

PROGRAMMING  
PROVERBS



HENRY F.  
LEDGARD



HAYDEN

*Principles of Good Programming*  
with Numerous Examples to  
*Improve Programming Style and Proficiency*



PROGRAMMING  
PROVERBS



**HENRY F.  
LEDGARD**



HAYDEN BOOK COMPANY, INC.  
Rochelle Park, New Jersey

## ACKNOWLEDGMENTS

- To Michael Flynn, my constant advisor at Johns Hopkins  
To Rao Kosaraju, whose friendship, intellect, and lightheartedness have been a source of delight  
To Will Fastie, who conscientiously devoted his imaginative talents towards writing the original draft of this work, and whose ready wit and energy lightened the task  
To Leslie Chaikin, whose hard work added significantly to the substance of each chapter  
To Joseph Davidson, who admirably served as a consultant on some of the deeper issues and who created some of the less forgotten lines  
To Lee Hoevel and Ian Smith, who contributed to a sound analysis of the issues in programming  
To members of the Institute for Computer Science at the National Bureau of Standards, who helped provide a solid intellectual environment for this work  
To students, faculty, and secretaries in the Computer and Information Science department at the University of Massachusetts.

### Library of Congress Cataloging in Publication Data

Ledgard, Henry F  
Programming proverbs.

(Hayden computer programming series)  
Bibliography: p.

Includes index.

1. Electronic digital computers--Programming.

I. Title.

QA76.6.L368 001.6'424 74-22058


ISBN 0-8104-5522-6

Copyright © 1975 by HAYDEN BOOK COMPANY, INC. All rights reserved.  
No part of this book may be reprinted, or reproduced, or utilized in any form or by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying and recording, or in any information storage and retrieval system, without permission in writing from the Publisher.


Printed in the United States of America

3 4 5 6 7 8 9 PRINTING

76 77 78 79 80 81 82 YEAR



## FOREWORD



By necessity, computer science, computer education, and computer practice are all embryonic human activities, for these subjects have existed for only a single generation. From the very beginning of computer activities, programming has been a frustrating black art. Individual abilities range from the excellent to the ridiculous and often exhibit very little in the way of systematic mental procedure. In a sense, the teaching of programming through mistakes and debugging can hardly be regarded as legitimate university level course work. At the university level we teach such topics as the notion of an algorithm, concepts in programming languages, compiler design, operating systems, information storage and retrieval, artificial intelligence, and numerical computation; but in order to implement ideas in any of these functional activities, we need to write programs in a specific language. Students and professionals alike tend to be over-optimistic about their ability to write programs or to make programs work according to pre-established design goals.

However, we are beginning to see a breakthrough in programming as a mental process. This breakthrough is based more on considerations of style than on detail. It involves taking style seriously, not only in how programs look when they are completed, but in the very mental processes that create them. In programming, it is not enough to be inventive and ingenious. One also needs to be disciplined and controlled in order not to become entangled in one's own complexities.

In any new area of human activity, it is difficult to foresee latent human capabilities. We have many examples of such capabilities: touch typing, speed writing, and 70-year-old grandmothers who drive down our highways at 70 miles an hour. Back in 1900 it was possible to foresee cars going 70 miles an hour, but the drivers were imagined as daredevils rather than as grandmothers. The moral is that in any new human activity, one generation hardly scratches the surface of its capabilities. So it will be in programming as well.

The next generation of programmers will be much more competent than the first one. They will have to be. Just as it was easier to get into college in the "good old days," it was also easier to get by as a programmer in the "good old days." For this new generation, a programmer will need to be capable of a level of precision and productivity never dreamed of in years gone by.

The new generation of programmers will need to acquire discipline and control, mainly by learning to write programs correctly from the start. The debugging process will take the new form of verifying that no errors are present, rather than the old form of finding and fixing errors over and over (otherwise known as "acquiring confidence by exhaustion"). Programming is a serious logical business that requires concentration and precision. In this discipline, concentration is highly related to confidence.

