognize and apply regularities, which exist in example texts, should be included into this knowledge. This part of the universal knowledge, proposed in our work, is described in the following sections. Logic-programming language Sampletalk is included into this knowledge in order to implement the synthesized algorithms. Sampletalk is described in the following sections. Representation and processing of text samples is described in Section 3. Proofs of the included theorems are given in Appendix.

Note that we don’t introduce any a-priori assumptions for limiting text examples, which are used for building algorithms. Relation of this approach to linguistics is illustrated in natural language processing applications of Sampletalk language, described in sections 3 and 4. Psychology and modeling of the user's way of thinking, his gift for generalization and mental associations, possibilities to involve computers in this creativity, are the main points of our consideration.

# 2. **Data example based programming**

Sampletalk may be considered as simplification and modification of Prolog language. Sampletalk was introduced on /15/ -- /17/. It involves elements related to Refal /38/ and Planner /24/ languages. The main feature of this language is the similarity of the constructions for writing algorithms to the data being processed. For this purpose, we sometimes use special notation with underlines rather than square brackets for designation of nested lists.

Syntax of Sampletalk language is intentionally made extremely simple for logic programming. As the modification of Prolog, Sampletalk may be defined by the following conventions.

1.1. The only basic data type in Sampletalk is char, i.e. any Sampletalk data element is either a single character or a list of Sampletalk data elements.

1.2. Prolog terms, such as [A1,A2,...,Ak], where each Ai is either a single character constant or a single character variable name (i.e. an upper-case alphabetic letter, in accordance with common Prolog convention), are designated as A1A2....Ak. Other variable names currently have not been used. These terms are called samples; their sub-lists (on any nesting level) are called sub-samples. Uppercase single-character constants and several constants used also for syntax purposes ([, ], {, }, double dot, double comma, neck-symbol :-) are designated in a special way. Nested samples are marked either by brackets or by underlining.

**EXAMPLES.** Prolog lists **['a', '+', 'b', '=', C], [A, '+', B, '=', C]** and **[['(','a', '+', 'b',')'], '\*', ['(',M, '+', N,')']]** have been written as samples **a+b=C, A+B=C** and **[(a+b)]\*[(M+N)]** respectively. Here **C, A, B, M, N** are variable names. Samples in brackets and the asterisk are elements of the last sample. Another notation for this sample is **(a+b)\*(M+N)**. Samples **(a+b)\*, (a+b), +N** are examples of sub-samples (on different levels) of the last sample.

1.3. The only predicate in Sampletalk is some 1-place predicate with an empty name. Predicates have been written in clauses without parentheses (simply by writing their arguments), separated by neck-symbol **:-** and double commas, terminated by double dot.

So, clauses in Sampletalk have been written in the form *w* **:-** *w1***,,***w2***,,** *...* **,,***wn***..***,* where *w, w1, w2, ... ,wn* are samples.

1.4. While matching two samples, a variable may match not only an element of the opposite sample (like in Prolog). It may match a segment, or sub-sample, formed by an adjacent sequence of such elements.

Since there may be many possibilities to match two samples, we introduce the following rule, which will affect backtracking: *the more left an occurrence of a variable in a sample, the less size of a sub-sample that is considered for instantiation of this variable (starting from 1-character size).*

**EXAMPLES.** Sample **a/b/c** may match sample **A/B** in two ways: 1) **A=a**, **B=b/c**; **2) A=a/b**, **B=c**. For matching a sample **begin spoke rim hub wheel frame end** to sample **C A B**, possible bindings of variables are represented in Fig. 2.1.

| **begin** | **spoke** | **rim** | **hub** | **wheel** | **Frame** | **end** | *(1-st attempt)* |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **C** | **A** | **\ - - - - - - - - - - - - - - - B - - - - - - - - - - - - - - - /** | | | | |  |

| **begin** | **spoke** | **rim** | **hub** | **wheel** | **Frame** | **end** | *(2-nd attempt)* |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **\ - - - - C - - - - /** | | **A** | **\ - - - - - - - - - - - B - - - - - - - - - - - /** | | | |  |

**…**

| **begin** | **spoke** | **rim** | **hub** | **wheel** | **Frame** | **end** | *(last attempt)* |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **\ - - - - - - - - - - - - - - - C - - - - - - - - - - - - - - - /** | | | | | **A** | **B** |  |

**Fig. 2.1.** Matching samples **C A B** and **begin spoke rim hub wheel frame end**

Variable **A** can not bind a sub-sample, which contains 2 or more underlined words, since **A** is underlined, and underlying structures of the matching samples must match. Variables **C** and **B** (which are not underlined) may match any sub-samples. See the last program example of this section, where this matching is used.

1.5. Any sample may be considered as a program goal. Resolved goal (that is, the result of replacing all variable names in this goal by values obtained during performance of the program) is considered the result, or the output of the program. Usually we place program goals at the top of the programs.

Only the user’s wishes and imagination limit the expressing power of a language with so trivial syntax and so various possibilities for interaction of language constructions. For example, a program for sorting numbers may contain fragment **A{N}{M}B ==> A{M}{N}B :- {M<N}** (see program examples below); a program for reducing homogeneous items in an analytic expression may contain fragment **A + M\*X + N\*X + B ---> A + L\*X + B :- L = M + N** with respective clauses involving constants **\***, **+**, **=** and the concept of number (**\*** and **+** are considered as ordinary symbols and may be later related to some machine-oriented operations).

Program 1 shifts quantifiers into the formula prefix for a disjunction of logic formulas:

| % Goal (lines beginning with \% are comments):  **(forall x0)[a(x0,y)] or (forall x0)[b(x0,t)] --> W..**  % Program:  **(Q X)[F] or (Q X)[H] --> (Q X)(Q Z)([F] or [G]):-** (1)  **[X] is variable,,**  **[Z] is variable,,**  **not(F contains Z),,**  **(Z/X)[H]=[G]..**  **[x0] is variable..**  (2)  **[X10] is variable:-[X0] is variable..**  (3)  **AXB contains X..**   (4)  **(Y/X)[AXM]=[AYN]:-(Y/X)[M]=[N]..**  (5)  **(Y/X)[A]=[A]..**   (6) |
| --- |

**Program 1. Sampletalk program for logic formula transformation**

Clause (1) almost literally quotes a well-known formal rule, which describes transformation of logic formulas, along with informal constraints, which describe application of this rule. Let us especially note that we turned these informal constraints into formal components of a working algorithm. Clauses (2) and (3) generate new names for logic variables in a logic calculus, such as **x0, x10, x110** etc. The predicate **not(***C***)**, where *C* may be any sample, is built-in Sampletalk construction, which succeeds if and only if goal *C* fails (like negation in Prolog language). Clause (4) states that word *F* contains word *Z* if *F* can be represented as *AZB*, where *A* and *B* are words that surround *Z* in *F.* Clauses (5) and (6) represent common notation for substitution of terms (more detailed consideration of these clauses will be given in the next section).

Consider performance of this program. Its goal matches head of clause (1), so **W** will be bound to **(forall x0)(forall Z)([a(x0,y)] or [G])**, where **Z** and **G** are still non-instantiated variables (remind that uppercase letters in samples are Sampletalk variable names). All occurrences of **Q** and **X** in (1) will be replaced with **forall** and **x0** correspondingly. Then, generated sub-goal **x0 is variable** and **Z is variable** matches (2), and **Z** will bind **x0**. Then, sub-goal **F contains Z** matches (4), and (1) fails, since **F** binds **a(x0,y)** which contains **x0**. After backtracking, **Z** will match **x10**, all the sub-goals of (1) will be resolved, and **W** will be bound to **(forall x0)(forall x10)([a(x0,y)] or [b(x10,t))]**. The output will be **(forall x0)[a(x0,y)] or (forall x0)[b(x0,t)] --> (forall x0)(forall x10)([a(x0,y)] or [b(x10,t)])**. In the next Section we consider such samples as stimuli for synthesis of Sampletalk clauses.

