KNOWLEDGE-BASED PROGRAMMING

Enn Tyugu

Knowledge-Based Programming

Enn Tyugu

Institute of Cybernetics
Estonian Academy of Sciences
Tallinn, USSR

TURING INSTITUTE PRESS
in association with

ADDISON-WESLEY PUBLISHING COMPANY
Wokingham, England · Reading, Massachusetts
Menlo Park, California · New York · Don Mills, Ontario
Amsterdam · Bonn · Sydney · Singapore · Tokyo
Madrid · Bogota · Santiago · San Juan

This book is based on Professor Tyugu's work *Kontseptualnoe programmirovanie*, originally published in the USSR by Nauka in 1984.

Cover design by Crayon Design, Henley-on-Thames.
Typeset by Advanced Filmsetters (Glasgow) Limited.
Printed in Great Britain by The Bath Press, Avon.
First printed 1987.

British Library Cataloguing in Publication Data

Tyugu, E.
Knowledge-based programming.—(Turing Institute Press Knowledge engineering tutorial series).
1. Computer software—Development
2. Artificial intelligence
I. Title
005.1'2 QA76.76.D47

ISBN 0-201-17815-X

Library of Congress Cataloging in Publication Data

Tyugu, Ė. Kh. (Ėnn Kharal 'dovich), 1935–
[Kontseptual 'noe programmirovanie. English]
Knowledge-based programming/E. Tyugu.
p. cm.—(Turing Institute Press knowledge engineering tutorial series)
Translation of: Kontseptual 'noe programmirovanie.
Bibliography: p.
Includes index.
ISBN 0-201-17815-X
1. Artificial intelligence—Data processing. 2. Programming (Electronic computers) I. Title. II. Series.
Q336.T9813 1988
006.3—dc19 87-19389
CIP

Knowledge-Based Programming

Turing Institute Press Knowledge Engineering Tutorial Series

Managing Editor Dr Judith Richards
Academic Editor Dr Peter Mowforth

The Turing Institute, located in Glasgow, Scotland, was established in 1983 as a not-for-profit company, named in honour of the late Alan M. Turing, the distinguished British mathematician and logician whose work has had a lasting influence on the foundations of modern computing.

The Institute offers integrated research and teaching programmes in advanced intelligent technologies – in particular, logic programming, computer vision, robotics and expert systems. It derives its income from research and training contracts, both governmental and industrial, and by subscription from its Industrial Affiliates. It assists Affiliates with the transfer of technology from research to application, and provides them with training for their technical staff, a wide range of software tools and a comprehensive library and information service.

The Turing Institute is an Academic Associate of the University of Strathclyde, and its research staff work closely with different departments of the University on a variety of research programmes.

Other titles published in association with the Turing Institute Press

Applications of Expert Systems
J. Ross Quinlan (Editor)

Structured Induction in Expert Systems
Alen D. Shapiro

Preface

This book is intended for readers who are interested in applying artificial intelligence to programming practice. It shows how a computer can be used even at the problem-specification phase of programming. This approach is called **knowledge-based programming**.

The following features of knowledge-based programming are the most important for us:

- using a knowledge base for accumulating useful concepts;
- programming in terms of a problem domain;
- using the computer in the whole problem-solving process beginning with the description of a problem;
- synthesizing programs automatically.

In order to distinguish our approach to problem solving from the approaches to program development in which knowledge is used for adapting algorithms to particular computers and for selecting data structures suitable for computations, we shall use a more specific term, **conceptual programming**, for our approach, because it extensively uses concepts as pieces of knowledge.

In conceptual programming we define concepts for a computer and then use them to describe the problems which are to be solved on the computer. We also expect the computer to construct automatically the programs for solving problems, using the knowledge we have given to it in the specifications.

It is common knowledge that whereas computer performance has increased about 1000 times, that of programmers has increased only 10 times. This difference in performance cannot be sufficiently improved, either by making software distribution easier, or by increasing the number of people involved in computer programming. There are two quite promising ways of increasing the productivity of programming:

- the development of intelligent software packages;
- the implementation of very high-level languages.

Conceptual programming is a combination of these ways which enables a user to build his own intelligent software capable of understanding the language of the user.

Conceptual programming is a way of using the computer as an intelligent partner for problem solving. It includes:

1. specifying new concepts to the computer;
2. representing problems to it in terms of these concepts.

The aim of this book is to introduce conceptual programming to the reader and to teach him how to use the computer as a partner which can understand to a certain degree the problem to be solved. This degree of understanding is determined by two factors:

- the available deductive mechanism;
- the amount of knowledge possessed by the computer.

The user must be aware of the computer's degree of understanding when he specifies problems. This is a difficulty of conceptual programming. Common experience of programming is of no help here. A FORTRAN programmer can be sure that the computer could execute almost any syntactically correct FORTRAN program. In our case the syntactic correctness of a specification of a problem is not sufficient. A problem may be unsolvable because the knowledge included in the specification is incomplete. In this case the specification either of the concepts intended to represent the problem or of the problem itself must be extended. This is a completely new technique of program development, and it is discussed throughout this book and illustrated with a considerable number of examples.