In simple illustration, consider a child who knows how to play a perfect game of tic-tac-toe but does not know that he knows. If you ask him to play for something important, like a candy bar, he will say to himself, "I hope I can win." And sometimes he will win, and sometimes not. The only reason he does not always win is that he drops his concentration. He does not realize this fact because he regards winning as a chance event. Consider how different the situation is when the child *knows* that he knows how to play a perfect game of tic-tac-toe. Now he does not say, "I hope I can win"; he says instead, "I know I can win; it's up to me!" And he recognizes the necessity for concentration in order to insure that he wins.

In programming as in tic-tac-toe, it is characteristic that concentration goes hand-in-hand with justified confidence in one's own ability. It is not enough simply to know how to write programs correctly. The programmer must *know that he knows* how to write programs correctly, and then supply the concentration to match.

This book of proverbs is well suited to getting members of the next generation off to the right start. The elements of style discussed here can help provide the mental discipline to master programming complexity. In essence, the book can help to provide the programmer with a large first step on the road to a new generation of programming.

*Harlan D. Mills*

Federal Systems Division, IBM  
Gaithersburg, Md.



## PREFACE




Several years ago I purchased a small book called *Elements of Style* written by William Strunk, Jr. and revised by E. B. White. Originally conceived in 1918, this book is a manual on English style. It is noted for its brevity, rigor, and deeply rooted faith in concise, clear English prose. I have read this manual several times. Each time I am again challenged to write better prose. In part, that small book is the motivation for this work.

When I began teaching courses on programming languages, I was struck by the tremendous need for style and quality in student programs. Reminded of Strunk's little book, I became concerned with the need to motivate an interest in program quality. I believe that introductory programming courses should be intimately and overtly concerned with elements of style in computer programming. This concern was brought to fruition in the summer of 1972, when the basic draft of this book was written. It was meant as a brief for people who write computer programs and who want to write them well.


Recently, there has been an increasing concern within the computing community about the quality of software. As a result, a new methodology is emerging, a harbinger of further changes to come. The ideas presented in this book depend heavily on the work of many competent researchers. Notable are the works of Dijkstra, Mills, Strachey, Hoare, Wirth, Weinberg, Floyd, Knuth, Wulf, and many others.

Several qualities other than academic ones have been deliberately sought for in this book. First, there has been an attempt to be lighthearted. It is primarily through a zest for learning that we do our best work and find it most rewarding. Second, there has been the goal of being specific. The book is written primarily in the imperative mood, and there are many examples.

This book is designed as a guide to better programming, not as an introduction to programming. As such, it should be of value to programmers who have some familiarity with programming but no great proficiency. The book may thus be used as a supplementary text in undergraduate courses where programming is a major concern. It should also be of value to experienced programmers who are seeking an informal guide to the area of quality programming. As an offshoot of its aims, I hope that programmers will begin to read and analyze programs written by others and try to reduce the intellectual effort



