McGraw-Hill
PERSONAL COMPUTER PROGRAMMING ENCYCLOPEDIA

LANGUAGES AND OPERATING SYSTEMS

Second Edition

William J. Birnes, Editor

McGraw-Hill PERSONAL COMPUTER PROGRAMMING ENCYCLOPEDIA

LANGUAGES AND OPERATING SYSTEMS

Second Edition

William J. Birnes, Editor

William P. Woodall, Technical Editor

Nancy Hayfield, Production Editor

McGraw-Hill Publishing Company

New York St. Louis San Francisco Auckland Bogotá
Caracas Colorado Springs Hamburg Lisbon
London Madrid Mexico Milan Montreal
New Delhi Oklahoma City Panama Paris
San Juan São Paulo Singapore
Sydney Tokyo Toronto

The following language and operating systems names appearing in this book are trademarks, with the owner's name given in parentheses. Failure to include any name here should not be construed as pertinent to its possible trademark status. • Ada (Department of Defense) • APL (IBM Corp.) • Microsoft BASIC (Microsoft Corp.) • ZBASIC (Zenith Corp. and Microsoft Corp.) • Compiled BASIC (Digital Research Inc.) • S-BASIC (Topaz Inc.) • Applesoft BASIC (Apple Computer Inc. and Microsoft Corp.) • Atari BASIC (Atari Inc.) • TI Extended BASIC (Texas Instruments Inc.) • Cobol (Codasyl Inc.) • COMAL (COMAL Users Group) • Forth (Forth Interest Group) • PC Forth (Forth Interest Group) • Fortran (IBM Corp.) • HyperCard and HyperTalk (Apple Computer) • Terrapin Logo (Krell Corp.) • RPG (IBM Corp.) • Smalltalk (Xerox Corp.) • dBASEII (Ashton-Tate Inc.) • VisiCalc (VisiCorp) • SuperCalc (Sorcim) • Multiplan (Microsoft Corp.) • Lotus 1-2-3 (Lotus Development Corp.) • Paradox (Ansa, Borland) • Postscript (Adobe Systems) • Symphony (Lotus Development Corp.) • Framework (Ashton-Tate Inc.) • UNIX (AT&T Bell Laboratories) • MS-DOS/MS-DOS 3.0 (Microsoft Corp.) • PC-DOS (IBM Corp. and Microsoft Corp.) • Z-DOS (Microsoft Corp.) • Commodore DOS (Commodore Computers Inc.) • Applesoft DOS 3.3 (Apple Computer Inc.) • ProDOS (Apple Computer Inc.) • TRS-DOS (Tandy Corp. and Logical Systems Inc.) • Macintosh Operating System (Apple Computer Inc.)

Library of Congress Cataloging-in-Publication Data

McGraw-Hill personal computer programming encyclopedia

Bibliography: p
Includes index.
1. Microcomputers—Programming. 2. Programming languages (Electronic computers) 3. Operating systems (Computers)
I. Birnes, William J. II. McGraw-Hill Book Company.
QA76.6.M414 1989 005.26 88-8410
ISBN 0-07-005393-6

1234567890 DOC/DOC 8954321098

ISBN 0-07-005393-6

Jacket design by Edward J. Fox

Printed and bound by R. R. Donnelley and Sons

For more information about other McGraw-Hill materials, call 1-800-2-MCGRAW in the United States. In other countries, call your nearest McGraw-Hill office.

Contributors

Morgan Adair, Software Consultant, New York City

Jonathan Amsterdam, Artificial Intelligence Laboratory, Massachusetts Institute of Technology

William Bailey, BASIS, Inc., Chatham, New Jersey

Scott Berfield, Parity Productions, Chicago, Illinois

Jeffrey D'Ambrosia, DHI Contracting, Greenbrook, New Jersey

David Dameo, Software Consultant, Perth Amboy, New Jersey

Brig Elliott, True BASIC, Inc., Hanover, New Hampshire

Ian Harac, Software Consultant, Fords, New Jersey

Bridgford Hunt, Software Consultant, Shelter Island, New York

Michael Iannone, Department of Mathematics and Computer Science, Trenton State College

