Gregers Koch
Datalogisk Institut Københavns Universitet
Sigurdsgade 41
DK-2000 København N

NATURAL LANGUAGE PROGRAMMING

1. Aims

This is a study of automated programming. We are aiming at the stepwise development of a programming environmemt consisting of an automatic translation system to translate texts in natural language (e.g. software requirement specifications) into certain logical formulae according to some semantic theory, as well as into executable programs. The only semantic theories considered here are logical theories, whether they be related to the lambda calculus or to the predicate calculus, and we shall henceforth talk about the level of logical representation (rather than semantic representation).

The paradigm of the method may be guessed from the figure 1. The horisontal axis displays the gap between the human user and the computer (that is the so-called man-machine communication problem). The vertical axis indicates the level of abstraction, from low levels of abstraction up to higher ones. The areas of natural language, programming language, and the predicate calculus are indicated with an overlap between the latter two, as some predicate calculus expressions are executable and other are not.

Natural Language Programming
Gregers Koch
Proceedings of NODALIDA 1983, pages 132-144

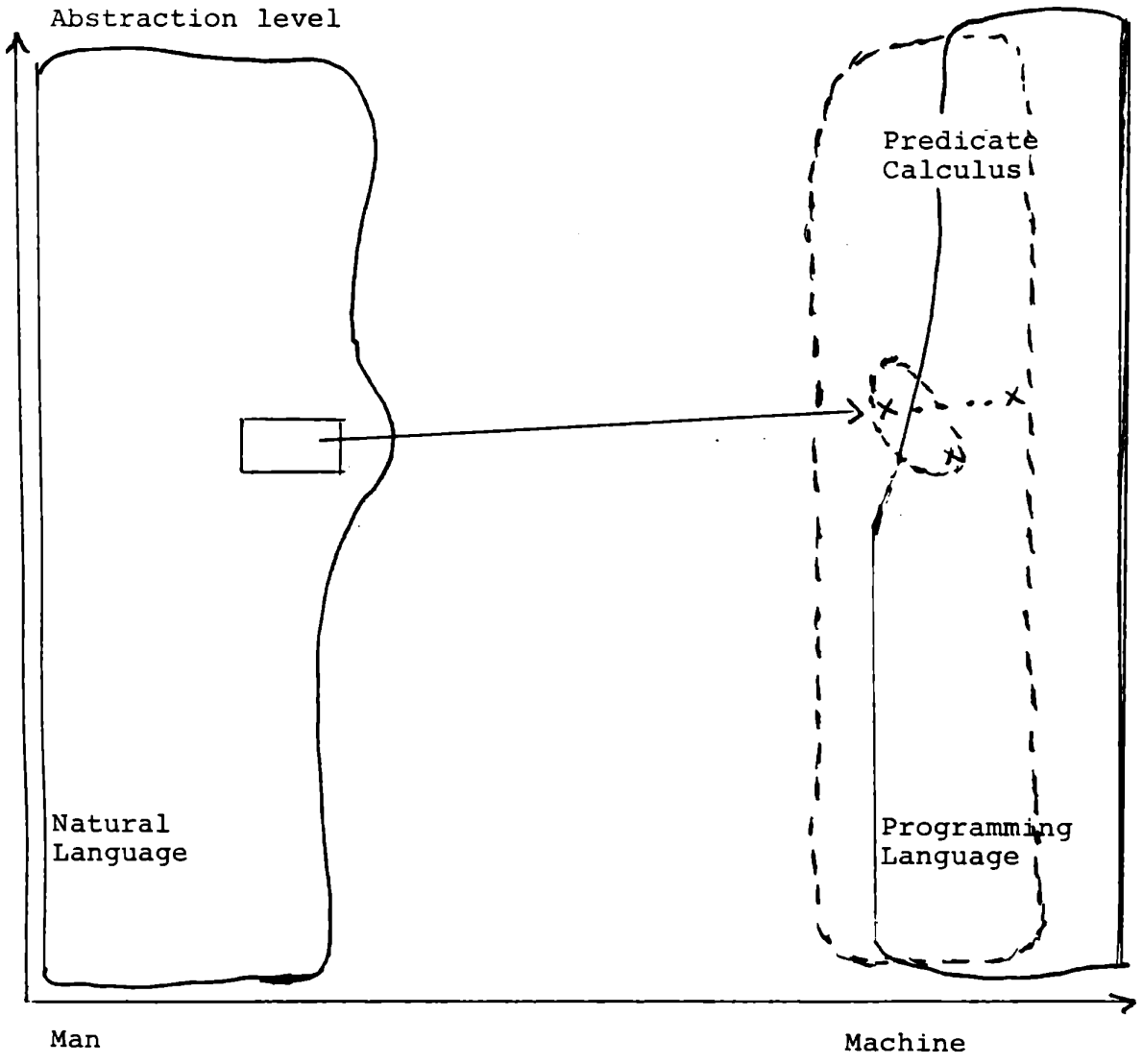


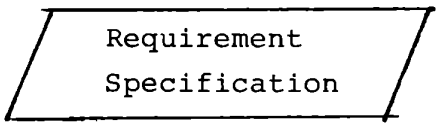
Figure 1.

The aim of this project is to develop a rigorously defined "square" sublanguage of natural language with a corresponding system performing automated translation into the predicate calculus.

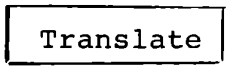
2. Developing natural sublanguages

Starting with whatever semantic theories are available in the literature the aim here is to develop step by step a computational sublanguage of natural language. This development is performed on the basis of carefully selected examples (and it will be illustrated by examples). The textual formulation of the requirement specification will thus be translated automatically into a logical representation.

The figure 2 shows the same thing in a diagrammatic form. Here are indicated the documents occuring during the process, for instance



and a few processes to be discussed, for instance



Several aspects of the figure 2 will be commented upon in the rest of the paper.

3. Reverse translation

Generating i.e. reverse translation back into natural language, makes it possible to check the quality of the textual translation process.
It is a characteristic property of logic programming languages that (at least in principle) the user may apply the same program for translation and for generation.

4. Logical alternatives

The particular choice of logical representation is essentially arbitrary, but the scientific literature heavily

