Arab Gulf J. Science. Res., **14**(1), pp. 79-100 (1996)

Knowledge Based Programming: An Experiment in Selecting a Data Type

Pavol Navrat[1]* and Viera Rozinajova[2]

[1]Department of Mathematics, Kuwait University,
P.O. Box 5969, Safat 13060, Kuwait
[2]Department of Computer Science and Engineering,
Slovak Technical University, Ilkovicova, Bratislava, 3, 812 19, Slovakia

ABSTRACT. The paper gives a short overview of the area of knowledge based programming. Research direction toward an intelligent support to software development is identified as an important aim. The paper concentrates on an original work in the area of knowledge based programming. The work reported in the paper relates to a tool to assist trainee programmers. A knowledge base on programming was built for a special area of programming expertise related to selecting a data type in the process of program formation. Properties of the knowledge base are discussed and its representation is devised. Description of one experiment is given in detail, showing user-system interaction. In the experiment, which relates to processing of data under a specific strategy, the user describes the relevant properties of the problem (such as whether the data elements will be processed more than once, whether their number can be determined in advance) and the system tries to infer what data type(s) would be appropriate to represent data with such properties. This can be regarded as an adviser to the user. The results show that the system is able to offer qualified advice. This is also an important consideration in the process of learning programming.

Viewing the historical developments in the area of computer programming, we identify a constant endeavour to incorporate the computer as a tool in the task of programming. The reason is not a desire for recursion, but a necessity to cope somehow with the complexity of the task. However, the (meta–) task of making the

*(Current address: Slovak Technical University; navrat@elf.stuba.sk)

computer assist humans in the task of writing programs appears to be extremely complex and therefore difficult. There have been taken various different approaches to it, ranging from automatic program synthesis to computer aided software engineering (part 2; for more detailed elaboration, see [Navrat 1993]). Often, methods borrowed from the field of artificial intelligence are found useful. One approach is to make use of as much available knowledge about both how to program and the respective problem domain as possible. The knowledge must be represented explicitly, however. There is much research done in the area of knowledge based programming. There are several important questions to be investigted here. As far as the knowledge itself is concerned, we can roughly distinguish problem dependent and problem independent knowledge. The latter includes the knowledge on programming. Their respective roles in the process of forming a program should be properly identified. Proper forms for their representation should be found. Ways of capturing knowledge pieces are to be sought. An architecture of the system to support the human programmer should be devised, as well as mode of their interaction. But all this would not work without building real knowledge bases that would incorporate at least part of the knowledge possessed by the experts of the particular domain.

We were interested in the form and content of the programming knowledge. Our goal was to build a base of knowledge related to certain programming procedures and to perform experiments on how such knowledge could be useful in the program formation. Our concern is also a possible use of such knowledge in learning of programming. It should be noted that such a base can be treated as a model of the programming process. We report here on an experiment involving the selection of proper data type (part 3; for more detailed elaboration of the background, see [Rozinajova and Navrat 1993, Navrat and Rozinajova 1993]).

From Automating Program Formation to Knowledge Based Programming

Since the very beginnings of the use of computers, which requires accurate and logical programming to solve given problems, an endeavour can be traced to automate that task. Frequently, attempts have been made to achieve automatic program snythesis (Burstall and Darlington 1977, Green and Barstow 1978, Manna and Waldinger 1980, Vojtek *et al.* 1986, Molnar *et al.* 1987). This is indeed an extremely challenging task. Some properties of it were investigated *e.g.*, (Molnar *et al.* 1986). On the other hand, high level language translators, editors and other tools have long ago become part of the programmer's everyday's practice, and these tools automate part of the program formation process. Another example is the syntax directed editor, *e.g.* (Navrat and Klaudiny 1983).

The motivation for automating programming or more generally the software development process is twofold: to improve quality of software and to improve quality of the process of its development. Of course, the notion of quality is understood here in quite a general sense, encompassing many dimensions: efficiency, correctness, reliability, *etc.*

The reason why it has been so often very difficult to achieve an acceptable level of quality is the great complexity of the programming task and the insufficiency of the human ability to cope with it. The vision of automatic synthesis of programs is based on an assumption that we can separate what is the problem from how it can be solved. What is to be solved is expressed in a problem specification. How it can be solved is a matter of algorithmisation.

In order to be able to synthesise programs automatically, the problem specification must be expressed in a formal way. They have been devised various different ways of expressing the problem specification. Problems have been specified as pairs of input and conditions, or pairs of input and output examples, or as sets of computation traces, *e.g.* (Biermann and Krishnaswamy 1976, Siklossy and Sikes 1975, Molnar and Navrat 1985). Appropriate methods have been proposed to synthesise programs from such specifications. The results can be described as modest at best, having in mind the high expectations and real needs of the programming practice. The reason for this situation can be identified as follows:

- high expectations imposed on the outcome of the endeavour,
- low maturity of the field,
- high complexity of the task of constructing a program,
- low utilisation of any additional knowledge on programming.

