Powerful ideas in computational linguistics - Implications for problem solving and education

Gerhard Fischer, Institut fuer Informatik
Universitaet Stuttgart, West-Germany

Abstract

It is our firm belief that solving problems in the domain of computational linguistics (CL) can provide a set of metaphors or powerful ideas which are of great importance to many fields. We have taught several experimental classes to students from high schools and universities and a major part of our work was centered around problems dealing with language. We have set up an experimental Language Laboratory in which the students can explore existing computer programs, modify them, design new ones and implement them. The goal was that the student should gain a deeper understanding of language itself and that he/she should learn general and transferable problem solving skills.

1. Introduction

Problem solving with the computer for the non-computer expert is slowly recognized as an important activity in our educational system. It is done best in a project-oriented course in which the student learns to solve problems in different domains. In the past, activities of this sort have been centered around numerical problems, physics problems and the standard computer science problems (eg like writing a sorting procedure).

2. The state of the art

The relevance of problems from linguistics has been ignored. The reasons for this fact are easy to explain:

1) the **educational community in the language-oriented fields** has very little knowledge about using a computer to write interesting programs to gain a deeper understanding of the problems in their domain

2) the **computer experts** were not familiar with linguistics

3) the most commonly used **programming languages and systems are inadequate** to deal with the data structures and dialog requirements which are relevant for language processing

4) new fields like **artificial intelligence, cognitive science and computational linguistics** were not widely known

The level of ignorance can best be seen by using ELIZA as an example: many people thought that it was a program which would "understand" the contents of a dialog. It was not evident to them that ELIZA represents nothing more than an exercise in pattern matching and symbol manipulation, where certain keywords trigger a few prestored answers. It may also serve as an example for how little machinery is necessary to create the illusion of understanding.

In our interdisciplinary research project (KLING et al, 1977) we have tried to overcome these problems by providing opportunities for the student to explore powerful ideas in the context of non-trivial problems and by showing that the computer prescence can do much more for education than improve the delivery system for curricula established independently of it.

3. Cognitive Science and Programming

In recent years the view has emerged that the language of computation is the proper dialect to describe basic issues in psychology, linguistics and education. Research in Cognitive Science has demonstrated that the phenomena surrounding computers are deep and obscure, requiring much experimentation. Cognitive Science theories about problem solving, representation of knowledge and other cognitive abilities provide the foundation for our understanding of programming.

We believe that the whole enterprise of programming can be much better explained with concepts from CL than with those from mathematics. Problems in CL are often ill-defined, algorithms are seldom given and programming is more a design task than it is a coding of a known algorithm. The problem formulation phase is more relevant than the execution of a program and systems are needed to support this phase of the problem solving process. Successive formulation of programs serve as stepping stones towards the goal of defining the specification of a problem.

Humans have a good intuitive understanding of the problems in CL and they can do the things (like communicate in natural language, deal with vast amounts of knowledge, infer new knowledge from exiting one) - even if they do not know how they do it. Programming can be understood as an effort to make our own knowledge explicit and can provide us with adequate metaphors to describe our own mental functions.

4. Design of a Language Laboratory

The design of learning environments is an important goal for the educational theorist and the teacher. The computer as a new technology has created almost unlimited possibilities to create new and challenging environments. The Turtle world (PAPERT 1979) and the simulation world of Smalltalk (KAY 1977) provide good models of what can be done.

In our project we have set up an experimental Language Laboratory in which the students can explore existing programs, modify them, design new ones and implement them. We took great care in our design (by following the tradition of the LOGO projets as opposed to CAI approaches) that the students could work in an **active** mode and develop ideas in a **personal** way (not limited by the teachers approach). Our teaching style was not to provide answers but the learners were encouraged to use their own language knowledge to find a solution. Their work had to rely on self motivation which seems a more reasonable goal in CL where the products (eg poems, horoscopes, question/answering systems etc) can be more interesting and aesthetically pleasing than a set of numbers appearing as a result in numerical mathematics. With our Language Laboratory we wanted to create an environment in which the student's task is not to learn a set of formal rules (eg about the syntax of a programming language), but to give them a world in which they could develop sufficient inside into the way they used language to allow the transposition of this self-knowledge into programs.

The students were exposed to different formalisms (primarily to LOGO, but also to LISP, ATNs, semantic networks, MICRO-PLANNER) and could explore the range of possible models which could be implemented in a **cognitively efficient** way with these formalisms. We tried to engage them in problems of moderate complexity (the students were no researchers working full-time in a project) and we created micro-versions of programs by ommiting features which were not essential for a conceptual understanding.