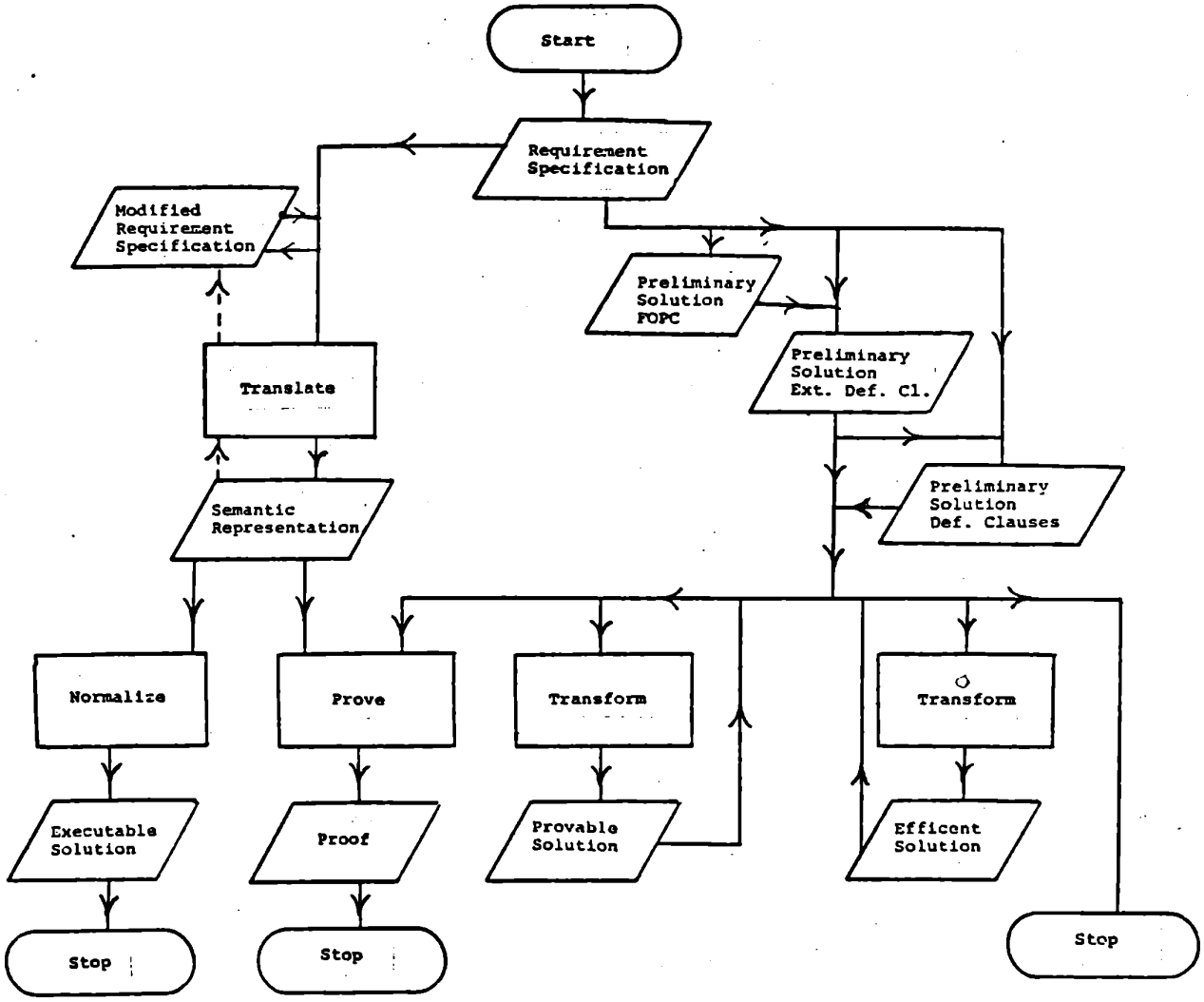


Figure 2

supports two kinds of representations. Here we chose a logic grammar (like [4]). An interesting alternative is a intensional grammar á la PHLIQA1 [3,7,14,16] . Conceivably the latter alternative may be characterized as a prototypical lambda calculus method and the former alternative as a prototypical predicate calculus method.

Also in a more narrow context of parsing (instead of full translation) the following method is probably preferable to that of the Uppsala Chart Parser [17] as far as modifiability, extensibility, portability, and experimentation with regard to grammatical descriptions are concerned.

The stepwise development of the computational sublanguage of ordinary English used a number of carefully selected benchmark problems.

5. An example

The first benchmark problem is simple enough to allow a fairly thorough presentation.
By the example of the fallible Greek we also demonstrate the idea of expressing a grammar as a logic program.

The problem consists in having the computer respond in a sensible way to the following ordinary English text (or natural language requirement specification):

Turing is human.
Socrates is human.
Socrates is Greek.
every human is fallible.
which human is Greek and is fallible ?

i.e. the program should answer the query in the last sentence.
We select the following simple context-free grammar

```
<Sentence>        ::=<Term><Verbphrase>
<Term>            ::=<Propername>|<Determiner><Nounphrase>
<Nounphrase>      ::=<Noun>|<Noun><Relativeclause>
