MATHEMATICAL PROBLEMS IN COMPUTATION THEORY
BANACH CENTER PUBLICATIONS, VOLUME 21
PWN – POLISH SCIENTIFIC PUBLISHERS
WARSAW 1988

HOW TO TREAT DECISION PROBLEMS BY DIFFERENT PROGRAMMING METHODOLOGIES

G. CIONI
A. MIOLA

Institute of Systems Analysis and Informatics, CNR Rome, Italy

This paper presents an overview of the most relevant programming styles.

In this overview we will consider several classical programming methodologies and discuss their features and their limitations from the point of view of a specific class of applications, namely the so-called decision problems.

The main goal is to characterize the fundamental aspects of different programming approaches in order to describe effective tools for the solution of problems in the area of applications we have chosen.

In the last section of this paper we will also refer to the main current research directions devoted to obtaining an integration of different programming styles and of different programming formalisms in a unique programming environment described by a unique semantics.

1. Introduction

The field of applications of computer science has been enlarged in a tremendous manner in the last few years. However, the impressive significance of this enlargement is not only the number of different areas where the computer science is now applied, but the width and depth of any single area of applications.

One of the most significant application areas is that of the so-called decision problems. Such problems can be described, for instance, by the following elements:

- a set of states S;
- a set of initial states $I \subset S$;
- a set of final states $F \subset S$;

– a transition function T: $S \rightarrow S$ which describes the evolution of the system from a state to the next one. Such function is generally not deterministic, and a strategy to choose among different next states is necessary.

The most important characteristic of this kind of problems is the possibility for the system to evolve following several different sequences of states. The name "decision problem" comes essentially from this intrinsic need of making different choices at the successive stages of the computational strategy, in order to achieve in the best way the solution, i.e., the final state.

Because of this intrinsic need of heuristics, problems of this kind are generally very difficult to solve automatically by a purely algorithmic approach. However, even if the use of the heuristic approach could help very much, new computational problems arise when using backtracking on the evolution states graph.

Among the different kinds of decision problems two can be quoted as interesting examples: the time schedule problems for school classes and the systems to the on-line control of trains running on highly busy railways. These two problems are characterized by an initial static and permanent data base, by a similar dynamic data base and, finally, by a set of rules to be applied to reach the chosen goal. These rules are generally based on the long experience of a well-skilled man who operates the system and solves real problems. Such problems are also characterized by their very big dimensions in terms of data and rules. Let us consider the very complicated situation which occurs when a train is on delay and comes into collision with some other trains, or the confused conditions which are imposed by teachers who ask for a specific time schedule, may be opposite to that proposed by students.

These kinds of problems are real, effective problems and must be solved, hopefully, by a very efficient system. However, their obvious high complexity does not allow us to make clear and simple considerations on the programming methodologies to be used. Therefore for our purpose we will consider games which have the same characteristics, but are simpler to describe, and in this paper we will refer to two classical and well-known puzzles: the Hanoi tower and the cannibals and missionaries problem [9]. In the problem analysis we will distinguish three different elements:

1. the rules of the problems, which explain what the problem means;
2. the moves or actions executed by the player to solve the problem, i.e., to reach the solution;
3. the strategies and the tricks which can help the player to reach the solution in less time.

Note that sometimes it is not easy to distinguish between a general strategy and specific tricks.

In the following we will discuss how different programming styles can face these problems. However, some preliminary considerations on hardware

and software environments are necessary to give a more complete frame to our discussion.

At the present time, machines on the market are generally based on the classical, well-known von Neumann architecture, with basic operations referring to the concepts of variables and of assignment of values to variables, and purely sequential execution.

New architecture, based on the concepts of function and of related mechanism of applying functions to arguments, is also available [18]. The new machines with such an architecture will be easily used also for parallel computations. Some specific products of this kind are already on the market: for instance, Symbolics and Lambda LISP-machines. Furthermore, the projects of the so-called V-th generation [17], [20], [12] should bring faster and more powerful machines based also on the principles of automatic inferences.