To read this book no deep knowledge of programming or of artificial intelligence is needed. A reader must have a general acquaintance with computers as well as with some programming language. Some knowledge of logic is needed, but only for Section 1.3, in which the logical basis of structural synthesis is explained.

The first chapter contains a discussion of a formal representation of problems, which is followed by a description of automatic program synthesis methods for solving problems. Knowledge representation for program synthesis is considered in the second chapter. A reader not interested in program synthesis can skip the first two chapters, except Section 1.1, where the problems are discussed. However, in this case he must just believe that the problems given as examples in the book can be solved automatically.

The third chapter is a description of a language for conceptual programming, and it must be read thoroughly in order to understand the following chapters, where the conceptual programming technique is presented.

The author is convinced that one can only learn to solve problems by practice in solving them. This book therefore contains a considerable number

of examples from various problem domains and special attention is paid to the specification of useful concepts.

The fourth chapter contains specifications of many concepts from geometry and physics which are taught to children in school. It is hoped that this kind of knowledge will soon be a common part of the knowledge base of every computer. A technique for building large intelligent software systems is presented in the fifth chapter, and in the last chapter this technique is applied to specify concepts in several different domains.

Enn Tyugu
Tallinn, USSR

Contents

Chapter 1

Problems and Programs

Programming can be considered to be the first stage of problem solving, in which a plan of action for solving a problem is developed. Given this view, the process of transcribing a program into a programming language acceptable to a computer is only the last and the simplest task of programming. We may draw an analogy between a programmer and a production engineer who designs a tooling process for a machine part. Both of them must take into account the available facilities and make a reasonable plan to produce a result. The result considered by the production engineer is a machine part, usually specified by a drawing. For a programmer, the result is described by a problem specification. We start this book with a discussion of problems and present a precise definition of computational problems, that is, of those problems the book is intended to deal with.

1.1 Representation of problems

According to the point of view taken in this book, programming starts from a problem and not from an algorithm or a flow chart. Therefore, we must define the notion of problem as precisely as possible. It is a difficult task because the world where the problems arise is extremely diverse, and it may even be impossible to find a universal form applicable to all problems. We shall start with computational problems, and later show that a large number of problems can be represented in a similar way.

1.1.1 COMPUTATIONAL PROBLEMS

Computational problems contain variables and the variables are denoted by identifiers, for instance, *AX*, *x1*, *x*, *AREA*. We assume that values of variables are data – in particular, pieces of text and numbers. The values may vary both in form and in meaning, i.e. they may be of various types. A more detailed

discussion of variables, values and data types will be presented in the second chapter. We shall represent a computational problem in the following form:

compute $y1,\ldots,yn$ **from** $x1,\ldots,xm$ **knowing** M.

The identifiers $x1,\ldots,xm$, $y1,\ldots,yn$, M are variables. The words **compute, from** and **knowing** have predefined meanings which must be understood by a problem solver. They show that this is a problem statement and they separate the different kinds of variables from each other:

- input variables: $x1,\ldots,xm$;
- output variables: $y1,\ldots,yn$;
- variable M, the value of which represents the problem conditions.

There are no explicit associations between the problem conditions denoted by M and input and output variables of the problem. Nevertheless, we assume that the problem conditions contain all necessary specifications for the input and output variables. In particular, we assume that problem conditions determine the set of variables from which the input or output variables may be drawn.

The data which constitutes the value of M represents the knowledge needed for solving the problem. Hence, the whole complexity of a problem description is hidden in the value of M. New results in artificial intelligence, especially in knowledge representation, can be used for encoding the knowledge needed for problem solving and this will be discussed in the second chapter.

Let us consider some examples of computational problems.

1. Compute the area of a triangle from its sides – this problem can be represented using variables S,a,b,c to denote the area and sides of a triangle:

   compute S **from** a,b,c **knowing** *triangle*.

   It is important to remember that the concept of a triangle must be specified for the solver before this problem can be solved, and it must contain just the same variables S,a,b,c for the area and sides.

2. Construct a proof of a formula in a theory: this problem is represented by problem conditions:

   compute *proof* **from** *formula* **knowing** *theory*.

3. Find the names of all young employees of an institute knowing the staff of the institute and the conditions which determine a young employee. In this case we must take care over the form of output of the problem.

We specify a variable which we shall call

names-of-young-employees.

(This is also an identifier.) This variable will have a set of names of all young employees as the value. Denoting the problem conditions as *staff* we can write the problem as follows:

compute *names-of-young-employees* **knowing** *staff.*

This problem statement does not contain input variables – all the knowledge needed for solving is hidden in problem conditions called *staff.*

Let us introduce a shorter form of a problem representation:

$(M \vdash x1, \ldots, xm \rightarrow y1, \ldots, yn)$

where the variables $M, x1, \ldots, xm, y1, \ldots, yn$ have the same meaning as before.