<Relativeclause>::= that <Verbphrase>
<Verbphrase>      ::=<Transitiveverb><Term>|<Verbphrase>and <Verbphrase>
<Determiner>      ::= every | which
<Noun>            ::= human
<Transitiveverb>::= is | isn't
<Propername>      ::= Turing | Socrates | Greek | fallible |  human
```

If we just want a parser (to accept or reject the input sentence) we may simply change the grammar into the logic program of figure 3, where the variables function as pointers to the input string.
To solve our problem of the fallible Greek we need a somewhat more sophisticated translation of the accepted input sentence, as shown in figure 4. For the sake of clarity we have here omitted the variables functioning as pointers (as in figure 3). We have only given the variables designating the focus and result(s).

Translate:

```
Sentence(x,z) if Term(x,y) & Verbphrase(y,z).
Term(x,y) if Propername(x,y).
Term(x,z) if Determiner(x,y) & Nounphrase(y,z).
Nounphrase(x,y) if Noun(x,y).
Nounphrase(x,z) if Noun(x,y)&Relativeclause(y,z).
Relativeclause(x,z)if Check(that,x,y)&Verbphrase(y,z)
Verbphrase(x,z)if Transitiveverb(x,y)&Term(y,z).
Verbphrase(x,w) if Verbphrase(x,y)&Check(and,y,z)
   & Verbphrase(z,w).
Determiner(x,y)if Check(every,x,y).
Determiner(x,y)if Check(which,x,y).
Noun(x,y)if Check(human,x,y).
Transitiveverb(x,y)if Check(x,y)
Transitiveverb(x,y)if Check(isn't,x,y).
Propername(x,y)if Check(Turing,x,y).
Propername(x,y)if Check(Socrates,x,y).
Propername(x,y)if Check(Greek,x,y).
Propername(x,y)if Check(fallible,x,y).
Propername(x,y)if Check(human),x,y).
```

Figure 3.