The same program may be used for inverse transformation of logic formulas: elimination of parentheses and extra quantifiers from the formula prefix. For this purpose we simply replace the program goal with **W --> (forall x0)(forall x10)([a(x0,y)] or [b(x10,t)]).** Variable **W** will be bound to **(forall x0)[a(x0,y)] or (forall x0)[b(x0,t)]**.

In the next version of this program (see Program 2) we change constants, which define rule description syntax, and add rule identification (see variable **R** in the goal). This example shows that ***we can limitlessly develop similarity of strong algorithmic constructions to natural language descriptions of formal objects***. As important is the following opposite observation: ***we can take unconstrained natural language description of an object or algorithm and transform it into a working program***.

| %Goal: **according to the rule R, the result of shifting quantifiers in the formula (/\ x0)[a(x0,y)] \/ (/\ x0)[b(x0,t)] is formula W..**  % Program: **according to the rule 2a from chapter 5, the result of shifting quantifiers in the formula (Q X)[F] \/ (Q X)[H] is formula (Q X)(Q Z)([F] \/ [G]) :- X is notation for variable,, Z is notation for variable,, not(word F contains word Z),, the result of replacing X by Z in formula H is G..**  **x0 is notation for variable.. X10 is notation for variable :- X0 is notation for variable..**  **word AXB contains word X..**  **the result of replacing X by Y in formula AXM is AYN :- the result of replacing X by Y in formula M is N..**  **the result of replacing X by Y in formula A is A..** |
| --- |

**Program 2. Another Sampletalk program for logic formula transformation**

Program 2 provides another version of Program 1. Note that no previous parsing of goals, building special data types, memory structures, or any other programming effort is required for forming and running these programs!

Samples in these programs are built as results of generalization of the corresponding data processing examples. The user in this case makes the generalization. In the next sections, we consider such programs as objects of program synthesis.

Convention 1.4 makes an essential difference between Sampletalk and Prolog: such various bindings are not allowed while matching terms and predicates in Prolog. Nevertheless, they may be naturally incorporated into the term unification procedure and do not contradict the paradigm of logic programming. The following theorem illustrates similarity and difference between Sampletalk and Prolog:

**THEOREM 2.1.** For any Sampletalk program **S**, an equivalent Prolog program **P** consists of Prolog notations for all clauses from **S** and a set of clauses, which are built from pairs of samples (**l,t**) of **S** (the caller and the one being called) by including additional sub-list forming brackets.

Proof of this theorem (see Appendix) provides a way to implement Sampletalk interpreter using Prolog interpreter. Another way (used in our implementation) consists in addition of new possibilities for list matching to the standard term unification procedure.

Although we eliminated important features of Prolog (other data types, special syntax for terms, predicate names, cut operator, side effects), Sampletalk is universal programming language, as stated in the next theorem:

**THEOREM 2.2.** Let **S** be alphabet of single-character constants of Sampletalk language, **M** be any Markov algorithm written in terminal alphabet **U** and non-terminal alphabet **V**, where **U** U **V** is subset of **S**. There exists Sampletalk program **SM**, which, given any goal *w***==>W** (where *w* is a word, **W** is a Sampletalk variable), does the following:

a) transforms it into output *w***==>***w****M***, where *w****M*** is result of application of **M** to *w*, if *w* is a recognized word for algorithm **M**;

b) stops with undefined results if word *w* leads **M** to a deadlock stop, and

c) shall not stop if **M** is not applicable to word *w*.

(See Proof in Appendix). Implementation of Markov algorithms written in other alphabets may be done by encoding in the alphabet **S**. So, for any Markov algorithm, an equivalent Sampletalk program can be constructed.

The reader may get the wrong impression that Sampletalk can be used only for symbol processing. Other data types may be represented using symbolic notation of other language elements or by incorporating built-in machine oriented Prolog terms and predicates (which is done in our current implementation). Refer to /16/ for details. Our objective at this point, however, is not study of Sampletalk in these directions, but automatic synthesis of Sampletalk clauses.

# 3. **Representation, structurization, generalization and composition of text samples**

Below we define constructions used for discovery of regularities exposed in text samples. They enable us to derive new useful samples and clauses from the existing ones.

**DEFINITION 3.1.** Let *u* be a clause. Result of underlining a sub-sample *w* in any of its samples is called **structurization** of *u*. For two sub-samples *w1* and *w2* of the same sample of *u*, simultaneous underlines are possible either if *w1* is sub-sample of *w2, or w2* is sub-sample of *w1*, or *w1* and *w2* have no common sub-samples in *u*.

**DEFINITION 3.2.** Let *C* be a clause, and *w1,w2, ... ,wk* are (non-intersecting) sub-samples of any of its samples. Any result of replacing *w1,w2, ... ,wk* with unique variable names *V1,V2, ... ,Vk* (which did not occur in *C* before) is called **generalization** of *C* by converting its sub-samples into variables. Clause *E* is called **common generalization** of *C* and *D* if *E* is generalization of *C* and generalization of *D*, and **minimal common generalization** of *C* and *D* if turning any variable of *E* into a constant violates the property of being common generalization.

**DEFINITION 3.3.** **Initial representation** of a text sample *w* is the fact (i.e. the clause without body) *w*. **Representation** of *w* is any Sampletalk clause, whose head is a result of structurization and generalization of *w*.

**EXAMPLES** of samples and their representations are given in Table 3.1:

| ***No.*** | ***Sample*** | ***Possible representation*** |
| --- | --- | --- |
| 1 | **a=>a** | **[A]=>[A]..** |
| 2 | **ab=>ba** | **AB=>BA..** |
| 3 | **xy=>yx** | **[AB]=>[BA]..** |
| 4 | **abc=>cba** | **[ABC]=>[CBA]..** |
| 5 | **reverse of a is a** | **reverse of [A] is [A]..** |
| 6 | **reverse of a/b is b/a** | **reverse of [A/B] is [B/A]..** |
| 7 | **reverse of a/b/c is c/b/a** | **reverse of [A/B/C] is [C/B/A]..** |
| 8 | **grass is a plant** | **X is a plant..** |
| 9 | **Green is a color** | **X is a color..** |
| 10 | **grass is green** | **X is Y:-X is a plant,,Y is a color..** |
| 11 | **(y/x)a=a** | **(Y/X)[A]=[A]..** |
| 12 | **(y/x)axb=ayb** | **(Y/X)[AXB]=[AYB]..** |
| 13 | **(y/x)axbxc=aybyc** | **(Y/X)[AXBXC]=[AYBYC]..** |
| 14 | **polish(a+b) is +ab** | **polish([A]+[B]) is +[A] [B]..** |
| 15 | **polish(a\*b) is \*ab** | **polish([A]\*[B]) is \*[A] [B]..** |
| 16 | **polish((a+b)\*(c+d)) is \*+ab+cd** | **polish([A]Z[B) is Z[A] [B]..** |
| 17 | **a fish lives in a sea; where does a fish live? in a sea** | **a X Ys in a Z; where does a X Y? in a Z..** |
| 18 | **a tree grows in a forest; where does a tree grow? in a forest** | **a X Ys in a Z; where does a X Y? in a Z..** |
| 19 | **axb contains x** | **AXB contains X..** |
| 20 | **new york is situated in america** | **X is situated in L..** |
| 21 | **a book is situated in the bag** | **X is situated in L..** |
| 22 | **where is a book? in the bag** | **where is X? in L..** |
| 23 | **where is new york? in america** | **where is X? in L..** |
| 24 | **x0 is variable** | **[x0] is variable..** |
| 25 | **x10 is variable** | **[X10] is variable..** |
| 26 | **(forall x0)a or (forall x0)b -->**  **(forall x0)(forall x10)(a or b)** | **(forall X)[F] or (forall X)[H] -->**  **(forall X )(forall Z)([F] or [H])..** |
| 27 | **(forall x0)a(x0,y) or (forall x0)b(x0,t) -->(forall x0)(forall x10)(a(x0,y) or b(x10,t))** | **(forall X)[F] or (forall X)[H]-->**  **(forall X)(forall Z)([F] or [G])..** |
| 28 | **Permutation for 123456 is 412356** | **permutation for [A[N]B] is [NW]..** |

**Table 3.1. Text samples and their representation as Sampletalk clauses**

Clause 2 from this table is minimal common generalization of samples 2 and 3. Clause 3 is result of structurization of clause 2. Clauses **reverse of A/M is N/A** and **reverse of M/C is C/N** are minimal common generalizations of samples 6 and 7. Clause **reverse of [A/M] is [N/A]:-reverse of [M] is [N]** is generalization of clause **reverse of [A/B/C] is [C/B/A]:-reverse of [B/C] is [C/B]**. Here sub-samples **B/C** and **C/B** have been replaced with variable names **M** and **N** respectively.

