Build a program immediately from data processing examples

Sampletalk Language & Programming Technology:

Data processing example abstraction immediately becomes a useful program if text matching is in focus

A. Gleibman

sampletalk (at) gmail . com

See also:

The philosophy of alternative programming

Introduction

This is a description of the AI language Sampletalk and of the corresponding programming technology.

Using Sampletalk, one can transform a set of sample texts into a working program, which employs *regularities*, contained in the texts, and behaves with other texts according to thus developed *understanding* of the texts. The sample texts are unrestricted. They may contain natural language descriptions of some data, linguistic rules, biological code sequences, rules for combining the data, symbolic math expressions, complex data examples in any text form, etc.

The running of a Sampletalk program resembles a matching of objects in a human brain. Program elements (we call them *samples*), and the data being processed, should have some

similarities, which are employed for driving the process.

In [1] and [2] we prove that any algorithm can be created in this way. Rather than *encoding* the algorithm one can *compose* it from some sample data without programming in a common sense, where data structures and the order of operations are to be specified.

Description of the language and technology

Sampletalk language is characterized by a trivial syntax and by an extreme similarity of the program code elements to demo samples of the data being processed.

The following Sampletalk program example demonstrates a small knowledge base. All its facts, rules and queries are extracted from English phrases. Note that only uncapitalized English words are used here; capital letters and the symbol "_" stand for *variables.*

```
where is new york?..                    % Goal (query)
where is X? in Y :- X is situated in Y..  % Inference rules
who is X? Y :- X is Y..
new york is situated in america..       % Facts in a natural language form
st. petersburg is situated in russia..
book is situated on table..
joe is son of maria and peter..
nick is son of maria and peter..
julia is daughter of maria and peter..
jack is son of julia and nick..
```

Note that words **where**, **is**, **situated**, **who** and the question signs are *not* key constructions of the language. Their meaning is defined just here, *via their usage*. You can consider the symbol **:-** as a separator between a procedure head and a procedure body, or as an ***if*** condition: the head holds if the body holds. Here you can find 2 heads with bodies (inference rules) and 7 heads without bodies (they are called facts; they always hold). Although this program consists only of 9 clauses, you can ask a number of questions (queries), e.g.:
where is new york? (the result will be **where is new york? in america**)
who is julia? (the result will be **who is julia? daughter of maria and peter**)
X is son of maria and Y (**X** will match **joe**, then **nick**)
Jack is Y of Z (**Y** will match **son**, **Z** - **julia and nick**)
X is son of _ (**X** will match **joe**, then **nick**, then **jack**.

So, the goal is *expanded* according to the inference rules and facts. You can easily enrich this knowledge for processing other questions like **what is on table?**, for introducing desired syntax or semantic tags, for generating explanations etc.

Symbol **:-** between a clause head and a clause body and the double-dot **..** is almost all the language syntax. ***Matching*** between the program elements controls the algorithm. These elements may contain *variables*, like **X** and **Y** in the example above. You can imagine programming in Sampletalk as extracting the reasoning rules from natural sources. Consider the rule

were is X? in Y :- X is situated in Y.

This is a rephrasing of the following reasoning:

I know that **X is situated in Y.** Then, if asked **"where is X?",** I should answer **"in Y"**.

This reasoning, in its turn, can be considered as a generalization of the following, more specific reasoning:

I know that **Paris is situated in France.** Then, if asked **"where is Paris?"** I should answer **"in France"**.

Sampletalk language is related to AI, Linguistics, Mathematics, and more specifically, to Logic Programming, Program Synthesis, Corpus Linguistics, Machine Translation. It raises some interesting technological and philosophical questions and provides a natural way of programming in many areas, beginning from text processing. It appeals to human intuition and demonstrates an unexpected power for the creation of algorithms.

Sampletalk language simplifies the concepts from Pattern Matching, Databases, Knowledge Representation and Natural Language Processing in that you can apply many of related algorithms in the simplest possible way. The data are hierarchical text structures with the possibilities of text matching, text generalization and text composition. The algorithmic knowledge is taken *immediately* from expert-domain expressions: natural language descriptions, formulas, code lines etc., so the programming is surprisingly straightforward: ***build a program immediately from data processing examples*.** I believe that this is an ideal instrument for creation of interpreters, compilers, and natural language processing software; see numerous working examples below and in our experimental version of Sampletlak interpreter.

Below Sampletalk program examples are given. Some theoretic analysis of the language and the technology is provided in papers [1], [2], [4], [8]. You can skip the next 3 sections and proceed to examples.

Syntax and semantics of Sampletalk language are intentionally made to be so simple that their description fits into the next half-page.

The main language construction (named *clause*) is written as follows:

head :-- sub-goal,, sub-goal,, . . . ,, sub-goal..

(note the double-commas and the double-dot, used as separators), or simple as

head..

Here ***head*** and ***sub-goal*** (which are called *samples*) can be any words (lists of symbols) in some

alphabet. Sub-words in the samples can be *underlined*; we denote this by square brackets **[** and **]**. The underlines define a nesting structure on the samples. The first program clause is called *program goal* and must have the form ***head..*** (we can use other sources for program goals: a keyboard input, a file of goals etc).