Let us denote the fact that a problem is solvable by writing

solvable $(M \vdash x1, \ldots, xm \rightarrow y1, \ldots, yn)$.

Actually, this is a predicate which has three arguments:

- problem conditions M;
- list of input variables $(x1, \ldots, xm)$;
- list of output variables $(y1, \ldots, yn)$.

It is true if and only if the value of M is such that on its basis a program can be constructed which solves the problem.

Computational problems with identical problem conditions can be compared with each other and some conclusions about their complexity can be drawn. Let $in(P)$ denote a set of input variables and $out(P)$ denote a set of output variables of a computational problem P. We say that a problem $P1$ is less than $P2$ ($P1 < P2$) if and only if $in(P1) = in(P2)$, $out(P1) \subset out(P2)$ and both problems have identical problem conditions. This ordering is reasonable. Indeed, if the problem $P2$ can be solved, then $P1$ can be solved also because, due to $out(P1) \subset out(P2)$, the solution of $P2$ contains the solution of $P1$.

There is also another ordering of problems – $P1 \ll P2$ if and only if $in(P2) \subseteq in(P1)$ and $out(P1) \subseteq out(P2)$, where one of the inclusions is strict and the problems have identical problem conditions. In this case the correspondence between solvability and ordering is not so straightforward. Sometimes a problem with a larger set of input variables may be more difficult to solve due to the redundancy and contradictions of input variables.

1.1.2 OTHER PROBLEMS

Not all computational problems fit directly into the form specified above. For instance, the problem:

compute *everything-that-can-be-computed* **from** $x1,\ldots,xm$ **knowing** M

is different. In order to formalize this problem in our way, we must introduce a new variable – *everything-that-can-be-computed.*

The value of this variable is a set of computable variables with their computed values, i.e. a set of pairs of the form

⟨variable name⟩ = ⟨value⟩.

If we are interested only in a solution of a problem for particular values $v1,\ldots,vm$ of the input variables $x1,\ldots,xm$ and not in a program for computing values of $y1,\ldots,yn$ from any given values of $x1,\ldots,xm$, then we can include the following equations among the problem conditions:

$$\begin{aligned} x1 &= v1,\\ &\vdots\\ xm &= vm. \end{aligned}$$

Then the problem representation will not contain input variables, being simply

compute $y1,\ldots,yn$ **knowing** M,

or in the abbreviated form:

$(M\vdash y1,\ldots,yn)$.

Introducing new variables can be useful in various cases when something must be computed. One more example of a problem of that kind is

compute *all-that-is-needed* **from** $x1,\ldots,xm$ **knowing** M.

all-that-is-needed must be specified by the value of M. In particular, all changes caused by new values of $x1,\ldots,xm$ may be asked for. (This particular problem is rather difficult to solve.)

There are problems of a quite different nature – the problems in which some real-world activities are needed. For example, to bring a small green box from the leftmost room of a building. Generally speaking, these are not problems for a computer, but a computer can be used to plan the solution of such problems. The plan which is needed can be designated by a variable, and

this will be an output variable of the problem for the computer. However, in this case some difficulties may arise, because the problem conditions may change during the course of action. In other words, the problem conditions are not sufficient to form the whole plan. In this case planning and real-world activities can be performed alternately and repetitively. Planning for a single stage of activities is done with the assumption that the environment does not change, and consequently the problem conditions are fixed for that stage.

In this section we have shown that many essentially different problems can be represented in the form

$$(M \vdash x1, \ldots, xm \rightarrow y1, \ldots, yn).$$

We discussed some modifications of this problem (omitting input variables and asking for all that can be computed) and proposed a method of introducing a new variable to denote the desired result. The main restriction on the use of this problem representation is the requirement that problem conditions must be completely specified before the solution planning begins.

We have barely touched here on the problem conditions M. (We shall also call this problem specification.) The remaining part of this chapter and the second chapter are devoted mainly to the question of how to represent and use knowledge about problems. This subject concerns just the problem conditions M.

1.2 Logical foundations of program synthesis

There are three different approaches to program synthesis:

- **deductive synthesis**, which uses automatic deduction of a proof of solvability of a problem and derives a program from the proof;
- **inductive synthesis**, where a program is built on the basis of examples of input–output pairs or examples of computations;
- **transformational synthesis**, where a program is derived stepwise from a specification by means of transformations.

In this book we shall use deductive synthesis of programs. In this case the way from a problem to a program is as follows:

problem in source form →
problem represented in a formal theory →
proof of solvability of the problem →
program.

The first step – transforming a problem into a language of a formal theory – can be done by means of a translation technique. The most complicated step is the construction of a solvability proof of the problem. In order to discuss