We can see that generalization of a sample always matches this sample.

**DEFINITION 3.4.** Let *H1****:-****B*1 and *H2****:-****B2* be such results of structurization and generalization of clauses *C1* and *C2* respectively, that all variable names of *H2* belong to *H1****:-****B1* (*B1* and *B2* designate bodies of the corresponding clauses, they may be empty). Combined clause *H1****:-****B1,,H2* is called **composition** of clauses *C1* and *C2*. Initial clauses are called parents of the composition.

**EXAMPLES.** Clause **reverse of [A/M] is [N/A] :- reverse of [M] is [N]** is composition of clause 7 with clause 6 from the above table. Here *H2* is the sample **reverse of B/C is C/B**. During generalization, sub-samples **B/C** and **C/B** were replaced with variable names **M** and **N** respectively. Other examples of compositions are presented in Table 3.2.

| ***No.*** | ***Parent*** | ***Parent*** | ***Composition*** |
| --- | --- | --- | --- |
| 29 | 4 | 2 | **[AM]=>[NA] :- [M]=>[N]..** |
| 30 | 7 | 6 | **reverse of [A/M] is [N/A] :- reverse of [M] is [N]..** |
| 31 | 7 | 6 | **reverse of [M/C] is [C/N] :- reverse of [M] is [N]..** |
| 32 | 13 | 12 | **(Y/X)[MXC]=[NYC] :- (Y/X)[M]=[N]..** |
| 33 | 13 | 12 | **(Y/X)[AXM]=[AYN] :- (Y/X)[M=[N]..** |
| 34 | 16 | 14 | **polish([M]\*[(C+D)) is \*[K] [+CD] :- polish(M) is K..** |
| 35 | 34 | 14 | **polish([M]\*[N]) is \*[K] [L] :- polish(M) is K,, polish(N) is L..** |
| 36 | 22 | 21 | **where is X? in L :- X is situated in L..** |
| 37 | 23 | 20 | **where is X? in L :- X is situated in L..** |
| 38 | 27 | 24 | **(Q X)[F] or (Q X)[H]-->(Q X)(Q Z)([F] or [G) :- [X] is variable..** |
| 39 | 38 | 24 | **(Q X)[F] or (Q X)[H]-->(Q X)(Q Z)([F] or [G) :- [X] is variable ,, [Z] is variable..** |
| 40 | 39 | 33 | **(Q X)[F] or (Q X)[H]-->(Q X)(Q Z)([F] or [G]) :- [X] is variable ,, [Z] is variable ,, (Z/X)[H]=[G]..** |

**Table 3.2. Composition of Sampletalk clauses**

Note that clauses 30 and 31 (as well as 32 and 33) are essentially different compositions of the same clauses. Note also that clauses 36 and 37 are identical compositions of parent clauses of different origins.

Composition of clauses may involve new recursion. Therefore, we obtain much more powerful clauses for text processing than the initial representations of samples. Indeed, group of clauses, consisting of a composition and one of the parents or another simple clause, can serve as the recursive rule and the boundary condition for reversing any word (pair 29,1), reversing any list of words separated by slash-symbols (pair 30,5), for replacing non-intersecting occurrences of a sub-word X of a word W by a word Y for any X,W,Y (pair 33,11), for processing of natural language phrases, for transformation of mathematical and logical expressions etc.

To clarify semantics of such programs, consider Program 3, constructed from pair (33,11), with some goal. (See also Program 1 where these clauses are applied).

| % Goal (W stands for the result of the replacing):  **(z2/z1)[(sin(z1)+cos(z1))]=[W]..**  % Program:  **(Y/X)[AXM]=[AYN] :- (Y/X)[M]=[N]..** (33)  **(Y/X)[A]=[A]..** (11) |
| --- |

**Program 3. Representation of common notation for term substitution in an expression. The result will be (z1/z2)[(sin(z1)+cos(z1))]=[(sin(z2)+cos(z2))]**.

This program performs global replacing of sub-words and may be used for changing variables in logical or arithmetical expressions. Sub-words **z1** in word **(sin(z1)+cos(z1))** will be globally replaced with sub-words **z2**. Consider this in more detail. Program goal matches the head of clause 33, then instance **(z2/z1)[(sin(z1)+cos(z1))]=[(sin(z2N]** of this head is constructed. After this matching, **Y** is **z2**, **X** is **z1**, **A** is **(sin(**, **M** is **)+cos(z1))**, **N** is undefined. The sub-goal in the body of this clause will have the form **(z2/z1)[)+cos(z1))]=[N]** (remind that “**=**” is an ordinary character rather than a specific construction of Sampletalk language).

Once more, this sub-goal matches (33), and we have: **A** is **)+cos(**, **M** is **))**, **N** is undefined. The next sub-goal **(z2/z1)[))]=[N]** does not match (33) but matches (11), and successive bindings will give us the following: **N** is **))**; return; **N** is **)+cos(z2))**; return; **W** is **(sin(z2)+cos(z2))**, and the output of the program will be **(z1/z2)[(sin(z1)+cos(z1))]=[(sin(z2)+cos(z2))]**. Compare this output to samples 11-13 from the above table and note its similarity to them. It may be considered as their inductive extension for the domain of arithmetic expressions.

Pair (32,11) does not fit the task of global replacing variables in an expression: the program, consisting of this pair of clauses, replaces only the first occurrence of **X** by **Y**.

| % Goal (W stands for the result of the replacing):  **(z2/z1)[(sin(z1)+cos(z1))]=[W]..**  % Program:  **(Y/X)[MXC]=[NYC] :- (Y/X)[M]=[N]..** (32)  **(Y/X)[A]=[A]..** (11) |
| --- |

**Program 4. This program replaces only the first occurrence of z1 with z2. The result will be (z1/z2)[(sin(z1)+cos(z1))]=[(sin(z1)+cos(z2))]**.

This is due to the standard strategy of matching samples (see Convention 1.4 from previous section). Likewise, a program, consisting of the pair (31,5), shifts the first element of a word list into the end of the list rather than reversing the list, as the pair (30,5) does.

The following two programs operate with word lists.

| %Goal  **reverse of [abc/def/gh] is X..**  % Program  **reverse of [M/C] is [C/N] :- reverse of [M] is [N]..** (31)  **reverse of [A] is [A]..**  (5) |
| --- |

**Program 5. Shifting the first element of a word list into the end of the list. The result will be the reverse of [abc/def/gh] is [def/gh/abc]..**

| %Goal  **reverse of [abc/def/gh] is X..**  % Program  **reverse of [A/M] is [N/A] :- reverse of [M] is [N]..** (30)  **reverse of [A] is [A]..**  (5) |
| --- |

**Program 6. Reversing list of words separated by /. The result will be reverse of [abc/def/gh] is [gh/def/abc].**

We recommend the reader to consider pair (30,5) in detailed performance (at least in mind if you don't have a Sampletalk interpreter). This program reverses words rather than lists, so the output will be **reverse of [ab/cd/ef] is [fe/dc/ba]**, similar to (29,1). Example applications of such clauses for reversing words were considered in previous sections.

Some of the program examples, considered in Section 2.2.3, can be obtained from clauses taken from tables 3.1 and 3.2.