The *performance* of a Sampletalk program consists in an attempt to match the program goal with the first clause head for which this matching is possible (starting from the top of the program excluding the goal itself). Then the interpreter attempts to do the same with the sequence of sub-goals of the clause being found and so on, until matching all of them, if this is possible. Note that this is a recursive definition.

The process of *matching* of the samples consists in an attempt to make them equal by a global replacing of *variable names* (the words starting from upper-case alphabetic letters and separated from other words by spaces or the underlines) with suitable constructions. If this is impossible for a current clause, the performance of this clause fails and the interpreter looks for other possibilities to resolve the goal.

The *result* of the program performance is the program goal. However, the variables in the program goal will have values, which are obtained during the performance. We can get all possible results by choosing a special option for the interpreter. Like in Prolog language, if some matching is impossible, the interpreter tries to find another clause, whose head matches the current sub-goal, then to match all its sub-goals, and so on. If a performance of some sub-goal is impossible, the interpreter backtracks and considers other possibilities for matching previous sub-goals. Unlike programming in Prolog language, we are not interested in the *side effects* of the program performance. The instantiated goal itself is the main result of the program performance.

Square brackets **[ ]** in the samples are used in order to define the nested substrings. They constrain the matching. Nesting structures of matching samples must match. Comments (line fragments, which do not affect the program performance) are designated by the % sign at the beginning. In this version of the language and compiler we cannot directly use symbols **:- „ ..** , brackets and braces in the text samples as ordinary symbol constants, since they are scanned as the key constructions of the language.

Sampletalk as modification of Prolog language: The main feature of Sampletalk is a *strong similarity of the constructions for writing algorithms to the data being processed*. For this purpose we sometimes use our special notation with underlines rather than the brackets for designation of the nested words.

As the modification of Prolog, Sampletalk can be defined as follows. The only basic data type in Sampletalk is *string*. Prolog terms such as [A1, A2, …, Ak], where each Ai is either a basic type or a variable, are designated as A1 A2 .... Ak. (without the quotation marks defining strings in Prolog). These terms are called *samples*; their sub-lists (list segments in any nesting level) are called sub-samples. The uppercase single-character constants and several constants, used also for the language syntax ([,], {, }, double dot, double comma, neck-symbol :-) are designated in a special way. The nested samples are marked either by brackets or by underlining.

EXAMPLES. Prolog lists **['a', '+', 'b', '=', C], [A, '+', B, '=', C]** and **[['(','a', '+', 'b',')'], '*', ['(',M, '+', N,')']]** are written as samples **a + b = C, A + B = C** and **[(a + b)] * [(M +**

N)] respectively. Here **C, A, B, M, N** are variable names. The samples in square 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 (at different levels) of the last sample.

Any Prolog term can be incorporated into a text sample simply by surrounding this term with braces. (This is used for providing Sampletalk with numbers and other machine-oriented terms). So, any built-in Prolog predicate can be used as a sub-goal. See examples of the incorporated Prolog terms and predicates in Sampletalk clauses below (examples for number list permutations and sorting).

The only predicate in Sampletalk is a 1-place predicate with an empty name. The predicates are written in clauses without parentheses, simply by writing their arguments, separated by the neck-symbol **:-** and the double-commas, and terminated by the double-dot.

While matching two samples, a variable can match not only an element of the opposite sample (like in Prolog). It can match a sub-sample (list segment) formed by a sequence of such elements. Since there can be many possibilities of matching two samples, we introduce the following rule, which affects the backtracking: *the more left an occurrence of a variable in a sample, the less is the size of a sub-sample that is considered for instantiation of this variable, starting from 1-element segment.*

There are various possibilities for matching text samples, and sometimes we meet unexpected and even unwanted results. For example, a sample **a / b / c** can match a sample **A / B** in two ways: 1) **A = a, B = b / c**; 2) **A = a / b, B = c**. However, you may find useful the exhaustion of all such possibilities for matching. For instance, matching samples like **begin [spoke] [rim] [hub] [wheel] [frame] end** and **C [A] B** enables us to process the fragments of samples as lists of words (see demo program PARTSIN2.SAM):

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 -**	

Matching of samples **C A B** and **begin spoke rim hub wheel frame end**

Variable **A** can not bind a sub-sample containing 2 or more underlined words, since it is

underlined. Underlying structures of the matching samples must match. Variables **C** and **B**, which are not underlined, can match any sub-samples.

Only the user's wishes and imagination limit the expressing power of a language with such trivial syntax and so rich possibilities for the interaction of the language constructions. For example, a *number sort* program may contain a fragment **A{N}{M}B ==> A{M}{N}B :- {M<N}** (see sort examples below); a program for reducing homogeneous items in an analytic expression may contain a fragment **A+M*X+N*X+B ---> A+L*X+B :- L=M+N** with respective clauses involving the constants **<, +, =** and the concept of number.

Compiler Options: Current version of Sampletalk compiler considers the first clause of any program as the goal, and all the following clauses as the clauses for resolving this goal.

By default, the compiler writes its output on the screen.

DIRECT OUTPUT INTO OUTPUT.SAM option: Using this option, you can redirect the output into a file with the standard name OUTPUT.SAM.

DIRECT OUTPUT ON THE SCREEN option: the compiler output is returned into the default status.