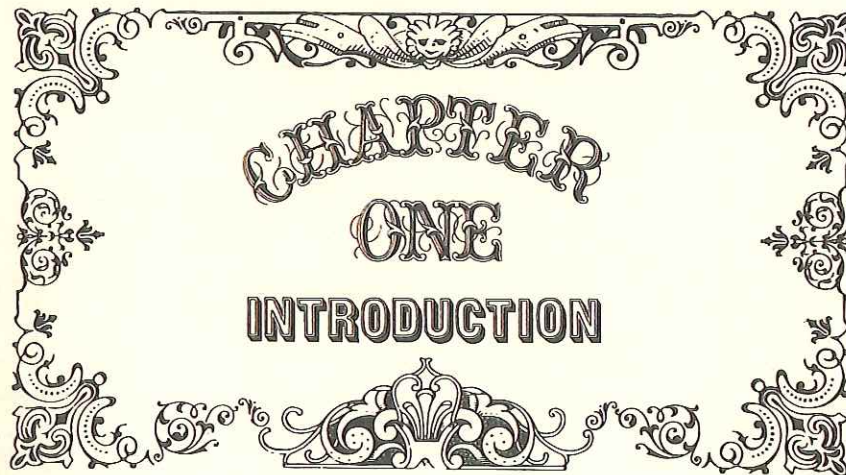
# CONTENTS



<b>Chapter 1</b>	<b>Introduction</b> .....	<b>1</b>
<b>Chapter 2</b>	<b>Programming Proverbs</b> .....	<b>3</b>
	<i>Proverb 1</i> Define the Problem Completely	5
	<i>Proverb 2</i> Think First, Program Later	10
	<i>Proverb 3</i> Use the Top-Down Approach	11
	<i>Proverb 4</i> Beware of Other Approaches	13
	<i>Proverb 5</i> Construct the Program in Logical Units	16
	<i>Proverb 6</i> Use Procedures	16
	<i>Proverb 7</i> Avoid Unnecessary GOTO'S	19
	<i>Proverb 8</i> Avoid Side Effects	24
	<i>Proverb 9</i> Get the Syntax Correct Now, Not Later	26
	<i>Proverb 10</i> Use Good Mnemonic Names	27
	<i>Proverb 11</i> Use Intermediate Variables Properly	29
	<i>Proverb 12</i> Leave Loop Variables Alone	31
	<i>Proverb 13</i> Do Not Recompute Constants within a Loop	32
	<i>Proverb 14</i> Avoid Implementation-Dependent Features	33
	<i>Proverb 15</i> Avoid Tricks	35
	<i>Proverb 16</i> Build in Debugging Techniques	36
	<i>Proverb 17</i> Never Assume the Computer Assumes Anything	38
	<i>Proverb 18</i> Use Comments	40
	<i>Proverb 19</i> Prettyprint	42
	<i>Proverb 20</i> Provide Good Documentation	43

	<i>Proverb 21</i>	Hand-Check the Program before Running It	44
	<i>Proverb 22</i>	Get the Program Correct before Trying to Produce Good Output	48
	<i>Proverb 23</i>	When the Program Is Correct, Produce Good Output	48
	<i>Proverb 24</i>	Reread the Manual	48
	<i>Proverb 25</i>	Consider Another Language	50
	<i>Proverb 26</i>	Don't Be Afraid to Start Over	51
	<i>Exercises</i>		51
<b>Chapter 3</b>	<b>Top-Down Programming</b>		64
	A Payroll Problem		67
	Kriegspiel Checkers		77
	<i>Exercises</i>		92
<b>Chapter 4</b>	<b>Miscellaneous Topics</b>		95
	Use of Mnemonic Names		95
	Prettyprinting		98
	Representation of Algorithms and Tricky Programming		102
	Procedures, Functions, and Subroutines		108
	Recursion		116
	Debugging Techniques		121
	Some Parting Comments		125
	<i>Exercises</i>		128
<b>Bibliography</b>			131
<b>Index</b>			133





CHAPTER  
ONE  
INTRODUCTION

“The purpose of this here book is to learn programmers, especially them who don’t want to pick up no more bad habits, to program good, easy, the first time right, and so somebody else can figger out what they done and why!”

For those readers who appreciate diamonds in the rough, the paragraph above represents this introduction as originally conceived. The following pages merely display the same diamonds cut, polished, and in a fancier setting.

An indication of the current state of the art of computer programming is the proud exclamation, “It worked the first time!” That this statement is conceivable but rarely heard indicates one prime fact: writing programs that work correctly the first time is *possible* but *unusual*. Since programmers undoubtedly try to write programs that work the first time, the question arises, “If it is possible, why is it unusual?” The answer to this question is twofold: First, programming is difficult, and second, there are very few standard methods for developing and writing good programs. Since few standard methods exist, each programmer must develop personal methods, often haphazardly. The success of these ad hoc methods depends on how well-suited they are to the problem at hand. For this reason the quality of programs varies not only between programmers, but also between programs written by the same programmer.

In reality, the state-of-the-art is considerably worse than is implied by the fact that most computer programs do not work right the first time. While there are many programs that never work at all, many more work only most of the time. More importantly, of those that work correctly, many are difficult to understand, change, or maintain.

This book is predicated on the thesis that programming is entering a new and exciting era and that programmers can and should write programs that work correctly the *first* time. For those who are accustomed to hours, days, or even

weeks of debugging time, this goal might seem idealistic. However, I am committed to the idea that well-founded principles can be invaluable in achieving it. Many important programming techniques are seldom obvious, even to experienced programmers. Yet with a fresh approach, many programmers may be surprised at the improvement in their ability to write correct, readable, well-structured programs.