Program 7 represents interesting program example, constructed from clauses like 20-23 and 36:

| % Goal:  **where is new york? in L..**  (2.3.1)  % Knowledge base:  **who is X? R :- X is R..** (2.3.2)  **where is X? in L :- X is situated in L..** (2.3.3)  **joe is son of maria and peter..**  (2.3.4)  **julia is daughter of maria and peter..** (2.3.5)  **peter 2 is son of maria and peter 1..** (2.3.6)  **jack 2 is son of julia and jack 1..**  (2.3.7)  **ann is a teacher..** (2.3.8)  **new york is situated in america..** (2.3.9)  **st.petersburg is situated in russia..** (2.3.10)  **a book is situated in the bag..** (2.3.11)  **a tree is situated in the forest..** (2.3.12) |
| --- |

**Program 7. Small reasoning system built from natural language phrases**

In the next section, synthesis of such programs will be discussed. This program represents a reasoning system working with natural language. All its clauses are initial representations, generalizations, and compositions of some natural language expressions, which describe a knowledge base; associations among them simultaneously serve as the language parser, grammar, and as an inference engine. Using this approach, the programmer does not need to design a grammar, parser, knowledge model, and inference engine as separate program components: they are represented *implicitly* in samples.

The program allows one to ask various questions (in the form of program goals), like:

**X is son of maria and Y..** ? (**X** will match **joe**, then **peter 2**, so we can use natural language questions to immediately inquire the knowledge base);

**jack N is Y of Z..** ? (**Y** will be instantiated with son, **Z -**-- with **julia and jack 1**, so we can extract from the knowledge base not only the subjects' names, but also the user's concepts);

**X 2 is son of Y and X 1..** ? ( **X** will be instantiated with **peter**, then with **jack**, so we can express a situation, in which different objects have similar names);

**who is julia? R..** ? (**R** will be instantiated with **daughter of maria and peter**, so we can obtain more complex descriptions of objects from the knowledge base, if such descriptions were provided by knowledge engineer);

**where is new york? in L..** ? (**L** will be instantiated with **america**, and we get the output **where is new york? in america..**, so we can extract not only fragments of the facts, but also compositions or derivations of that fragments);

**what is in the bag? R**.. ? (**R** will be instantiated with nothing; we should include into the database additional knowledge like **what is in X? R:-R is situated in X..** so that we could obtain the result **what is in the bag? a book..**; so, we can immediately use fragments of a natural language discourse and their abstractions in order to enlarge our inference engine).

The knowledge base itself has natural language form and may be easily enlarged in the same manner; we may as easily obtain natural language explanation of derived samples if we use special Sampletalk interpreter option, which prints information about all intermediate matching. For the goal (2.3.1), the explanation will look like the following:

**where is new york? in L;**

**new york is situated in america;**

**where is new york? in america.**

In this way, Sampletalk language ties up some very important programming concepts: Logic Programming, Databases, Knowledge Representation, Generalization, Natural Language Modeling, Symbol Manipulation, Pattern Matching. These concepts are represented in the language implicitly, without complex syntax and semantics. This feature (along with its simplicity) allows us to consider Sampletalk, in the family of AI languages, in a position analogous to the position of Basic language in conventional programming language families. The user may be unaware that simple programs, which he creates using this technology, may be exceptionally complex for other approaches and programming languages.

# 4. **Incremental Sampletalk clause synthesis**

Compare Table 1.2 from our Introduction with Tables 3.1 and 3.2. By means of structurization, generalization and composition of samples (still made manually), we can represent hypotheses about regularities, which are manifested in data processing samples. Sampletalk programs serve to transform these regularities into programs. In this section, we will outline some paths to automatic or semi-automatic construction of such programs, and describe the current state of our experiments with Sampletalk program synthesis.

Consider Program 7 once more. This program (or its program equivalent) can be constructed by the following sequence of actions:

1). Take samples (2.3.4 -- 2.3.12). Include their initial representations into the created program.

2). Take samples: **who is joe? son of maria and peter** and **joe is son of maria and peter**. Construct their composition: **who is joe? son of maria and peter:-joe is son of maria and peter**.

3). Take samples: **who is ann? a teacher** and **ann is a teacher**. Construct composition: who **is ann? a teacher:-ann is a teacher**.

4). Find minimal common generalization of those compositions. It will be sample **who is X? Y:-X is Y**. Include this clause into the created program (clause 2.3.2).

5). Obtain clause, equivalent to (2.3.3), in similar way, from samples involving the concepts **where** and **situated**. Include it into the program.

We can vary this program if we vary initial samples and clause generation strategy. For instance, we may consider samples: **who is joe? son of maria and peter** and **who is ann? daughter of evelyn and ivan** as a stimulus for the generalization, construct clause **who is X? Y of Z and T** from them and then obtain a clause **who is X? Y of Z and T:-X is Y of Z and T** as some composition instead of (2.3.2). This clause will match another class of goals, but while associating with facts (2.3.4 -- 2.3.12) it will work as well as (2.3.2), for any questions but questions concerning **ann**.

So, we have a few actions for transformation of data processing examples into data processing clauses. This transformation can be automated if we allow the computer to form plausible hypotheses and do not bore the user with too difficult actions for evaluating them. For simple samples, such as 1 -- 4 from Table 3.1, all possible representations, generalizations and compositions can be found by simple exhaustion. The task of the user, who composes a program using the synthesizer, in this case is reduced to evaluation of generated compositions against other examples (e.g. test whether the generated program can reverse a test word **abcdef**). This testing can be automated using positive (correct) and negative (incorrect) examples of text transformation.

In more complex case, when such exhaustion is impossible, we may use an ordered teaching sequence of samples, as well as some previously constructed clauses, where useful (from the user's point of view) regularities are already discovered and stored as a useful knowledge. Here we assume that the user can evaluate each synthesized clause as successful or unsuccessful. He can test it against other examples of correct and incorrect text processing, or use special evaluation functions for this purpose. Some problems with this approach are mentioned below.

Let *w1,w2,...,wk* be an incremental sequence of samples. We do not define here the concept of incremental sequence of samples in a formal way. Informally, we assume that there exists some similarity of any *wi* to a *wj* (*j<i*) and *wj* is simpler than *wi* (*wj* may be an empty word). Probably, the user can provide the synthesizer with such a sequence of samples, based on his intuitive concept of text similarity and complexity.

We assume that the synthesizer can take into account already created clauses for synthesis of new clauses. Various ways to organize a synthesizer by using composition and generalization of clauses are possible. A general scheme of our synthesizer is defined by the following procedure:

**ALGORITHM 4.1.** (Synthesis of a program **S**).

Given text samples ***w1,w2,...,wk***, perform the following steps, starting from ***w1*** and terminating with ***wk*** :

1. Build some representation ***Ci*** of sample ***wi***, which was not built before, using structurization, generalizations and compositions of ***C1,C2,...,Ci-1*** and of initial representations (i.e. facts**) *w1,w2,...,wi***.

If the user approves ***C****i*, place it at the top of the program **S** and go to Paragraph 2. Otherwise, go to Step 1.

2. Choose the next sample and go to Step 1. If ***i = k***, then the program **S** is built.

The following theorem, along with Theorem 2.2, shows that this synthesizer can be used for synthesis of any algorithm.

**THEOREM 4.1.** For any Sampletalk program **S**, an equivalent Sampletalk program **S1** can be constructed by applying Algorithm 4.1 to some sequence of samples ***w1,w2,...,wk***.

(see Proof in Appendix).

This theorem has interesting implications related to programming technology. It shows that we can build any algorithm without programming at all, simply by combining data examples that expose behavior of the desired algorithm. Programming in Sampletalk language assumes that we build the algorithm from examples manually. Now we find that we don't even have to build the program manually. Algorithm 4.1 can do this for us. Our task is only to provide examples for the program synthesizer and organize the examples in such a manner that the necessary regularities will be discovered in the necessary order.

