UNC Report No.
TR-76-007
May 1976

STRUCTURED PROGRAMMING EXAMPLES FOR USE IN AN INTRODUCTORY BUSINESS PROGRAMMING COURSE

by

ALAN D. BERNARD

A thesis submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Master of Science in the Department of Computer Science.

Chapel Hill

1976

ACKNOWLEDGEMENTS

I would like to thank Dr. Donald F. Stanat for his advice and help during the development of this thesis. Dr. S. M. Pizer and Dr. M. Jazayeri also contributed comments and suggestions which greatly improved the final version. I am very grateful to all for their prompt review of early drafts. In addition I would like to thank my wife, Mary Jo, who acted as secretary and editor throughout the months of writing and revising this thesis.

ALAN D. BERNARD

Structured Programming Examples for Use in an Introductory Business Programming Course

(Under the direction of DONALD F. STANAT)

ABSTRACT

This thesis is designed for use by instructors of introductory programming courses. It contains examples demonstrating the use of stepwise refinement in problem solving. Although the examples are business oriented, they are written so that they can be understood by instructors and students without business backgrounds. A list of additional problems for refinement is included.

TABLE OF CONTENTS

Chapter 1 - Introduction

This thesis is intended for use by instructors of an introductory computer programming course. It contains several examples demonstrating the use of stepwise refinement in problem solving. Where language dependent, the language used is PL/1; however, the examples could readily be adapted to any general purpose language.

The examples are oriented toward students with a business background. Because of the diversity of students' backgrounds, however, I have written the examples in such a way that a business background is not necessary for understanding the problems. For example, in a problem dealing with depreciation of fixed assets, I have given the formulas for computing depreciation by the methods required, rather than assume that the students or instructors would be familiar with the techniques used.

An introductory business programming course poses many problems for teachers and students alike. The instructor in such a course is very often a graduate teaching assistant from the Department of Computer Science or the School of Business. Teaching the course may be the first classroom experience the instructor has had from the teacher's side of the desk. To further complicate matters, the instructor may have a very limited business background; he may be unfamiliar with the types of problems in which the students are likely to be interested.

Another problem faced by teaching assistants is lack of expertise in the programming language used in the course. First year graduate students recruited to teach may very well know less about the language used than their more knowledgeable students.

The students in an introductory course are faced with a variety of problems as well. The clatter of keypunch machines and the atmosphere of the computation center can be overwhelming for the novice programmer. As well as familiarize himself with a new routine, the student must learn new techniques of communicating. Although they have been solving problems all their lives, many students find a formal approach to problem solving difficult to implement. Learning a computer programming language presents difficulties, many more, perhaps, for students in a business curriculum than for those in science and math.

The best way to eliminate problems facing the students is to eliminate problems facing the instructors. G. Polya, in How To Solve It, says that there are two things every teacher should know: the material he is to teach, and a little bit more than that. The material contained in this thesis has been chosen to meet both of these demands.

Of the many problems the instructors face, most can be solved by the use of existing literature. There is ample material available concerning language syntax; there are many introductory texts on programming languages. These may

be used to supplement language details given in the course textbook.

But when we come to problem solving techniques, the existing literature is not adequate. To be sure, structured programming has become a popular topic in computer science in recent years. There have been many articles on topics ranging from how to comment programs to how to write GOTO-less programs. But these materials give an instructor of an introductory course very little help.

Most introductory programming textbooks cover language syntax. The better texts, however, also include discussions of program format, program documentation, program design, and program testing. The general approach to problem solving is very good, but there are simply not enough examples given to show a beginning student how to write good programs. It is the lack of good examples towards which this thesis is directed.

The problems in this thesis are presented in terms of a problem specification, problem clarification, and problem refinement. These correspond to a problem statement, explanation, and solution, respectively.

The problem specification is a statement of the problem we wish to solve. I have attempted to specify problems in a manner that is similar to what one can expect outside the classroom. Thus, there are many ambiguities which the programmer must resolve.

The section on problem clarification deals with these ambiquities. Examples are given to indicate what processing is required; the input and output are discussed in detail, with remarks on output which will be useful when testing and debugging the program.

The heart of each example is the stepwise refinement of the problem. Each refinement is presented in the form of a PL/1 comment outline. The comments in the outline are indented to show the refinement level of the subproblems they represent.

In most examples the problem is refined to the level above the introduction of PL/1 statements. In this way, I have tried to show that the approach to understanding and solving a problem is not entirely dependent upon the programming language to be used. The final refinements given could be used to write programs in any general purpose language.

The refinements are intended for use as models of program development, but there are other solutions which may be quite satisfactory. Instructors may choose to demonstrate this fact by presenting alternate methods of solution for one or more problems.

In the first example the problem is refined into a complete PL/1 program. The program uses some of the more complex PL/1 concepts, but instructors could easily modify it for use prior to the introduction of sophisticated PL/1

techniques (e. g., LIST I/O could be used in place of EDIT I/O). The format of the program will be as follows:

```
Comment describing function of program
Program-name:PROCEDURE OPTIONS(MAIN);
   Declaration of all variables
   Program statements
   END; /* Program-name */
```

The examples selected for refinement fall into three categories: simple business applications, simulation, and file processing. There are two examples in each category.

The first example in simple business applications involves calculating depreciation on fixed assets. The problem is relatively easy to solve, and it serves mainly as a vehicle for introducing the method of stepwise refinement.

The second example involves calculating various financial ratios. This problem is much more complex than that developed in the first example because the user has many options available. The problem becomes one of writing an interpreter for a special-purpose language.

The third and fourth problems involve simulation. The game of craps is simulated in the third example. It has been chosen to demonstrate the use of a random number generator. Because students show great interest in playing games on the computer, they can be expected to follow the class discussion of the problem very carefully, learning about problem refinement and simulation at the same time.

The fourth example is much more complex, but not too difficult if the previous example is understood. It involves simulating activity in a store to determine the optimum number of checkout counters. A great deal of clarification is needed before the problem specification can be understood. This problem is excellent for demonstrating the necessity of working a few examples before plunging into the refinement.

The problems selected for file processing are common business problems. The first of these is an inventory problem. The problem clarification and discussions of input and output are emphasized.

The sixth example is a payroll problem. Most students find payroll concepts very easy to understand, but the example demonstrates that there may be a great deal more to a problem than is first realized. The level of complexity must be controlled by the problem solver, and is virtually unlimited by the problem specification.

A list of additional problems is provided in the appendix. These problems are suitable for class assignments or term projects. It is my hope that more daring instructors will use these problems for class examples, in the same spirit as Phaedrus, in Zen and the Art of Motorcycle Maintenance:

> He felt that by exposing classes to his own sentences as he made them, with all the misgivings and hang-ups and erasures, he would give a more

honest picture of what writing was like than by spending class time picking nits in completed student work or holding up the completed works of masters for emulation.

Chapter 2 - Simple Business Applications

2.1 Depreciation

In the course of providing goods and services, most businesses acquire relatively long-lived resources, such as buildings, machinery, etc. These resources are called fixed assets. With some exceptions, fixed assets have a limited useful life. For example, a car cannot be expected to run forever, nor can a small warehouse be expected to satisfy a growing company's needs for a long period of time. During the time a fixed asset is used, the cost of the asset may be charged as an expense for tax purposes. The accounting process for reducing the value of a fixed asset by the estimated value "used up" is called depreciation.

In order to determine the depreciation on a fixed asset during a year, a company must determine four things:

1. Cost of the asset. This is the purchase price of the asset or the cost incurred in building the asset in-house.

2. Service life of the asset. This is the period of time during which the asset will be useful to the company.

3. Salvage value of the asset. This is the resale value of the asset at the end of its useful life.

4. Method of depreciation. There are several techniques for computing depreciation. In practice, we want to select the method which most accurately reflects the

usage of the asset. In this problem, we will compare three commonly used methods of depreciation.

Problem Specification

Compute the depreciation cost in each year of a fixed asset's service life. Use the following methods of depreciation for each asset:

1. Straight Line : This method treats an asset as if it provides the same amount of service in each year of its useful life. We charge an equal amount of the cost each year.

   Dep/yr = (cost - salvage value)/(service life).

An asset is often more useful in its early years than at the end of its service life. When this is the case, we want to use a depreciation method which reflects the greater usefulness in early life. The two methods below are examples of such methods:

2. Double Declining Balance : This method gives the fastest depreciation allowed under present tax laws. The depreciation each year is computed by taking a percentage of the book value of the asset at the beginning of the year. The book value is defined as the cost of the asset less accumulated depreciation. The percentage used (rate of depreciation) is double the rate used in the straight line method. Thus if an asset has a useful life of ten years, the depreciation

rate under the straight line method would be 10%/year; under the double declining balance method, the rate would be 20%/year. Depreciation in the last year is the difference between the book value and the salvage value.

Dep in yr 1 = (2/life) * cost

Dep in yr 2 = (2/life) * (cost - accum dep)

.
.
.

Dep in last yr = cost - accum dep - salvage value

3. Sum of Year's Digits : This method is used for assets that do not lose their usefulness as quickly as the double declining balance method would indicate.

sum = 1 + 2 + ... + (life-1) + life.

Dep in yr 1 = (life/sum)*(cost - salvage value)

Dep in yr 2 = ((life-1)/sum)*(cost - salvage value)

.
.
.

Dep in last yr = (1/sum)*(cost - salvage value)

Problem Clarification

Examples

It is important to impress upon students the need for working examples before starting the problem refinement. The details of the problem become clear once we calculate depreciation using the methods specified. Instructors may find it useful to work this example for all ten years of the asset's life.

Assume an asset costs $1100, has an expected useful life of 10 years, and has a salvage value of $100. The accumulated depreciation at the end of ten years will be cost - salvage value, which is $1000. This is the same for each of the three methods.

Straight Line Method

Using the equation given in the problem specification, depreciation/year= (1100-100)/10 = $100/year.

Double Declining Balance Method

The rate of depreciation is 2/10 = .20.

The undepreciated value in the first year is $1100. Depreciation in the first year = .20 * $1100 = $220.

The undepreciated value in the second year is $1100-220 = $880. Depreciation in the second year = .20 * 880 = $176.

