Part 2

Language Design and Acquisition of Programming

One important consequence of the increasing number of programming languages and the evolution of program design environments is the number of new issues they raise. As developed in the first part of this volume this evolution is due both to the necessity to solve new classes of problems and to have more powerful tools to support the programming activity. However, this evolution does not necessarily make the programming activity more easy to practice and to learn. Are some languages really better than others? If so, what for? What kind of languages are better for learning and/or for educational purposes? What kind of support environment is really efficient for professionals?

This section examines programming languages, their design, their use, their structural and semantic properties and their learning. It raises what are hoped to be relevant issues for language designers and educators through presentation of data on the ways in which experts and novices behave in programming. What is common to both is that no one programming language (or a programming environment) is a panacea. Rather they are tools that facilitate certain tasks by supporting expert and novice programming behaviour in an appropriate manner, while in some cases they may constitute an obstacle to efficient planning activities.

The first two chapters deal with language design. In Chapter 2.1 Petre discusses the distance separating language designers' and expert programmers' criteria of what constitutes a 'good' programming language. Key features differentiating them involves the interpretation of abstraction: designers prefer high-level expressions, whereas the experts feel the need to choose their level of abstraction and to be able to manipulate hardware when necessary. Designers tend to emphasize 'well-foundedness' and correctness, whereas expert programmers stress utility, control and efficiency. Chapter 2.2 by Green concentrates on the detail of language design and describes how notational aspects influence the process of programming. A program is viewed as an information structure. The programmer is the user of this structure who is called upon to accomplish different tasks: obtain information from a program, add new information, re-organize it, debug it, modify it, etc. The main argument is that the structure of information should match the structure of the task. A set of cognitive dimensions of notation is developed in order to analyse how a combination of a notation and the environment in which it is used affect usability. There is no universally good notation system, only adequate notations for specific tasks. These dimensions can be seen as initial guidelines to cognitive requirements in language design.

The next two chapters explore learning issues. The relationship between task structure and programming language semantics is discussed as a critical component of programming learning in Chapter 2.3 by Hoc and Nguyen-Xuan. Novices must learn not only new notations and new means of expression but also the operating rules of the processing device that underlies the language. It is argued that learning by doing and learning by analogy are privileged mechanisms of acquisition of these rules. The notion of the representation and processing system (RPS) is developed in order to identify and analyse the gap between novices' existing RPSs and those they have to construct and to suggest the training situations that facilitate the development of RPSs. The acquisition of operating rules is only the first step in programming learning: novices have to also learn programming concepts and structures. Chapter 2.4 by Rogalski and Samurçay examines the cognitive difficulties encountered by novices in learning the concepts and structures commonly presented in introductory

programming classes. It is shown how dynamic mental models related to action execution act as precursors by playing both productive and reductionist roles, and become an obstacle when more static representations are needed. It is argued that training paradigms tend to place emphasis on the computational and procedural aspects of programming which prevents novices from learning problem modelling and programming as a function specification.

Chapter 2.5 by Mendelsohn, Green and Brna discusses issues related to the use of computers for general educational purposes. After examining the 'transfer of competence' and 'acquisition of new knowledge' hypotheses, the authors put forward an alternative proposition: the new representation and processing system that people acquire via programming may modify their strategy in analysing the objects on which they have programmed.

All the chapters in this part analyse current issues in programming learning and the cognitive requirements of programming activity in language design. The study of learning mechanisms is faced with unresolved theoretical and methodological problems. Observation of learning activities is generally too short to assess the real characteristics of activity—with obvious consequences on validity (the importance and duration of errors, etc.).

A number of questions still remain open. Although we know that individual differences are important factors affecting the learning process and performance in the use of specific programming languages and environments, very few concepts have been defined to characterize these differences systematically. Little work has been done as well on the retraining of professional programmers. How quickly and thoroughly can they acquire a new language or a design environment? How do previously acquired representations and procedures affect ease of acquisition of new knowledge?