STOP BETWEEN LINES option: the current line in Sampletalk windows will be printed only after pressing any key. This option is used to slow down the process for the analysis and for debugging. Otherwise, the lines will be printed without stopping.

SENSITIVE MATCHING BACKTRACKING: during backtracking, the compiler will analyze all the possibilities for matching any current pair of samples. *This option does not have an analogy in Prolog language*. If the option is unchecked, then only one possibility for matching the current pair of samples is searched for. In case of backtracking, the current sub-goal fails without considering other possible matching of the current samples.

FIND ALL (ONE) DECISIONS (toggle): the compiler considers, through the backtracking, all possible results of a program performance (i.e. results of replacing bound variables in the program goal by their values). If the toggle is off, only one result will be searched for.

WRITE INTERMEDIATE RESULTS: if this option is unchecked, then the compiler avoids writing the information about matching of the intermediate samples. If the program performance is successful, then only the resolved goal is printed. Otherwise, the information about every Sampletalk clause-call is printed. This option is used for testing and debugging of Sampletalk programs. Another usage of this option is getting an explanation and a history of the program performance.

Sampletalk Program Examples:

Natural language question understanding

Imagine that you have plenty of facts like **jack eats apples in the corridor**, represented in a natural language form, and would like to develop a question-answering system that allows a user to derive consequences of such facts and to initiate other actions, related to such reasoning, e.g., to qualify eating as a consumption. Sampletalk provides the simplest possible way for doing this. You will never need to make a separate parser, grammar, semantic models, and reasoning modules. Everything is obtained via the *universal* concept of *matching text samples*, where the samples may contain variables, e.g. **X** (standing for **Jack** or somebody else, as in our example), **A** (standing for verb **eat** or for some other description of action), **F** (standing for **food** or for some other object) etc. In order to implement such reasoning, all other knowledge-processing systems require much more complex programming.

```
% Goal:
what does jack eat?..
% Knowledge base (inference rule with semantic patterns):
what does X A? F :-
        X As F in P,,
        consumption: Aing,,
        food: F,,
        place: P..
% Semantic knowledge:
consumption: drinking..
consumption: eating..
food: apples..
food: bananas..
beverage: coca cola..
place: corridor..
place: room..
% Known facts:
jack eats apples in the corridor..
mary drinks coca cola in the room..
```

The output (or, better, the answer) of this program will be **what does jack eat? apples.** On the way to derive this answer the program will print intermediate results: **consumption: eating, food: apples, place: the corridor.**

A more precise representation of English syntax, semantics and morphology can be achieved in the same way: see the following examples.

Abstraction of natural language reasoning

This program example demonstrates how we can obtain inference rules immediately from NL reasoning examples. Consider the following 3 obvious reasoning examples:

What is on roof? Bird (if object bird is situated on roof);
Who is Socrates? Human (since person Socrates is Human);
Where is Paris? In France (since object Paris is situated in France).

In the program below we simply replace specific object names with unique variable names in the form {Something} (variable names in Sampletalk start from the upper-case letters) and obtain the rules, which are much more general. They provide additional reasoning like

What is on a table? Book (since object book is situated on a table);
Who is Julia? Daughter of Maria and Peter (since person Julia is a daughter of Maria and Peter);
Where is St. Petersburg? In Russia (since object St. Petersburg is situated in Russia)

and more, given program goals like the goals given below. The program is as simple as this:

```
% Possible program goals
who is julia? Function..
where is new york? in Place..
person Somebody is son of maria and His_father..
person jack N is Relative of Somebody..
person Son 2 is son of Y and X 1..
what is on table? X..
% Reasoning rules
what is on Roof? Bird :- object Bird is situated on Roof..
who is Socrates? Human :- person Socrates is Human..
where is Paris? in France :- object Paris is situated in France..
% Factual knowledge
person joe is son of maria and peter..
person julia is daughter of maria and peter..
person peter 2 is son of maria and peter 1..
person jack 2 is son of julia and jack 1..
object new york is situated in america..
object st.petersburg is situated in russia..
```

```
object book is situated on table..
```

The outputs for the corresponding goals will be:
who is julia? daughter of maria and peter..
where is new york? in america..
person joe is son of maria and peter..
person peter 2 is a son of maria and peter 1..
person jack 2 is son of julia and jack 1..
person peter 2 is son of maria and peter 1..
person jack 2 is son of julia and jack 1..
what is on table? book..

The inference **where is Paris? in France :- object Paris is situated in France**, where the capitalized words stand for *variables*, is used for the inference **where is new york? in america :- object new york is situated in america**.

Morphological and syntactical analysis of a sentence

This program takes an English sentence and produces a tree, which represents a grammar analysis of this sentence. The output represents the parsing tree structure along with morphological tags. (Here we don't consider the problems of ambiguity in NL morphology and parsing).