High expectations were caused may be partly be a too ambitious description of it. But soon it was realised that this would be nothing else but a shift from the task of program formation to that of specification formation. Many problems remained, appearing possible on a different level: instead of verifying the program correctness, the specification completeness and consistency has become the property to be verified.

Theory of programming as the scientific basis for any achievements in this field is very young, when measuring with standards of more traditional disciplines. It is therefore natural that it has not been able to develop until now a sufficient theoretical and methodological apparatus. Clearly, this supports our observation on the low maturity of the field.

On the other hand, the programming task is considered to be very complex. This is true for programming in the small, and it holds also for programming in the large where complex software, information, control and management systems are involved. Once again, we are faced with the recurring reason for most of our difficulties.

It can be argued that we know if fact a lot more than what has actually been formalised by the theory so far. For example, there are many programming techniques or rules which have proven to be very useful in practice but which have the nature of heuristics, not any axiomatic knowledge.

An interesting parallel was drawn between automatic program synthesis and automatic problem solving as studied in artificial intelligence, cf. (Navrat *et al.* 1988, 1989). However, it has been more frequent that conversely, results from artificial intelligence influenced attempts to automate program formation. In parallel with the developments in the area of automatic program synthesis, there has been in artificial intelligence a shift from problem solvers of a general nature to very specialised tools supporting problem solving in special and very limited domains. These so called expert systems are based on a large body of heuristic knowledge rather than on axiomatic system expressing the 'first' principles of the particular domain. Of course, this does not mean that any principles known would or should be excluded from such a knowledge base. Much more it means that provisions have been made to allow much wider scope of knowledge to participate in the problem solving process. First of all, languages for expressing domain knowledge were developed. These knowledge representation formalisms include not only production rules, but also frames, semantic networks, concepts *etc.* and allow to express procedural as well as declarative knowledge. Expert systems, when their knowledge base has been filled with knowledge acquired usually from experts from the domain, are able to support quite effectively the search for a solution of the given problem. Their role is thus to support a human in the problem solving process, and not to automate the process entirely. It is another shift in the point of view. It is also to be noted that while the expert systems are definitely one of the major application results of the work in artificial intelligence so far, at the same time they are not considered a mature methodology and much more research is needed.

Similarly to the shift of attention from automating the entire problem solving process to providing support to it as we witnessed in artificial intelligence, there has also been a shift in the program development process from automating it to providing support to the programmer. Of course, this should not be understood to imply that there has not been other research directions besides automatic program

synthesis before and that recently this research direction has been abandoned entirely. In fact, none of these are true. We are only trying to identify the shift in attention. The attention to providing support to the entire software development process is quite apparent now. Even the more traditional approaches like computer aided software engineering (CASE) seek progress in incorporating techniques of artificial intelligence, first of all those incorporating knowledge bases *e.g.*, (Aslett 1991).

As can be seen from the above, there are several directions of research all within the general endeavour to create methods and tools to support the software development process in a form of a computer assistance to humans. It is quite clear that the area is too broad to be covered in a single line of research. We shall present our specific approach devoted to making the programming knowledge and its representation explicit.

The idea of explicit representation of programming knowledge is not new. There have been several attempts to identify and formalize programming knowledge *e.g.* (Barstow 1979), (Anderson and Skwarecki 1986), (Soloway and Ehrlich 1984). But it is obvious that the knowledge base must be divided into specialized subdomains, as the manipulation is then much more flexible and this approach can be made more effective. The division of knowledge base is useful from the point of view of its creation as well as from the point of view of its using.

The Goal

From the different kinds of knowledge involved in knowledge based software development, we will concentrate here on knowledge of programming. Our goal is to build an experimental knowledge base. Its contents reflect the know-how applied by a programmer during program formation. Its form is constrained by our choice to investigate the possibility of using a commercially available knowledge representation tool, rather than a specially tailored tool *e.g.*, (Navrat *et al.* 1989, Navrat and Fric 1991, Gasparovic and Navrat 1991). We were interested in finding out to what extent we can actually proceed without interferring substantially with possible limitations of the chosen tool.

We were interested in a particular framework for our experiments. Our objective is to support a student who is learning programming. This is a particular situation where the programming knowledge is of special importance. For professional programmers, it is expected that the basic principles of programming have been fully understood and therefore it becomes much more important to understand the problem domain. It should be noted however, that the task of learning programming

has many facets and we are aware that externalisation of the relevant body of knowledge is only one of them (Navrat 1987).