Translate:

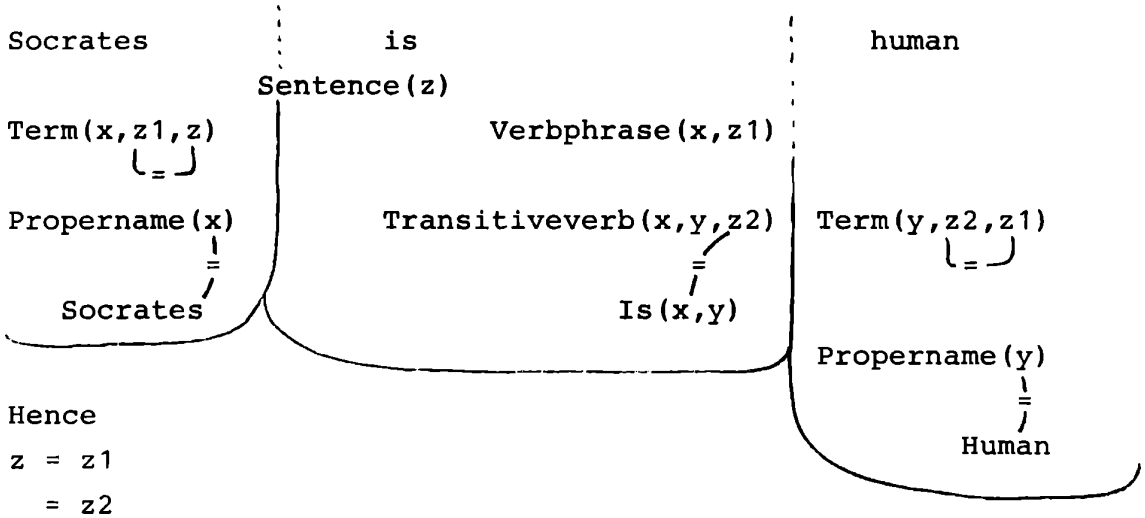
```
Sentence(z)if Term(x,z1,z)&Verbphrase(x,z1).
Term(x,z,z)if Propername(x).
Term(x,z1,z)if Determiner(x,z2,z1,z)&Nounphrase(x,z2).
Nounphrase(x,z)if Noun(x,z).
Nounphrase(x,z1&z2)if Noun(x,z1)&Relativeclause(x,z2)
Relativeclause(x,z)if Check(that)&Verbphrase(x,z).
Verbphrase(x,z)if Transitiveverb(x,y,z1)&Term(y,z1,z)
Verbphrase(x,z1&z2)if Verbphrase(x,z1)&Check(and)
   & Verbphrase(x,z2).
Determiner(x,z1,z2,∀x[z1⇒z2])if Check(every).
Determiner(x,z1,z2,Which(x,z1&z2))if Check(which).
Noun(x,Is(x,Human)) if Check(human).
Transitiveverb(x,y,Is(x,y)) if Check(is).
Transitiveverb(x,y,¬Is(x,y))if Check (isn't).
Propername(Turing)if Check(Turing).
Propername(Socrates)(if Check(Socrates).
Propername(Greek) if Check(Greek).
Propername(Fallible)if Check(fallible).
Propername(Human)if Check(human).
```

Figure 4.

The computational processing of the second and the second last sentences are displayed in the figures 5 and 6, respectively.

In conclusion the output from the translation program Translate is shown in figure 7.