Given the sentence **this large apple is tasty since it has ripened** as a program goal (see below), the program produces the output **tree for sentence [this large apple is tasty since it has ripened] is sent[sent[noun[determiner[this] aj[large] noun[apple]] be[is] aj[tasty]] logic[since] sent[pronoun[it] have_sg[has] past_verb[ripened]]]**. This output represents the parsing tree structure along with the morphological tags.

```
%Goal (variable T is to be filled by the parsing tree; brackets "[" and "]" restrict possible matching):
tree for sentence [this large apple is tasty since it has ripened] is T..

% Main rule:
tree for sentence [X] is T :-
        morphological tagging for sentence [X] is Y,,
        tree for tagged sentence [# Y #] is T..

% Morphologic tagging: Recursive rule
morphological tagging for sentence [A X] is T[A] Y :-
```

```
        word A has morphological tag T,,
        morphological tagging for sentence [X] is Y..

% Boundary condition for the last 2 words:
morphological tagging for sentence [A B] is T1[A] T2[B]:-
        word A has morphological tag T1 ,,
        word B has morphological tag  T2..

% Building the tree: recursive rule
tree for tagged sentence [A X B] is W :-
        T[X] is a grammar pattern ,,
        tree for tagged sentence [A T[X] B] is W..
% Boundary condition:
tree for tagged sentence [# X #] is X ..

% Gramamar patterns:
noun[determiner[_] aj[_] noun[_]] is a grammar pattern..
noun[determiner[_] noun[_]] is a grammar pattern..
noun[aj[_] noun[_]] is a grammar pattern..
sent[noun[_] past_verb[_]] is a grammar pattern..
sent[noun[_] be[_] aj[_]] is a grammar pattern..
sent[pronoun[_] have_sg[_] past_verb[_]] is a grammar pattern..
sent[sent[_] logic[_] sent[_]] is a grammar pattern..
% Morphologic dictionary:
word   this        has morphological tag determiner..
word   large       has morphological tag aj..
word   apple       has morphological tag noun..
word   is          has morphological tag be..
word   tasty       has morphological tag aj..
word   since       has morphological tag logic..
word   it          has morphological tag pronoun..
word   has         has morphological tag have_sg..
word   ripened     has morphological tag past_verb..
```

Have you noticed that we are *programming* in English, not in C++ or even in Prolog?! Can you imagine more simple and expressive implementation of the parsing algorithm than what is immediately composed from linguistic terms like these?

Parts Inventory

This is a Sampletalk version of a classic Prolog program by Clocksin & Mellish. Using a description of a bike, this program produces the list of all its basic parts. The main difference from the Prolog version is the natural language form: we don't have to use a specific Prolog

syntax, and we can be as expressive in our NL expressions as we want. I even used an English check speller for debugging this program.

```
% Goal (L stands for the list of basic parts of a bike, which is being generated here)
parts of [bike] are L..
% Knowledge base
bike contains [wheel] [wheel] [frame]..
wheel contains [spoke] [rim] [hub]..
frame contains [rear frame] [front frame]..
front frame contains [fork] [handles]..
hub contains [gears] [axle]..
axle contains [bolt] [nut]..
basic part (rim)..
basic part (rear frame)..
basic part (gears)..
basic part (nut)..
basic part (spoke)..
basic part (handles)..
basic part (bolt)..
basic part (fork)..
 % Inference rules
parts of [X] are X :- basic part (X)..
parts of [X] are P :- X contains S „ parts of list [S] are P..
parts of list [P T] are H G :- parts of P are H „ parts of list [T] are G..
parts of list [[T]] are G :- parts of [T] are G..
```

As a result, this program will print

parts of [bike] are [spoke] [rim] [gears] [bolt] [nut] [spoke] [rim] [gears] [bolt] [nut] [rear frame] [fork] [handles].

Such programs will never need detailed documentation. Since ***text matching*** is the main programming concept, the reader of the program will always *see* the explanation in the program itself. As important, the universal concepts like **contains**, **is**, **parts of**, as well as any kind of jargon language designations, can be easily included into the program, along with the constraints and explanations on how the corresponding objects should behave.

Logic Formula Transformation (shifting quantifiers)

The main clause of this program almost literally quotes a well-known formal rule from a student textbook for Mathematical Logic, along with the informal constraints on how to apply this rule. See also the next 2 programs (modifications of this one) and the miscellaneous Sampletalk program examples below -- for the explanation of the last clauses of these programs.

```
% Goal:
(/\ x0)[a(x0,y)] \/ (/\ x0)[b(x0,t)] --> W..
% (here /\ denotes universal quantifier; W stands for the resulting formula).
% Program (knowledge base): transformation rule
(Q X)[F] \/ (Q X)[H] --> (Q X)(Q Z)([F] \/ [G]) :-
        X is variable,,
        Z is variable,,
        not(F contains Z),,
        (Z/X)[H]=[G]..
% rules for definition of logic variables:
x0 is variable..
X10 is variable :- X0 is variable..
% explanation what is "contains" by example:
AXB contains X..
% rules for change of variables:
(Y/X)[AXM]=[AYN] :- (Y/X)[M]=[N]..
(Y/X)[A]=[A]..
```

Output: **(/\ x0)[a(x0,y)] \/ (/\ x0)[b(x0,t)] --> (/\ x0)(/\ x10)([a(x0,y)] \/ [b(x10,t)]).**

This and the next two programs demonstrate that such complex algorithms as the algorithms for logic formula transformations can be derived just from their ***semi-formal*** descriptions!

Inverse Logic Formula Transformation (elimination of parentheses and distribution of quantifiers).