This book is also predicated on the thesis that the ideas presented here should go *hand-in-hand* with learning any new computer language. The reader who dismisses the overall objective of this book with the comment, "I've got to learn all about my language first" may be surprised to find that the study of good programming practices along with the basics of the language may reap quick and long-standing rewards.

This book is organized in three major parts. Chapter 2 is a collection of simple rules, called *proverbs*. The proverbs summarize in terse form the major ideas of this book. Each proverb is explained and applied. A few references to later chapters are made where various ideas are more fully explored.

Chapter 3 is an introduction to a strict top-down approach for programming problems in any programming language. The approach is oriented toward the writing of correct, modular programs. It should be read carefully, because some of its details are critical and not necessarily intuitive. The approach, clearly related to an approach called "Stepwise Refinement" (see Reference W4 in the Bibliography) and an approach called "Structured Programming" (Ref. D1), hinges on developing the overall logical structure of the program first. Specific decisions, such as data representation, intermediate variables, and the like, are delayed as long as possible in order to achieve maximum flexibility.

Chapter 4 elaborates on several techniques discussed in the chapter on programming proverbs and contains a section on recursion as well. The use of these techniques should make programs easier to read and understand. They should also expedite error detection and program modifications should they become needed.

ALGOL 60 (see Reference N1 in the Bibliography) and PL/I (Ref. Z3) are used throughout the text, sometimes almost repetitiously. In a few ALGOL 60 programs, some simple, free-format input/output statements have been added to the strict ALGOL 60 language.

The reader may observe the absence of flowcharts in this book. This omission is deliberate. In the author's opinion, the use of flowcharting techniques as a method of program development has been overestimated, mainly because flowcharts can readily lead to an undue preoccupation with flow of control. The objective here is to emphasize numerous other programming techniques that have little need for flowcharts, but it must be admitted that the judicious use of flowcharts can be a valuable part of the programmer's repertoire.



CHAPTER  
TWO  
PROGRAMMING PROVERBS

“Experience keeps a dear school, but fools will learn in no other”  
Maxim prefixed to *Poor Richard's Almanack*, 1757

Over two centuries ago Ben Franklin published his now familiar *Poor Richard's Almanack*. In it he collected a number of maxims, meant as a guide for everyday living. Analogously, this chapter is intended as a simple guide to everyday programming. As such, it contains a collection of terse statements that are designed to serve as a set of practical rules for the programmer. In essence, the programming proverbs motivate the entire book.

As with most maxims or proverbs, the rules are not absolute, but neither are they arbitrary. Behind each one lies a generous nip of thought and experience. I hope the programmer will seriously consider them all. At first glance some of them may seem either trivial or too time-consuming to follow. However, I believe that experience will prove the point. Just take a look at past errors and then reconsider the proverbs.

Before going on, a prefatory proverb seems appropriate.

“Do Not Break the Rules before Learning Them”

By their nature, the programming proverbs, like all old saws, overlook much important detail in favor of an easily remembered phrase. There are some cases where programs should not conform to standard rules, that is, there are *exceptions* to every proverb. Nevertheless, I think experience will show that a programmer should not violate the rules without careful consideration of the alternatives.

A list of all the proverbs is given in Table 2.1. It is hard to weigh their relative importance, but they do at least fall into certain categories. The relative importance of one over another depends quite markedly on the programming problem at hand.