For the purpose of our discussion in this paper, we may underline that the original rigid dependence of programming languages design on a fixed given architecture has now been made free. In fact, the actual tendency is to design architectures on the base of the features and of the evaluation mechanisms offered by the programming languages to be made available on such architectures.

From these brief considerations we derive a further stimulation for research on programming languages, being aware that in future the hardware and software problems will be more and more strictly connected.

2. Programming methodologies

In order to attack, and possibly to solve, decision problems we do need specific and important features from programming languages for computers of today and of tomorrow.

Such languages must allow to describe several different kinds of knowledge. At the same time, the programming style must be more declarative than imperative, giving to the user the possibility of expressing the given problem in terms of "what" must be pursued, and not in terms of "how" that could be performed.

After that, the automatic execution of a request, declaring a given problem, will be completely hidden to the user, and therefore the programming language we expect must be implemented with specific demands for security, correctness and efficiency.

We could synthetize these considerations by saying that the research is being developed toward the effective possibility to transfer, as much as possible, to a basic automatic level, the work generally done by the programmer [24], [2].

In the last few years the main research activities in programming methodologies have been characterized by the following elements [13]:

1. modularity and control abstraction;
2. data abstraction, encapsulation and hierarchy of definitions of objects;
3. expressiveness of the formalism, flexibility and easiness-to-use;
4. efficiency and security.

1. Modularity is absolutely necessary to use a programming language effectively in the large spectrum of new application fields. Modularity means having the possibility to decompose a problem into many small subproblems each of which represents a simple function, and is linked together with the others by clear and completely defined interfaces.

To speak about abstraction means to isolate in the problem only some fundamental basic elements, putting aside all the details which can be specified later. The following step will be the refinement, in which the details, previously disregarded, are added to complete the specification of the problem.

Modularity and abstraction are, in some sense, independent of the programming language used, but can be promoted by some very modular languages.

The first important feature, which is now present in every modern language (from Pascal on), is the separation between the declarative and executive parts; that allows one to use, in a natural way, a top-down methodology for the development of programs. The declarative part corresponds to program specification and also defines the connections with other specification parts. The executive part contains, and isolates from the rest of the world, the way in which the operations are to be executed.

The second feature is the existence of many kinds of units suitable for different objectives and programming styles. For examples a "block", which defines the scope of visibility of objects and separates the different components of a program, and a "process", which realizes the specific unit which accomplishes a specific task, maybe in parallel with some other tasks which interact together. These two kinds of units, which we have quoted as typical, are, for example, present in ADA [1].

2. The second element, if we view a big program as a set of independent modules, is to think over the data which are an element going from one module to another and having important and well-defined characteristics.

Data today means abstract data types [19]; i.e., the definition of the set of elements allowed (which we can regard as the syntax of the data) and of operations admissible on these elements (which can be viewed as the semantics of the data). That means encapsulation in all different moments of the software life. If a programming language admits constructs like classes or

packages, the declarative part of these units represents the specification of the type of objects of our problem.

The implementation level is an extension of this level and can be realized by a class which is a construct typical to define complex data structures of our interest, together with the admissible operations.

Strictly connected with encapsulation there is, generally, the requirement of a hierarchy, which follows directly from the definition of abstract data types; it is very important to have the possibility to define a data structure as an extension of a simpler one, whose properties it retains, adding to them some new ones. We obtain, in this way, a hierarchical structure of abstract data types, very easy to test and implement. As a typical example of this point, we quote SIMULA, with its very powerful and flexible constructs, namely the construct "class" [11].

3. In order to represent several different types of knowledge we certainly need languages with a very high expressive power. In fact, we do have to describe facts and their consequences in the most natural way. Furthermore, the process of making inferences from some given facts must be carried out in a given contex where the successive implications could be verified. Logic programming languages [7] offer such characteristics, even if they have still a main limitation, namely the lack of flexibility. We mean here the possibility, offered by a programming language, of being adapted to treat several different objects with different approaches.