The program is the same as the previous one, but its goal is "opposite" (W at the beginning is to be filled by the resulting formula):

```
% Goal
W --> (/\ x0)(/\ x10)([a(x0,y)] \/ [b(x10,t)]).
% Knowledge base:
(Q X)[F] \/ (Q X)[H] --> (Q X)(Q Z)([F] \/ [G]) :-
        X is variable,,
        Z is variable,,
        not(F contains Z),,
        (Z/X)[H]=[G]..
x0 is variable..
X10 is variable :- X0 is variable..
AXB contains X..
(Y/X)[AXM]=[AYN] :- (Y/X)[M]=[N]..
(Y/X)[A]=[A]..
```

Output: the same as in the previous example.

Logic Formula Transformation Revisited: Description in Natural Language Form

In this version of the same program we use more natural description language. A rule identification (see variable R in the goal) is added. You can run these programs using the special FIND ALL DECISIONS option. The programs will form words x10, x110, x1110 etc. for the representation of the logic variable Z under the second quantifier /\. Note that no previous parsing of the goal is required!

```
%Goal:
according to rule R, the result of shifting quantifiers in the formula (/\ x0)[a(x0,y)] \/ (/\ x0)
[b(x0,t)] is formula W..
```

```
% Knowledge base:
according to 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..
```

Output:

according to the rule 2a from chapter 5, the result of shifting quantifiers in the formula (/\ x0)[a(x0,y)] \/ (/\ x0)[b(x0,t)] is formula (/\ x0)(/\ x10)([a(x0,y)] \/ [b(x10,t)]).

Have you noticed already that we are *programming* in a natural language?!

Logic reasoning about polygons, expressed in English

This example was immediately extracted from a polygon axiomatization, kindly provided by S.S.Lavrov just in English. Possible goals:

[is the number of diagonals of a square less than five ?] ..
[is a square a quadrangle ?]..
[is a rhombus a quadrangle ?]..
[is a square a rhombus ?]..
[is a rectangle a parallelogram ?]..

Given a goal of this kind, the program generates the output, related to the intuitive understanding of the polygons expressed below directly in English.

```
% Logic reasoning expressed in a natural language (knowledgebase):
definition: [a rectangle is a parallelogram with equal diagonals]..
definition: [a rhombus is a parallelogram with equal adjacent sides]..
```

```
definition: [a square is a rectangle with equal adjacent sides]..
definition: [a parallelogram is a quadrangle with two pairs of parallel sides]..

 [a X is a Y] :- definition: [a X is a Y with P]..
[a X is a Y] :- statement [a X is a Y] can be derived from definition ..
[is a X a Y ?] :- statement [a X is a Y] can be derived from definition ..

statement [a A is a C] can be derived from definition :-
        definition: [a A is a B with X] „
        definition: [a B is a C with Y]..
statement [a A is a D] can be derived from definition :-
        definition: [a A is a B with X] „
        definition: [a B is a C with Y] „
        definition: [a C is a D with Z]..
statement [a A is a D] can be derived from definition :-
        definition: [a A is a B with X] „
        definition: [a B is a C with Y] „
        definition: [a D is a C with X]..

[is the number of D of a S less than N ?] :-
        statement [a S is a Q] can be derived from definition „
        [the number of D of a Q is N1] „
        fact about numbers: [N1 is less than N] ..

 [the number of A of a B is N] :-
        numerical property of B : [a B has just N A] ..
numerical property of quadrangle : [a quadrangle has just two diagonals] ..
fact about numbers: [one is less than two]..
fact about numbers: [two is less than three]..
fact about numbers: [three is less than four]..
fact about numbers: [four is less than five]..
fact about numbers: [five is less than many]..
fact about numbers: [N1 is less than N2] :-
        fact about numbers: [N1 is less than X] „
        fact about numbers: [X is less than N2]..
```

This program depends on the clause order. Its development for a wider axiom set is easier using the Sampletalk option for breadth-first application of clauses, briefly described below (see the section about sequential and parallel processing in Sampletalk).

Recognition of regularities in word codes

```
%Possible goals:
a12zb and cz21de have inversion fragments X and Y..
axyzxyzxyzxyzb contains repetition of Z..
[##1#3#2##4#9# and |0|1|2||3|4|5||6|7|||8|9|] have common subsequence W..
% Knowledge base:
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 AB is BA..
AXB contains repetition of Z :- X is repetition of Z..
XX is repetition of X..
ZX is repetition of X :- Z is repetition of X..
[AXV and BXW] have common subsequence X S :-
        [V and W] have common subsequence S..
[A and B] have common subsequence ..
```

After running this program, **X** will match **12z**, **Y** - **z21**, **Z** - **xyzxyz**, then **xyz**, then **yzx** etc., **W** will match the sequence **1 3 4 9** and then the sequence **1 2 4 9**. This program can be used for detecting sub-word repetitions, inversions, and other special features of symbol lines, which represent a genetic structure. Note the natural language form of this program: it is constructed from expressions occurring in literature about genetics, without involving any special knowledge about computer programming!