So, ***we can build any algorithm without programming in a computer language***: all we need is to form the necessary sequence of data processing examples, which is characteristic for the algorithm being built. This way of programming assumes an alternative way of thinking about objects and processes and can lead to powerful programming techniques. Sampletalk program examples illustrate this for natural language processing and math applications.

Today, however, people don’t use data generalization while writing programs. Probably, for some applications, forming a characteristic sequence of data examples is not an easy task. This method needs experimental approval and experience.

Algorithm 4.1 represents only a general program synthesis scheme and may generate too many program versions to be useful in applications. We should modify it for special text fields or for obtaining special kinds of programs. Many options for such modification are possible.

We can do without initial representation of samples ***w1, w2, ..., wi-1***, and use only clauses ***C1, C2, ..., Ci-1***and***wi*** at each step of the synthesis; the version space is reduced. Another way of reducing the version space consists in applying only minimal common generalizations to the initial samples. We may apply special constraints while underlining and generalizing sub-samples. Interesting generalization strategy consists in underlining each replaced sub-word in resulting clauses, so that the user will be able to control the process of synthesis by including matching and mismatching sub-words into initial samples.

The result of this interactive program synthesis depends on the initial sequence of samples and on evaluation of generated clauses by the user. Each step may lead to an explosion. However, since the user is the source of the initial information, we can expect a reasonable combination of his informal assumptions about samples, with formal properties of those samples. Proper sequence of samples and correct strategy for underlining perspective sub-words may drastically reduce exhaustion of possible compositions and generalizations.

Synthesizer may be organized in such a way that any Sampletalk program can be generated without exhaustion at all, provided with a proper sequence of samples. The synthesis in this case is deterministic and is similar to compilation. However, providing such a detailed and precise sequence of samples may be a very complex task.

So, we have two extreme approaches to interactive program synthesis:

* Simple sequence ***w1,w2,...,wk*** and complex user's actions for evaluating generated clauses that may lead to explosion;
* Complex incremental sequence ***w1,w2,...,wk*** (beginning from basic sub-words for structurization of further constructions) and trivial user's actions (if any) for evaluating clauses.

A reasonable synthesis technology is, probably, somewhere between. We can simplify preparation of samples and make the computer work out a reasonable class of constructions.

The next important possibility for composing new programs consists in using already created and verified programs rather than single clauses. Suppose, for instance, that we want to build a program for transformation of analytic expressions into a canonical form. Usually this task involves sorting and reduction (summation) of homogeneous items. We may use 1) an already created program for sorting a list of items; 2) samples for reduction of sub-lists of homogeneous items. We should not mix samples for sorting with samples for reduction, since they may interact in undesired ways and provide unpredictable results. Technically, we can build these program fragments into the generated program by marking clauses with a special prefix (which plays the same role as unique predicate names for groups of Prolog clauses). In this way, we can reduce exhaustion and help the user to control the process of synthesis.

So, the system, depicted here, is not a simple Machine Learning system, but one, which learns from a clever tutor: the result of learning heavily depends on text samples and on applied strategy of synthesis.

Current experiments with Sampletalk language interpreter have lead us to new programming technology, which can be described as follows: ***data processing example abstraction immediately provides a useful program if matching of text abstractions is in focus***. Now we have various applications with symbol-numeric processing in linguistics, music and algebra (see /15/,/20/). In experiments with synthesis, we implemented a procedure for building common and minimal common generalizations for a set of samples. Other steps of the synthesis algorithm are currently performed manually, in order to find and verify reasonable strategy of algorithm synthesis for given classes of samples. This is our appeal for the reader: detail or modify Algorithm 4.1 for your data samples and, probably, you will get new programming technology for processing your data.

# **Related work**

Generalization technique in logic calculus was systematically described in Michalsky /33/ and applied and investigated in various works, e.g. Langley et al /30/ (for discovery of new chemical compounds), Gleibman /20/ (for discovery of equations with complex term structure). This technique causes considerable interest and applications in latest research. Furtado /13/ presents a Prolog program, which builds the most specific generalization (minimal common generalization in our terminology) for a pair of Prolog structures which may contain various combinations of identical variables.

Generalization of sub-samples, composition and generalization of clauses, described here, may be considered as application of this technique in Sampletalk context. Other kinds of generalization, proposed by Michalsky (e.g. generalization of antecedent), probably, will be used in future versions of our synthesizer.

The paradigm of inductive program synthesis has been investigated in various works (beginning from synthesis of Turing machines and recursive functions). The approach of Barsdin, Kinber and Brasma /4/, /5/, /28/, /29/ is based on generalization of computation examples. Authors propose special programming language for expressing sample computations and algorithms, and then introduce techniques for detecting so-called syntactical analogies in examples. The analogies are basically related to fragments of arithmetical progressions, that implicitly reflect assignments in the form x:=x+c and x:=x-c. Reasoning and program synthesis are related to arrays. Set of expressions like “a(3)<b(1)? yes, then a(3)->c(3)” is generalized into an array processing program (for sorting, merging arrays etc). The implementation language for generated algorithms (so called graphic language) contains traditional programming constructions for cycles, arithmetic operators, logic conditions and so on. This work is related to program synthesis from full information-logical history of constructed programs (see Bierman /6/, Angluin & Smith /3/).

In Vityaev /40/, Prolog program synthesis is related to discovery of probabilistic regularities in Prolog expressions. Shapiro's /36/ approach to Prolog program synthesis is based on covering the input set of Prolog clauses by more general clauses.

There is a work in synthesis of formal language grammars, regular expressions, and templates by examples. Angluin /2/ introduced an algorithm for constructing the most specific template for a group of text examples and suggests using this algorithm in data input systems and text editors (this is a proposal to organize text editing by example).

Chuzhanova /8/ presents a system for building pattern grammars for the analysis of genetic texts. Synthesis is based on detection and inductive generalization of specific types of regularities (repetitions, inversions, and common subsequences) in a finite set of positive examples. These types of regularities represent a-priori information for the system. (Is this a universal knowledge in the sense of Table 1 from our Introduction?). In our approach, we try to minimize applied universal knowledge as much as possible. The mentioned types of regularities for the genetics texts might be expressed by the following Sampletalk clauses with recursion:

**word [AXB] contains repetition of the word [Z]:-**

**[X] is repetition of [Z]..**

**[XX] is repetition of [X]..**

**[ZX] is repetition of [Z]:- [X] is repetition of [Z]..**

**words [AXB] and [CYD] have inversion fragments [X] and [Y]:-**

**reverse of [X] is [Y]..**

**reverse of [AM] is [NA]:-reverse of [M] is [N]..**

**reverse of [A] is [A]..**

**words [AXV\*] and [BXW\*] have common subsequence (X,S.):-**

**words [V\*] and [W\*] have common subsequence (S.)..**

**words [\*] and [\*] have common subsequence (.)..**

(see also clauses 29,1, 30 and 5 from Tables 3 and 4 in Section 2. The asterisks play the same role as empty list **[ ]** in Prolog programs. On the other hand, concepts of repetition, inversion and common subsequence are fundamental for research in genetics, and may be regarded as the universal knowledge.

This example shows that our minimalist approach to universal knowledge does not hinder us from expressing specific kinds of universal knowledge. This confirms the simplicity and power of our knowledge-processing tool, in comparison to other knowledge processing tools.

Many papers describe grammar induction. In Vanlehn and Ball /39/, version space for possible grammars, consistent with a set of examples, is investigated. Jansen-Winkeln /26/ describes the grammar rule generation system LEGAS, which is based on maximum and minimum specific conjunctive generalization of example string sets. From the example set D= (lady -> ladies), (body -> bodies) the following rules are constructed: str,[dy] -> str,[dies] (maximum specific); str,[y] -> str,[ies] (minimum specific). In Sampletalk notation, the same rules are represented as generalizations of samples lady ---> ladies and body ---> bodies into Xdy ---> Xdies or Xy ---> Xies. However, we are not so concentrated in formal language grammar (is this a universal knowledge concept?).I believe that this concept should be excluded from explicit use anywhere it's possible. Our natural language processing programs (refer to Program 6 in Section 2 and to the program for morphologic analysis of Finnish language [17]) do not use this concept and therefore are simple.