All languages must be easy to understand, to learn and to use; for example, an explicit request of the American Defense Department tells that ADA has to share all these characteristics.

4. In many real situations efficiency and solvability of a given problem are essentially the same. In fact, when a problem is very complex, it is solvable only if an efficient solution can be obtained by some very efficient tools. The efficiency of a programming language is strictly connected to the implementation of different features and constructs, to the quality of control checks, generally performed at compilation time, and to the way the run-time system works, in particular, for the memory management.

However, it is important to notice that also security is a crucial parameter in respect of the quality of a language. It is in fact necessary to avoid dangling reference, which is frequently present during programming at-large with dynamic languages and dynamic objects. To obtain a good degree of security, one can ask for a totally correct semantics and related implementation. Unfortunately, that is not the case for many of the recently defined languages (for this topic, see the two papers of the author quoted in the reference list [4], [5]).

3. The main programming styles

Classically two main categories of programming styles are considerated: the imperative and the declarative one.

The imperative style states how it is possible to solve a problem, i.e., how to derive an output from a given input while, generally, what the problem has to solve is completely hidden.

On the other hand, the declarative style states what a program is supposed to compute, i.e., what is the connection between the input and the output and it does not specify how the computer can solve the problem.

A very simple example of these two styles is given in the following two programs which compute the minimum between two input data. The first version is given in BASIC.

```
10 INPUT N1, N2,
20 IF N1 < N2, THEN GOTO 30 ELSE GOTO 50,
30 PRINT N1,
40 GOTO 60,
50 PRINT N2,
60 END.
```

The second program is written in a logic language like PROLOG.

```
lesser-of (-N1 -N2 -N1)
lesser-of (-N1 -N2 -N1) if
        less (-N1 -N2)
lesser-of (-N1 -N2 -N2) if
        less (-N2 -N1)
```

Note that in both programs built-in functions are used to compare two numbers ("<" and "less"). Independently of the syntactical aspects, these two programs show the main differences between the two styles.

The categories considered are related to the following classes of programming languages:

1. functional languages;
2. logic languages;
3. imperative languages;
4. object-oriented languages.

Let us give here, for each of these classes, only some general considerations, together with their main limitations.

Our goal in this paper is to move in the only possible direction, namely the integration of different programming styles. In fact we do think that is the only way to guarantee the treatment of sophisticated classes of problems.

3.1. Functional programming languages. The functional programming approach is an original approach in computing, based on the mathematical

notion of function and on the computing mechanism of applying a function to some arguments (even functional arguments), mechanism which is typical of the abstract model of lambda calculus [31], [26].

The simple subproblems which are the functional decomposition of a complex real application can be seen as corresponding to different modules implementing different functions. In this sense it is natural to regard the functional programming as the most immediate way to program, with a high level of modularity and security.

To illustrate the functional programming style, let us consider the following program which solves the Hanoi tower problem [31].

```
DEFUN TOWER-OF HANOI (N) (TRANSFER 'A 'B 'C N))
(DEFUN MOVE-DISK (FROM TO)
        (LIST (LIST 'MOVE 'DISK 'FROM FROM 'TO TO)))
(DEFUN TRANSFER (FROM TO SPARE NUMBER)
        (COND ((EQUAL NUMBER 1) (MOVE-DISK FROM TO))
            (T (APPEND (TRANSFER FROM
                SPARE
                TO
                (SUB1 NUMBER))
            (MOVE-DISK FROM TO)
            (TRANSFER SPARE
                TO
                FROM
                (SUB1 NUMBER))))))
```

In order to improve this approach and in order to obtain clear interfaces between modules, some recent work, has been made to introduce data types in a functional programming methodology [3].