Transformation of infix notation of arithmetic expressions into (Polish) prefix notation

```
% Possible goals:
[[[a + b] * c] * [u2 + u3]] => _..
[[a + b] * c] => ..
[[a + b] * [c - d]] => ..
[[a + b] * [[a - b] / c]] => ..
```

% Symbols "_" are to be replaced with the prefix notations. Outputs of the program with these goals will be as follows:

[[[a + b] * c] * [u2 + u3]] => * * + a b c + u2 u3..
[[a + b] * c] => * + a b c..
[[a + b] * [c - d]] => * + a b - c d..
[[a + b] * [[a - b] / c]] => * + a b / - a b c..

```
% The program is as simple as this:
[A Z B] => Z X Y :-
        A => X ,,
        B => Y..
A => A..
```

As a result, you will get the polish notations instead of the symbol "_" in the goals.
Note that the symbol "**=>**" is not a language construction. You can write "**-->**", or something more expressive like "**is translated to**", "**is**" - anything you want, to express the meaning of this transformation. For Sampletalk language, only matching of the symbol "**=>**" to other occurrences of this symbol is important. For example, the following version of this program might be more expressive:

```
% Transformation of infix notation of arithmetic expressions into prefix notation.
% Possible goals:
prefix form for expression [[[a + b] * c] * [u2 + u3]] is expression X..
prefix form for expression [[a + b] * c] is expression X..
prefix form for expression [[a + b] * [c - d]] is expression X..
prefix form for expression [[a + b] * [[a - b] / c]] is expression X..
% The program: Recursive rule:
prefix form for expression [A Z B] is expression Z X Y :-
        prefix form for expression A is expression X ,,
        prefix form for expression B is expression Y..
% Boundary condition:
prefix form for expression A is expression A..
```

Output for the given goals correspondingly:
prefix form for expression [[[a + b] * c] * [u2 + u3]] is expression * * + a b c + u2 u3
prefix form for expression [[a + b] * c] is expression * + a b c
prefix form for expression [[a + b] * [c - d]] is expression * + a b - c d
prefix form for expression [[a + b] * [[a - b] / c]] is expression * + a b / - a b c

Probably the latest version is preferable: it is well self-documented, and its clauses can be included into other programs without the loss of readability.

Word Inversion 1

Compare this program to the next two programs. The under-lines denote the nesting structure on the samples.

```
% Goal:
reverse of the word abcdef is X..
% Program (knowledge base):
reverse of the word AM is NA :- reverse of the word M is N..
reverse of the word A is A..
```

The output will be **reverse of the word abcdef is fedcba.**

This program demonstrates how a word processing algorithm can be constructed from the natural language phrases, describing this algorithm.

Word inversion 2

This program is similar to the previous one but has a more compact code.

```
[abcdef]□[X]..                    % Goal
[AM]-->[NA] :-                    % Recursive rule
        [M]-->[N]..   [A]-->[A].. % Boundary condition
```

The output will be **[abcdef]-->[fedcba].** These examples show that we can easily introduce your own syntax like **[]-->[]** for expressing our favorite designations. The output will be **[abcdef]-->[fedcba].**

Inversion of a list of words (separated by /)

Compare this program to the previous two programs for word inversion. The only essential difference consists in two symbols / in the recursive rule, but the performance is quite different.

```
reverse of the list abc/def/gh is X..                    %Goal
reverse of the list A/M is N/A :- reverse of the list M is N..   % Program
reverse of the list A is A..
```

Output: **reverse of the list abc/def/gh is gh/def/abc.** Note the readability of such programs: it is hard to imagine a program, which is more readable than what is composed from the natural language descriptions of the algorithm.

Replacing sub-words in a word

The goal for this program has the form

(Word2/Word1)SourceExpression=ResultingExpression

This program replaces all occurrences of **word1** in the source expression by **word2** (upper-case letters stand for text variables). See also the applications of this program in our program examples for logic formula transformation.

```
% Goal:
((y+2)/x7)[sin(2x7)+cos(4x7)]=_..
% Program:
(Y/X)[AXM]=[AYN] :-
        (Y/X)[M]=[N]..
(Y/X)[A]=[A]..
```

Output: **((y+2)/x7)[sin(2x7)+cos(4x7)]=[sin(2(y+2))+cos(4(y+2))].**

Note the similarity of the program constructions to the processed data: they are built just from examples of such data!

Palindrome test

This program tests a word whether it is a palindrome and finds its 1-st half (the goal is the 1-st line; the half will replace variable **W**).

```
% Goal
[abcddcba]->W..
% Program
[AXA]->AW :-
        [X]->W..
[AA]->A :-
        ~~[A]=[CD]..
[A]=[A]..
```

The output: **[abcddcba]->abcd..** Once more, notice that notations **[***something***]->***something* and **[***something***]=[***something***],** which express a transformation of entities and equality of entities correspondingly, are *defined* just here. You can use any other expressive (for you) notation, and the program will do the same, provided all your notations match each other in a similar way.