5. Powerful ideas

There is little doubt that we will be unable to solve the problems of coverage in our school and university subjects and of predicting what specific knowledge our students will need in thirty or forty years. Despite the fact that we would like to have more empirical evidence that problem solving skills can be taught, we have little choice, because we don't have any real alternative (for a detailed discussion of this issue, see SIMON 1978).

Cognitive Science and Artificial Intelligence have contributed to our understanding of problem solving processes and we believe that general problem solving skills, crystallized as powerful ideas, can be taught explicitly in the context of a rich environment of problems. The main goal of this paper is to show that CL provides this rich context (which if it is not superior than mathematics, at least complements mathematics).

Powerful ideas are nuggets of knowledge, which are universally useful, which appear over and over in different disciplines and which can be connected in a natural and illuminating way with a large complex of other ideas.

One example of a powerful idea is the heuristic: **"divide and conquer"**. It appears to be an almost universal truth, but how it is done in the context of a concrete problem situation is far from being trivial. Many of the typical problems (like writing a program to compute factorial, to sort a set of objects or to solve a trivial puzzle) are too simple, so there is little need to use this heuristic. Furthermore many traditional programming systems are not build for (or do not even support) this problem solving approach, whereas in our work the heuristic took on a concrete meaning and was the only successful way to solve a problem.

In the following parts of this section we briefly describe a set of powerful ideas which can be explored in the context of realistic problems and research areas in CL (the projects are fully described in BOECKER/FISCHER, 1978):

1) **difference between syntax and semantic** (eg in the context of writing a program to generate poetry, in solving word problems in algebra)

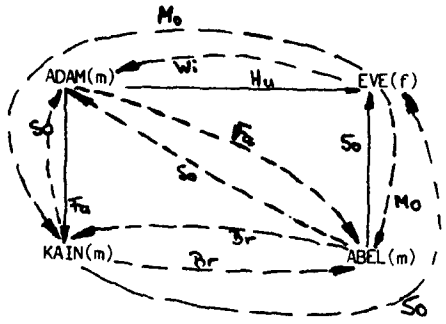
2) **rule systems** (a sort of production systems; they are useful for the organisation of programs to derive the plural form of an English noun given the singular, to conjugate French verbs, to characterize the rules and heuristics in a game or to implement the evaluation procedure of LISP)

3) **design and implementation of a mini-language** (this allows us to describe a problem in terms which are characteristic for the problem instead of being forced to use a given general purpose programming language; examples are: production systems, pattern matcher, etc); in programming, it is a natural task to design your own representational system whereas in mathematics people mostly use the representational system given to them

4) **experimentation with a wide variety of different grammars** (eg to generate and analyse algebraic expressions; to transform arabic

numerals into roman ones and vice versa; to explore transition networks and augmented transition networks in dealing with natural language)

5) **knowledge representation:** eg to derive implicit knowledge and to study the impact of processing at read-time (antecendent theorems) versus question-time (consequent theorems) in a system which dealt with family relations (a system of this sort can be contrasted with ELIZA or a program to cast horoscopes); the following diagram illustrates how 9 implicit relationships (---) can be derived from 3 explicit ones within a family of four persons:



6) exploration of **statistical properties** of languages in the context of a wide variety of different language games (eg like designing the distribution of a Scrabble game, information content of vowels in different languages etc)

7) **general computational ideas** (eg like backtracking, which is encounterd in parsing non-deterministc grammars and which could be applied to pattern matching and tree like data structures)

6. Pattern Matching - an example for the design and implementation of a mini-language

A matching capability can be a key element for many problem solving tasks involving the computer to make otherwise large, complicated efforts reachable. The following powerful ideas can be investigated in the context of this project:

1) **incremental design:** we can start with a pattern matcher which is basicly an EQUAL predicate. The next steps could be: a membership predicate, a pattern with slots of fixed size, a pattern with slots of arbitrary size (which creates the need for back-up), the possibilty for simultanous assignment of matched elements to pattern variables, the restriction of matching by using predicates etc

2) **the problem is ill-defined:** the specification of the pattern matcher should be derived from the needs of using it to simplify problem solving tasks. A partial implementation can be an important help for a further specification or for a revision of already existing parts, ie the problem formulation is an important part of the problem solving process

3) **definition of a new language layer:** the pattern matcher can be used as a new language layer between the problem and the programming language and it can help to reduce the distance between the two.