Considering these examples, note that researchers make hard efforts to transform natural language descriptions into formal objects as relational databases (e.g. see Alshawi /1/), frames, grammars and so on. The objective of this transformation is application of reasoning tools that are more usable for formal objects than for NL phrases. Our examples show that, sometimes, the direction of this transformation should be opposite: we would better represent necessary reasoning in the form of natural language reasoning, and then compose a special reasoning system from abstractions of these texts. This is my recommendation for natural language processing designers. Sampletalk seems to be quite useful for implementation of this approach.

Approach to analysis and synthesis of formal constructions on the ground of informal human reasoning has its own history and applications. Zavalishin and Muchnik /43/ investigated human mechanisms of analysis of complex graphic images (including geometrical illusions). They experimented with verbal image descriptions. The authors present a table of graphic images with informal hypotheses about them, which is similar to our Table 2 from Section 1 (for graphic case), and try to build a shape dictionary for a graphic language.

There is research in synthesis of new descriptions based on analogies (e.g. Winston /42/; Haraguchi and Arikawa /23/). In the second paper, pairing of terms for two Prolog programs is defined. It is used for constructing new clauses, and may involve new recursion. Our means of generating new recursive rules are based on text matching rather than analogies, although these concepts have much in common.

Botta /7/ introduces the concept of conjunctive maximally specific characterization (MSC) of a set of examples, along with algorithms for generation of MSC. The author represents examples from a block world domain. The MSC, the most specific generalization, our minimal common generalization of samples, and other constructions of this kind (e.g. Michalsky /33/) are analogous constructions that highlight similar elements among instances in various fields. Usage of this tool in discovery and application of regularities is developed in AI research and applications. Critical point of view is expressed by Macdonald and Witten /32/: they believe that discovery based learning (at all) is quite immature and does not yet form a foundation for practical knowledge acquisition techniques.

The technique of *synthesis from examples* has been investigated in many other works. Gallant /14/ describes the generation of a medical expert system from examples of diseases and treatments. The induction is based on so-called learning matrix (a matrix of dependencies between terminal variables and goal variables). This matrix is constructed by analysis of the dependencies in the example treatment descriptions; the generated expert system is built up on this matrix. This approach can be used for any problem domain, which may be described by means of discriminate characteristics; the synthesized inference engine is based on dependencies between them. There are various approaches to automatic generation of formal descriptions and generalizations in such fields. Examples of descriptive rules formation for a large set of experimental data with discriminate characteristics are described, for instance, in Ho et al /25/. The generation is based on a priori information about attributes of observed objects and on set-theory relations between classes of objects. This approach is suitable for a description of discrete events or facts, which are not bound by functional relations. For working with text data (as in our paper) or continuous physical phenomena, it is more suitable to use a logic calculus or logic programming language than set theory, for complex concepts and interrelations can be implicitly expressed in them. Delgrande /10/ describes an approach to formation of hypotheses about relations on domains that are represented by a set of positive and negative facts. For representation of hypotheses a special formal language is used which contains functors analogous to traditional set-theory operations and logic copulas. From our point of view, this set of functors also isn't suitable for working with arbitrary text data.

In Gleibman /20/, a formal approach to hypothesis formation and generalization is introduced, for learning the structure of a continuous physical world interconnected by complex functional relations. The proposed discovery component deals with simultaneous differential equations; some heuristics are given for forming new equations from case revelations of the physical phenomena. This approach leads us to a conceptual generalization (of the concepts which are implicitly represented in the equations), while the present work is more close to a descriptive generalization (in the sense of Michalsky /33/). There are various projects concerning reasoning about equations (see review in Silver /37/). Silver describes inductive generation of new equation processing rules on the basis of correct inference examples provided by the user.

Constructing Sampletalk language, we have eliminated syntax features of Prolog language, which hinder the synthesizer (as well as the human programmer) from assuming analogies and comparing predicates with different names and term structures. At the same time, symbol constants in samples, such as constants **)**, **(**, **=**, or more expressive constants such as **is**, **contains**, **not** etc., may constrain Sampletalk program performance in the similar way as Prolog specific constructions constrain Prolog performance (see Sampletalk version of Clocksin and Mellish Prolog program for working with parts inventory in Section 1 and our representation of merge sort algorithm in Appendix 2. These programs present results of direct rewriting of the corresponding Prolog programs in Sampletalk language).

The advantage of notation with underlines for designation of sub-samples consists in keeping text entirety and readability; this notation may be considered as an attempt to involve graphic elements into the programming language area (we feel that such elements, as well as indices and fonts may not only improve readability of the program code, but provide a new expressive power for programming languages). We believe that it is an interesting research problem. Some consideration of this problem is given in D.Knuth's article "Literate Programming".

Although Sampletalk language initially was designed only for experiments with automatic program synthesis, it turned out (surprisingly for us) to be useful for programming by a human. Sampletalk is extremely simple. At the same time, Sampletalk comprises and ties up the most expressive elements of such languages as Prolog (clause call, recursion, backtracking, unification), Refal and Planner (list matching and involving variables into the further processing). Curiously, the mentioned programming technology, based on the human ability to generalize his data, became clear only after debugging the interpreter on samples from different fields. Our implementation and experiments with Sampletalk language demonstrate its power for various classes of problems (especially linguistic problems) and domains.

Some technical relation may be mentioned, between our approach to generalization and some other approaches. In systems GLAUBER, NGLAUBER (see Langley et al /30/ and also Jones /27/), generalization of statements is done by adding and expanding the effect of quantifiers in the first order calculus. The obtained hypotheses are verified according to input statements (that are represented similar to Prolog facts). In our case, replacing sub-samples with variables does the similar action.

An important problem, which arises with the inductive program synthesis, is verification of the generated programs. This problem is analogous to the problem of evaluation of hypotheses in other discovery systems (e.g., see Mitchell et al. /32/). Probably, the technique of programming, proposed in our work, is more feasible for easy creation of program prototypes with non-critical reliability or for creation of programs whose working results are immediately analyzed by the user (e.g. formation of plausible hypotheses for semi-automatic recognition of an ancient natural language text on the basis of bilingual text samples; an immediate synthesis of a prototype of a translator for some artificial language represented by sample expressions and their translations, etc.). Development of new methods for a formal verification of the synthesized programs can be connected with formal analysis of text samples.

Application of Sampletalk language and technology to linguistic problems seems to be very promising. In Gleibman and Kirsanov /17/, check-spelling problems for such a highly inflected language as Finnish is considered. Sampletalk program is presented for resolving this problem. The program is written directly in terms of linguistics rather than in programmer's slang, and consists of clauses like

**correctness of the word [A] is [correct(A)]:-**

**X of [Y] will be [A],,**

**[Y] is a stem..**

**present tense 3rd person singular of [A] will be [DVV]:-**

**[A] ends in vocal [V], begins with [D]..**

**[Y] is a stem:-stems X/Y/Z..**

**stems //ole/tule/saa/voi/sano/mene/n\"ake/pit\"a/tiet\"a/teke//..**

Analogous program for Russian check spelling was written by the author. Using such clauses, we can describe morphology of many languages, as well as inference rules for deduction of new constructions (e.g. translations) from existing ones. Note that knowledge in the computer programming area is not required for the author of this program. Important is understanding of the process of *matching samples.* Now we can imagine clause synthesis technology for automatic or semi-automatic extraction of such clauses from linguistic literature.

Semiautomatic Sampletalk program synthesis resembles language grammar retrieval based on the examples of language expressions. For example, rather than designing a natural language grammar, we can use sample words such as *embedded, emphasized* for generating samples *emXed, Xed*, which may implicitly represent concepts of prefix, suffix and tense. In explicit form, these concepts may be not in use at all in the program. Its designer may do without these concepts. Bilingual translation samples may be used for developing a natural language text translator without programming, simply by choosing and structuring suitable samples.