List permutations

This is (probably) the shortest representation of an algorithm for generating list permutations, possible for the universal programming languages. You can even exclude the words "permutation for", "is", and the program will do the same. Use the compiler option for finding all decisions while running this program.

```
% Goal
permutation for ({1}{2}{3}{4}) is [W]..
% Program
permutation for A{M}B is [{M}W] :- permutation for AB is [W]..
permutation for () is [ ]..
```

Output:

```
permutation for ({1}{2}{3}{4}) is [{1}{2}{3}{4}] .
permutation for ({1}{2}{3}{4}) is [{1}{2}{4}{3}] .
permutation for ({1}{2}{3}{4}) is [{1}{3}{2}{4}] .
permutation for ({1}{2}{3}{4}) is [{1}{3}{4}{2}] .
…
```

Generating words of form W [W-1] W:

Generation of words of the type W [W-1] W (see the comment below), which is impossible for CF grammars.

```
%Goal
[A][B][A]..
%Program
[X][X][X] :- letter [X]..
[AX][XB][AX] :-
        [A][B][A] „
        letter [X]..
letter [a]..
letter [b]..
```

Comment: The program generates words **[a][a][a]**, **[b][b][b]**, **[aa][aa][aa]**, **[ab][ba][ab]**, **[abab][baba][abab]** etc, where the 2-nd word is inverse of the 1-st and 3-rd words which are equal.

Bubble sort of a digit list

In this program, just finding the digits in the ascending digit list tests the order of digits.

```
sort of begin [2] [4] [5] [9] [3] [7] [4] [3] [5] end is [W]..      % Goal
sort of A [M] [N] B is [W] :-                                      % Program
        [M]<[N],,
        sort of A [N] [M] B is [W]..  sort of A is [A]..
[M]<[N] :-
        digits [_ M _ N _]..
digits [# 0 # 1 # 2 # 3 # 4 # 5 # 6 # 7 # 8 # 9 #]..
```

Output: **sort of begin [2] [4] [5] [9] [3] [7] [4] [3] [5] end is [begin [9] [7] [5] [5] [4] [4] [3] [3] [2] end]**

Such programs will never need detailed documentation. Since ***text matching*** is the main programming concept, the reader will always find the explanation of program elements in other similar elements.

Bubble sort of numbers

In this program, the order of numbers is tested by incorporated Prolog predicate <. Designation {...} stands for Prolog terms imbedded into Sampletalk programs (machine-oriented numbers in this case),

```
% Goal:
sort of [begin {2.7} {1024} {9.8} {3.14} end] is W..
% Program:
sort of [A {M} {N} B] is W :-
        {M<N},,
        sort of [A {N} {M} B] is W..
sort of [A] is A..
```

Output: **sort of [begin {2.7} {1024} {9.8} {3.14} end] is begin {1024} {9.8} {3.14} {2.7} end.**

Merge sort

Sorting a list of numbers by dividing the list into two parts, sorting these parts, and then merging the results.

```
% Goal (variable W stands for the sorted list):
sort of the list {7}{7}{7}{128}{2}{12}{10}{-130}{1993}{1492}{5743}* is list W..
% Program. Boundary conditions for two-element and one-element lists:
sort of the list {M}{N}* is list {M}{N}* :- {M<N} ..
sort of the list {M}{N}* is list {N}{M}* ..
sort of the list {M}* is list {M}* ..

% Recursive rule:
sort of the list A is list W :-
        list A consists of even elements [E] and odd elements [O] ,,
        sort of the list E is list X ,,
        sort of the list O is list Y ,,
        result of merging lists [X] and [Y] is list W ..

% Clauses for dividing a list into two parts - for even and odd elements - recursive rule:
list {X}{Y}A consists of even elements [{X}B] and odd elements [{Y}C] :-
        list A  consists of even elements [B] and odd elements [C] ..
```

```
% Boundary conditions for 1-element and 0-element lists:
list {X}* consists of even elements [{X}*] and odd elements [*] ..
list * consists of even elements [*] and odd elements [*] ..

% Clauses for merging sorted lists - recursive rules:
result of merging lists [{K}A] and [{L}B] is list {K}W :-
        {K<L} „
        result of merging lists [A] and [{L}B] is list W ..
result of merging lists [{K}A] and [{L}B] is list W :-
        {K = L} „
        result of merging lists [A] and [{L}B] is list W ..
result of merging lists [{K}A] and [{L}B] is list {L}W :-
        result of merging lists [{K}A] and [B] is list W ..
% Boundary conditions for 0-element and 1-element lists:
result of merging lists [*] and [A*] is list A* ..
result of merging lists [A*] and [*] is list A* ..
```

Output: **sort of the list {7}{7}{7}{128}{2}{12}{10}{-130}{1993}{1492}{5743}* is list {-130} {2} {7} {10} {12} {128} {1492} {1993} {5743}***

Once more, this is almost a pure natural language description of a merge sort algorithm. Due to *matching samples*, this is a working program. So, we are *programming* in a natural language!

About sequential and parallel processing in Prolog and Sampletalk: limitations of sequential object matching

Consider the example of logic reasoning about polygons (see above), expressed in English, once more. We can express its fragment, related to polygon types, in a simpler way:

```
% Knowledge base about polygons:
a rectangle is a parallelogram with equal diagonals..
a rhombus is a parallelogram with equal adjacent sides..
a square is a rectangle with equal adjacent sides..
a parallelogram is a quadrangle with two pairs of parallel sides..
```

```
a A is a B :-
        a A is a B with P..
a A is a C :-
        a A is a B „
        a B is a C..
a A is a D :-
        a A is a B with X „
        a B is a C „
        a D is a C with X..
```

Then, we would like this program to behave correspondingly: Given the following goals, the program should find the positive decisions:

a rhombus is a quadrangle.. **a square is a quadrangle..**
a square is a rhombus..
a rectangle is a parallelogram..
a rectangle is a quadrangle..