Ann Joslin, Idaho State Library

Guy M. Kelly, Chairman of Forth Standards Team, Chairman San Diego Chapter of the Forth Interest Group, La Jolla, California

Yong M. Lee, Department of Mathematics and Computer Science, Trenton State College

Paul D. Lerhman, Musician and Software Consultant, Arlington, Massachusetts

Robert Levine, UNYSIS, New York City

David Lewis, Department of Mathematics and Computer Science, Trenton State College

Len Lindsay, COMAL Users Group USA, Ltd., Madison, Wisconsin

Lawrence Mahon, Software Consultant, Somerville, New Jersey

Contributors

Gary Markman, BDI Systems, Pawling, New York

Norman Neff, Department of Mathematics and Computer Science, Trenton State College

Karl Nicholas, Physical Acoustics Corporation, Princeton, New Jersey

Ross Overbeek, Argonne National Laboratories, Lisle, Illinois

P. J. Plauger, Whitesmiths, Ltd., Westford, Massachusetts

Mark J. Robillard, Systems Consultant, Milford, New Hampshire

Al Rubottom, New Technica, Inc., San Diego, California

Stephen E. Seadler, Uniconsult, New York City

Thomas Sheldon, Author and Consultant, Santa Barbara, California

Kern Sibbald, Auto CAD, San Francisco, California

Ernest R. Tello, President, Integral Systems, Santa Cruz, California

Paul Thomas, Force Field, San Francisco, California

Michael Tilson, Vice President, Human Computing Resources Corporation, Toronto, Ontario

Charles R. Walther, President, New Century Educational Corporation, Piscataway, New Jersey

Linda Weisner, Research Assistant, Department of English, Trenton State College

Michael Wetmore, Chief of Engineering, Pierce-Phelps, Inc., Philadelphia, Pennsylvania

Robert Wharton, Professor of Mathematics, Department of Mathematics and Computer Science, Trenton State College

Roy Woodall, AT & T, Western Electric, Piscataway, New Jersey

William P. Woodall, Software Specialists, Somerville, New Jersey

Preface to the Second Edition

The years since 1985, when the first edition of the *Personal Computer Programming Encyclopedia* was published, have witnessed a second revolution in the field of personal computing. The introduction of the IBM AT, shortly after the *Encyclopedia* was completed, and the development of "386" machines, 100-plus Mbyte hard drives, WORM and DRAW CDROM technology, low-cost laser printers, desktop graphics scanners, and the Macintosh II have effectively enclosed the computing power and applications capability of a reasonably sized mainframe within the deceptively small box of a desktop personal computer. In addition, the release of multitasking operating systems for both PC compatibles and Macintoshes and the development of reasonably priced local area network packages for desktops have changed the personal computing environment from the individual stand-alone workstation connected to a larger system only by dedicated telephone lines to a truly integrated system. Finally, the development of high-end graphics-based desktop publishing packages for professional and commercial facilities and the introduction of desktop CAD stations based on Reduced Instruction Set Computer chips have created entirely new industries in just three short years.

This second edition of the now-standard *Programming Encyclopedia* includes new entries on these applications as well as entries for Apple's HyperTalk language, which is already revolutionizing the way personal computers are customized by even the most nontechnical of users; neural networks, the most daring and boldest attempt yet to recreate a human reaction and learning system within the strict limitations of binary logic; games and entertainment, and computer-generated music. The *Encyclopedia* has updated the core entry on BASIC by taking the programming language back to its historic roots as a teaching language and examining the way it has evolved. In addition, there are completely revised entries for Forth and for Pascal and new entries for AWK, SNOBOL, and Paradox.

We have maintained the first edition's format of single-column entries for introductory articles and applications and double-column entries for high-level language glossaries, operating systems, and assembly-language instruction sets. We have also preserved the primary goal of the *Encyclopedia* from the first edition, which is to provide a basic, single-volume, cross-indexed, desktop reference to the major personal computer implementations of high-level programming language interpreters and compilers, command language applications software, and operating systems. In its revised and enlarged second edition, the *Encyclopedia* continues to be one of the single most valuable reference tools for the entire spectrum of endusers, from programming professionals to nontechnical hobbyists.