Our version of the synthesizer is interactive. It is using evaluation of generated clauses by the user. In other versions, this evaluation may be automated by statistical analysis, heuristic evaluation functions, by synthesis and evaluation of characteristic data examples for evaluation of given clauses, etc.

The synthesizer may be organized so that only clauses *C1,C2,...,Ci-1* and the initial representation of the sample *wi* will be used for creating *Ci*. Such a synthesizer is still able to generate any algorithm, but requires another approach for choosing samples and evaluation of the generated program. What kind of synthesizer is suitable for which sort of problems?

Probably, participation of clauses, which were already included into the program being created, is preferable for the synthesis process, in comparison to their text origins. Experiments with various synthesizers and sets of samples may clarify this question and give us a pragmatic concept of their applicability.

Paradigm of *creating something new on the basis of observing something* instead of (or, to some extent, along with) using sophisticated a priori knowledge, is rather promising not only in relation to inductive learning, discovery and generalization. Many interactive computer systems may be related to this paradigm if we consider the user as a part of the system (e.g., systems for interactive creation of graphic images from basic icons: the user himself represents the universal knowledge for creation of new images). Well-known Query-by-Example language (Zloof /44/) is related to analysis of the user's actions with a model of a relational table on the screen; the system's actions with real tables are generated automatically from these actions.

The most important feature of such systems (from our point of view) is the possibility to create essentially new complex objects without programming, simply by combining and transforming existing ones "by hand", in accordance with their own automatic properties. Where are the limits and advantages of this approach?

In Gleibman /22/, this approach is investigated for creation and animation of graphic images by using basic images (graphic samples and their abstractions) under constraints influenced by other graphic images and some a priori types of transformation and composition of samples. The latter types represent a universal knowledge for the graphic synthesizer (linear and other transformations, composition of graphic objects by gluing "point to point", "a point slides along a line", by gluing lines with rolling friction; animation of compositions by gluing objects with some earlier animated objects, etc). This approach allows one to create complex technical scenarios without programming, simply by manual processing and controlling samples. Technology of program synthesis, described here, may be considered as an attempt to expand this approach into the data processing area.

# **Summary**

We have shown that various kinds of useful text processing programs can be created by providing the synthesizer with demo samples of data processing, instead of writing program code.

Sampletalk is a universal programming language. Sampletalk programming and program synthesis are essentially based on data examples and their structurization, generalization and composition. Natural language reasoning is one of the prospective applications for Sampletalk.

Technique for program synthesis, described here, can be used for producing any word processing algorithm, based on a proper set of text samples.

# **Appendix. Proofs**

**1. Proof of the Theorem 2.1.** Construction of the program **P** we begin by writing each clause of **S** in the Prolog form in accordance with Conventions 1.2 and 1.3 (we choose some unique predicate name pname for all predicates). Then, for each matching pair of samples l, t of **S** (caller and the one being called), we construct all possible groups of Prolog clauses in the form

pname(L):-pname(L1,M,N,K), (1)

pname(L1,M,N,K):-pname(T1,M,N,K), (2)

pname(T1,M,N,K):-pname(T), (3)

Here list L (T) is the Prolog notation of the sample l (t), list L1 (T1) is formed from the list L (T) by adding zero or more pairs of square brackets so that pname(L1) matches pname(T1) as Prolog predicates (let us call them specifications of the original samples), M is the number of a calling clause, N is the number of a call in the clause body, K is the number of a clause being called. Place these groups in the program P in accordance to Convention 1.4 of Section 1.

This set of clauses is finite due to the finiteness of the set of all possible specifications of the samples (we do not allow lists like [[L]] within these lists).

As the goal for this program we will use predicate pname(WP), where WP is the Prolog representation of the goal of **S**.

Now let us show that the program **S** is equivalent to the program **P**. For any successful Sampletalk call (l,t), where l is a sample from a clause body, t is the head of a clause being called, there exist such specifications L1 and T1 that L1 match T1 according to the Prolog unification rules. In the program **P** the corresponding call is implemented by the successive calls (1), (2), (3). These calls make the same environment change (i.e. bindings of free variables) as the above Sampletalk call.

Likewise, for each successful sequence of Prolog calls in the form (pname(L), pname(L1,M,N,K), pname(T1,M,N,K), pname(T)) in the program **P**, the correspondent environment change is made by the Sampletalk call (l,t), where l and t are Sampletalk notations of the predicates pname(L) and pname(T). So, the programs **S** and **P** are equivalent.

**2. Proof of the Theorem 2.2.** Compose the program **SM** of the clauses in the form

**X**ai**Y==>W :- X**bi**Y==>W..** (1)

for the normal (Markov) rules ai -> bi of the algorithm **M**, and of the facts in the form

**X**aj**Y==>X**bj**Y..** (2)

for the terminating rules aj ->. bj of the algorithm **M**. The order of the clauses in **SM** corresponds to the order of the rules in **M**. Clause (fact) **X==>Z..** we add at the bottom of **SM**. Here **X**, **Y**, **W** are Sampletalk variable names.

Now let us show that the working of **SM** is equivalent to the working of **M**. Let word **w==>W** be the goal for **SM**. If **w** is a recognized word for **M** and one of the rules ai -> bi or aj ->. bj is applied to **w**, then **w** can be represented in the form **X**ai**Y** or **X**aj**Y**, and therefore in **SM** one of the clauses (1) or (2) is called. In the case (1), a sub-goal *w'* **==> W** is formed, where w' is a word produced by the algorithm **M** from **w**. In the case (2), the program will resolve the goal and stop. In this case, variable **W** gets value **X**bj**Y** with **X** and **Y,** which were bound before. This corresponds to the result of applying **M** to **w**.

Due to Convention 1.4 (Section X.X.1), all the variables **X** in the clauses (1), (2) will match sub-words of minimal possible sizes. This corresponds to the canonical application of the substitution rules in Markov algorithms.

Consider the case when **w** leads **M** to the deadlock stop, i.e. when there is some derivation w' of **w** for which no rule of **M** is applicable. In this case, program **SM** comes to a sub-goal w' **==> W** which does not match any clause head of **SM** but the last one. This last clause **X==>Z** serves for preventing backtracking; variable **W** matches the undefined variable **Z**, and the program stops.

At last, let word **w** be such that **M** will never stop to process it. This means that for each derivation w' of **w** only rules in the form (1) are applicable. In the program **SM** this process corresponds to successive bindings **W=**w', **W=**w'' etc., and the program will never stop.

**3. Proof of the Theorem 4.1.** Let *C1,C2,...,Cn* be the clauses of the program **S** in reverse order. To build **S1**, we construct an equivalent clause for each *Ci* and keep the order of these clauses, beginning from the bottom of **S**.

First, we construct a sequence E of samples for **S1** using the following procedure:

For *i = 1,2,...,n,* perform the following steps:

1) If *Ci* is a clause without body (i.e. fact), construct two instances *ui* and *wi* of *Ci* so that *Ci* is their minimal common generalization. This can be done by globally replacing all variables of *Ci* with two sets of unique samples (one set for *ui* and one set for *wi*). Add *ui* and *wi* to E.

2) If *Ci* has a body, construct a unique instance *wij* for each its sample b\_j (including the head) by globally replacing each its variable with some unique sample, so that similar variables correspond to similar samples. Add all such *wij* to E.

Now let us apply Algorithm 4.1 to the sequence E. For each *Ci* there exists a pair of instances *ui* and *wi* or a set of instances *w*ij in E such that *Ci* is their minimal common generalization or composition. Thus, some clause *Ci'*  can be created, such that the only differences between *Ci* and *Ci'* are the names of the corresponding variables, so that these clauses are equivalent. Therefore, the program **S1** is equivalent to the program **S**.

## ***REFERENCES***

1. Alshawi, H. Creating Relational Databases From English Texts. Artificial Intelligence Applications. The Engineering of Knowledge-Based Systems. Ch. R. Weisbin, ed. IEEE CS Press. North-Holland, 1985, pp. 390-395.