Continuing in the same manner, the undepreciated value after 9 years is $147. The depreciation in the tenth year will be $147-100 = $47.

Sum of Years' Digits Method

The life of the asset is 10 years. The sum of the year's digits is 1+2+3+...+9+10=55. A better method for calculating the sum of the years' digits is to use the formula

$$1+2+\ldots+N=N(N+1)/2,$$

where N is the number of years.

Depreciation in year 1 = ((10-1+1)/55) * 1000 = $182.

Depreciation in year 2 = ((10-2+1)/55) * 1000 = $164.

And so forth.

Input

In working with the example, we saw that the input information must include the purchase cost, expected useful life, and salvage value of the asset. All other information needed can be computed from this information. In addition, we will include the name of the asset.

Although the problem specification indicates that we need only compute depreciation schedules for one asset, the program will be much more useful if we modify it to work for more than one asset.

Output

For each asset, we must print the amount of depreciation each year of its expected useful life, using each of the three methods. In addition, we will print the accumulated depreciation for each year using each method.

Problem Refinement

Having investigated the problem requirements, we will now develop a solution to the problem. We will do this by refining the problem into a sequence of subproblems. The sequence of subproblems must have the property that solving the sequence is equivalent to solving the original problem.

The value of stepwise refinement is that we can consider each subproblem as a problem in itself and refine it further. By doing this we can break the original problem into a sequence of subproblems, each of which can be readily solved. Let's begin with a statement of the basic problem.

```
/* COMPUTE DEPRECIATION ON FIXED ASSETS */
```

Since we expect more than one asset, this will be a repetitive process. We can refine each repetition into two subproblems:

```
/* COMPUTE DEPRECIATION ON FIXED ASSETS */
   /* REPEAT UNTIL ALL ASSETS PROCESSED */
      /* READ NAME,COST,LIFE,SALVAGE VALUE OF ASSET */
      /* COMPUTE DEPRECIATION ON THIS ASSET */
```

We will not refine the read subproblem further before we write program statements. We can now concentrate on /* COMPUTE DEPRECIATION ON THIS ASSET */. We will want to compute the depreciation for each year, comparing the three methods. This is a repetitive process, and for each year we will want to compute the depreciation and accumulated depreciation, and print the results.

```
/* COMPUTE DEPRECIATION ON THIS ASSET */
   /* REPEAT FOR EACH YEAR OF USEFUL LIFE */
      /* COMPUTE DEPRECIATION FOR CURRENT YEAR */
      /* AND ACCUMULATED DEPRECIATION          */
      /* PRINT RESULTS */
```

To refine /* COMPUTE DEPRECIATION FOR CURRENT YEAR AND ACCUMULATED DEPRECIATION */, we can simply indicate that we must compute depreciation by three methods.

```
/* REPEAT FOR EACH YEAR OF USEFUL LIFE */

   /* COMPUTE DEPRECIATION FOR CURRENT YEAR */
   /* AND ACCUMULATED DEPRECIATION          */

      /* STRAIGHT LINE METHOD */

      /* DOUBLE DECLINING BALANCE METHOD */

      /* SUM OF YEARS' DIGITS METHOD */

   /* PRINT RESULTS */
```

We will not refine the subproblems for each of the methods of depreciation at this point. We can do so when we are ready to write program statements.

The last refinement needed before we can write the program is to indicate what we want to print.

```
/* PRINT RESULTS */

   /* PRINT DEPRECIATION AND ACCUMULATED */
   /* DEPRECIATION FOR EACH METHOD       */
```

The refinement up to this point is listed below:

```
/* COMPUTE DEPRECIATION ON FIXED ASSETS */

   /* REPEAT UNTIL ALL ASSETS PROCESSED */

      /* READ NAME,COST,LIFE,SALVAGE VALUE OF ASSET */

      /* COMPUTE DEPRECIATION ON THIS ASSET */

         /* REPEAT FOR EACH YEAR OF USEFUL LIFE */

            /* COMPUTE DEPRECIATION FOR CURRENT YEAR */
            /* AND ACCUMULATED DEPRECIATION          */
```

```
      /* STRAIGHT LINE METHOD */

      /* DOUBLE DECLINING BALANCE METHOD */

      /* SUM OF YEARS' DIGITS METHOD */

   /* PRINT RESULTS */

      /* PRINT DEPRECIATION AND ACCUMULATED */
      /* DEPRECIATION FOR EACH METHOD       */
```

Program Development

Now that we have a detailed outline of the problem solution, we can develop a complete PL/1 program. We will use the comment outline as the skeleton for the PL/1 program.

As mentioned in the introduction, our program will have the following structure:

```
Comment describing function of program
Program-name:PROCEDURE OPTIONS(MAIN);
   Declaration of all variables
   Program statements
   END; /* Program-name */
```

We will take the first comment in the outline as the description of the program. Selecting a reasonable name for our program, we can continue the program development, as follows:

```
/* COMPUTE DEPRECIATION ON FIXED ASSETS */
DEPREC:PROCEDURE OPTIONS(MAIN);
   /* REPEAT UNTIL ALL ASSETS PROCESSED */
      /* READ NAME,COST,LIFE,SALVAGE VALUE OF ASSET */
```

```
      /* COMPUTE DEPRECIATION ON THIS ASSET */

         /* REPEAT FOR EACH YEAR OF USEFUL LIFE */

            /* COMPUTE DEPRECIATION FOR CURRENT YEAR */
            /* AND ACCUMULATED DEPRECIATION          */

               /* STRAIGHT LINE METHOD */

               /* DOUBLE DECLINING BALANCE METHOD */

               /* SUM OF YEARS' DIGITS METHOD */

            /* PRINT RESULTS */

               /* PRINT DEPRECIATION AND ACCUMULATED */
               /* DEPRECIATION FOR EACH METHOD       */

   END; /* DEPREC */
```

The first design decision comes when we look at

```
/* REPEAT UNTIL ALL ASSETS PROCESSED */

   /* READ NAME,COST,LIFE,SALVAGE VALUE OF ASSET */

   /* COMPUTE DEPRECIATION ON THIS ASSET */
```

Because this is a repetitive process, we will want to use a DO loop. There are three ways we can determine when to exit from the loop.

1. Count the number of assets to be processed, and include the count as the first item in the input; use the count to control the number of items read and processed.

We would like to avoid this method because counting is a tedious chore, and miscounting would cause errors in the processing.

2. Add a dummy item as the last item in the input.

When this item is read, we know that all assets have been processed.

This method is preferable to the first, but it can only be used if we can find a dummy value which would never appear as valid input to the program. While this method is suitable for this problem, instructors will find the third method useful in introducing the ENDFILE condition. In addition, the instructor can discuss the concept of maintaining the integrity of the file - should we allow invalid data in our asset file if it can be avoided?

3. Stop processing when there are no more assets to be processed.

This is the most natural way to stop the program. The PL/1 ENDFILE condition will signal that there is no more data in the input stream. We will use the ENDFILE condition to terminate the loop.

Using the ENDFILE condition to terminate the loop, we have two possibilities:

1. ON ENDFILE go out of loop
   DO WHILE (forever);
   read asset data
   process this asset
   END;
2. read asset data
   DO WHILE (ENDFILE condition not raised);
   process this asset

read next asset data

END;

We will select the second method because it reflects the basic iterative process:

1. Set the condition (read asset data)
2. Test the condition (DO WHILE ...)
3. Perform the body of the loop (process this asset)
4. Modify the value of the condition (read next asset data)
5. Go back to step 2. (END)

Having selected the approach we will use, we can now refine our program to the following level:

```
/* COMPUTE DEPRECIATION ON FIXED ASSETS */
DEPREC:PROCEDURE OPTIONS(MAIN);
   DECLARE ANOTHER_ASSET FIXED DECIMAL; /* STOP FLAG */
   ANOTHER_ASSET = 1;
   ON ENDFILE(SYSIN) ANOTHER_ASSET = 0;
   /* READ NAME,COST,LIFE,SALVAGE VALUE OF ASSET */
   /* REPEAT UNTIL ALL ASSETS PROCESSED */
   DO WHILE (ANOTHER_ASSET=1);
      /* COMPUTE DEPRECIATION ON THIS ASSET */
         /* REPEAT FOR EACH YEAR OF USEFUL LIFE */
            /* COMPUTE DEPRECIATION FOR CURRENT YEAR */
            /* AND ACCUMULATED DEPRECIATION          */
               /* STRAIGHT LINE METHOD */
               /* DOUBLE DECLINING BALANCE METHOD */
               /* SUM OF YEARS' DIGITS METHOD */
```

```
            /* PRINT RESULTS */

               /* PRINT DEPRECIATION AND ACCUMULATED */
               /* DEPRECIATION FOR EACH METHOD       */

         /* READ NAME,COST,LIFE,SALVAGE VALUE OF ASSET */

         END;

      END; /* DEPREC */
```

In the refinements that follow, we will list only the parts of the program that are being refined and the declarations of the variables introduced in the refinements. These declarations and refinements will then be combined and collated fo form the final program.

We can now refine /* READ NAME,COST,LIFE,SALVAGE VALUE OF ASSET */. We will select variable names of NAME, COST, LIFE, and SALVAGE for the values to be read. NAME will hold character strings of length up to 15 characters; LIFE will hold integer values. We are not sure what the largest cost can be, so we will declare COST and SALVAGE as FLOAT DECIMAL.

We can use LIST or EDIT input. To eliminate the use of quotes around asset names, we will use EDIT input. The format used must be reasonable for the values to be read.

```
DECLARE NAME CHARACTER(15); /* ASSET NAME */
DECLARE COST FLOAT DECIMAL; /* INITIAL COST OF ASSET */
DECLARE LIFE FIXED DECIMAL; /* YEARS ASSET CAN BE USED*/
DECLARE SALVAGE FLOAT DECIMAL; /* SALVAGE VALUE */

/* READ NAME,COST,LIFE,SALVAGE VALUE OF ASSET */
   GET EDIT (NAME,COST,LIFE,SALVAGE)
       (COL(1),A(15),X(1),F(8),X(1),F(2),X(1),F(6));
```

This refinement will be used in both places in the program where the read comment occurs. The variables will be declared only once, however.