Preface to the First Edition

The concept of personal computing has its beginnings in the late 1970s with the assembly of the first microcomputers based on the technology of the integrated circuit. The initial commercial success of the microprocessor-driven units converged with the ongoing development of high-level computer programming languages. The appearance of these languages in the 1950s took programming, once a field open only to the engineers and designers who had originally developed computers, out of the laboratory and into the business marketplace. By the late 1960s there were a number of high-level languages with applications in the sciences, engineering, business, and even elementary and secondary education. When the first BASIC interpreters were bundled with the new "personal computer" packages in the 1970s, the two development streams were officially joined, and the personal computer revolution began.

Over the next eight years, each success in personal computing technology prompted a new surge of development. The machines themselves grew in power and capability from 8-bit 8K PET computers with membrane-type keyboards and onboard cassette recorders to the IBM PC-AT with its 20 Mbytes of hard-disk storage and true 16-bit speed. By the end of 1984, the LISA technology developed by Apple for its 32-bit office machines was repackaged to appeal to a home market and quickly caught on as the Macintosh. In a lockstep with the development of new hardware systems was the invention of new software systems and applications programs. The BASICs of the 1960s and 1970s gave birth to more powerful versions that had built-in graphics and music commands and system utilities that allowed even novice programmers a sophistication and efficiency of code that previously could only have been found in assembly language routines. However, beyond BASIC, versions of Pascal, C, and Forth were developed which put enormous programming power into the hands of personal computer users and allowed them to emulate the processing capabilities of large mainframes at their desktop terminals. And this is only the beginning. Languages such as Ada, the Department of Defense's new projected standard information-processing language, and Prolog, which is at the center of artificial intelligence research and development, have recently been implemented on personal computers and will become more popular as succeeding generations of more powerful computers find their way into the home and business markets.

This proliferation of computing language implementations has created a serious need for a single reference volume which not only introduces the various languages and indexes all of their command words and statements, but provides for a cross-referencing of applications and a comparison of the languages' capabilities. This is the purpose of the *Personal Computer Programming Encyclopedia*. The *Encyclopedia* illustrates the capabilities of each language with overviews of the language's design and architecture. By comparing the operational differences of each language through sample programs, the *Encyclopedia* demonstrates the different ways applications can be addressed by programmers within the different language environments.

The *Encyclopedia* is divided into two types of sections: the double-column sections which cover the high-level languages, operating system commands, and assembly-language commands, and the single-column sections which contain background and introductory material. In the single-column sections, readers will find articles which explain the architecture and design of computer programs, examine the user-oriented issues of software development in business and education, and explore

Preface to the First Edition

the newest areas of development in graphics, robotics, and artificial intelligence. In choosing these different types of entries for the *Encyclopedia*, recognition has been given to the major areas of personal computing that affect users: (1) the need to understand the logic of program design; (2) the current trends and issues in the most important areas of software development; (3) the diversity of languages that are available to personal computer users and the primary applications of these languages; and (4) the relationship of hardware systems to software systems. The result is a volume that addresses the needs of the entire personal computing community from the business user and consumer of professional software products to the home user learning about a type of information technology that promises to transform the ways people organize their lives.

The *Encyclopedia* provides background histories of the high-level programming languages, operating systems, and applications software cited. It relates the development of these various software tools to the current computing environment as well as to the historical period during which the software was originated and marketed. The result of this approach is a social history of personal computing in which programmers, personal computer users, and general readers will discover the underlying reasons for must of the product development in the marketplace. The *Encyclopedia* explains the different trends in languages, operating systems, and applications software over the past five to ten years, and the effects of these products on users in different professional fields.