The Approach

First, there is the question of choosing a suitable way of representing and processing knowledge pieces. Often, some form of production rules is chosen and indeed, for many kinds of knowledge this way of representing is quite straightforward. To some extent, this applies also for programming knowledge. However, there are situations which call for other methods of structuring. We find in the context of programming especially that the so called stereotypical situations arise *e.g.*,

- programming input of a sequential file
  - take care of its end:
    - EOF
    - specific value (sentinel)
    - explicit number of items

- programming a loop:
  - find its invariant
  - choose how to devise the halting condition.

We can make a more general observation that mixing both precedural and declarative knowledge in a common structure is desirable. The issue has been extensively studied by (Molnar 1989) who suggested that relevant knowledge should be organised in "program frames".

We have created an experimental knowledge base coding one relatively well known area of programming problems. In fact, we have coded part of the knowledge relevant to the problem of the choice of a data type.

This topic is one of the most important ones in the introductory courses of programming, independent on programming language. It is hard to be learned just by studying the textbooks, neither by mere explanation. One usually needs quite some experience to be able to select right data types.

We have also conducted several experiments using the knowledge base. The aim of the experiments was to establish how useful the approach is in supporting the students to acquire programming skills, and how closely we have modelled the respective domain.

For the former, our basic source of comparison was our own teaching experience. We have been giving various Computer Programming courses for several years at the Slovak Technical University, and recently at the Kuwait University. We are fully aware that more experiments with more extensive involvement of students will be necessary to be able to make other than the most obvious conclusions in this regard. We believe the latter aspect is more important for the time being in the sense that without having the domain knowledge model (*i.e.* source of new skills) it is hard to investigate the process of acquiring skills from that domain.

For the latter aspect, we have devised experiments which involve nontrivial decisions. We compare the solutions recommended by our system with those expected by the professional programmers.

Structured data types are often categorized as either static or dynamic:

Structured data type
- static
  - array
    - one_dimensional
    - two_dimensional
  - table
    - with_sorted_elements
    - hash
- dynamic
  - linked_list
  - graph
  - tree
    - search_tree
  - queue
  - stack

Each of these types has properties which suggest certain class of applications. In order for a programmer to choose the most suitable data type she or he (from now on, she) needs not only to know the respective data types but some experience in their use is also necessary. We shall show that this knowledge can be identified, formalized and properly represented.

The above outline of data types can be viewed also as a system of classes and subclasses in the sense of the object-oriented approach.

Using classes and subclasses coupled by inheritance in the sense of the object-oriented approach is the natural way of representing the whole system of data types used in most programming languages.

For example, when the following properties of a superclass:

– all the elements are of the same type,
– there exists a mapping from the set of ordinals to the set of elements,
– frequency of operations over more than one element will be high,

are all set to true then the system proceeds to its subclass, which in this case is static data structure and then directly to one dimensional array. This process is very straightforward. It would be possible to implement the system also without an object-oriented tool, but using it the process of reasoning is made more efficient and effective.

This approach has the advantage of offering such (meta–) properties as inheritance of represented properties within a common class, multiple inheritance and polymorphism.

In our experiments, introducing several classes proved to be appropriate. They describe the set of basic entities that we work with during a program development:

– a problem specification defined by means of input and output relations,
– processing of data within the program being developed, and
– problem solving method used in solving the specified problem.

In designing these classes we took an advantage of polymorphism: although the structure of the classes is always the same, by activating them each time other slots are active. This depends on input conditions of the session.

It is necessary to describe pieces of programming knowledge within a conceptual framework of the above entities. A system of rules appears to be the most suitable from specifically for these knowledge pieces. A rule is for example (in English paraphrase).

Rule 7:
IF 1. Use of structured data type is recommended
2. There exists mapping from the set of ordinals to the set of elements
3. All elements are of same type

4. Order of processing of elements cannot be decided in advance
THEN Use of one dimensional array is recommended

We present all the rules in English paraphrase. The whole set used at the current stage of experiments can be found in Appendix A. The rules express basically different conditions that suggest use of a particular data type. Some rules refer to properties of data (*e.g.*, rule 12), or to properties of the expected processing (*e.g.*, rule 14), or to properties of the problem solving method (*e.g.*, rule 21).

To implement our system, we have chosen the NEXPERT OBJECT tool because it offers facilities to work with both procedural and declarative representation of knowledge and provides for certain inheritance of properties. Its availability (and lack of it in case of some other tools) played its role, too.