We can now refine /* REPEAT FOR EACH YEAR OF USEFUL LIFE */. This is a repetitive process and will be replaced by a DO loop. Although we can use a DO WHILE loop, the self-incrementing loop will better reflect the activity.

```
DECLARE YEAR FIXED DECIMAL;
      /* REPEAT FOR EACH YEAR OF USEFUL LIFE */
      DO YEAR = 1 TO LIFE;
         /* COMPUTE DEPRECIATION FOR CURRENT YEAR */
         /* AND ACCUMULATED DEPRECIATION          */
            /* STRAIGHT LINE METHOD */
            /* DOUBLE DECLINING BALANCE METHOD */
            /* SUM OF YEARS' DIGITS METHOD */
         /* PRINT RESULTS */
            /* PRINT DEPRECIATION AND ACCUMULATED */
            /* DEPRECIATION FOR EACH METHOD       */
         END;
```

We can now insert the code for computing depreciation and accumulated depreciation. The code follows from the formulas given for each method of depreciation. (Instructors may want to expand the comments used in declare statements since they will have 72 card columns available rather than the 60 column limitation of this paper.)

```
DECLARE STRT_DEP FLOAT DECIMAL; /* STRAIGHT-LINE */
```

```
DECLARE TOTAL_STRT FLOAT DECIMAL; /* ACCUMULATED STRT */
DECLARE BOOK_VALUE FLOAT DECIMAL; /* COST-DEPRECIATION*/
DECLARE DBL_DEP FLOAT DECIMAL; /* DOUBLE DECLINING */
DECLARE TOTAL_DBL FLOAT DECIMAL; /* ACCUMULATED DBL */
DECLARE SUM_OF_DGTS FIXED DECIMAL; /* 1+2+...+LIFE */
DECLARE DGTS_DEP FLOAT DECIMAL; /* SUM OF YEARS' */
DECLARE TOTAL_DGTS FLOAT DECIMAL; /* ACCUMULATED YEARS*/

        /* COMPUTE DEPRECIATION FOR CURRENT YEAR */
        /* AND ACCUMULATED DEPRECIATION          */

           /* STRAIGHT LINE METHOD */

              STRT_DEP=(COST-SALVAGE)/LIFE;
              TOTAL_STRT = TOTAL_STRT + STRT_DEP;

           /* DOUBLE DECLINING BALANCE METHOD */

              BOOK_VALUE = COST - TOTAL_DBL;
              IF YEAR ¬= LIFE
                 THEN DBL_DEP=2*BOOK_VALUE/LIFE;
                 ELSE DBL_DEP=BOOK_VALUE-SALVAGE;
              TOTAL_DBL=TOTAL_DBL + DBL_DEP;

           /* SUM OF YEARS' DIGITS METHOD */
              SUM_OF_DGTS=LIFE*(LIFE+1)/2;
              DGTS_DEP=(LIFE-YEAR+1)*(COST-SALVAGE)
                       /SUM_OF_DGTS;
              TOTAL_DGTS=TOTAL_DGTS + DGTS_DEP;
```

We are not yet finished with this part of the program. The problem is that we must initialize the values for accumulated depreciation. We will want them to be set to 0 each time we process a new asset. The initialization will have to come before /* REPEAT FOR EACH YEAR OF USEFUL LIFE */, but after DO WHILE (ANOTHER_ASSET=1).

```
DO WHILE (ANOTHER_ASSET=1);
   /* COMPUTE DEPRECIATION ON THIS ASSET */
      TOTAL_STRT=0; TOTAL_DBL=0; TOTAL_DGTS=0;
      /* REPEAT FOR EACH YEAR OF USEFUL LIFE */
```

```
DO YEAR = 1 TO LIFE;

   /* COMPUTE DEPRECIATION FOR CURRENT YEAR */
   /* AND ACCUMULATED DEPRECIATION          */

      /* STRAIGHT LINE METHOD */

         STRT_DEP=(COST-SALVAGE)/LIFE;
         TOTAL_STRT = TOTAL_STRT + STRT_DEP;

      /* DOUBLE DECLINING BALANCE METHOD */

         BOOK_VALUE = COST - TOTAL_DBL;
         IF YEAR ¬= LIFE
            THEN DBL_DEP=2*BOOK_VALUE/LIFE;
            ELSE DBL_DEP=BOOK_VALUE-SALVAGE;
         TOTAL_DBL=TOTAL_DBL + DBL_DEP;

      /* SUM OF YEARS' DIGITS METHOD */
         SUM_OF_DGTS=LIFE*(LIFE+1)/2;
         DGTS_DEP=(LIFE-YEAR+1)*(COST-SALVAGE)
                  /SUM_OF_DGTS;
         TOTAL_DGTS=TOTAL_DGTS + DGTS_DEP;

   /* PRINT RESULTS */

      /* PRINT DEPRECIATION AND ACCUMULATED */
      /* DEPRECIATION FOR EACH METHOD       */

   END;
```

We can now add the PL/1 statements to print our results. Instructors should stress the fact that the output must be carefully laid out rather than haphazardly printed. We want the output to be in a readable format, with variables printed in the most useful order.

```
/* PRINT DEPRECIATION AND ACCUMULATED */
/* DEPRECIATION FOR EACH METHOD       */

   PUT SKIP EDIT (YEAR,STRT_DEP,DBL_DEP,
                  DGTS_DEP,TOTAL_STRT,
                  TOTAL_DBL,TOTAL_DGTS)
                 (F(5),X(2),F(11,2),X(4),
                  F(11,2),X(7),F(11,2),
                  COL(56),F(11,2),COL(71),
```

```
                                        F(11,2),COL(90),F(11,2));
```

The output will not be very useful unless we indicate which asset we are processing and print headings over each column. This will have to be done before the printing of the values. We can print headings before we compute the depreciation for an asset.

```
/* PRINT HEADING */
   PUT SKIP(3) EDIT (NAME,'COST=',COST,
                     'SERVICE LIFE=',LIFE,'YEARS',
                     'SALVAGE VALUE=',SALVAGE)
                    (A,X(3),A,F(8),X(3),A,F(2),
                     X(1),A,X(3),A,F(6));

   PUT SKIP EDIT ('CURRENT DEPRECIATION',
                  'ACCUMULATED DEPRECIATION')
                 (COL(19),A,COL(67),A);

   PUT SKIP EDIT ('STRAIGHT','DOUBLE DECLINING',
                  'SUM OF YEARS''','STRAIGHT',
                 'DOUBLE DECLINING','SUM OF YEARS''')
                 (COL(10),A,COL(21),A,COL(40),A,
                  COL(58),A,COL(69),A,COL(89),A);

   PUT SKIP EDIT ('YEAR','LINE','BALANCE','DIGITS',
                                'LINE','BALANCE','DIGITS')
                (COL(3),A,COL(12),A,COL(24),A,COL(43),
                 A,COL(60),A,COL(72),A,COL(92),A);

/* COMPUTE DEPRECIATION ON THIS ASSET */
```

Improvements

We now have a complete PL/1 program for this problem. However, there are several changes we may want to make. In order to make the program more efficient, we could reduce output by printing headings only once per page. This would involve the use of the ENDPAGE condition and would not be advisable in an introductory course.

We will, however, want to remove calculations from loops where possible. In this way, we can perform calculations once instead of many times. The place to look is in the innermost loop, DO YEAR=1 TO LIFE.

Under /* STRAIGHT LINE METHOD */, we are computing STRT_DEP = (COST - SALVAGE)/LIFE. Since COST, SALVAGE, and LIFE do not change values during the computation, we can move this statement outside the loop.

In computing double declining balance depreciation, we compute DBL_DEP = 2*BOOK_VALUE/LIFE. We can set DBL_RATE=2E0/LIFE outside the loop, and change the calculation to DBL_DEP = DBL_RATE*BOOK_VALUE. Note that we must use 2E0/LIFE rather than 2/LIFE because we want a floating point result.

The final changes occur in computing sum of years' digits depreciation. We can move the calculation of SUM_OF_DGTS outside the loop. Furthermore, we can compute USEFUL_VALUE = COST - SALVAGE outside the loop and change the calculations for STRT_DEP and DGTS_DEP to use USEFUL_VALUE rather than COST - SALVAGE.

After adding DBL_RATE and USEFUL_VALUE to the declare statements, we can group the declare statements so that they are more readable.