The *Encyclopedia* contains an index to all of the keywords and statements in the high-level languages that are cited. This index is an important reference tool for programmers and serious users because it provides an immediate cross-reference between the different languages. Programmers seeking to translate source code from one compiler to another, or from one dialect of BASIC to COMAL, or to structured BASIC will find the cross-index a handy tool. General readers interested in the history and intellectual backgrounds of programming languages will find in the cross-index of high-level language keywords a generic approach to the types of commands that are used in programming. Teachers will also find this system a valuable reference tool for use in comparative programming.

The *Encyclopedia* also examines the different corporate cultures from which the most popular types of personal computers have evolved and evaluates the dynamic relationships between manufacturer and the manufacturer's product history, the hardware system and supporting software, and the user market the computer was targeted to reach. Readers will find an interesting perspective on the current trends of technological development in the areas of hardware, software, and operating environments. There is a capsule summary of the history of personal computers from the first attempts to market basic user-assembled kits to the 32-bit supermicros that will be making their appearance within the next several years. Their history, brief as it is, will provide a needed background to the dynamic microcomputer marketplace and the different products that are announced in the computer magazines and newspapers.

Preface to the First Edition

The articles in the *Personal Computer Programming Encyclopedia* are written by individuals from a variety of backgrounds. This diversity of opinions is reflected in the different levels of emphasis within the entries and the broad perspective of the volume in general. In short, the *Encyclopedia* embraces the types of related informational materials that spread across the traditional boundaries often found in a reference book on science and technology. This is what makes the volume an innovative reference tool.

While a number of computer dictionaries and comparative reference books on programming languages have appeared recently, the *McGraw-Hill Personal Computer Programming Encyclopedia* is the only single-volume reference to provide a comprehensive introduction to the entire personal computing environment both as a science and as a commercial industry. Thus, it will become a valuable reference both for the computer professional and for the novice. Business users, students, teachers, hobbyists, and home users will find the *Encyclopedia* a most useful desktop computer reference.

William J. Birnes
Editor

CONTENTS

CONTENTS

5

CONTENTS

6

7

8

1

PROGRAM DESIGN AND ARCHITECTURE

As in most creative work, the fundamental aspect of writing good applications and systems programs lies in the preliminary design and architecture. It is on this level that some of the most important thinking takes place. Goals are defined, pathways are mapped out, logical relationships between data types are developed, and the rules that will govern the decisions the machine will make are stipulated. It is here, at the very heart of a well-designed program, that a definition of truth is implemented, and the execution of this program, the processing of line after line of code, is a test for that truth. And as a statement of truth and a test for that truth's existence in the data that flows through it, the computer program takes its place right alongside literature and art as a form of creative expression. There is a practical creativity, to be sure, but a computer program is no less creative and structured in its disciplined expression of truth than a line of poetry by Keats, a portrait by Albrecht Dürer, or a Bach concerto.

What takes place in the design and architecture of a computer program? On the most obvious and visible level, it is a patient sequence of logic that expresses the complexity of human thought in terms of an organized pattern of connected decisions to be implemented ultimately as a series of electronic pulses. On an even more fundamental level, it is nothing less than the definition of the reality that the computer will understand as its truth. The design of a computer program, therefore, is a microcosm of the physical universe, and as an amalgamation of artistic creativity and technical precision, it is the marriage of C. P. Snow's "Two Cultures."

Designing a computer program is analogous to designing a building: the programmer must create a structure performing various functions in an efficient, pleasing manner. Frequently program design is also referred to as program architecture, and with good reason: the designer of a program must usually take a multidisciplinary approach to the task. While the knowledge required for the successful design of a particular program (or system, within this context) depends in a good measure on the task to be performed, at a minimum a good understanding of logic, language, and computer fundamentals is prerequisite. More commonly, the designer must also know a goodly amount about the particular objective that the program is to solve, and also about seemingly esoteric subjects including psychology. Programming a computer is an exercise in abstract thinking; unlike many other jobs, with a computer the only tangible results are points of light on a screen, marks on paper, or other effects that are the result of the process. In this respect, programming a computer can be considered an art form, since only the final effort will ever be seen, and only rarely will the component parts ever be viewed.