The NEXPERT OBJECT tool allows users to develop models which can be classified as "deep worlds". It provides several distinct representation mechanisms. It allows the user to capture facts and procedural knowledge of the domain of interest in rules, and provide descriptive knowledge about the problem domain in classes and objects.

These features were useful when looking for appropriate representation of knowledge about data types. We have represented the data types outlined above as a system of classes and subclasses in the NEXPERT OBJECT system. The knowledge of an expert-programmer is represented by rules. Rules usually have a complex right hand side, composed of a hypothesis and actions. The system is able to perform both backward and forward chaining.

Objects represent the knowledge used for reasoning about by the rules. Objects describe variables in the knowledge base. Hierarchical relationships can be defined among objects to give rules greater reasoning flexibility over objects. We use objects, properties, and classes in these relationships.

A rule or a hypothesis can become relevant simply because an external event justifies its evaluation, even when the system is currently evaluating another part of the knowledge base. The tool provides inference control mechanisms that can either be set globally or incorporated into the rules themselves.

NEXPERT OBJECT is an agenda-based system. In particular, it processes events according to how they were generated rather than merely according to some fixed strategy, such as LIFO or FIFO. The system keeps a prioritized list of hypotheses to evaluate. This allows to modify dynamically a list of events with

varying priorities. It also incorporates the notions of conflict resolution, which is a decision between different possible inference paths, and nonmonotonic reasoning, which allows one to change previous conclusions which have been reached. This feature is important to allow reasoning that would respect facts supplied during a session in form of an user's answer without ending up in a inconsistency.

A subset of our experimental knowledge base (list of 23 rules) used in some of the experiments is given in Appendix A.

Experiments

We have conducted several experiments with the knowledge base. From the system user's (a student programmer) view, we have distinguished two possible situations. First, the user has no idea which data type she should use. She merely inputs the relevant data describing the problem specification. The system starts a forward reasoning process. It asks the user to supply additional information in cases where it either does not directly have it or it cannot deduce it. Second, she is able to make a guess and what she in fact wants from the system is either confirmation or rejection of her hypothesis. The system is able to provide a suggestion in case it rejects the user's hypothesis, which amounts to a qualified advice.

In the former type of experiments, we were interested in finding out how such system can model decisions made by a programmer during the program formation process. The system was provided by comparable input information as a programmer would have at that stage of the development process. The latter type of experiments was concerned with a question, how supportive in fact can the system be for a programmer who proposes a solution to a particular programming (meta–) problem, *e.g.*, how to implement representation of some structured data involved in given problem. The system is presented a hypothesis and attempts to either confirm or refute it in a process of deductive reasoning, which may involve inquiries to user that are related to problem properties. We presented some such experiments elsewhere (Navrat and Rozinajova 1993).

Here, we present other experiment not published elsewhere that aimed to select a data type for a problem which involves processing according to so called first-in first-out strategy. We give a detailed record of the user-system interaction which should give an idea about the system behaviour. Of course, there is the usual comfort including windows environment *etc.* provided.

The given problem requires processing in order described as "first in first out". Initially, following condition was set:

During processing multiset of element first-in first-out strategy will be employed

According to this, only conditions of the rules 9 and 10 become relevant:

The first condition of rule 9 is the recommendation of use of structural data, so the system starts up deductive reasoning to find out if this condition is true. To perform this, the rule 17 has to be evaluated as the first one:

Question (Q): *"Is input a multiset of elements?"*
Answer (A): *YES*
Q: *"While processing multiset: will elements be processed more than once?"*
A: *YES*
Rule 17 is set to true:
–> * *Use of structured data type is recommended**
Now both conditions of rule 10 are true, so:
Rule 9 is set to true:
–> **Use of queue is recommended**
*& Action (Ac): *Use of stack is not recommended**
Now the system proceeds to rule 10, the first condition of which is the recommendation of dynamic data structure. To prove or to reject this, again deductive reasoning is needed:
Rule 1: The first condition – the use of a structure data type (from now on abbreviated as SDT) is true.
Q: *"Can cardinality of multiset be estimated in advance?"*
A: *NO*
Rule 1 is set to true:
–> **Use of dynamic data structure is recommended**
From this comes the deduction that rule 10 is set to true, which confirms that:
–> **Use of queue is recommended**
The first appropriate data type was proposed. However, the processing goes on. The system investigates other possibilities, *e.g.*, whether queue is the only data type which suits this problem. Next rule to be processed is rule 21: The first condition – SDT is true.
Q: *"Does the problem solving method presuppose organizing multiset of elements as matrix?"*
A: *NO*
Rule 21 is set to false.
Rule 22: The first condition – SDT is true.
Q: *"Does the specification presuppose organizing multiset of elements as matrix?"*