If a standard Prolog strategy for resolving goals is applied, then the last three program clauses will lead to an infinite recursion for some of the goals. We can rearrange these clauses. However, we will not find their order, in which all the above goals will be resolved.

Since we are ***programming*** in natural language, we would like to have the options for more natural application of the sequential object comparison. In Sampletalk, the additional interpreter options allow the breadth-first search for goal matching. This reminds us of our intuitive notion of the matching objects: Comparing objects in our brain, we avoid an infinite application of cyclic inferences. Instead, we try to consider the alternative models.

Metaphoric Conclusion about *Sampletalk* programming technology

CAR DESK METAPHOR

Let us try to compare Sampletalk programming technology to some other technologies metaphorically. Programming with complex syntax and semantics resembles driving a car without a control desk. You want to turn --- take a spanner and revolve a special bolt. Need to stop --- take another spanner and turn a gas tap (and what about a brake? find a vice, read its manual, and grip the axle). Such actions have almost nothing in common with the country map and the journey plan, although can help in a slow, may be dangerous as for possible collisions, but sometimes successful motion.

Situations with complex syntax and semantics are the same. Typically they have nothing in common with the problem in question, nor with the intuitive plan of the program being created, although can help in a slow, may be dangerous as for errors, but sometimes successful programming.

One needs a *steering wheel* and a *speed lever*. To some extent, Prolog is such a tool. Unification (read: the *steering wheel*) and performance strategy (read: the *speed lever*), its basis, have some similarity to such features of a human thinking as matching of intuitive images and exhaustion of variants for problem solving. For many applications this provides more effective programming than the conventional programming languages.

We go further: we create a country map (*samples*) from the country pictures (text examples), and allow the user to control the world automatically, simply by combining such objects. So, imagine you are in a car and can drive it just by matching the map to what you see in the window…

Perspectives for the Future

SAMPLETALK technology, described here, can provide an unexpected power for software creation and for incorporation of a scientific knowledge into the software projects.

The advantage of the technology is a reduction of the time and cost for software development, and, still more, for software maintenance. The technology is based on the methods of automatic identification and extraction of algorithms from text documents, containing data examples and their formal and semi-formal explanations. Even in its today form the technology can be applied for fast prototyping of the algorithms for logic inference, parsing, machine translation and knowledge representation (see the above examples).

The technology provides a way to extract algorithms automatically or semi-automatically, from document examples, without any efforts for *writing* the program code. This feature turns programming into a human activity, which if available for every intelligent individual ***independently on his computer language skills***.

Some forms of the technology can be related to patent-style formulations of the algorithms and to immediate usage of such formulations in programs. This means that the user can verbally describe the application, which he wants to build. After this, he can get a working prototype of the application automatically.

Connecting such a system to an existing database of patented algorithms and to a published literature, where algorithms are described in a semi-formal form has many perspectives. ***So, imagine a patent text, accompanied with an immediately and automatically constructed prototype, which is described in this patent.*** Such text can be immediately incorporated into new software, then modified or combined with other texts in order to construct new algorithms in a similar way.

The technology can reduce the linearity of programming, meaning that today we write and debug the algorithms linearly, statement after or before another statement, breakpoint after breakpoint, etc. This linearity is inconsistent with our intuitive images of complex data and

algorithms. We often need to recall such images, and we need to match numerous tiny program details to those mental images. This problem is not well understood in computer business. Emerging programming technologies almost do not address this. In my opinion, this is an important reason for failures of some computer projects.

Related publications:

1. A. Gleibman. Intelligent Processing of an Unrestricted Text in First Order String Calculus. In: M.L. Gavrilova et al. (Eds.): Trans. on Comput. Sci. V, LNCS 5540, pp. 99–127, 2009. © Springer-Verlag Berlin Heidelberg 2009. https://link.springer.com/chapter/10.1007/978-3-642-02097-1_6

2. A. Gleibman. Knowledge Representation via Verbal Description Generalization: Alternative Programming in Sampletalk Language. In: Workshop on Inference for Textual Question Answering. July 09, 2005 – Pittsburgh, Pennsylvania, pp. 59-68. AAAI-05 -- the Twentieth National Conference on Artificial Intelligence. http://www.aaai.org/Papers/Workshops/2005/WS-05-05/WS05-05-010.pdf

3. A. Gleibman. Reasoning About Equations: Towards Physical Discovery. In: The Issue of the Institute of Theoretical Astronomy of the Russian Academy of Sciences No.18, 1992, 37 pp. In Russian.

4. A. Gleibman. Synthesis of Text Processing Programs by Example: The SAMPLE Language. In: The issue of the Institute of Theoretical Astronomy of the Russian Academy of Sciences. No.15, 27 pp., 1991. In Russian.

5. A. Gleibman. Automatic construction of equations for celestial mechanics on the ground of observation data. Theses. of papers of the Soviet conference "Methods for Computer Modeling of a Classic and Celestial Mechanics", Leningrad, 1989, p.30. In Russian.

6. A. Gleibman. Object Recognition System Design in Computer Vision: a Universal Approach. https://arxiv.org/abs/1310.7170

7. A. Gleibman. Delegating Custom Object Detection Tasks to a Universal Classification System. https://arxiv.org/abs/1401.6126

8. 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.* English translation