Table 2.1 The Programming Proverbs

---

*Approach to the Program*

1. DEFINE THE PROBLEM COMPLETELY
2. THINK FIRST, PROGRAM LATER
3. USE THE TOP-DOWN APPROACH
4. BEWARE OF OTHER APPROACHES

*Coding the Program*

5. CONSTRUCT THE PROGRAM IN LOGICAL UNITS
6. USE PROCEDURES
7. AVOID UNNECESSARY GOTO'S
8. AVOID SIDE EFFECTS
  
9. GET THE SYNTAX CORRECT NOW, NOT LATER
10. USE GOOD MNEMONIC NAMES
11. USE INTERMEDIATE VARIABLES PROPERLY
12. LEAVE LOOP VARIABLES ALONE
13. DO NOT RECOMPUTE CONSTANTS WITHIN A LOOP
  
14. AVOID IMPLEMENTATION-DEPENDENT FEATURES
15. AVOID TRICKS
16. BUILD IN DEBUGGING TECHNIQUES
17. NEVER ASSUME THE COMPUTER ASSUMES ANYTHING
  
18. USE COMMENTS
19. PRETTYPRINT
20. PROVIDE GOOD DOCUMENTATION

*Running the Program*

21. HAND-CHECK THE PROGRAM BEFORE RUNNING IT
22. GET THE PROGRAM CORRECT BEFORE TRYING TO PRODUCE GOOD OUTPUT
23. WHEN THE PROGRAM IS CORRECT, PRODUCE GOOD OUTPUT

*In General*

24. REREAD THE MANUAL
  25. CONSIDER ANOTHER LANGUAGE
  26. DON'T BE AFRAID TO START OVER
-

I must not close this introduction to the proverbs without noting why we use the word, *proverb*, rather than the more accurate word, *maxim*. Proverbs and maxims both refer to short pithy sayings derived from practical experience. Proverbs, however, are usually well-known, whereas maxims are usually not. Admittedly, the programming proverbs are not popular sayings. However, the title was chosen with an eye to the future, when hopefully some of these sayings might become true programming proverbs. And, of course, I think that "Programming Proverbs" just sounds better!

### Proverb 1 DEFINE THE PROBLEM COMPLETELY

At first glance, this proverb seems so obvious as to be worthless. As the saying goes, "It's as plain as the nose on your face." True enough, but there is a tendency to take your nose for granted. Similarly, there is a tendency to assume that a problem is well defined without really examining the definition. As a result, all too often programmers begin work before they have an *exact* specification of the problem.

Consider Example 2.1, which is stated in plain English. The statements here range from the somewhat vague definition given in 2.1a, which even an experienced cook would deem ambiguous, to that of 2.1d, which is so completely specified that even the average cook should be able to follow it easily. Statement 2.1d even specifies 350°F, as opposed to 350°. Who knows, someone just might have a centigrade oven.

#### Example 2.1 Successively Better Problem Definitions

---

<i>Statement 2.1a</i>	Cook the chicken.
<i>Statement 2.1b</i>	Roast the chicken.
<i>Statement 2.1c</i>	Roast the chicken in a 350° oven until done.
<i>Statement 2.1d</i>	Roast the chicken in an oven at 350°F. Roasting times should be about 30 minutes per pound according to the following timetable:

<i>Weight</i>	<i>Time</i>
2 lb	1 hr
2-3 lb	1-1½ hr
3-4 lb	1½-2 hr
4-5 lb	2-2½ hr

---

More typically, consider the following simple program specification:

“Write a program that reads in a list of nonzero integers and outputs their mean.”

At first glance, this specification sounds complete. On closer analysis, there prove to be a number of vague points.

- (1) How long is the list? If the length of the list is to be read in explicitly, will the length be the first integer, or is the list terminated by a blank line, a special symbol, or the number zero?
- (2) What is the formula for the mean and what is to be printed if the list is empty?
- (3) Is the input free format or fixed format? If fixed format, what is it?
- (4) What is the output to be? A message along with the mean? To how many decimal places should the mean be computed?

In the real world, programmers are often given some latitude in the final input/output characteristics of a program. Since all languages have rigid rules for the execution of programs, programmers must be specific to the last detail. In general, if something is left unspecified in the original definition, the programmer will eventually have to face the consequences. Changes made while writing the program can be annoying and distracting. In addition, some of the code already written may have to be scrapped due to oversights in the original problem definition. As a result, any critical omitted information should be defined by the programmer *before* programming.

Consider next the definition of Example 2.2a. Certain questions remain unanswered, for example, the form of the input data, the form of the check stub, and the formulas for calculating the gross and net pay. More importantly, the problem is partly defined by a specific *algorithm* stating the order of the calculations. This kind of definition should be avoided unless the implementation of a specific algorithm is actually part of the problem. The specification of an *unnecessary* algorithm clouds a program specification and restricts the class of possible solutions. A better definition is given in Example 2.2b.