3.2. Logic programming languages. Logic programming languages belong to the category of declarative languages, since they state what the program is supposed to compute, and they do not explicitly specify how the computer has to solve the problem. Unlike functional programming languages, they may give many solutions for a given problem, i.e., they admit non-deterministic computations. Another important feature is the reversibility of logic programs, while in the other languages the input and output are clearly distinguished [22], [23].

From the external point of view, the user has only to define his own problem in terms of facts and rules, without caring of the way the system could reach the solution of the problem.

If we carefully observe logic programming languages, the most known of which is PROLOG [30], we can deduce that we deal with an approximation of logic and that there exist some limitations:

1. the unification is not always correct (the occur check is sometimes missed);
2. the search rule (non-determinism) is unfair because it is based on backtracking, and on the order of the successive choices which is determined by the order of the clauses (given by the user);
3. much heuristics, which can be realized by admissible extra-logical features, is dangerous, because it forces the user to mix declaration and control, and it does not always maintain a correct backtracking (as an example, take the "cut" operator);
4. also semantics is not satisfactory, because not completely conforming to that of the Horn clauses;
5. the original declarative programming style tends to obscure programming, sometimes even more than in the old imperative languages.

As an example, look at the following program which is taken from [8] and which has to solve the classical Hanoi tower problem.

```
hanoi (N): – hanoi (N, L, [ ]), N1,
    write (L), N1,
hanoi (N) -->put (N, A, B, C),
put (0, –, –, –)-->[ ].
put (N, A, B, C)-->{M is N–1}.
    put (M, A, C, B),
    move (A, B),
    put (M, C, B, A),
move (X, Y)-->[from, X, to, Y].
```

As you can see, efficiency and readability are completely lost.

However, the main advantage of the actual implementation of logic programming languages (see PROLOG) is its simplicity and efficiency, at least on a sequential stack based machine.

Moreover, both at the system and the application level, there exist some software components which are intrinsically procedural, and also some primitive data types are procedurally defined. For example, operating systems and programming tools. A declarative definition of these components would be very unnatural and less efficient.

3.3. Imperative programming languages. In this section we will not refer to the old imperative languages, but only to the newest ones: ADA can be considered the representative of the "classical" line while in the object-oriented direction we can consider PARAGON [29] and LOGLAN [25]. The features of these languages are taken from several different programming styles, and therefore they can hardly be classified as strictly imperative.

It is thus important to list their features rather than classify them. In

particular, the following are the characteristics which we believe fundamental for the treatment of complex decision problems:

1. strong typing and consistency checking;
2. secure and efficient memory management, user-dependent, and based on system invariants;
3. hierarchical abstract data types;
4. dynamically defined operations depending on data structures;
5. high level clearness and flexible modularity;
6. primitives for unit activation and deactivation, user-dependent;
7. separate compilation.

Having in mind these characteristics, we can consider the following skeleton example to solve the cannibals and missionaries problem using a LOGLAN-like language.

```
program
   unit backtrack: class
         unit node: coroutine
               unit virtual leaf: function
   unit virtual answer: function
   unit virtual lastson: function
   unit virtual nextson: function
   unit virtual equal: function
   unit virtual cost: function
         unit ok: function
         unit purge: procedure
         unit elem: class
         unit virtual insert: procedure
         unit virtual delete: function
         unit killall: procedure
   unit bestsearch: backtrack class
         unit exnode: node class
         unit virtual delete: function
   . . . . . .
   . . . . . .
         unit virtual insert: procedure
   . . . . . .
   . . . . . .
   begin
   . . . . . .
   . . . . . .
         pref bestsearch block
   unit state: exnode class
```

```
unit virtual answer: function
. . . . . .
unit virtual leaf: function
. . . . . .
    . . .
    . . .
  . . . .
unit virtual less: function
. . . . . .
unit display: procedure
```

Looking at this example we can distinguish three different parts:

– the backtracking class, which we call HR because it corresponds to the heuristics realized directly by the implementation in logical languages,

– the bestsearch class, which we call MK because it defines a specific strategy based on a meta-knowledge,

– the main block, which we call KB because it corresponds to the user problem and it is based on the specific knowledge of the problem, i.e., on its knowledge base.