4) **glass-box approach:** in many situations, we are primarily interested in using the pattern matcher. But by making use of an already existing program the student is not confined to a black box (like it would be in CAI environment); at any time he/she can look inside the program, open it up, change it to his/her own needs etc. A prerequisite for a program to be a glass-box is that it is implemented in a formalism the student is familiar with.

5) **recursive control structure:** a pattern matcher is a good example for the power of recursive definitions and control structures which can be used in many other situations

A pattern matcher can be used in all projects where symbolic structures have to be dissected and identified, eg for the translation from infix to prefix, for parsing and translating processes, for morphological analysis, for simple I/O routines (eg the identification of keywords), for ELIZA like programs and for symbolic manipulation of algebraic expressions.

We do not have the space to document the problem solving processes (including all the incomplete versions) which occured in the context of implementing the pattern matcher (see BOECKER/FISCHER 1978) but we want to give examples of its use. The simplification with the help of a pattern matcher can be demonstrated by a program for infix to prefix translation (written in LOGO; the program also nicely shows the power of recursive definitions):

```
TO PREFIX :INFIX
10 LOCAL "A "B
20 IF (EQUAL COUNT :INFIX 1) THEN OUTPUT :INFIX
30 IF MATCHP [?A + ?B] :INFIX
      THEN OUTPUT (SENTENCE "SUM PREFIX :A PREFIX :B)
40 IF MATCHP [?A - ?B] :INFIX
      THEN OUTPUT (SENTENCE "DIFFERENCE PREFIX :A PREFIX :B)
50 IF MATCHP [?A * ?B] :INFIX
      THEN OUTPUT (SENTENCE "PRODUCT PREFIX :A PREFIX :B)
60 IF MATCHP [?A / ?B] :INFIX
      THEN OUTPUT (SENTENCE "QUOTIENT PREFIX :A PREFIX :B)
70 EXIT [WRONG SYNTAX]
END
```

The following testruns show how the program works:

```
?PRINT PREFIX [U + V]
SUM U V
```

```
?PRINT PREFIX [A + B * C / A - D]
SUM A DIFFERENCE PRODUCT B QUOTIENT C A D
```

This version of the program can be extended easily to include other operators like ">" or "<":

```
65 IF MATCHP [?A > ?B] :INFIX
      THEN OUTPUT (SENTENCE "GREATERP PREFIX :A PREFIX :B)
67 IF MATCHP [?A < ?B] :INFIX
      THEN OUTPUT (SENTENCE "LESSP PREFIX :A PREFIX :B)
```

It is an instance in the class of rule systems which we mentioned earlier. The ordering of the rules takes care for the precedence conventions of infix notation. We have chosen this application specifically to support our claim that many problems considered to be mathematical can be more clearly understood by looking at them from a linguistic viewpoint (and the APL experience shows that changing the precedence rules for the evaluation of arithmetic expressions poses a non-trivial problem).

Another application of the pattern matcher would be to parse sentences in a language where the grammer is given. For this purpose we assume that the pattern may contain predicates (which are marked by "<" and ">"):

```
PRINT MATCHP [A <NUMBERP> B <ZEROP>] [A 3 56 B 00]
TRUE
```

The following grammer may serve as an example (it describes the language of at least one "0" followed by at least one "1"):

```
<SENT>  —>  <S0> <S1>
<S0>    —>  0 | 0 <S1>
<S1>    —>  1 | 1 <S1>
```

SENT, S0 and S1 can be implemented with the pattern matcher as followed:

```
TO SENT :INPUT
10 OUTPUT MATCHP [<S0> <S1>] :INPUT
END

TO S0 :INPUT
10 IF MATCHP 0 :INPUT THEN OUTPUT "TRUE
20 OUTPUT MATCHP [0 <S0>] :INPUT
END

TO S1 :INPUT
10 IF MATCHP 1 :INPUT THEN OUTPUT "TRUE
20 OUTPUT MATCHP [1 <S1>] :INPUT
END
```

A few testruns show the working of the parser:

```
PRINT SENT [0 0 1 1 1]      PRINT SENT [1 0 1]
TRUE                        FALSE

PRINT SENT [0 0 0 1]        PRINT SENT [0 1 0 1]
TRUE                        FALSE
```

7. Implications for problem solving and education

Powerful ideas have the potential to lead to a breakdown of the traditional boundaries between established scientific disciplines and reduce the division of school knowledge into disjunctive compartments. By working on some of the projects described above our students found that the knowledge which they acquired or discovered was not only useful in the context of a specific task but could be successfully used to understand and solve problems in other domains as well, which should be illustrated through the following two specific examples:

1) the students became aware that the evaluation of arithmetic expressions (as it is commonly used in mathematics) is not something determined by God but that it is only a convention and that the laws behind it can be easily explained by the use of a grammar.