### Example 2.2 Proposed Definitions of a Payroll Problem

---

#### 2.2a Poor Problem Definition

Read an employee data card.  
Calculate gross pay.  
Calculate net pay by deducting  
    4% taxes and 1.75% for social security  
Print a payroll stub for the employee  
If there are more cards,  
    then go back and repeat the process,  
    otherwise exit the program.

---

2.2b Better Problem Definition

*Input:* A sequence of employee data cards with the following data:

Columns	Meaning	Format
1-5	RATE of pay per hour	dd.dd
11-15	HOURS worked per week	dd.dd

*Output:* A payroll stub for each employee, printed according to the following format:

line 3 →	PAYROLL STUB			
line 12 →	RATE	HOURS	GROSS	NET
line 14 →	dd.dd	dd.dd	ddd.dd	ddd.dd
	↑	↑	↑	↑
	col 5	col 20	col 35	col 50
	Rate of	Hours	Gross	Net
	pay	worked	pay	pay

$$\text{NETPAY} = \text{RATE} * \text{HOURS} * (1 - 0.04 - 0.0175)$$

As a third example, consider the definition of Example 2.3a, which defines a program to aid a prospective homeowner in determining the financial arrangements of a mortgage loan. This definition is quite adequate, but on close analysis certain points need to be resolved. The formula that relates the values of the principal, interest rate, number of years, and monthly payment may not be readily available to the programmer. The formats for the input and output are not exactly clear, and several exceptional conditions that can arise in the computation are not mentioned. The definition of 2.3b resolves each of the above issues. It is a bit long, but far more precise than the definition of 2.3a.

**Example 2.3 Proposed Definitions of a Mortgage Problem**

*Example 2.3a Adequate Problem Definition*

We wish to devise a program to help potential home-owners consider the finances of mortgaging a home. There are four basic factors to be considered: the principal, the interest rate, the number of years for the mortgage, and the monthly payment. The program must input values for any three of the above quantities, output the fourth quantity, and also output a table indicating how the amount of the first monthly payment of each year is divided between principal and interest.

The input to this program is a line (or card) containing three of the above four figures:

<i>Columns</i>	<i>Quantity</i>
1-5	Principal
8-11	Interest rate
14-15	Number of years
18-22	Monthly payment

The principal and number of years are given as integers, the interest rate and monthly payments are given as fixed-point real numbers. The missing quantity is given as an integer or fixed-point zero.

The output is to be a line indicating the value of the missing quantity, and a table giving, for the first monthly payment of each year, the amount contributed to decreasing the principal and the amount paid as interest.

*Example 2.3b Better Problem Definition*

(1) *Problem Outline:* We wish to devise a program to help potential homeowners consider the finances of mortgaging a home. There are four basic factors to be considered:

- P The principal amount of the mortgage
- I The yearly interest rate for the mortgage
- N The number of years for the duration of the mortgage
- M The (constant) monthly payment required to pay back the principal P over N years at the interest rate I

The above quantities are related by the equation:

$$M = \frac{P * i * (1 + i)^n}{(1+i)^n - 1}$$

where

- $i = I/12$  = monthly interest rate
- $n = 12 * N$  = number of monthly periods in N years

Briefly, the program is to input any three of the above quantities, compute and print the fourth quantity, and also print a table specifying how the first monthly payment of each year is divided between interest and principal.

(2) *Input:* The input to this program is a line (or card) of the form

column	1	8	14	18
	↓	↓	↓	↓
	dddd	d.dd	dd	ddd.dd
	P	I	N	M



where the d's represent decimal digits such that

- P = the principal in dollars  
 I = the percentage interest rate computed to two decimal places  
 N = the number of years in integer form  
 M = the monthly payment in dollars and cents

The value of P, I, N, or M to be computed is given as zero. Leading zeros for any value may be replaced by blanks.