2. Angluin, D. Finding patterns common to a set of strings. J. Comput. Systems Sci. - 1980, V.21, pp.46-62.

3. Angluin, D. and Smith, C.H. Inductive Inference: Theory and Methods. Comput. Surveys, 1983, V.15, No. 3, pp. 237-364.

4. Barsdin, J.M., Brasma, A.N. and Kinber, E.B. Inductive Synthesis: State, Problems, Perspectives. Kibernetika, No.6, 1987, pp.81-86,90. In Russian.

5. Barsdin, J.M. Some rules of inductive inference and their role for program synthesis. Information Processing 83. Amsterdam: North Holland, 1983, pp. 333-338.

6. Bierman, A.W., Krishnaswamy R. Constructing programs from example computations. IEEE Trans. Software Eng. - 1976, No.2, pp. 141-153.

7. Botta, M. Constructive learning in conjunctive concept characterization. Methodologies for Intelligent Systems, 3. Z.M.Ras, L. Saitta, eds., North Holland, 1988.

8. Chuzhanova, N.A. Inductive synthesis of programs for symbolic sequences processing. Analogical and Inductive Inference. Intern. Workshop AII'89 Proc., Reinhardsbrunn Castle, East Germany, Spr.-Ver. 1989, pp.317-327.

9. Clocksin, W.F. and Mellish, C.S. Programming in Prolog. Springer-Verlag, 1984.

10. Delgrande, J.P. A formal approach to learning from examples. Int. J. Man-Machine Studies, V.26, 1987, pp. 123-141.

11. Dontas, K. and Zemankova, M. APPLAUSE: An Experimental Plausible Reasoning System. Methodologies for Intelligent Systems, 3. Z.M.Ras, L. Saitta, eds., North Holland, 1988.

12. Falkenhainer, B.C., Michalsky, R.S. Integrating Quantitative and Qualitative Discovery: The ABACUS System. Machine Learning, V.1, pp.367-401, 1986.

13. Furtado, A.L. Analogy by generalization --- and the quest of the grail. ACM SIGPLAN Notes, V.27, No. 11, January 1992, pp.105-113.

14. Gallant, S.I. Automatic generation of expert systems from examples. Artificial Intelligence Applications. The Engineering of Knowledge-Based Systems. Ch.R.Weisbin, ed. IEEE CS Press. North-Holland, 1985, pp. 313-319.

15. Gleibman, A.H. Sample: New Programming Technology and AI Language (Subtitle: Data processing example abstraction immediately becomes a useful program if text matching is in focus). *ACM SIGSAM Bulletin No.2, 1992, 10 pp.*

16. A.H.Gleibman. Sample: The Simplest Language for Artificial Intelligence Problems. *Preprint of the Institute of Theoretical Astronomy of the Russian Academy of Sciences No.19, 1992, 54 pp. In Russian.*

17. Andrew H. Gleibman and Nikolai O. Kirsanov. Data generalization as a Programming Technology. *Manuscript, 8 pp. In publishers.*

18. A.H.Gleibman. Synthesis of Text Processing Programs by Example: The Sample Language. *Preprint of the Institute of Theoretical Astronomy of the Russian Academy of Sci. No.15, 27 pp.,1991. In Russian.*

19. A.H.Gleibman. Text Example Generalization: Inductive Synthesis of Sample Language Programs. *Manuscript, 30 pp. Submitted for publication.*

20. A.H.Gleibman. Reasoning About Equations: Towards Physical Discovery. *Preprint of the Institute of Theoretical Astronomy of the Russian Academy of Sciences No.18, 1992, 37 pp. In Russian.*

21. A.H.Gleibman. Automatic construction of equations for celestial mechanics on the ground of observation data. *Thes. of papers of the Soviet conference "Methods for Computer Modeling of Classic and Celestial Mechanics", Leningrad, 1989, p.30*.

22. A.H.Gleibman. Computer Graphics System Design on the ground of samples. *In book: Applied Informatics, No.2 (Moscow, 1986, pp. 5-18). In Russian.*

23. Haraguchi, M and Arikawa, S. A Foundation of Reasoning by Analogy: Analogical Union of Logic Programs. Lecture Notes in Computer Science, V.264, pp.58-69, 1986.

24. Hewitt, C. Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot. TR-258, - AI Lab.,MIT, Cambridge, Mass., 1983.

25. Ho, T.B.; Diday, E.; Getler-Summa, M. Generating rules for expert systems from observations. Pattern Recognition Letters No.7, 1988, pp. 265-271.

26. Jansen-Winkeln, R.M. LEGAS -- Inductive Learning of Grammatical Structures. Artificial Intelligence and Expert Systems. S.Savory, ed. Ellis Horwood Ltd, John Wiley & Sones, 1988, pp. 206-217.

27. Jones, R. Generating predictions to aid the scientific process. Proc. of AAAI-86: Fifth Nat. Conf. on AI, Philadelphia, PA, USA, 11-15 Aug. 1986, pp. 513-517.

28. Kinber, E.B. Syntactical inductive synthesis of iterative programs. Computers and Art. Int., V.8,1989, No.6, pp.565-580.

29. Kinber, E.B. Syntactical Inductive Synthesis According to Examples of Programs Containing Conjunctive Conditions in Loops. Kibernetika, 1988, No. 6, pp.23-28. In Russian.

30. Langley, P.; Bradshaw, G.; Simon, H.A. Rediscovering Chemistry with the Bacon System. Machine Learning: An Artificial Intelligence Approach, R. Michalski, J.G. Carbonell, and T.M. Mitchell, eds., Tioda Press, Palo Alto, Calif., 1983.

31. Lenat, D.B. The Nature of Heuristics. Artificial Intelligence, V.19, No.2, pp.189-249, 1982.

32. Macdonald, B.A., Witten, I.H. A Framework for Knowledge Acquisition Through Techniques of Concept Learning. IEEE Transactions on Systems, Man, and Cybernetics. V.19, No.3, 1989, pp. 499-512.

33. Michalski, R.S. A Theory and Methodology of Inductive Learning. Artificial Intelligence, V.20, No.2, pp.111-161, 1983.

34. Mitchell, T.B.; Keller, R.M.; Kedar-Cabelli, S.T. Explanation-Based Generalization: A Unifying View. Machine Learning, V.1, No.1, pp. 47-80, 1986.

35. Rose, D; Langley, P. Chemical Discovery as Belief Revision. Machine Learning, V.1, No.4, 1986, pp. 423-451.

36. Shapiro, E. Algorithmic program debugging. Cambridge (Mass.): MIT Press, 1983 (122 p).

37. Silver, B. Meta-level inference. North-Holland-Amsterdam-New York-Oxford, 1986.

38. Turchin, V.F. Algorithmic language of recursive functions (REFAL). Institute of Applied Mathematics of USSR Ac. of Sci. Moscow, 1968 (in Russian).

39. Vanlehn, K. and Ball, W. A version Space Approach to Learning Context-free Grammars. Machine Learning, V.2, 1987, pp. 39-74.

40. Vityaev, E.E. Inductive synthesis of Prolog programs by discovery of regularities. Problems of Program Synthesis, Verification and Debugging. Riga, LGU, 1986, pp. 65-66. In Russian.

41 Vigant I. Natural Language Processing by Examples. Master Thesis. Comp. Sci. Dept., Technion, Haifa 1997.

42. Winston, P.H. Learning new principles from precedents and exercises. Artificial Intelligence, V.19, pp.321-350, 1983.

43. Zavalishin, N.B. and Muchnik, I.B. Visual perception models and image analysis algorithms. Moscow, "Nauka", 1974, p. 205. In Russian.

44. Zloof, M. Query by Example. AFIPS Conf. Proc. V. 44, 1975, pp. 431-438.

45. Zytkow, J.M.; Simon, H.A. A theory of historical discovery: The construction of componential models. Machine Learning, V.1, No. 1, pp. 107-136.