Example

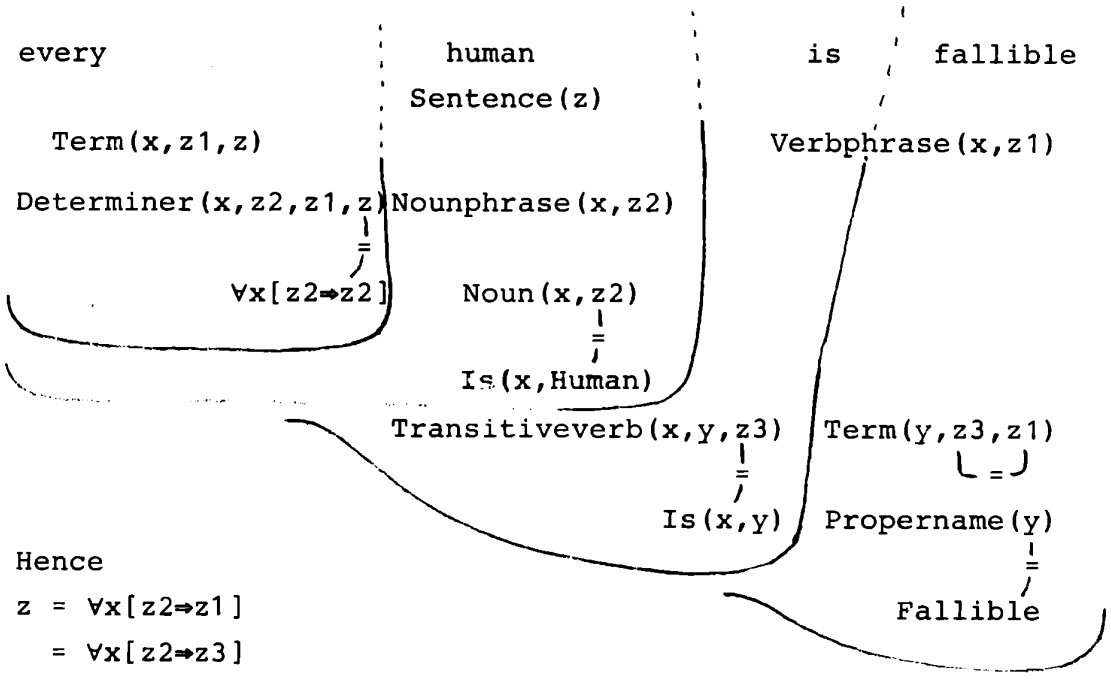


Hence

z = z1
  = z2
  = Is(x,y)
  = Is(Socrates, Human)

Figure 5

Example



Hence

z = ∀x[z2⇒z1]
  = ∀x[z2⇒z3]
  = ∀x[z2⇒Is(x,y)]
  = ∀x[Is(x,Human)⇒Is(x,Fallible)]

Figure 6

Translation output:

```
Is (Turing, Human).
Is (Socrates, Human).
Is (Socrates, Greek).
∀x[Is(x,Human)⇒Is(x,Fallible)].

Which(x,Is(x,Greek)&Is(x,Fallible)&Is(x,Human)).
```

Figure 7.

6. Normalization

The further transformation of formulae in the logical representation depends heavily upon the particular choice of logic programming language. In case of a dialect of Prolog the transformation should constitute a normalization into conjunctive normal form (or clausal form). Here the question is raised whether or not the sublanguage actually will generate logical formulae in a clausel form that are definite (Horn clauses). (An interesting problem would be to characterize the sublanguages satisfying this requirement).

In the first benchmark problem we get the normalized formulae of figure 8.

Clausal form (conjunctive normal form):

```
Is(Turing,Human).
Is(Socrates,Human).
Is(Socrates,Greek).
Is(x,Fallible)if Is(x,Human).
Print(x) if Is(x,Greek)&Is(x,Fallible)&Is(x,Human).
```

Figure 8.

7. Verifying handwritten programs

The logical representation may be used in the context of verifying handwritten logic programs, as indicated in figure 2. As an example of verification we have the second benchmark problem which is a variant of the Alpine Club problem from the artificial intelligence literature [15].

8. Natural language programming

The logical representation allows direct execution on the computer, which means that we are really developing an automated programming system based on natural language. It seems likely that this level of logical representation should be

considered as yet another level (a fifth) in the context of the four levels dealt with in the EUROTRA project [13]. Actually it is an obvious possibility to extract this kind of information from the third, so-called logico-semantic level and build the recommended logical representations from that information. Unfortunately, I tend to be very pessimistic as to whether this task will actually be realised by the EUROTRA participants.

It is relatively easy to supplement this automated programming system with specific rules concerning the problem domain to make it a knowledge-based system, whether the rules are formulated in a notation akin to the chosen logical representation, or in the form of additional texts in natural language.

So this system may constitute the kernel of a knowledge based automated programming system, where frame information specific to the universe of discourse are to be added. This kind of programming may very well be termed "programming in natural language" or "natural language programming"(hence the title of this paper).