2 Program Design

How important is good design in a program? As in constructing a building, a quick on-the-fly job may suffice for the moment, and might just last for future generations. More likely, it will collapse in the first big storm to come along, which in the programming world is the first moment a user requests a noticeable change in how the program works. Just as the overwhelming majority of buildings benefit from the services of an architect, so too do most programs from a designer.

The various skills of a program designer are an amalgamation of learned and experienced techniques. At the forefront of a designer's skills is that of communications: to be able to communicate both with the programmer(s) who will write the actual program and with the users of the program. Knowledge of a computer programming language is decidedly secondary. Logic is needed to handle the precise, unswerving manner in which a computer persistently executes only the instructions defined for it. A basic awareness of the rudiments of computer operation is of aid in handling the logic flow, and the data structures of a program. And for those programs which interact directly with the user, through keyboard and display, or other interactive media, some knowledge of psychology may be needed to enhance the computer's communications with humans. System design and analysis courses have been a staple of computer science curricula from the beginning, from graduate degree programs right down to trade schools and high schools.

The choice of the actual programming language and the subsequent coding of the program, while important, take place after the logic of the program has been designed. Programming languages by themselves are only forms of machine code generators. By definition, they are a source of code that is either compiled or interpreted by the machine and translated into the sets of instructions that the machine can process. As a source of code, they help the programmer implement a coherent and executable logical design and serve as a matrix for the actual commands. In addition, high-level computing languages provide for the definition of the types of data and, in some cases, particularly COBOL, the specific machine environments. But, while an indicator of program efficiency, the programming language is not, and should not be considered, the ultimate indicator of quality. The best programs are good, not because they are written in C rather than BASIC, or in LISP rather than Pascal, but because they are designed from concept through code to be disciplined and creative tests for validity and truth.

As an implementation of a logical structure, the original programming environments in the early days of digital computing were required to be designed efficiently and completely because they were written on a machine level which was unforgiving of mistakes. High-level languages, which were developed later, were oriented more toward the natural language of speakers than they were toward the opening and closing of electronic circuits. High-level languages addressed compilers and interpreters which in turn generated the machine code. As a result, high-level languages often have built-in mechanisms which, though quick to trap errors in the usage and syntax of source code, can sometimes be quite forgiving of fundamental mistakes in design and logic. And it is these hidden structural mistakes which ultimately surface in the program's execution to make debugging the program a seemingly impossible task. Therefore, all professional computing training curricula, whether on a secondary school, college, or vocational level, usually begin with a unit on program design and architecture.

But program design and architecture begin with an understanding of commonsense sequential logic. In other words, to design a program, an individual need not have the actual high-level language code at his or her fingertips; rather the person must understand fundamentally how the computer of choice will operate and how to define logically the complete task of the program from beginning to end. It is this task definition and the realistic design of an operation from beginning to end that is necessary in order to write a good program. Whether the task is designing an operating system or writing a program to calculate the principal and interest payments on a loan, the programmer begins with a list of items the program must accomplish and ends with

a chart showing the order in which the program will do just that. This section will introduce you to the elements of programming design and architecture from a programmer's point of view and will look at the elements of goal-setting, evaluating the programming tools to be used, and the construction of a programming structure.

Principles of Program Design

In principle, designing a program is not different from approaching any other task. We require a goal that is clearly defined because we need to know what we want to accomplish, we require an understanding of our beginning resources, and we must have a thorough understanding of the means we will use to reach the goal. As an example, consider the act of going to work in the morning. The goal is to get to work. The beginning resource is your home. The means to reach the goal is some form of transportation. The only difference between this process and the process of writing a