A: *NO*
Rule 22 is set to false.
Rule 23: The first condition – SDT is true.
Q: *"Are all the elements of multiset of the same type?"*
A: *YES*
Q: *"Does the mapping from the set of ordinals to the set of elements exist?"*
A: *NO*
Rule 23 is set to false.
–> **Use of two diemensional array is not recommended**
Rule 18: The first condition SDT is true.
Q: *"Does accessing of an element depend on the contents of element?"*
A: *NO*
Rule 18 is set to false:
–> **Use of table is not recommended**
As the use of a table is one of the conditions of rules 4 and 19, these rules are now set to false:
–> **Use of hash table is not recommended**
–> **Use of table with ordered elements is not recommended**
Rule 11: The first condition – SDT is true.
Q: *"Is ordering upon the set of elements defined?"*
A: *NO*
Now four rules are set to false – No. 11,12,13,14:
–> **Use of search tree is not recommended**
Rule 8: The first condition – SDT is true.
Q: *"Does mapping from the set of ordinals to the set of elements exist?"*
A: *YES*
The third condition is true: all the elements are of same type.
Q: *"Can order of processing of elements be decided in advance?"*
A: *NO*
Rule 8 is set to true:
–> **Use of one dimensional array is recommended**
Rule 7: The first condition – SDT is true.
Q: *"Is ordering upon the set of elements defined?"*
A: *NO*
The third condition is true: all elements are of same type.

Q: *"Will the frequency of operations remove and insert element be high?"*
A: *NO*
Rule 7 is set to false.

This result does not support the recommendation of a one dimensional array. But, as we have seen, other arguments support it, so in the final report one dimensional array comes into consideration for this problem.

Rule 6: The first condition – SDT is true.

Q: *"Will order of processing be sequential most of the time?"*

A: *NO*

Rules 5 and 6 are set to false:

–> **Use of linear list is not recommended**

Rule 2: The first condition – SDT is true.

Q: *"Does the problem require an explicit representation of a binary relation?"*

A: *NO*

Rules 2 and 3 are set to false:

–> **Use of graph is not recommended**

Rule 20: As the use of a graph is one of its conditions, it is set to false:

–> **Use of tree is not recommended**

As we can see, the system has proposed two data types after investigating many possibilities:

–> *Queue* <–

and the second one is:

–> *One dimensional array* <–

This is annotated description of the system's performance which includes also user-system interaction. System recommends use of the dynamic data type queue after inductive reasoning, but "allows" use of the one dimensional array as well.

This result is in coincidence with an advice that would probably be received from an experienced programmer:

When a queue is suitable for the problem, usually an one dimensional array can be used as well. The dialogue given here is just one sample of interaction with our tool. Other working modes are pssible in the NEXPERT OBJECT tool, *e.g.* specific properties of objects or classes can be volunteered, hypotheses can be suggested or knowledge bases loaded, or one has access to working memory to investigate the values of data which is being processed. However, we find this kind of dialogue quite effective: the consultation is fast, the system communicates in a (quasi–) natural language and asks only the questions necessary for successful reasoning. Another advantage of using the tool as described in this paper is that the student can learn how to use the tool.

Results and Discussion

Results concern the experimental knowledge base that is presented here. We have shown that coding programming knowledge is possible, and can be done in a way which allows meaningful manipulation. Here, two facts perhaps should be stressed. The base is very modest in volume, and it is written in a general tool not specifically tailored to this kind of expertise. We have benefitted greatly from the advanced facilities of the tool, especially of its representation language, allowing expression of class-like relationships.

The experience of using non-dedicated tool was an interesting one: we found the tool appropriate. However, we have detected also some limitations of using this tool: the representation of programming knowledge by a system of classes and subclasses is not always suficiently rich. The representation does not reach the level known from typical frame-based knowledge representation languages used in artificial intelligence (Bobrow and Winograd 1977). Seen from another perspective, we must note that working with more complex expertise would require different language structures and modes of reasoning. In particular, incorporating problem specifications and problem solving methods requires that alternative kinds of representation would have to be sought. Programming schemes, for example, would probably require defining specific unification algorithms in order to be represented appropriately. An excellent proposal was made in this direction by (Rich and Feldman 1992). Even more, a programming method probably defines its own strategies when and how such schemes are to be combined and used. To represent such a meta-level knowledge requires more flexibility in the representation language itself.