The final program appears below.

```
/* COMPUTE DEPRECIATION ON FIXED ASSETS */
DEPREC:PROCEDURE OPTIONS(MAIN);
```

```
DECLARE NAME CHARACTER(15), /* ASSET NAME */
       (COST, /* INITIAL COST OF ASSET */
        SALVAGE) /* SALVAGE VALUE OF ASSET */
                 FLOAT DECIMAL,
        LIFE FIXED DECIMAL, /* YEARS ASSET USEFUL */

       (STRT_DEP,   /* STRAIGHT LINE DEPRECIATION */
        TOTAL_STRT, /* AND ACCUMULATED DEPRECIATION */

        DBL_DEP,   /* DOUBLE DECLINING BALANCE */
        TOTAL_DBL, /* AND ACCUMULATED DEPRECIATION */

        DGTS_DEP,   /* SUM OF YEARS' DIGITS DEP */
        TOTAL_DGTS, /* AND ACCUMULATED DEPRECIATION */

        BOOK_VALUE, /* COST-ACCUMULATED DEPRECIATION */
        DBL_RATE, /* HIGHEST ALLOWABLE RATE */
                  /* OF DEPRECIATION */
        SUM_OF_DGTS, /* 1+2+...+LIFE */
        USEFUL_VALUE) /* COST-SALVAGE VALUE */
                      FLOAT DECIMAL,

        ANOTHER_ASSET FIXED DECIMAL, /* END OF DATA */
        YEAR FIXED DECIMAL; /* LOOP INDEX VARIABLE */

ANOTHER_ASSET = 1;
ON ENDFILE(SYSIN) ANOTHER_ASSET = 0;

/* READ NAME,COST,LIFE,SALVAGE VALUE OF ASSET */
   GET EDIT (NAME,COST,LIFE,SALVAGE)
       (COL(1),A(15),X(1),F(8),X(1),F(2),X(1),F(6));

/* REPEAT UNTIL ALL ASSETS PROCESSED */

DO WHILE (ANOTHER_ASSET=1);

   /* PRINT HEADING */
      PUT SKIP(3) EDIT (NAME,'COST=',COST,
                        'SERVICE LIFE=',LIFE,'YEARS',
                        'SALVAGE VALUE=',SALVAGE)
                       (A,X(3),A,F(8),X(3),A,F(2),
                        X(1),A,X(3),A,F(6));

      PUT SKIP EDIT ('CURRENT DEPRECIATION',
                     'ACCUMULATED DEPRECIATION')
                    (COL(19),A,COL(67),A);

      PUT SKIP EDIT ('STRAIGHT','DOUBLE DECLINING',
                     'SUM OF YEARS''','STRAIGHT',
                    'DOUBLE DECLINING','SUM OF YEARS''')
                    (COL(10),A,COL(21),A,COL(40),A,
                     COL(58),A,COL(69),A,COL(89),A);
```

```
      PUT SKIP EDIT ('YEAR','LINE','BALANCE','DIGITS',
                              'LINE','BALANCE','DIGITS')
                    (COL(3),A,COL(12),A,COL(24),A,COL(43),
                     A,COL(60),A,COL(72),A,COL(92),A);

   /* COMPUTE DEPRECIATION ON THIS ASSET */

      TOTAL_STRT=0; TOTAL_DBL=0; TOTAL_DGTS=0;

      USEFUL_VALUE=COST-SALVAGE;
      DBL_RATE=2E0/LIFE;
      SUM_OF_DGTS=LIFE*(LIFE+1)/2;

      /* STRAIGHT LINE DEPRECIATION PER YEAR */
         STRT_DEP=USEFUL_VALUE/LIFE;

      /* REPEAT FOR EACH YEAR OF USEFUL LIFE */

      DO YEAR = 1 TO LIFE;

         /* COMPUTE DEPRECIATION FOR CURRENT YEAR */
         /* AND ACCUMULATED DEPRECIATION          */

            /* STRAIGHT LINE METHOD */
               TOTAL_STRT = TOTAL_STRT + STRT_DEP;

            /* DOUBLE DECLINING BALANCE METHOD */

               BOOK_VALUE = COST - TOTAL_DBL;
               IF YEAR ¬= LIFE
                  THEN DBL_DEP=DBL_RATE*BOOK_VALUE;
                  ELSE DBL_DEP=BOOK_VALUE-SALVAGE;
               TOTAL_DBL=TOTAL_DBL + DBL_DEP;

            /* SUM OF YEARS' DIGITS METHOD */
               DGTS_DEP=(LIFE-YEAR+1)*USEFUL_VALUE
                        /SUM_OF_DGTS;
               TOTAL_DGTS=TOTAL_DGTS + DGTS_DEP;

         /* PRINT DEPRECIATION AND ACCUMULATED */
         /* DEPRECIATION FOR EACH METHOD       */

            PUT SKIP EDIT (YEAR,STRT_DEP,DBL_DEP,
                           DGTS_DEP,TOTAL_STRT,
                           TOTAL_DBL,TOTAL_DGTS)
                          (F(5),X(2),F(11,2),X(4),
                           F(11,2),X(7),F(11,2),
                           COL(56),F(11,2),COL(71),
                           F(11,2),COL(90),F(11,2));

         END;
```

```
      /* READ NAME,COST,LIFE,SALVAGE VALUE OF ASSET */
         GET EDIT (NAME,COST,LIFE,SALVAGE)
              (COL(1),A(15),X(1),F(8),X(1),F(2),X(1),F(6));

      END;

   END; /* DEPREC */
```

2.2 Financial Ratios

Financial analysis revolves around two major accounting reports - the firm's balance sheet and its income statement. The balance sheet is a statement of the firm's financial condition at a specified point in time (e.g. the end of the year). The income statement is a record of the firm's activity during a period of time (e.g. one year). Financial analysts relate the two reports by means of financial ratios.

A financial analyst may be interested in one particular ratio. For instance, he may want to know the return on investment for a firm, in which case, he would look at the return on net worth ratio (net profit after taxes/net worth).

In some situations the analyst may be interested in a group of ratios. For instance, before a banker gives a short term loan, he may want to know how quickly the firm's assets can be turned into cash and whether this cash can be used to repay the loan. For this purpose, he will look at the ratio of current assets (assets which are expected to be converted into cash in a short period of time) to current liabilities (debts which must be paid in a short period of time). He also may want to look at the ratio of (current assets - inventory) / current liabilities.

The finance executive in a company will want to see how his company compares to others in the same industry. To do this, he can compare his company's ratios with the averages of the financial ratios of companies in his industry.

We will deal with a problem to compute the more commonly used ratios.

Problem Specification

Compute financial ratios requested by the user. The ratios are divided into the following categories:

1. Liquidity Ratios

   Current: (current assets) / (current liabilities)

   Quick: (current assets - inventory) / (current liabilities)

2. Leverage Ratios

   Debt: (total liability) / (total assets)

   Interest: (profit before taxes + interest charges) / (interest charges)

3. Activity Ratios

   Inventory Turnover: sales / inventory

   Collection Period: receivables / (sales per day)

   Fixed Asset Turnover: sales / (fixed assets)

   Total Asset Turnover: sales / (total assets)

4. Profitability Ratios:

   Profit Margin on Sales: (net profit after taxes) / sales

Return on Total Assets: (net profit after taxes) / (total assets)

Return on Net Worth: (net profit after taxes) / (net worth)

Problem Clarification

The ratios to be calculated are standard ratios used in managerial finance. The figures used in the ratios come from balance sheets and income statements. The breakdown is as follows:

Balance Sheet (at a specified point in time)	Income Statement (during the specified period)
receivables	sales
inventory	interest charges
current assets	profit before taxes
fixed assets	net profit after taxes
total assets	
current liabilities	
total liability	
net worth	

Based on our discussion of the uses of the ratios, we can anticipate the requests we will receive from users. We should have the capability of printing one ratio, a group of ratios, or all the ratios. In addition, the user should be able to specify the industry average for any ratio requested, and our program should be able to print the company ratio alongside the industry average.

We can expect the user to make more than one request, so we should write the program in such a way that many requests can be processed. It is also possible that the user will want to compute ratios for more than one company, so we should allow him to read values from more than one set of financial statements.

Input

The input will consist of commands to calculate various ratios. The previous discussion indicates that we should supply command names for each group of ratios (e.g. a command named LIQUIDITY would require that we print the current ratio and the quick ratio). Commands should specify whether the industry average is to be printed.

In addition, we must supply a command to read balance sheet and income statement figures. We will also need to read the industry averages.

Output

We should print the name and value of every ratio we compute. When a group of ratios is requested, we should print the name of the group, and the names and values of the ratios in the group. We should also print the industry average when it is requested.

Problem Refinement

A straightforward statement of the problem is

/* COMPUTE FINANCIAL RATIOS */.

This is a repetitive problem which can be broken into two parts.

```
/* COMPUTE FINANCIAL RATIOS */
   /* REPEAT UNTIL NO MORE COMMANDS */
      /* READ A COMMAND */
      /* EXECUTE THE COMMAND */
```

The problem /* EXECUTE THE COMMAND */ can be broken into parts according to the three basic types of commands.

```
      /* EXECUTE THE COMMAND */
         /* COMMANDS TO READ DATA */
         /* SINGLE RATIO COMMANDS */
         /* GROUP COMMANDS */
```

There are three commands to read data. The commands and the information read for each of them are given in the next refinement.

```
         /* COMMANDS TO READ DATA */
            /* BALANCE SHEET COMMAND */
               /* READ VALUES FOR RECEIVABLES,INVENTORY, */
               /* CURRENT ASSETS, FIXED ASSETS,          */
               /* TOTAL ASSETS, CURRENT LIABILITIES,     */
               /* TOTAL LIABILITY, NET WORTH             */
               /* PRINT NAMES AND VALUES OF ITEMS READ */
            /* INCOME STATEMENT COMMAND */
               /* READ VALUES FOR SALES,INTEREST CHARGES,*/
               /* PROFIT BEFORE TAXES,                   */
               /* PROFIT AFTER TAXES                     */
```

```
            /* PRINT NAMES AND VALUES OF ITEMS READ */
         /* INDUSTRY AVERAGES COMMAND */
            /* READ INDUSTRY AVERAGES FOR ALL RATIOS */
```

The second type of command asks for a single ratio to be computed. There will be a routine for each ratio, but the refinement of each is the same.

```
      /* SINGLE RATIO COMMANDS */
         /* PRINT NAME OF RATIO */
         /* COMPUTE AND PRINT VALUE OF RATIO */
         /* IF REQUESTED, PRINT INDUSTRY AVERAGE */
         /* FOR RATIO                            */
```

The last group contains commands to execute more than one ratio. The groups conform to the types of ratios as indicated in the problem specification.

```
      /* GROUP COMMANDS */
         /* LIQUIDITY RATIOS */
         /* LEVERAGE RATIOS */
         /* ACTIVITY RATIOS */
         /* PROFITABILITY RATIOS */
         /* ALL RATIOS */
```

We must now refine each of the group headings. There are two essential activities in each group: print the name of the group, and compute the ratios included in each group. When we compute each ratio, we will perform the same

activities as we do when the command for that ratio is given. It is certainly worth pointing out to the students that if we use subroutines for the single ratios, this segment of the program can be written almost exclusively with subroutine calls.

```
/* LIQUIDITY RATIOS */

   /* PRINT NAME OF GROUP */

   /* COMPUTE CURRENT AND QUICK RATIOS */

/* LEVERAGE RATIOS */

   /* PRINT NAME OF GROUP */

   /* COMPUTE DEBT AND INTEREST RATIOS */

/* ACTIVITY RATIOS */

   /* PRINT NAME OF GROUP */

   /* COMPUTE INVENTORY TURNOVER, COLLECTION */
   /* PERIOD, FIXED ASSET TURNOVER,          */
   /* TOTAL ASSETS TURNOVER                  */

/* PROFITABILITY RATIOS */

   /* PRINT NAME OF GROUP */

   /* COMPUTE PROFIT MARGIN ON SALES, */
   /* RETURN ON TOTAL ASSETS,         */
   /* RETURN ON NET WORTH             */

/* ALL RATIOS */

   /* COMPUTE LIQUIDITY, LEVERAGE, ACTIVITY, */
   /* AND PROFITABILITY RATIOS               */
```