With this many-level organization the user can obtain a high degree of efficiency, without caring of the general problems of strategy. As is better illustrated in [6], a very high level programming languages can be useful for the solution of decision problems if it has the fundamental features presented in the previous part. In this case the user can approach his problem with the same philosophy as in the logic programming languages, having also the benefits of the imperative style.

3.4. Object-oriented programming languages. This class of languages has the main feature to be based on the definition of objects to which a specified behaviour is associated [21]. These objects are represented by data structures and their behaviours can be described by a given set of operations and predicates. They can also exchange messages corresponding to operation requests. In this sense the entire program becomes a set of definitions of objects and of descriptions of messages among them. Therefore there is no form of explicit control any more.

If we carefully consider this kind of languages, we may discover that they have strong qualities, such as modularity, abstraction mechanism, hierarchization of data and objects.

However, it must be stressed that at the present time, from the implementation point of view, the general approach that has been followed is the one of functional programming implementation techniques. See for instance the well-known case of SMALLTALK [14], [15]. Moreover, the implementation choices can be different and we can call object-oriented also some languages like LOGLAN and PARAGON.

4. Integration of Different Programming Styles

Many of the elements we have discussed so far go in the direction of an integration process of different programming styles. Besides, we wish to notice that the world is integrated, i.e., real applications can be decomposed into declarative programs (which would be able to produce informations by inference on a knowledge base) and algorithms (which would produce results in a more standard way). We can say in this sense that none of the available tools (functional, logic, imperative or object-oriented), all considered as "pure", is the best or unique tool.

The idea of integrating different programming styles could take several approaches. Two different possible directions are the following:

1. One is that of admitting an alternation between logic predicates (typically PROLOG predicates) and functions (typically LISP functions), mainly to increase efficiency. For example, when the resolution has to treat predicates with some variables not yet instantiated, a functional evaluation is clearly more efficient. In this line we can consider LOGLISP [27] and HLOG [28] as effective results. In these cases PROLOG programs are compiled by the LISP compiler and in this sense we can say that it is as superimposing different kinds of control on an existing logic language.

In the same context we can also consider POPLOG which is a complete system with logic and functional tools which are connected and can be alternately used.

2. Designing a completely new integrated language. In this line we have, for example, EQLOG [16]. See for the problems which are concerning the following example to solve with the EQLOG approach the cannibals and missionaries problem.

```
module NIGER using NAT, PATH = LIST[trip] is
  preds
    boat-ok: trip
    solve, safe: path
  fns
    boat: person-set  - -→ trip
    l-bank, r-bank; path  - → person-set
    mis-set, can-set: person-set  - → person-set
  vars
    Q: person-set, L: path, P, person, T: trip
  axioms
    boat-ok (boat (Q)):-length (Q) = 1;
     length (Q) = 2;
    l-bank(nil) = mis U can.
    r-bank(nil) = { }.
```

```
l-bank (cat (L, boat (Q))) = l-bank (L)-Q:-
 even(length) (L)).
r-bank (cat (L, boat (Q))) = r-bank (L) U Q:-
 even (length) (L)).
. . . . . .
. . . . . .
solve(L):- safe(L), l-bank (L) = { }.
endmod NIGER
```

In our Institute we approach the integration problem following two different directions which we intend to unify in future.

1. The first line starts from the logic approach and has the objective to extend PROLOG language by the introduction of the typing and of a more intelligent and flexible control. The purpose of this research is to increase correctness, readability and efficiency. In this line we can also refer to paper [16] and to the work which has been developed in ESPRIT-ALPES by many researchers, for example [10]. Here we have to do with a new way to approach logic programming, i.e., to see abstract data types as a central element of logic programming.