Our interest has not been so much to verify the very fact that building a knowledge base is possible, as there have been already presented reports on similar projects *e.g.* (Barstow 1979) which however do not take advantage of using classes in representing knowledge. Other works were interested in programming knowledge in the context of learning programming *e.g.,* (Anderson and Skwarecki 1986, Soloway and Ehrlich 1984). Partially, this is also our interest. We feel encouraged by our experiments and conclude that developing knowledge bases can aid in learning programming. Here, the approach can be from two sides. Either a teacher provides a knowledge base to the learner who learns from such externalisation. Or the learner builds a knowledge base which helps her organise and clarify the ideas about the subject matter. We find the former aspect important especially in such domains as programming, where studying an axiamatically built theory of programming is still not the usual procedure for learning programming, and obviously for good reasons, but where the need for more formal knowledge is recognised as urgent. The latter aspect is presented in *e.g.,* (Webb 1992).

In the future work we plan that the system would be a part of working environment which would help the student also in initial stages of program development – *i.e.* analysis and specification (CASE tool should be very useful in this concern). The system would maintain more information about the program being developed by the student. The presented version of the system is an experimental one. There were performed limited experiments, but till now it has not been used regularly in programming courses. However, it has shown already in the testing phase to be a valuable tool not only in advising students but also forcing them to think more deeply about this programming domain.

From a more general perspective, use of explicitly formulated knowledge is found an important direction in software engineering. In intelligent support of software development, attempt is made to provide assistance to professional programmers *e.g.*, (Aslett 1991, Fisher *et al.* 1992).

Acknowledgements

We should like to thank an anonymous referee for numerous constructive comments on an earlier draft of the paper.

Appendix A.

Experimental knowledge base (set of rules) written in natural language

Rule 1:
IF 1. Use of structured data type is recommended
2. Cardinality of multiset cannot be estimated in advance
THEN Use of dynamic data structure is recommended

Rule 2:
IF 1. Use of dynamic data structure is recommended
2. The problem requires explicit representation of binary relation
3. The multiset of elements is set of elements
THEN Use of graph is recommended

Rule 3:
IF 1. Use of structured data type is recommended
2. The problem requires explicit representation of binary relation
3. The multiset of elements is set of elements
THEN Use of graph is recommended

Rule 4:
IF 1. Use of table is recommended
2. Complexity consideration tells us that access time should not be dependent on size of the set of elements
THEN Use of hash table is recommended

Rule 5:
IF 1. Use of structured data type is recommended
2. It can be assumed that the order of processing of elements will be sequential most of the time
THEN Use of linear list id recommended

Rule 6:
IF 1. Use of dynamic data structure is recommended
2. It can be assumed that the order of processing of elements will be sequential most of the time
THEN Use of linear list id recommended

Rule 7:
IF 1. Use of structured data type is recommended
2. There exists mapping from the set of ordinals to the set of elements
3. All elements are of same type
4. Order of processing of elements cannot be decided in advance
THEN Use of one dimensional array is recommended

Rule 8:
IF 1. Use of structured data type is recommended
2. Ordering upon multiset of elements is not defined
3. All elements are of same type
4. Frequency of operations over more than one element is high
5. There exists mapping from the set of ordinals to the set of elements
THEN Use of one dimensional array is recommended

Rule 9:
IF 1. Use of structured data type is recommended
2. During processing multiset first-in-first-out strategy will be employed
THEN Use of queue is recommended

Rule 10:
IF 1. Use of dynamic data structure is recommended
2. During processing multiset first-in-first-out strategy will be employed
THEN Use of queue is recommended

Rule 11:
IF 1. Use of structured data type is recommended
2. Ordering upon set of elements is defined
3. This ordering is referred to in problem specification
4. Multiset of elements is set of elements
THEN Use of search tree is recommended

Rule 12:
IF 1. Use of dynamic data structure is recommended
2. Ordering upon set of elements is defined
3. This ordering is referred to in problem specification
4. Multiset of elements is set of elements
THEN Use of search tree is recommended

Rule 13:
IF 1. Use of structured data type is recommended
2. Ordering upon set of elements is defined
3. Frequency of operations remove element and insert element is high
4. Multiset of elements is set of elements
THEN Use of search tree is recommended

Rule 14:
IF 1. Use of structured data type is recommended
2. Ordering upon set of elements is defined
3. Frequency of operations remove element and insert element is high
4. Multiset of elements is set of elements
THEN Use of search tree is recommended

Rule 15:
IF 1. Use of structured data type is recommended
2. During processing of elements last-in-first-out strategy will be employed
THEN Use of stack is recommended

Rule 16:
IF 1. Use of dynamic data type is recommended
2. During processing of elements last-in-first-out strategy will be employed
THEN Use of stack is recommended

Rule 17:
IF 1. Input is multiset of elements
2. It cannot be ruled out that while processing the multiset, elements will be processed more than once
THEN Use of structured data type is recommended