(3) *Output:* The output from the program is to consist of two parts:

(a) The value to be computed using one of the formats:

PRINCIPAL	=	\$dddd
INTEREST RATE	=	d.dd
NUMBER OF YEARS	=	dd
MONTHLY PAYMENT	=	\$ddd.dd

(b) A table giving for the first monthly payment of each year the amount paid to principal and the amount paid to interest. The headings and formats for the table values are as follows:

YEAR	AMT PAID TO PRINCIPAL	AMT PAID TO INTEREST
dd	\$ddd.dd	\$ddd.dd

Leading zeros for any value should be replaced by blanks.

(4) *Exceptional Conditions:* If any of the input values are not of the prescribed format, or if any output value is not in the range indicated, the program is to print an appropriate message to the user.

(5) *Sample Input:*

20000 8.00 22 0.0

(6) *Sample Output for Above Input:*

MONTHLY PAYMENT = \$154.36

YEAR	AMT PAID TO PRINCIPAL	AMT PAID TO INTEREST
1	21.03	133.33
2	22.77	131.59
.		
.		
25	142.53	11.83

One important point of Example 2.3b is the inclusion of a sample of the input and output. Often a sample printout can be of great value to a computer programmer in giving a quick synopsis of the problem. In addition, a sample printout can often prevent surprises in cases where the program turns out to be quite different from the expectations of the person defining the problem. If a programmer is not given a sample of the input/output, he or she should try to provide a sample *before* programming.

Before closing this discussion, one critical point must be emphasized. In practice, a programmer is often given a somewhat vague problem description and left with decisions about input/output headings, the treatment of exceptional conditions, and other factors. In such cases, the programmer should *not* begin the program until all of these alternatives have been considered and resolved.

In summary, starting the program with a fully defined problem gives a programmer a solid head start. This is the first proverb because it should be the *first* programming consideration.

## Proverb 2 THINK FIRST, PROGRAM LATER

This proverb is intimately connected with a clear definition of the problem. The essence is to start thinking about the program as soon as possible, and to start the actual programming process only when the problem has been well defined and you have chosen an overall plan of attack.

Consider the first part of the proverb: *Think* means *think—do not program!* Examine the problem carefully. Consider alternative ways to solve the problem. Consider at least two different approaches. Examine the approaches in sufficient detail to discover possible trouble spots or areas in which the solution is not transparent. A top-notch program requires a top-notch algorithm. *First* means *immediately—before programming*. Start thinking as soon as possible, while the problem is fresh in your mind and the deadline is as far away as it will ever be. It is much easier to discard poor thoughts than poor programs.

The second part of the proverb is *program later*. Give yourself time to polish the algorithm thoroughly before trying to program it. This will shorten the programming time, reduce the number of false starts, and given you ample time to weed out difficult parts.

A common violation of this proverb lies in a phenomenon that we shall call the “linear” approach. In the linear approach, a programmer receives a problem and immediately starts typing or punching the code to solve it. Such an attack quickly leads to errors and patches to cover easily made mistakes. Avoid the linear approach unless you are sure the problem is really easy.

Remember Murphy’s law of programming: It always takes longer to write a program than you think. A corollary might be: The sooner you start coding your program (instead of thinking), the longer it will take to finish it.

**Proverb 3 USE THE TOP-DOWN APPROACH**

One major point of this book is to advocate the "top-down" approach to a programming problem. The top-down approach advocated here is probably *not* like conventional methods of programming. Furthermore, the top-down approach is itself subject to several interpretations, some of which we disagree with. Top-down programming is discussed at length in Chapter 3. The following description of the top-down approach is an excerpt from that chapter:

**1. Exact Problem Definition:**

The programmer starts with an *exact* statement of the problem. It is senseless to start any program without a clear understanding of the problem.

**2. Initial Language Independence:**

The programmer initially uses expressions (often in English) that are relevant to the problem solution, even though the expressions cannot be directly transliterated into the target language. From statements that are *machine and language independent*, the programmer moves toward a final machine implementation in the target language.

**3. Design in Levels:**

The programmer designs the program in *levels*. At each level, the programmer considers alternative ways to refine some parts of the previous level. The programmer may look a level or two ahead to determine the best way to design the present level.