Typing the axioms, apart from the well-known advantages in respect of correctness which are typical of all strong typed programming languages, also makes it possible to reach a higher degree of efficiency. Consider, as an example, the cannibals and missionaries problem. It is very easy to understand that if we want to obtain a solution in a less number of steps, a good "trick" is to carry the maximum number of passengers, conforming to the boat capacity, going from the left to the right side of the river and the minimum, not zero, coming back. This consideration can be seen as a particular strategy for this problem, but it is not at the same logical level of problem description nor of the general strategy. If the rules of the problem are typed, the general strategy, realized by the implementation of the logic programming languages, becomes different because the unification is made only with the rules of the same type. That means that, depending on the river bank in which the boat actually is, the rules are instantiated in different ways. And it is not necessary to add new rules in the problem description to distinguish the two different situations.

To illustrate the problem of intelligent control, let us start from the following program which solves the Hanoi tower problem.

```
path (X, X),
path ([A, B, C], [R, S, T]):-
    move ([A, B], [D, E]),
    path ([D, E, C], [R, S, T]);
    move ([A, C], [D, F]),
    path ([D, B, F], [R, S, T]);
```

```
    move ([B, C], [E, F]),
    path ([A, E, F], [R, S, T]);
    back (A, B, C], [D, E, F]),
    path ([D, E, F], [R, S, T]).
move ([A, B], [C, D]): – nil(B), car(A, A1),
    rem (A, C), cons (A1, B, D).
move ([A, B], [C, D]): – car(A, A1), car(B, B1),
    less-than (A1, B1),
    rem(A, C), cons(A1, B, D).
back ([A, B, C], [D, E, F]):-
    move ([B, A], [E, D]), F = C;
    move ([C, A], [F, D]), E = B;
    move ([C, B], [F, E]); D = A.
```

This program is perfectly adherent to the way of playing with this game. The data of our problem are lists and represent the states of the computation. The recursion generates the path which is the sequence of correct moves. Actions, like nil, car, cons, etc., are performed by the inferential engine according to the given rules. However, even if this program is correct, it does not terminate, because we have not imposed any condition to check against loop. If we want to solve this problem we can follow different directions.

– To redefine the goal, specifying that a correct solution is a path which terminates:

```
solution (Probl, Path):-
    path (Probl, Path), no-loop (Path).
```

Unfortunately to use this kind of approach it is necessary to admit a coroutining mechanism, which allows to activate "no-loop", to consume the new Path just produced by "path", checking in this way for the loop.

– To shift to a lower level the loop check, by the introduction of a new data structure (a "Done" list) to store the states already visited.

```
path ([A, B, C], [R, S, T], Done):-
    move ([A, B], [D, E]),
    no-loop ([D, E, C], Done),
    . . . . . .
no-loop (T, [ ]).
no-loop (T, [T| –]) :-!, fail.
. . . . . .
```

If a state already visited is reached, "no-loop" forces the "cut" operator to stop this path. With this kind of solution we introduce between the rules of the specific problem some others which are effectively at a different level. At the same time we obtain a decrease of efficiency, because the searching

process, executed on the new data structure, is made by a user algorithm written in PROLOG, which obviously can be not the best one.

– To operate on the rules base adding new states.

```
path [(A, B, C], [R, S, T]):-
    move ([A, B], [D, E]),
    not (just-done ([D, E, C])),
    assert (just-done ([D, E, C])),
    path ([D, E, C], [R, S, T]).
```

This solution must be carefully considered, because if it can give a higher degree of efficiency, it can also have dangerous effects for what concerns backtracking.

2. Another research direction starts from the features of very high level languages, and particularly from LOGLAN. Some elements have been presented in the previous section and in [6]. Here we add only that the next step is now the introduction of clauses directly as a new type in the language. In this way we can encapsulate in a complete way the elements which are connected with the definition of the problem and which are, generally, the only ones that the user has to consider.

References

[1] J. D. Ichbiah, *Preliminary ADA Reference Manual*, SGPLAN Notices 14, 6, 1979.