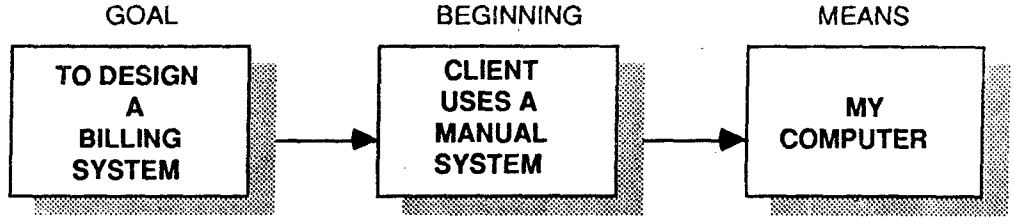

Fig. 1–1

computer program is that there's no need to make any part of the task of getting to work in the morning a conscious, consistent, and repetitive test for truth every day. However, how would these actions appear without a goal? Your normal activities such as waking up at 6:30, getting dressed in work attire, going to the train station, and so on, would seem silly indeed if you performed them on a Sunday morning or a holiday. The point is that most of the things we do need a goal in order for the actions to have any meaning. Designing a program is no exception.

Probably the least practical way to begin writing a computer program is to sit down at your computer and start writing code. Many beginning programmers, eager to

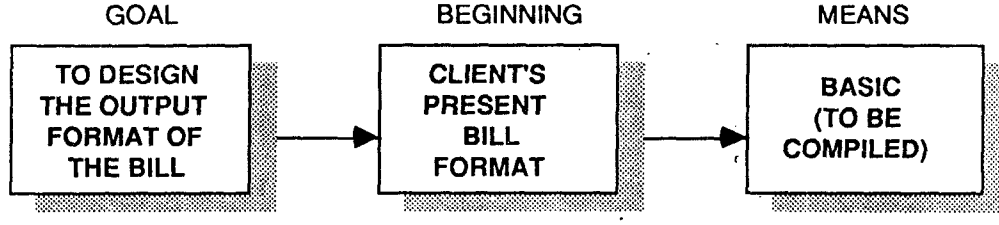

Fig. 1–2

see results, will do just that. The result is usually very awkward and inefficient code, poor documentation, and an absolute nightmare when debugging time comes (and it always does).

The three steps named above—goal, beginning resources, and means—are essential in the design of a good program. And the first language any beginning programmer should consider using is not BASIC, Fortran, Pascal, or COBOL, but rather English. Specifically, the way to begin writing a program is to identify in common language the goals, the beginning, and the means. Typically, the programmer will be working on part of an overall system which has been defined. However, for purposes of this example, consider that the programmer is also responsible for designing the system. Let's say that the task is to design part of a billing system for a client. If the design steps were identified as in Fig. 1–1, the programmer would have virtually no useful information with which to begin. It would be like the going-to-work

analogy without knowing where you worked. The goal, beginning, and means must be identified as specifically as possible. For example, in Fig. 1–2, the goal is specific, the beginning is defined, and the means (in this case the programming language) is specified. It is also important for the programmer to know that the BASIC to be used will be compiled, since compiled BASIC runs much faster than interpreted BASIC. This knowledge could influence the programmer's decision on designing the program since speed of execution would become a less important factor. The steps the programmer would take are:

1. Define the goal by designing the actual appearance of the bill.
2. Examine the input that this section of the program will receive.
3. Decide how to operate on that input by formatting the output in a programming language.

Steps to Design

At the outset, it's important that the computer architect know as much as possible about the task to be solved, and the conditions under which it must be solved. Just as the conventional architect must know the prime function of a building, so must the designer know the major function of his or her program. And while the building architect worries about the placement of a structure within an environment perhaps already shared by other buildings, and the limitation of cost and building materials and labor availability, so must the program designer. Only the names and physical nature of the tools and end products differentiate the two.

The logical place to start is with a definition of program goals. A program may have a single goal, such as ejecting paper from a printer, or printing a payroll check, or multiple goals, such as the complete supervision of a warehouse from receiving of component parts through the shipping of completed manufactured goods to a customer.

Goals are generally defined by the user of the information or service to be produced by the program. The user may or may not be the actual operator of the program; many programs execute unseen (especially those that make up a computer's operating system). Goals of a program may be stated in various ways, and they are often ambiguous. An initial goal of the designer, then, is to clarify the objective of the program to a coherent truth. A designer must also be prepared to accept that the final defined goal of a program may bear little if any resemblance to its initial presentation. And many programs must solve multiple problems, usually in a prioritized fashion.