**4. Postponement of Details to Lower Levels:**

The programmer concentrates on critical broad issues at the initial levels and postpones details (for example, input/output headings, choice of identifiers, or data representation) until lower levels.

**5. Insuring Correctness at Each Level:**

After each level, the programmer rewrites the "program" as a *correct formal statement*. This step is critically important. The program must be debugged to insure that all arguments to unwritten procedures or sections of code are then explicit and correct. Further sections of the program should be able to be written *independently*, without later changing the specifications or the interfaces between modules. The importance of being complete and explicit at each level is by far the most misunderstood aspect of the top-down approach.

**6. Successive Refinements:**

Each level of the program is successively refined and debugged until the programmer obtains the completed program in the target language.

Consider Example 2.4, which gives the initial levels of the design of a small payroll problem presented in Chapter 3. After examining the problem in detail, the programmer makes a general statement of the program,  $P_0$ . After further consideration, he decides on the overall approach to the program and obtains the

---

**Example 2.4 Use of the Top-Down Approach in a Payroll Problem**


---

P<sub>0</sub>

Process the Payroll Cards

P<sub>1</sub>

Initialize for program

A: read next employee card

if no more data

then calculate and print average hours worked  
stop

else process the card and check  
 update for weekly average  
 go back to (A) for next employee

P<sub>2</sub>

```
/* Initialize for program */
TAXRATE   = 0.04
SS_RATE    = 0.0175
TOTAL_HOURS = 0.0
NOS_EMPLOYEES = 0
```

```
/* read next employee card */
A: read CARD into (NAME, SS_NUM, WAGE, HOURS)
```

if no more data

then /\* calculate and print average, and stop \*/  
 AVERAGE = TOTAL\_HOURS/NOS\_EMPLOYEES  
 print (AVERAGE)  
stop

else /\* process the card and check \*/  
 GROSS\_PAY = WAGE\*HOURS  
 NET\_PAY = GROSS\_PAY - GROSS\_PAY\*TAXRATE - GROSS\_PAY\*SS\_RATE  
 print (NAME, NET\_PAY, SS\_NUM) on check stub

```
/* update for weekly average */
TOTAL_HOURS = TOTAL_HOURS + HOURS
NOS_EMPLOYEES = NOS_EMPLOYEES + 1
```

```
/* repeat for next employee */
goto A
```

---

more detailed description of  $P_1$ . The programmer now takes another step and refines the statements of  $P_1$  into the more formal description of  $P_2$ . The "program" of  $P_2$  *explicitly* specifies the decisions made in  $P_1$ . Each level is in some sense complete, and can be *debugged* as if it had been written in an actual programming language. While these initial levels are not yet written in the particular target language, successive refinements will take care of that.

Top-down programming has two distinct advantages. First, it initially frees a programmer from the confines of a programming language and allows him to deal with more natural constructs. Second, it leads to a structured modular approach, which allows the programmer to write statements relevant to the level of detail he is seeking. The details can be specified later in separate modules. In fact, the entire goal of top-down programming is just that: to aid the programmer in writing well-structured, modular programs.

#### Proverb 4 BEWARE OF OTHER APPROACHES

Traditionally, programmers have used many different approaches to a program. Consider the following list:

- (1) Bottom-up approach
- (2) Inside-out or forest approach
- (3) Linear approach
- (4) Typical systems analyst approach
- (5) Imitation approach

In the "bottom-up" approach, the programmer usually writes the lower procedures first and the upper levels later. The bottom-up approach is (with some critical exceptions) the mirror image of the top-down approach. It suffers severely from requiring the programmer to make specific decisions about the program *before* the overall structure is understood.

In between the top-down and the bottom-up approaches, we have the "inside-out" or "forest" approach, which consists of starting in the middle of the program and working down and up at the same time. Roughly speaking, it goes as follows:

##### 1. *General Idea:*

First we decide upon the general idea for programming the problem.

##### 2. *A Rough Sketch of the Program:*

Next we write any "important" sections of the program, assuming initialization in some form. In some sections we write portions of the actual code. In doing this, the actual intent of each piece of code may change several times, so that parts of our sketch may need rewriting.