[2] D. R. Barstow, H. E. Shrobe and E. Sandewall, *Interactive Programming Environments*, McGraw-Hill, 1984.

[3] R. M. Burstall, *Abstract Data Types in Functional Languages*, Edinburgh University Report, 1984.

[4] G. Cioni and A. Kreczmar, *Analysis of control structure, modularity and parallelism in ADA*, R. Istituto di Automatica N. 80–30, 1980.

[5] –, –, *Programmed deallocation without dangling reference*, IPL n. 18, 179–187, 1984.

[6] –, –, *How to solve logic programming problems in imperative language*, in preparation.

[7] W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer, 1981.

[8] H. Coelho, J. C. Cotta and L. M. Pereira, *How to solve it with PROLOG*, Laboratorio Nacional de Engenharia Civil, 3rd ed. 1982.

[9] K. L. Cooke, R. E. Bellman and J. A. Lockett, *Algorithms, Graphs and Computers*, Academic Press, 1970.

[10] J. Y. Cras, *The notion of Abstract Data Type in Logic Programming*, ESPRIT – ALPES Techn. Rep., 1984.

[11] O. J. Dahl, B. Myhrhaug and K. Nygaard, *The SIMULA 67 Common Base Language*, Publ. n. S-2, Norwegian Computing Center, Oslo 1968.

[12] E. A. Feigenbaum, *The fifth generation*, Addison-Wesley, 1983.

[13] C. Ghezzi and M. Jazayeri, *Programming Languages Concepts*, Wiley, 1982.

[14] A. Goldberg and A. Kay, *SMALLTALK-72 Instruction Manual*, Xerox Palo Alto Res. Center Rep. SSI 76-6, 1976.

[15] A. Goldberg and D. Robson, *SMALLTALK-80 the Language and its implementation. An introduction*, Springer, 1979.

[16] J. A. Goguen and J. Meseguer, *Equality, types, modules and (why not?) Generics for logic programming*, J. of Log. Progr. 2 (1984), 179–210.
[17] E. Goto and T. Soma, *Design of a LISP Machine-FLATS*, ACM 0-89791, 1982.
[18] R. Greenblatt, *The LISP Machine*, MIT Art. Int. Lab., Work, Paper 79, 1974.
[19] J. V. Guttag, *Abstract data types and the development of data structures*, Comm. ACM 20 (1977), 397–404.
[20] K. Hiraki, *Design of FLATS Machine*, Thesis, Fac. of Science, Univ. of Tokyo, 1984.
[21] F. Horowitz, *Programming Languages*, 2nd ed. Springer, 1984.
[22] R. A. Kowalski, *Predicate Logic as Programming Language*, IFIP 74, North-Holland, 1974, 569–574.
[23] –, *Algorithm = Logic + Control*, Comm. ACM 22, 7 (1979), 424–436.
[24] G. Levi, *Evolution of the Software Development Environment*, Software Engineering Applications, Capri 1980.
[25] Loglan-82, Warsaw 1982.
[26] J. McCarthy, *Recursive functions of symbolic expressions and their Computation by machine*, Part I, Comm. A.C.M. 3 (4) (1960), 184–195.
[27] R. Robinson and E. E. Sibert, *LOGLISP: an alternative to PROLOG*, in Machine Intelligence 10, Wiley, 1982, 399–414.
[28] D. Sartini, *Integrazione fra programmazione logica e funzionale: il linguaggio H-LOG*, AICA congress Rome, 1984.
[29] M. S. Sherman, *Paragon*, Lectures Notes in Comp. Science 189, 1982.
[30] D. Warren, *Implementing PROLOG*, Vol. I and II, DAI Res. Rep. 39 and 40, Edinburgh Univ., 1977.
[31] P. H. Winston and B. K. Horn, *LISP*, Addison-Wesley, 1981.

Presented to the semester
Mathematical Problems in Computation Theory
September 16–December 14, 1985