After the basic objectives of a program are defined, the designer begins to deal with the computer environment. Foremost at this stage is to develop a schema for data representation. Since practically all useful programs manipulate data at some level, a coherent, flexible design of data representation is crucial to the long-term success of a program. The delineation of data within the computer environment may little resemble conventional human organization of information.

For example, within a typical business, the name and address of a particular customer will be duplicated on virtually every piece of paper relating to that customer, be it order, invoice, packing slip, or statement. The computer probably contains but a single occurrence of this information, but, through the computer's ability to cross-reference data, the name and address of the customer is available to any module within the system that requires it. In another case, where a long list of yes-no answers to questions is desired, the computer representation may be but a single binary digit for each of the answers. The data design for a program may be a simple list of terms, or a multifile interlinked database system. But the basic goal of a robust and logical design remains the same in either extreme. Of course, if the program is expected to be abandoned at a known point in the future, then perhaps this goal may be modified. But designers should also be somewhat wary of such predictions, and generally strive for the best possible design. If the data structures are too cumbersome, at some point the system must be torn down and rebuilt. It is possible for a design to be both flexible and

complex at the same time; an examination of the histories of some mass-market utility software will show this to be true.

In developing a data representation, the designer should know both the source and the destination of data manipulated by the program, and, potentially, other uses of the data. The source of information may influence data representation, most especially in the case of machine-specific functions, such as display control, and destination of the information, as in printer control, may also influence the design. Additional considerations will be security of information, both from prying eyes and from operator and/or system error.

Also around the stage of data design, the choice of programming language begins to become relevant. While the best approach is to choose a language based on its inherent strengths to solve a particular task, most designers will be constrained to a choice of language based on a more mundane reason: availability. Nevertheless, the programming language(s) to be used will impact on data representation, as each programming language has its own quirks as to what is allowable. But there is a generally available common subset of data types available to all languages—character, data, and numbers.

The choice of programming language, however, is not so intensely crucial. Many tasks can be solved in any language, although the level of effort required to solve the task may vary widely.

The particular language or mix of languages used to solve a specific task is not a legitimate guide to the quality of the solution; more, it is a guide to what was available at that time and place. In many cases, familiarity with the quirks of a particular language may cause it to be chosen over some more "suitable" language. Academics may find cause to argue over the merits of one language to another, but with the continuing evolution of computers, over time, many of the wide variations in the problem-solving capabilities of languages are shrinking. Still, most languages are identified, rightly or wrongly, with strengths in particular areas. Fortran and PL/I are the major choices of scientific-engineering projects, with rich mathematical abstractions available. COBOL enjoys a reputation of rich data structures, easy readability by nonprogrammers, and wide acceptance in business applications programming. C is the choice for portability, and Pascal has its own body of adherents. BASIC is the most widely installed language, although it suffers from a lack of standards. Assembly language, the native language of a specific computer, is still the vehicle of choice for small, extremely fast programming. And other languages, with some proprietary to a particular environment, others perhaps lacking academic or industry support, exist for solving various disparate situations.

Once a conceptualization of data representation has been made, but perhaps prior to the choice of a programming language, the designer must seriously consider the flow of information through the program.

Obviously the program must perform whatever manipulation is needed to meet its objectives. Also, the design should take into account error conditions, whether caused by invalid data, operator error, or machine error. A program which dies due to simple operator error, such as a printer running out of paper, definitely missed something at the design stage. The designer should anticipate every possible sort of data corruption, and place safeguards against it where practicable.

Various tools exist to assist the designer at this stage. Perhaps the most ubiquitous is the flowchart, a stylized schematic diagram of program control. In its purest state, the flowchart is drawn with a template, using special symbols to denote certain precise forms of activity. A decision is represented with a diamond, a process or calculation with a rectangle, a sort/merge operation with a triangle. Arrows direct the flow of information through the flowchart.

Logic of the Flowchart

In the course of structuring the logic and the flow of control within the actual program, the most important tool is the formal flowchart. A flowchart is a diagram