The complete refinement is listed below.

```
/* COMPUTE FINANCIAL RATIOS */

   /* REPEAT UNTIL NO MORE COMMANDS */

      /* READ A COMMAND */
```

```
/* EXECUTE THE COMMAND */

   /* COMMANDS TO READ DATA */

      /* BALANCE SHEET COMMAND */

         /* READ VALUES FOR RECEIVABLES,INVENTORY, */
         /* CURRENT ASSETS, FIXED ASSETS,          */
         /* TOTAL ASSETS, CURRENT LIABILITIES,     */
         /* TOTAL LIABILITY, NET WORTH             */

         /* PRINT NAMES AND VALUES OF ITEMS READ */

      /* INCOME STATEMENT COMMAND */

         /* READ VALUES FOR SALES,INTEREST CHARGES,*/
         /* PROFIT BEFORE TAXES,                   */
         /* PROFIT AFTER TAXES                     */

         /* PRINT NAMES AND VALUES OF ITEMS READ */

      /* INDUSTRY AVERAGES COMMAND */

         /* READ INDUSTRY AVERAGES FOR ALL RATIOS */

   /* SINGLE RATIO COMMANDS */

      /* PRINT NAME OF RATIO */

      /* COMPUTE AND PRINT VALUE OF RATIO */

      /* IF REQUESTED, PRINT INDUSTRY AVERAGE */
      /* FOR RATIO                            */

   /* GROUP COMMANDS */

      /* LIQUIDITY RATIOS */

         /* PRINT NAME OF GROUP */

         /* COMPUTE CURRENT AND QUICK RATIOS */

      /* LEVERAGE RATIOS */

         /* PRINT NAME OF GROUP */

         /* COMPUTE DEBT AND INTEREST RATIOS */

      /* ACTIVITY RATIOS */

         /* PRINT NAME OF GROUP */
```

```
            /* COMPUTE INVENTORY TURNOVER, COLLECTION */
            /* PERIOD, FIXED ASSET TURNOVER,         */
            /* TOTAL ASSETS TURNOVER                 */

         /* PROFITABILITY RATIOS */

            /* PRINT NAME OF GROUP */

            /* COMPUTE PROFIT MARGIN ON SALES, */
            /* RETURN ON TOTAL ASSETS,         */
            /* RETURN ON NET WORTH             */

         /* ALL RATIOS */

            /* COMPUTE LIQUIDITY, LEVERAGE, ACTIVITY, */
            /* AND PROFITABILITY RATIOS               */
```

Program Development

The refinement of /* EXECUTE THE COMMAND */ shows what must be done when we execute a particular command, but it does not show how to determine which command must be executed. There are several possibilities, some of which are very simple, and some of which are very sophisticated.

The most obvious way to determine which command to execute involves the use of IF statements. The command read can be compared to each of the possible command names. When a match is found, the appropriate subroutine can be called. If no match is found, an error message should be printed indicating the use of an invalid command name.

```
/* EXECUTE THE COMMAND */

   /* COMMANDS TO READ DATA */

      IF command-name = 'BALANCE' THEN
         CALL subroutine to read BALANCE SHEET data;

ELSE IF command-name = 'INCOME' THEN
         CALL subroutine to read INCOME STATEMENT data;
```

```
   /* SINGLE RATIO COMMANDS */

ELSE IF command-name = 'QUICK' THEN
        CALL subroutine to compute QUICK RATIO;

             .
             .
             .

   /* GROUP RATIOS */

ELSE IF command-name = 'LIQUIDITY' THEN
        CALL subroutine to compute LIQUIDITY RATIOS;

             .
             .
             .

ELSE print invalid command name message;
```

Instructors may choose to use the above control structure with or without using subroutines. It may be useful to use this example before subroutines are introduced to the class and modify it to use subroutines later. In this way the instructor could give practice in writing subroutines while demonstrating their value.

If students are already proficient in the use of nested IF statements, instructors may choose to introduce the use of label variables or entry variables. While these techniques could be discussed in class, it may not be appropriate to use them in assignments in an introductory class.

Program Modification

Due to incorrect assumptions or changes in the problem specification, programs used in a business environment often

need to be modified many times during their useful life. One of the advantages of using stepwise refinement is that it simplifies program modification.

The most likely change in this program would be the addition of new ratios. Suppose, for example, that we wish to add a ratio to determine whether a business has enough cash on hand to meet its current liabilities. We will call this ratio CASH, and we will compute it as

CASH = (cash on hand) / (current liabilities).

The first modifications to the refinement are under /* COMMANDS TO READ DATA */. We will have to read a value for cash on hand. Because this is an item on the balance sheet we will modify the Balance Sheet Command to read CASH ON HAND in addition to the values presently read. Although we do not have to change the problem refinement of /* INDUSTRY AVERAGES COMMAND */, we will have to modify the actual subroutine so that we read an average for the CASH ratio.

Since CASH is a new ratio, we will have to add a subroutine to compute it. It will have the same basic structure as the other SINGLE RATIO COMMANDS.

It is likely that we will also want to include CASH in one of the groups of ratios. It naturally falls into the category of LIQUIDITY RATIOS.

These are the only changes necessary in the problem refinement. We will, however, have to modify the control

structure to allow execution of the new command. Using the nested IF statements this will simply involve the addition of a statement to test for the command name CASH.

Chapter 3 - Simulation

3.1 Craps Game

Most businesses are influenced by events which are not directly under their control. For example, the rental of hotel rooms in a resort area is closely related to the weather during the peak season, a factor over which the hotel industry has very little control. When a business cannot control events, it is often of great importance to be able to predict their effect.

Simulation is one method used to estimate what will happen under uncertain conditions. We will demonstrate simulation techniques in a problem concerned with a popular game of chance.

Problem Specification

Simulate the play at a craps game. Craps is a game played with two dice, each of which has faces numbered one through six. Rolling both dice gives a number ranging from 2 to 12. The rules of the game are as follows: Roll the dice. If the roll is a 7 or 11, you win the game. If the roll is a 2, 3, or 12, you lose the game. Otherwise the number rolled is called the point. Continue rolling the dice until you win by rolling the point again, or you lose by rolling a 7.

There are several possible side bets in the game, but we will restrict ourselves to the basic game as described above.

Problem Clarification

The problem specification omits much useful information, including such points as the input, output, or number of games to play. We will have to make many assumptions in order to write a satisfactory program. These assumptions can best be made after looking at several examples of craps games.

Examples

Investigation of the rules indicates that there are four possible results: win on the first roll, lose on the first roll, win on some roll after the first roll, or lose on some roll after the first roll. Following is a set of examples illustrating the four possible outcomes.

first roll: 11	You win on the first roll.
first roll: 3	You lose on the first roll.
first roll: 6	6 becomes the point, continue rolling.
additional rolls: 5,3,11,8,6	You win because the point was rolled again before a 7 was rolled.
first roll: 9	9 becomes the point, continue rolling.
additional rolls: 6,8,6,7	You lose because 7 was rolled before the point was rolled again.

Input

No input is specified in the problem. It is possible that we will want to read values for the rolls of the dice, but this would be recording the play at a craps game rather than simulating it. To simulate play, we should have the computer "throw the dice" itself. This can be done by using a random number generator. The random number generator should produce two numbers between one and six, one for each die.

We will use no input to this program but will construct the program in such a way that this does not compromise its versatility. Instructors should emphasize the fact that input could be used during the development and testing of the program. We could write the program and use numbers from input rather than from a random number generator for the rolls of the dice. This would allow us to control the roll of the dice in such a way that we could test the program to see if it behaves as it should when given various sequences of numbers. Once we were sure of the reliability of the program, we could substitute a random number generator in the final program. Proper problem refinement should make the interchanging of methods of rolling the dice trivial.

Output

As with input, the problem specification says nothing about the output of the program. What to print and how to print it are left to our discretion. There are, however, certain obvious choices for output.

We will certainly want to print how many games were played, and of these games, how many were won and how many were lost. In addition, we may want to print the number of games won as a percentage of the total number of games played.

Most of the excitement of craps comes from watching the dice, not just finding out the result of the game. For this reason, we should also print the value of each roll of the dice. We should indicate the beginning and end of a game and whether the game was won or lost. This information will also be useful when testing the program.

Termination

We have not yet decided how long we should play the game. There are two reasonable stopping criteria. The first is to play until the number of games won exceeds the number of games lost by some amount x, or vice versa. This would correspond to a player starting with x dollars and playing for one dollar per game until he had lost his bankroll or doubled it.

The second criterion is to set a limit of y games to be played. This would correspond to a player setting a limit on the number of games he will play or the amount of time he will play. There is, however, a more important reason for setting a maximum number of games, and that is to insure that the computer program will halt in a reasonable length of time. Probability theory tells us that eventually the player must go broke or double his money (stopping criterion 1); however, the length of time for this to occur may be arbitrarily long. Due to computer costs, we must be able to control the amount of time our program runs.

This reason is sufficient in itself, but there is a much more subtle reason for using the second stopping criterion. Random number generators do not generate an arbitrarily long sequence of random numbers, but rather, a series of random numbers that is repeated. Although it is unlikely, it is certainly possible that the random number generator used may generate a sequence of numbers such that the required difference between the number of games won and the number of games lost is never reached. That is, the program would never halt.

This discussion indicates that it is imperative that we use the second stopping criterion (limit the number of games played); however, due to the nature of the problem, we will use the first criterion as well. Thus we will play until we have doubled our money, lost our money, or played the

maximum number of games allowed.

Having decided upon our stopping criteria, we must decide how to specify this in the program. We can assign values to the bankroll and maximum number of games using constants within the program, but this means that we must change the program whenever we want to change the values used. To avoid this, we will read the values for bankroll and maximum number of games to be played from input. Thus we have determined how to stop the program without having decided how many games to play.