Rule 18:
IF 1. Use of structured data type is recommended
2. Accessing an element depends on the contents of this element
THEN Use of table is recommended

Rule 19:
IF 1. Use of table is recommended
2. Ordering upon set of elements is defined
3. Optimization of the accessing of element is required
THEN Use of table with ordered elements is recommended

Rule 20:

IF 1. Use of graph is recommended
2. For the relation predecessor successor defined on the set the inverse relation is one-one
3. Only one element exists, for which the relation predecessor successor is not defined (root)

THEN Use of tree is recommended

Rule 21:

IF 1. Use of structured data type is recommended
2. Problem solving method presupposes organizing multiset of elements as matrix
3. All elements are of same type
4. Matrix is not sparse matrix

THEN Use of two dimensional array is recommended

Rule 22:

IF 1. Use of structured data type is recommended
2. Specification presupposes organising multiset of elements as matrix
3. All elements are of same type
4. Matrix is not sparse matrix

THEN Use of two dimensional array is recommended

Rule 23:

IF 1. Use of structured data type is recommended
2. All elements are of same type
3. There exists mapping from the pair of ordinals to the set of elements
4. This mapping will be used for accessing an element

THEN Use of two dimensional array is recommended

References

Anderson, J.R. and **Skwarecki, E.** (1986) The Automated Tutoring of Introductory Computer Programming. *Communications of the ACM,* **29**(9): 842-849.

Aslett, M.J. (1991) *A Knowledge Based Approach to Software Development.* North Holland Publishing Co., Amsterdam, The Netherlands, 249 p.

Barstow, D.R. (1979) An Experiment in Knowledge-Based Automatic Programming, *Artificial Intelligence,* **12:** 73-119.

Biermann, A.W. and **Krishnaswamy, R.** (1976) Constructing Programs from Example Computations. *IEEE Transactions on Computers,* **C-24**(2):141-153.

Bobrow, D.G. and **Winograd, T.** (1977) An Overview of KRL, a Knowledge Representation Language. *Cognitive Science,* **1**(1): 3-46.

Burstall, R.M. and **Darlington, J.** (1977) A Transformation System for Developing Recursive Programs. *Journal of the ACM,* **24**(1): 44-67.

Fisher, G. , Girgensohn, A., Nakakoji, K. and **Redmiles, D.** (1992) Supporting Software Designers with Integrated Domain Oriented Design Environments. *IEEE Transactions on Software Engineering,* **18**(6): 511-522.

Gasparovic, L. and **Navrat, P.** (1991) Extalk - Smalltalk Based Expert Systems Development Tool. *In:* **Mrazik, A. (edt.).** Proc. East EurOOPe'91, Short Papers, Bratislava, 65-73 pp.

Green, C. and **Barstow, D.R.** (1978) On Program Synthesis Knowledge. *Artificial Intelligence,* **10:** 241-279.

Manna, Z. and **Waldinger, R.** (1980) A Deductive Approch to Program Synthesis. *ACM Trans. on Programming Languages and Systems,* **2**(1): 90-121.

Molnar, L. and **Navrat, P.** (1985) Automation of Program Creation and Methodology of Programming. (In Slovak). *The Journal of Eectrical Engineering,* **36**(4): 316-323.

Molnar, L., Navrat, P. and **Vojtek, V.** (1986) Heuristic Search with Global and Local Heuristics. *Computers and Artificial Intelligence,* **5**(5): 417-426.

Molnar, L., Navrat, P. and **Vojtek, V.** (1987) A System for Automatic Implementation of Abstract Data Types. *Computers and Artificial Intelligence,* **6**(5): 481-488.

Molnar, L. (1989) A Knowledge Based Program Creation. DrSc. Dissertation. Slovak Technical University, Bratislava.

Navrat, P. and **Klaudiny, M.** (1983) A Syntax Directed Editor of Pascal Programs. (In Slovak). In: Proc. Modern Programming 1983, Part 2, Zilina, 125-136 pp.

Navrat, P. (1987) Towards a Master Programmer: A Paradigm for Automated Tutoring of Programming. *In:* **Plander, I. (edt.).** Proc. Artificial Intelligence and Information Control Systems of Robots 87, North-Holland, Amsterdam, 375-379 pp.

Navrat, P., Molnar, L. and **Vojtek, V.** (1988) Using Automatic Program Synthesizer to Generate Solutions from Diverse Problem Environments. *Computers and Artificial Intelligence,* **7**(2): 139-146.

Navrat, P. and **Mlada, I.** (1989) What Knowledge is the Knowledge Based Programming Based on?": An Inquiry into Knowledge Sources. *In:* **Plander, I. (edt.).** Proc. Artificial Intelligence and Information - Control Systems of Robots 89, North-Holland, Amsterdam, 187-190 pp.