The third benchmark problem is a prototypical database query language problem [19]. The fourth benchmark problem is a small computer aided design problem in architectural design. These benchmark problems fall into the class of natural language programming. Further details may be found in the report [12].

9. Status remarks

The system here was written for English to develop an English computational sublanguage. An obvious alternative could be to develop a similar system in some (or every) Scandinavian language (and it might include the surface structures of the logic programming language). A related experiment using the same method in the automated translation from Japanese will be reported on (in [2]).

When developing the appropriate natural sublanguage certain difficulties showed up in connection with pronouns. They may be examplified by the sentence:

A man takes an apple and he eats it.

The difficulties concerned the scope rules of the quantification and they could certainly be overcome by extending the logical connectives into two-dimensional operators in a systematic manner [12].

As far as the plural of nouns and quantification are concerned, they were needed in the fourth benchmark problem. There seems to be essentially six different ways to extend the computational sublanguage with quantification, as illustrated in figure 9.

There is a need for gaining experience with the use of systems like this one. Virtually nothing is available in the scientific literature.

Such a system should not be considered completely trivial to use, although its potentiality for popularization should be recognised (Virtually everybody who is not an analphabet might learn to use it).

Decisions:

1: Should the translation be one-pass or two-pass (many-pass) ?
2: Should the result be expressed in higher order functions (or cardinality) ?
3. Should there be two truth-values or three (many) ?

1: Passes	2: Higher-order	3: Values	
1	No	2	(here)
1	No	3	-
1	Yes	2	(Intensional grammars and Lexical-Functional Grammars)
1	Yes	3	
2	Yes	2	(here)
2	Yes	3	(Logic grammars).

Figure 9.

10. References

A few related contributions from my institute are included in the list though not referred to in the paper.

[1] F. Als et al: Compiling in Prolog (in Danish), DIKU report 83/16, Institute of Datalogy. Copenhagen University, 1983.

[2] A. Bernth: Logics applied to the translation of Japanese (in Danish), these proceedings.

[3] W.Bronnenberg et al.: The question answering system PHLIQA1, in L. Bolc (ed.) Natural communication with computers Vol. 2, 1980.

[4] A. Colmerauer: An interesting subset of natural language, in Clark and Tärnlund (eds.) Logic programming, 1982.

[5] N.D.Jones and A.Mycroft: Stepwise development of denotational semantics for Prolog, DIKU report 83/1, Institute of Datalogy, Copenhagen University, 1983.

[6] P.H. Jørgensen and G. Koch: Two new methods of natural language database queries (in Danish), Proc. NordDATA Conf., Copenhagen 1981, 2, 227-232.

[7] G. Koch: Experimental formalization of Danish . Institute of Datalogy, Copenhagen University, 1979. DIKU report 79/19 (in Danish).

[8] G. Koch: A Prolog way of representing natural language fragments, DIKU report 80/16, Institute of Datalogy, Copenhagen University, 1980.

[9] G. Koch: A problem oriented software development method in computational linguistics (in Danish), Proc. De Nordiske Datalingvistikdagene, E. Lien (red.), Trondheim University, 1981, 47-63.

[10] G. Koch: Grammars and predicate calculus, DIKU report 81/16, Institute of Datalogy, Copenhagen University, 1981.

[11] G. Koch and K.B. Larsen: Logical prototyping in system development (in Danish), Proc. NordDATA Conf.,Göteborg, 1982, 1, 270-273.

[12] G. Koch: Stepwise development of logic programmed software development methods, DIKU report 83/5, Institute of Datalogy, Copenhagen University, 1983.

[13] B. Maegaard and H.Ruus; Multilingual syntax and morphology for machine translation, in K. Hyldgaard-Jensen and B.Maegaard (eds.): Machine translation and comptutational lexicography, Copenhagen 1982, 26-34.

[14] R. Montague: Formal philosophy, 1974.

[15] N.J.Nilsson: Principles of artificial intelligence, Springer-Verlag 1982.

[16] P.S.Olsen: Computational philosophy, future DIKU report, Institute of Datalogy, Copenhagen University, 1984.

[17] A.-L. Sågvall-Hein: A parser for Swedish, Uppsala University, UCDL-R-83-2.

[18] E. Upfal: Programming in predicate logic, DIKU report 81/9, Institute of Datalogy, Copenhagen University, 1981

[19] J.E. Ullman:Principles of database systems, 1980.