Problem Refinement

The basic problem can be stated as follows:

```
/* SIMULATE PLAY AT A CRAPS GAME */
```

This can be divided into two subproblems.

```
/* SIMULATE PLAY AT A CRAPS GAME */
   /* PLAY THE REQUIRED NUMBER OF GAMES */
   /* PRINT THE FINAL RESULTS */
```

We will now refine the subproblem /* PLAY THE REQUIRED NUMBER OF GAMES */.

Since we may be required to play many games, this will be a repetitive step. We will use the stopping criteria discussed above. The action that we repeat is the playing of the games and the recording of the outcome of each game.

```
   /* PLAY THE REQUIRED NUMBER OF GAMES */
```

```
/* SET THE VALUE FOR BANKROLL AND        */
/* MAXIMUM NUMBER OF GAMES TO BE PLAYED */

/* REPEAT UNTIL BANKROLL DOUBLED OR LOST */
/* OR MAXIMUM NUMBER OF GAMES PLAYED     */

   /* PLAY ONE GAME */

   /* RECORD WIN OR LOSS */
```

We have now refined the subproblem in such a way that we can concentrate on the play of one game. Examining the rules of the game, we can refine /* PLAY ONE GAME */.

```
/* PLAY THE REQUIRED NUMBER OF GAMES */

   /* SET THE VALUE FOR BANKROLL AND        */
   /* MAXIMUM NUMBER OF GAMES TO BE PLAYED */

   /* REPEAT UNTIL BANKROLL DOUBLED OR LOST */
   /* OR MAXIMUM NUMBER OF GAMES PLAYED     */

      /* PLAY ONE GAME */

         /* ROLL THE DICE */

         /* PRINT VALUE OF ROLL */

         /* CHECK FOR WIN OR LOSS ON FIRST ROLL */

         /* IF GAME NOT OVER, CONTINUE PLAYING */
```

Let's look at the subproblems in the above refinement. /* ROLL THE DICE */ has been discussed under the section on input. We will eventually solve this problem with a random number generator, but the refinement up to this point does not restrict us to this method. Thus we can use numbers from input to test the other steps in the refinement. For instance, we will certainly want to test /* PRINT VALUE OF ROLL */ with values that we know rather than with values

that have been randomly generated. The advantage to this approach is that if, at a later point, we find numbers such as 1, 13, or 6.5 being printed, we can be reasonably certain that the problem is in the random number generation process, not in the print section.

The problem /* CHECK FOR WIN OR LOSS ON FIRST ROLL */ can be programmed by looking at the rules of the game. We will not refine it further until we are ready to write program statements.

We are now ready to refine /* IF GAME NOT OVER, CONTINUE PLAYING */. Looking at the rules for craps, we see that this involves rolling until a 7 has been rolled or the point has been rolled again.

```
/* PLAY ONE GAME */
   /* ROLL THE DICE */
   /* PRINT VALUE OF ROLL */
   /* CHECK FOR WIN OR LOSS ON FIRST ROLL */
   /* IF GAME NOT OVER, CONTINUE PLAYING */
      /* SET POINT = FIRST ROLL */
      /* KEEP ROLLING UNTIL NEW ROLL=7 (LOSE) */
      /* OR NEW ROLL=POINT (WIN)              */
```

/* SET POINT = FIRST ROLL */ can be refined by program statements. /* KEEP ROLLING UNTIL NEW ROLL = 7 (LOSE) OR NEW ROLL = POINT (WIN) */ needs further refinement.

```
/* KEEP ROLLING UNTIL NEW ROLL = 7 (LOSE) */
/* OR NEW ROLL = POINT (WIN)              */
```

```
/* ROLL THE DICE */

/* PRINT VALUE OF ROLL */
```

The statements in the above refinement need no further refining until we are ready to write the program statements.

We have now completed the refinement of /* PLAY THE REQUIRED NUMBER OF GAMES */. We must now refine the problem /* PRINT THE FINAL RESULTS */. This involves specifying the information to be printed.

```
/* PRINT THE FINAL RESULTS */

   /* PRINT NUMBER OF GAMES PLAYED, */
   /* NUMBER OF GAMES WON,          */
   /* NUMBER OF GAMES LOST,         */
   /* PERCENTAGE OF GAMES WON       */
```

This completes the refinement of the problem. The complete refinement is listed below. Note that we have not written any of the PL/1 statements necessary to refine the solution into a computer program. Using this refinement we could easily write a computer program in any general purpose language with which we are familiar.

```
/* SIMULATE PLAY AT A CRAPS GAME */

   /* PLAY THE REQUIRED NUMBER OF GAMES */

      /* SET THE VALUE FOR BANKROLL AND       */
      /* MAXIMUM NUMBER OF GAMES TO BE PLAYED */

      /* REPEAT UNTIL BANKROLL DOUBLED OR LOST */
      /* OR MAXIMUM NUMBER OF GAMES PLAYED     */

         /* PLAY ONE GAME */
```

```
            /* ROLL THE DICE */

            /* PRINT VALUE OF ROLL */

            /* CHECK FOR WIN OR LOSS ON FIRST ROLL */

            /* IF GAME NOT OVER, CONTINUE PLAYING */

               /* SET POINT = FIRST ROLL */

               /* KEEP ROLLING UNTIL NEW ROLL=7 (LOSE) */
               /* OR NEW ROLL=POINT (WIN)              */

                  /* ROLL THE DICE */

                  /* PRINT VALUE OF ROLL */

         /* RECORD WIN OR LOSS */

/* PRINT THE FINAL RESULTS */

   /* PRINT NUMBER OF GAMES PLAYED, */
   /* NUMBER OF GAMES WON,          */
   /* NUMBER OF GAMES LOST,         */
   /* PERCENTAGE OF GAMES WON       */
```

3.2 Waiting Line Problem

In many situations, the length of time a customer must wait to be served has considerable impact on a business's success. Most of us have had the experience of waiting in line for a table at a restaurant, or worse, sitting at a table for an intolerable length of time waiting to be served. Unless the business has a monopoly on the product it provides, future sales will suffer due to poor service.

To prevent loss of revenue, a business must determine the balance between lost sales and added costs of facilities. This is not as easy as it may seem. The cost of the added facilities (waiters' salaries, tables, linens, etc.) may be easy to calculate, but the amount of sales lost due to customers waiting to be served is difficult to determine.

In this problem, we will assume that the cost of customer dissatisfaction and the cost of added facilities is known. We will be concerned with determining the number of added facilities which minimizes costs.

Problem Specification

Determine the optimum number of checkout counters to install in a store.[1] Assume the following:

---

[1] Michael Kennedy and Martin B. Solomon, eight statement pl/c (pl/zero) plus pl/one (New Jersey, 1972), p. 375.

1. There may be customers waiting in line before the simulation begins.

2. No customer, one customer, or two customers can arrive in any minute.

3. A customer will join the shortest line.

4. When a customer arrives in line, he remains in line until he is served.

5. A customer must be checked out before the next person in line can advance to the counter.

6. The store loses a certain amount in future purchases for each minute that a customer waits in line.

7. A salary must be paid to one clerk for each counter.

8. The initial cost of installing the counters will be ignored.

Problem Clarification

The problem given could be solved by an analytical approach (using results from stochastic analysis, queueing theory, etc.). Such a solution procedure would, in most cases, be both cheaper and more accurate than a computer simulation. Many problems exist, however, for which no analytical approaches are known, and for some of these a computer simulation is the only feasible way of estimating the answer. To illustrate the principles of simulation, we will determine the optimum number of counters by simulation of the activity in the store. The simulation should closely model the activities of customers entering a line, waiting in line, and being checked out.

Some of the assumptions in the problem specification are questionable. It is very possible that more than two customers would arrive in line in a given minute, particularly during the store's peak activity period. For simplicity, however, we will limit the number of customers arriving in any minute to two.

We will allow the user of the program to specify the probability of a person or persons arriving in line. We will also allow the user to specify the number of people waiting in line at the beginning of the simulation, the time required to check out one customer, the cost incurred by making customers wait in line, and the hourly wages of the clerks.

By specifying the above variables as program parameters rather than constants, we make the program much more versatile. Using program parameters, the user could use the program to simulate average store activity; or, if desired, he could simulate activity during the peak hours or slack hours; or he could even use the program to estimate how long it would take the lines to empty immediately after peak hours.

If our simulation is reasonable, then the longer the period of activity in the store simulated, the better our approximation of costs; however, the longer the period simulated, the longer the execution time of our program and the greater the cost of the simulation. Rather than

arbitrarily selecting a time limit, we will let the user specify the period of time to be simulated.

Having resolved the question of how much store activity to simulate, we must decide how many simulations to run, using a different number of counters each time. If the parameters used in the program resemble costs and activities in the real world, we can use personal observations to guess at the number of counters to simulate.

Consider the local grocery store. There will be at least one counter, and probably not more than five or six. Even if the number of counters is not optimum, we should certainly expect the ideal number of counters to be ten or less. Thus we should be able to restrict our simulations to a store with one to ten counters. We will, however, let the user specify the maximum number of counters to be simulated, thus assuring that the program will halt.

We must now determine how we will select the optimum number of counters. One method is to simulate a store with one counter, then with two counters, etc., until we have simulated a store with ten counters. In each simulation we will determine the costs of operating with the given number of counters. We can then compare the costs calculated. The simulation giving the lowest costs will indicate the optimum number of counters.

A little analysis on our part, however, will show us a better method. Assume that the optimum number of counters

is 5. The cost with 4 counters will be greater since 5 is the number of counters which minimizes cost. But since 4 counters is closer to optimum than 3 counters, the cost of operating with 4 counters should be less than the cost of operating with 3 counters. The same relationship holds with 3 counters vs. 2, and with 2 counters vs. 1. Thus we can expect the costs to follow the pattern:

> cost of 1 counter > cost of 2 counters > ... > cost of optimum number of counters < cost of optimum number + 1 counters.

This shows that as we increase the number of counters, the operating expenses will drop until we go beyond the optimum number of counters. Because of this, we can stop our simulation when the cost of operating increases; the optimum number of counters will be one less than the number of counters in the final simulation.