2) a student discovered why mathemeticians talk about one-to-one mappings (which never made any sense to him in mathematics) by trying to design secret codes in some of the language games (eg Pig Latin and other ones)

Another important feature of our approach was that the students extended the range of their **"subjectively computable"** problems, which helped them to replace their view of the computer being a giant adding machine with the more adequate view of being a general information processing device. We challenged their views thinking about the computer. Despite the fact that computation is still in its infancy there are many strong beliefs what computers are, what they can do and what they can not do.

By being exposed to the complex problems mentioned above the students got familiar with general problem solving ideas about representations, planning and debugging. The intuitive understanding which a person has about his/her own language provided the basis that debugging incomplete and incorrect programs becomes an easy-to-grasp activity, because bugs in language programs have a high **visibility** (ie we can discover them by inspection and not only by extensive calculations like it is the case in numerical computations).

Problems in CL provide good prototypes to understand the theoretical relevance of debugging. Opposed to the dominant view in

computer science, where many people regard bugs as an awkward obstacle (or as an indication that the programmer is unable to think clearly and carefully enough) we consider bugs as potentially informative friends and as a starting point to find out about the discrepancies between a specification (a model, a theory) and an implementation (a program). In CL, most people are aware that if a conflict arises we can not always conclude that the specifications are correct and the implementation is wrong (as in Galileo's case, where the theory was wrong and his data were correct).

Working on the projects described above, the students can do work which is close to the research front (if they would have done their work ten years earlier they could have earned a PhD degree with it). This makes this subject material once again more interesting than much of mathematics where the students have to think about what is not even close to the current research front.

9. Empirical findings

Most of the hypotheses and assertions of the previous sections are supported by the empirical work in our project. We have not made an effort to do any kind of formal evaluation, but we have carried out a large number of informal investigations to understand the impact of our approach. Students filled out questionaires, participated in think-aloud protocols for many problem solving situations and we tried to understand their programs and the bugs they produced during the solution of a complex problem. There is no space here to talk about this in detail; the information is documented in KLING et al (1977) and FISCHER (1978 and 1979).

We believe that our approach turned out to be very successful. The students enjoyed working in our laboratory and they learned a lot about language as well as general problem solving and programming skills. Especially students with little interest in mathematical problems were motivated by language-oriented applications. They could work in an active mode and investigate arbitrary formalisms and conjectures. They could see that ideas from linguistics could help them to understand problems in other domains, which supports our hypothesis that problems from CL can serve as an entry point and as a transient object to the world of problem solving, programming and mathematics.

Acknowledgements

I would like to thank H.-D. Boecker, A. Fauser, J. Laubsch and D. Roesner for many critical comments about earlier drafts of this paper.

References

Boecker,H.-D. and G. Fischer (1978): "Interaktives Problemloesen mit Computerhilfe: Problemaufgaben zur Linguistik, Informatik und Kuenstlichen Intelligenz", Forschungsgruppe CUU, Darmstadt

Fischer, G. (1978): "Probleme und Erfahrungen bei der Programmierausbildung im Informatik-Unterricht" in W. Arlt (ed): "EDV-Einsatz in Schule und Ausbildung", Oldenburg Verlag, Muenchen, pp 70-75

Fischer, G. (1979): "Fehlerdiagnose - Grundbaustein fuer ein Verstehen von Lehr- und Lernprozessen", in Beitraege zum Mathematikunterricht, Schroedel Verlag

Kay, A. (1977): "Microelectronics and the personal computer", Scientific America 1977, pp 231-244

Kling, U., Boecker H.-D., Fischer, G., Freiburg, D., Schneider, B. and Schroeder, J. (1977): "Projekt PROKOP", Forschungsgruppe CUU, Darmstadt

Papert, S. (1979): "The LOGO Book", unpublished draft, MIT AI Lab

Simon, H. (1978): "Problem Solving and Education", CIP Working Paper No. 391, Carnegie Mellon University