Navrat, P., Fric, P., Adamy, M. and **Mlada, I.** (1989) KEX: Computer Aided Knowledge Engineering System. *In:* Proc. Computers '89 Conference, Blahova, 156-162 pp.

Navrat, P., Molnar, L. and **Vojtek, V.** (1989) Using Automatic Program Synthesizer as a Problem Solver: Some Interesting Experiments. *In*: **Davenport, J. (edt.).** Proc. EUROCAL '87, LNCS 378, Springer, Berlin, 412-423 pp.

Navrat, P. and **Fric, P.** (1991) A Tool for Knowledge-Based Systems Development. I*n:* **Marik, V. (edt.)**. Proc. Artificial Intelligence Applications Conference AI '91, Prague, 351-360 pp.

Navrat, P. and **Rozinajova, V.** (1993) Making Programming Knowledge Explicit. Computers and Education, **21**(4): 281-299.

Navrat, P. and **Rozinajova, V.** (1983) Experiment in Knowledge Based Programming. Computer and Information Sciences - *Journal of King Saud University.*

Rich, C. and **Feldman, Y.A.** (1992) Seven Layers of Knowledge Representation and Reasoning in Support of Software Development. *IEEE Transactions on Software Engineering,* **18**(6): 451-469.

Rozinajova, V. and **Navrat, P.** (1993) Explicit Knowledge Representation in Support of Learning Programming. *In:* **Brna, P., Ohlsson, S., Pain, H. (edt.).** Proc. AI-ED '93 World Conference on Artificial Intelligence in Education (Edinburgh), Association for the Advancement of Computing in Education, Charlottesville, 584 p.

Siklossy, L. and **Sykes, D.A.** (1975) Automatic Program Synthesis from Example Problems. *In:* Advance Papers IJCAI4, Tbilisi, 268-273 pp.

Soloway, E. and **Ehrlich, K.** (1984) Empirical Studies of programming knowledge. *IEEE Transactions on Software Engineering,* **10**(5): 595-609.

Vojtek, V., Molnar, L. and **Navrat, P.** (1986) Automatic Program Syntheis Using Heuristics and Interaction. *Computers and Artificial Intelligence,* **5**(5): 427-442.

Webb, M. (1992) Learning by Building Rule - Based Models. *Computers and Education,* **18**(1-3): 89-100.

(Received 28/09/1993;
in revised form 11/07/1995)

البرمجة المبنية على المعرفة : تجربة في اختيار نوع البيانات

بافول نافرات[1]* و فييرا روزيناجوفا[2]

[1] *قسم الرياضيات - جامعة الكويت - ص .ب (٥٩٦٩)*
الصفاة ١٣٠٦٠ - الكويت
[2] *قسم الحاسب الآلي والهندسة - جامعة سلوفاك التقنية - إلكوفيكوفا ٣*
براتيسلافا - ٨١٢١٩ - سلوفاكيا

يقدم هذا البحث باقتضاب نظرة شاملة عن مجال البرمجة المبنية على المعرفة . إن البحث الموجه نحو تدعيم ذكي لتطوير البرمجة يعتبر هدفا هاماً . يركز هذا البحث على تقديم عمل مبتكر في مجال البرمجة المبنية على المعرفة . يرتبط العمل المقدم في هذا البحث بتقديم أداة لمساعدة المبرمجين المتدربين . لقد تم بناء قاعدة معرفة خاصة بالمهارة البرمجية التي تتعلق باختيار نوع من البيانات في عملية تكوين البرنامج . كما نوقشت خواص قاعدة المعرفة واستنبط تمثيل لها . ولقد أعطى وصف مفصل لتجربة واحدة مبيناً التفاعل بين المستخدم والنظام . يقدم المستخدم في التجربة التي تتعلق بمعالجة البيانات ضمن استراتيجية معينة وصفا للخواص التي هي وثيقة الصلة بالمسألة (مثل فيما إذا كانت البيانات ستعالج أكثر من مرة واحدة وفيما إذا كان يمكن تحديد عدد تلك العناصر مقدماً) ، ويحاول النظام أن يستدل على نوع (أنواع) البيانات الملائم لتمثيل البيانات التي لها تلك الخواص . يمكن اعتبار هذا كمرشد للمستخدم . تبين النتائج أن النظام قادر على تقديم نصائح قيمة . وإن لهذا أيضاً اعتبارات هامة في عملية تعلم البرمجة .

* العنوان الدائم : جامعة سلوفاك التقنية - الكوفيكوفا ٣ - براتيسلافا-٨١٢١٩