The validity of our assumption that costs will decrease as we approach the optimum number of counters depends upon the values of the program parameters and the method used to simulate customer arrivals. We will use a random number generator to determine the number of customers arriving in a given minute. We would like to compare runs (one counter, two counters, etc.) using the same sequence of random numbers. With a deterministic random number generator, such as that discussed in problem 3.1, this is relatively simple. We need only restart the random number generator each time

we add another store counter in the simulation. Thus we see that the cyclic nature of the random numbers was a hazard in the craps game simulation, but it will be a virtue in this problem.

We would like to be able to use a random number generator to simulate customer checkout as well as customer arrival. We cannot, however, use the same random number generator for both purposes. To do so would interfere with the sequence of numbers generated to determine customer arrival, and we would not be able to compare runs using the same sequences. Rather than use another random number generator, we will allow the user to specify the amount of time necessary to process one customer at the checkout counter.

Examples

This is a complex problem, more difficult than most problems that an introductory student will face. Working an example should help pinpoint problem areas and decisions that need to be made. We will simulate 10 minutes of activity in a store with 2 counters.

Before we can work the example, we have to make decisions about the order in which events occur. Our first assumption is that a customer enters a line at the beginning of a minute. This eliminates the problem of charging for part of a minute in line. Second, we assume that if two lines are equally short, the customer will enter the first

of the equally short lines he reaches. Third, we assume that loss of future revenue will be incurred only while a person is waiting in line, not once he reaches the checkout counter. Fourth, we assume that the customers in line at minute 0 have just arrived in line.

MINUTE	ACTIVITY
0	1 customer at counter 1 1 customer at counter 2
1	no customer arrives cost of clerks = $.08 both customers being served; no waiting expense total cost = $.08
2	1 customer arrives, enters line 1 cost of clerks = $.08 1 customer waiting to be served; cost $.80 total cost = $.88 accumulated cost = $.96
3	no customer arrives cost of clerks = $.08 1 customer waiting; cost = $.80 total cost = $.88 accumulated cost = $1.84
4	no customer arrives cost of clerks = $.08 1 customer waiting; cost = $.80 total cost = $.88 accumulated cost = $2.72
5	customer at counter 1 checked out customer in line 1 moves to counter customer at counter 2 checked out 1 customer arrives, moves to counter 2 1 customer arrives, enters line 1 cost of clerks = $.08 1 customer waiting; cost = $.80 total cost = $.88 accumulated cost = $3.60
6	1 customer arrives, enters line 2 cost of clerks = $.08 2 customers waiting; cost = $1.60

```
        total cost = $1.68
        accumulated cost = $5.28

  7     no customer arrives
        cost of clerks = $.08
        2 customers waiting; cost = $1.60
        total cost = $1.68
        accumulated cost = $6.96

  8     1 customer arrives, enters line 1
        cost of clerks = $.08
        3 customers waiting; cost = $2.40
        total cost = $2.48
        accumulated cost = $9.44

  9     customer at counter 1 checked out
        customer in line 1 moves to counter
        customer in line 2 checked out
        customer in line 2 moves to counter
        no customer arrives
        cost of clerks = $.08
        1 customer waiting; cost = $.80
        total cost = $.88
        accumulated cost = $10.32

 10     no customer arrives
        cost of clerks = $.08
        1 customer waiting; cost = $.80
        total cost = $.88
        accumulated cost = $11.20

Cost of 2 counters for 10 minutes = $11.20.
```

Input

The program parameters are the only input required. These will include the length of simulation, maximum number of counters to simulate, number of customers initially in line, probabilities of customer arrivals, time needed to check out a customer, cost incurred because of customers waiting in line, and clerk's salary.

Output

We are required to print out the optimum number of counters. In addition, we will print the cost of operating with this number of counters.

There is other output which will be useful during the testing and debugging of the program. In order to know if our program is behaving as expected, we can print the details of each minute, as we did in the example. This can show us whether costs are being calculated and accumulated correctly and whether customers are entering at a reasonable rate. Due to the large amount of output this would generate, however, we would like to be able to prevent this output during the actual simulation. This is an excellent opportunity to introduce the use of executable comments in a language such as PL/C.

Problem Refinement

The initial problem can be stated as

```
/* DETERMINE OPTIMUM NUMBER OF CHECKOUT COUNTERS */.
```

This problem involves simulating activity until we have found the optimum number of counters. We will also want to print our results when the simulation is finished.

```
/* DETERMINE OPTIMUM NUMBER OF CHECKOUT COUNTERS */

   /* VARY NUMBER OF COUNTERS IN SIMULATION OF */
   /* STORE ACTIVITY UNTIL OPTIMUM COST FOUND  */

   /* PRINT OPTIMUM NUMBER OF COUNTERS AND COST */
```

We now need to determine what must be done in the simulation. First, we must read parameters to be used during this simulation. Next, we must simulate the cost of operating with one counter, then two counters, then three counters, etc., until the cost of operating increases. We will know that the optimum number of counters was used during the simulation preceding the increase in cost.

```
/* VARY NUMBER OF COUNTERS IN SIMULATION OF */
/* STORE ACTIVITY UNTIL OPTIMUM COST FOUND  */

   /* READ IN LENGTH OF SIMULATION,          */
   /* MAX NUMBER OF COUNTERS TO SIMULATE,    */
   /* NUMBER OF CUSTOMERS INITIALLY IN LINE, */
   /* PROBABILITIES OF CUSTOMER ARRIVALS,    */
   /* TIME NEEDED TO CHECK OUT CUSTOMER,     */
   /* COST INCURRED BECAUSE OF CUSTOMERS     */
   /* WAITING IN LINE, CLERK'S SALARY        */

   /* REPEAT UNTIL COST OF X COUNTERS IS GREATER */
   /* THAN THE COST OF X-1 COUNTERS, OR MAX      */
   /* NUMBER OF COUNTERS SIMULATED               */

      /* ADD ONE COUNTER */

      /* DETERMINE COST OF OPERATING FOR THE */
      /* SPECIFIED LENGTH OF TIME            */

      /* PRINT COST OF OPERATING */
```

To determine the steps involved in simulation, we can look at the example given earlier. We need to put the initial customers in the checkout lines and simulate the activity for the time specified by the user. Unlike the example, we will compute the total cost for clerks only once, at the beginning of the simulation, rather than compute their cost each minute.

```
/* DETERMINE COST OF OPERATING FOR THE */
/* SPECIFIED LENGTH OF TIME            */

   /* COMPUTE COST OF CLERKS */

   /* PLACE INITIAL CUSTOMERS IN LINE */

   /* SIMULATE ACTIVITY FOR EACH */
   /* MINUTE OF SIMULATION       */
```

At last we have reached the subproblem which is at the heart of the problem: simulate activity for each minute. The example given earlier is most helpful here. Looking at the example, we see that we must allow customers that have completed the checkout process to leave, bring in new customers, and compute the cost of future business lost because of long lines.

```
/* SIMULATE ACTIVITY FOR EACH */
/* MINUTE OF SIMULATION       */

   /* CHECK CUSTOMERS OUT */

   /* PROCESS ARRIVING CUSTOMERS */

   /* COMPUTE COST OF BUSINESS LOST DUE TO */
   /* CUSTOMERS WAITING IN LINE            */
```

Checking a customer out involves two steps. We must move the old customer out of the line, and move the next customer in line to the counter.

```
/* CHECK CUSTOMERS OUT */

   /* REPEAT FOR EACH COUNTER */

   /* IF CUSTOMER AT COUNTER FINISHED,   */
   /* MOVE HIM OUT,MOVE NEXT CUSTOMER IN */
```

Finally, we must determine how to add a new customer to the line. We must first determine if there are any new arrivals, and if so, how many. Second, we must decide where to place the new customers.

```
/* PROCESS ARRIVING CUSTOMERS */
   /* DETERMINE WHETHER NEW ARRIVALS */
   /* IF SO,ADD THEM TO SHORTEST LINES */
```

The entire refinement is listed below. Further refinement would involve the actual PL/1 statements.

```
/* DETERMINE OPTIMUM NUMBER OF CHECKOUT COUNTERS */

   /* VARY NUMBER OF COUNTERS IN SIMULATION OF */
   /* STORE ACTIVITY UNTIL OPTIMUM COST FOUND  */

      /* READ IN LENGTH OF SIMULATION,          */
      /* MAX NUMBER OF COUNTERS TO SIMULATE,    */
      /* NUMBER OF CUSTOMERS INITIALLY IN LINE, */
      /* PROBABILITIES OF CUSTOMER ARRIVALS,    */
      /* TIME NEEDED TO CHECK OUT CUSTOMER,     */
      /* COST INCURRED BECAUSE OF CUSTOMERS     */
      /* WAITING IN LINE, CLERK'S SALARY        */

      /* REPEAT UNTIL COST OF X COUNTERS IS GREATER */
      /* THAN THE COST OF X-1 COUNTERS, OR MAX      */
      /* NUMBER OF COUNTERS SIMULATED               */

         /* ADD ONE COUNTER */

         /* DETERMINE COST OF OPERATING FOR THE */
         /* SPECIFIED LENGTH OF TIME            */

            /* COMPUTE COST OF CLERKS */

            /* PLACE INITIAL CUSTOMERS IN LINE */

            /* SIMULATE ACTIVITY FOR EACH */
            /* MINUTE OF SIMULATION       */

               /* CHECK CUSTOMERS OUT */

                  /* REPEAT FOR EACH COUNTER */

                  /* IF CUSTOMER AT COUNTER FINISHED,   */
                  /* MOVE HIM OUT,MOVE NEXT CUSTOMER IN */

               /* PROCESS ARRIVING CUSTOMERS */

                  /* DETERMINE WHETHER NEW ARRIVALS */

                  /* IF SO,ADD THEM TO SHORTEST LINES */

               /* COMPUTE COST OF BUSINESS LOST DUE TO */
               /* CUSTOMERS WAITING IN LINE            */

         /* PRINT COST OF OPERATING */

   /* PRINT OPTIMUM NUMBER OF COUNTERS AND COST */
```

Chapter 4 - File Processing

4.1 Inventory

In order to minimize the time needed to deliver goods ordered, businesses maintain inventories of goods to be sold. Inventory levels for an item are determined by the number of units produced and the number of units sold. Note that the term "item" distinquishes products rather than units of a product.

It is necessary to keep track of the flow of goods into and out of inventory. In this problem, we will be concerned with keeping track of inventory activity as it relates to the shipment of goods to customers.

Problem Specification

Write a program to process customer orders for a furniture manufacturer. Print the status of inventory items after the orders have been processed.

Problem Clarification

Processing customer orders involves several activities. If there is enough in inventory to cover the order, then a shipping invoice is written. If the order cannot be filled, an invoice is written for the quantity which can be shipped and a backorder is written for the remainder of the order.

We will try to fill backorders before new orders the next time we run the program.

Because we must look at inventory to see whether we can fill an order, we will have to update the inventory file during processing. If an invoice is filled out for an item, we must subtract the quantity shipped from the quantity on hand. Otherwise we might plan to send the same goods to more than one customer.

Since inventory maintenance is necessary with order processing, we will imbed order processing in an inventory system. There are three other phases involved in inventory processing.

The first phase must come before order processing. This is the addition of finished goods to the inventory. If goods were not added to the inventory before order processing, many items which could be shipped would be backordered rather than invoiced. This would increase inventory costs and decrease customer satisfaction.

After customer orders (those not filled on the previous run and new orders) are processed, we will want to print a reorder list for items that have been backordered. To reduce the number of backorders, we will also list items to be reordered which have not been backordered, but which have low inventory levels. The level below which an item will be reordered is called the reorder point. The quantity to be ordered is called the reorder quantity. These values will

vary for the different items in inventory.

The last phase of inventory processing will be printing the status of each item in inventory.

Input

There are two types of input. The first is a procurement record. This includes the item number and the quantity received for an item which was reordered. There will be one procurement record for each item received.

The second type of input record is the customer order. Each customer order will contain the customer name and address, item number, and quantity ordered. There will be one input record for each item ordered by a customer.

Output

When processing customer orders, we will have to print invoices and backorders. Invoices will contain the customer name and address, and the following information for each item shipped: item number, quantity shipped, unit price, and amount due. The total amount due will be printed after all items ordered by one customer have been processed.

Backorders will contain information on items which were ordered but not available for shipment. For each item the backorder will contain the customer name and address, item number, number of units backordered, and unit price.

After processing customer orders, we will print the reorder list and status list. The reorder list will contain the item number and quantity to order for each item which

must be reordered.

The status list will include the following information for each item in the inventory: item number, quantity on hand, number backordered, number shipped, and dollar value of item sales.

Inventory File

In early references to the inventory file, we did not discuss the information contained in the file. Now that we know the output required, we can determine what information will be needed before and during each run.

There will be one record in the inventory file for each item in inventory. We will know which item is being referred to in the record by keeping the item number in the record. Obviously, no two items may have the same item number.

The status list requires that we print the item number, quantity on hand, number backordered, number shipped and dollars in sales for each item. To simplify the program we will store this information in the inventory file rather than in a separate file. So that we may calculate the price of goods shipped, we must also include the unit price of each item in the inventory file.

To determine the information necessary for the reorder list, we need to know the reorder point and reorder quantity. These should also be kept in the inventory file.

If an item has already been ordered but not received, we must know this to prevent reordering items more than once. Therefore, the inventory file record will also contain the quantity on order for the item.

To summarize, the following information for each item will be kept in the master file:

Permanent:

item number
quantity on hand
unit price
reorder point
reorder quantity
quantity on order

Determined for each run:

number backordered
number shipped
dollar value of sales

Assumptions

We have made several assumptions in the problem clarification. First, we have assumed that the inventory file already exists. We have provided no means of creating this file; we have no mechanism for adding items to the file, deleting items from the file, or changing the price, reorder point, or reorder quantity for an item. These are events which will almost certainly take place during the life of the file, but we will not deal with them in this problem.

We will also assume that the inventory file is sorted by item number, that the transactions concerning shipments

from the factory to the warehouse are sorted by item number, and that the customer orders are sorted by item number within customer number (or name), and that backorders from the previous run precede new orders.

Problem Refinement

The problem we are dealing with is

```
/* INVENTORY MAINTENANCE AND ORDER PROCESSING */.
```

This problem can be refined into four parts, reflecting the four phases discussed in the problem clarification.

```
/* INVENTORY MAINTENANCE AND ORDER PROCESSING */
   /* PROCESS ITEMS RECEIVED */
   /* PROCESS BACKORDERS, THEN NEW ORDERS */
   /* LIST DEPLETED INVENTORY ITEMS */
   /* PRINT STATUS REPORT */
```

If possible, we should combine the third and fourth phases so that we do not have to make two passes through the master file. Since these phases are logically separate, however, we will treat them separately here.

The first subproblem is a repetitive problem. We must look at the inventory record for each item received. In addition to updating the inventory file, we will want to print an exception report for an item if the quantity received is not the same as the quantity on order.

```
/* PROCESS ITEMS RECEIVED */
```

```
/* REPEAT FOR EACH ITEM RECEIVED */
   /* ADD QUANTITY RECEIVED TO QUANTITY ON HAND */
   /* SUBTRACT FROM QUANTITY ON ORDER */
   /* IF QUANTITY RECEIVED DIFFERS FROM QUANTITY */
   /* ON ORDER PRINT MESSAGE ON EXCEPTION REPORT */
```

The second phase involves printing invoices and backorders. We will also have to update the inventory file to reflect the number of items shipped and backordered.

```
/* PROCESS BACKORDERS, THEN NEW ORDERS */
   /* REPEAT FOR EACH CUSTOMER */
      /* FILL OUT INVOICE AND BACKORDER IF NECESSARY */
         /* PRINT NAME AND ADDRESS ON INVOICE */
         /* REPEAT FOR EACH ITEM CUSTOMER ORDERED */
            /* PRINT MESSAGE IF ORDER UNUSUALLY LARGE */
            /* ADD ITEM TO INVOICE */
            /* ADD ITEM TO BACKORDER, IF NECESSARY */
            /* UPDATE INVENTORY RECORD */
         /* PRINT TOTAL COST ON INVOICE */
```

Adding an item to the invoice involves determining the quantity and price of each item to be shipped.

```
/* ADD ITEM TO INVOICE */
   /* DETERMINE QUANTITY TO BE SHIPPED */
   /* CALCULATE COST FOR THIS ITEM */
```

Backorders will not be filled out for each item. The quantity to be backordered is the difference between the quantity ordered and the quantity available for shipment.

```
/* ADD ITEM TO BACKORDER, IF NECESSARY */

   /* QUANTITY BACKORDERED=NUMBER ORDERED */
   /* LESS NUMBER SHIPPED                */

   /* IF QUANTITY BACKORDERED > 0 THEN */
   /* ADD ITEM TO BACKORDER            */
```

The third phase of processing involves determining which items must be reordered. We will not reorder an item if it is already on order. Otherwise, we will reorder if the quantity on hand is below the reorder point. This algorithm should be satisfactory if management has chosen an apropriate reorder point and reorder quantity.

Under ideal conditions there would be no backorders. The reorder point would be high enough to fill all customer orders arriving between the time an item is reordered and the time the quantity reordered is received. Because this condition cannot always be met, our program must have the capability of handling backorders.

If the reorder point or reorder quantity is too low, backorders can accumulate in such a way that the inventory for a particular item is seldom, if ever, sufficient to fill customer orders. It is therefore imperative that management carefully select the reorder point and reorder quantity for each item. In order to help them do this, when we list an

item in the status report, we will indicate whether the quantity backordered is greater than the quantity on order. This will point out items which may need adjustment in the reorder point or reorder quantity.

```
/* PRINT REORDER LIST */

   /* REPEAT FOR EACH ITEM IN INVENTORY */

      /* DETERMINE WHETHER ITEM SHOULD BE REORDERED */

         /* IF QUANTITY ON ORDER IS ZERO AND          */
         /* QUANTITY ON HAND IS BELOW REORDER POINT */

            /* UPDATE INVENTORY FILE */

               /* QUANTITY ON ORDER=REORDER QUANTITY */

            /* PRINT ITEM ON REORDER LIST */
```

The last phase of processing involves printing the status of each item in the inventory file. We will have to look at every item in the file.

```
/* PRINT STATUS REPORT */

   /* REPEAT FOR EACH ITEM IN INVENTORY */

      /* PRINT ITEM NUMBER, QUANTITY ON HAND, */
      /* NUMBER BACKORDERED, NUMBER SHIPPED,  */
      /* DOLLAR SALES OF SHIPMENTS            */

      /* IF QUANTITY ON ORDER IS LESS THAN */
      /* QUANTITY BACKORDERED, MARK THIS   */
      /* ITEM FOR EASY IDENTIFICATION      */
```

The complete refinement is listed below.

```
/* INVENTORY MAINTENANCE AND ORDER PROCESSING */

   /* PROCESS ITEMS RECEIVED */

      /* REPEAT FOR EACH ITEM RECEIVED */
```

```
         /* ADD QUANTITY RECEIVED TO QUANTITY ON HAND */

         /* SUBTRACT FROM QUANTITY ON ORDER */

         /* IF QUANTITY RECEIVED DIFFERS FROM QUANTITY */
         /* ON ORDER PRINT MESSAGE ON EXCEPTION REPORT */

/* PROCESS BACKORDERS, THEN NEW ORDERS */

   /* REPEAT FOR EACH CUSTOMER */

      /* FILL OUT INVOICE AND BACKORDER IF NECESSARY */

         /* PRINT NAME AND ADDRESS ON INVOICE */

         /* REPEAT FOR EACH ITEM CUSTOMER ORDERED */

            /* PRINT MESSAGE IF ORDER UNUSUALLY LARGE */

            /* ADD ITEM TO INVOICE */

               /* DETERMINE QUANTITY TO BE SHIPPED */

               /* CALCULATE COST FOR THIS ITEM */

            /* ADD ITEM TO BACKORDER, IF NECESSARY */

               /* QUANTITY BACKORDERED=NUMBER ORDERED */
               /* LESS NUMBER SHIPPED                 */

               /* IF QUANTITY BACKORDERED > 0 THEN */
               /* ADD ITEM TO BACKORDER            */

            /* UPDATE INVENTORY RECORD */

         /* PRINT TOTAL COST ON INVOICE */

/* PRINT REORDER LIST */

   /* REPEAT FOR EACH ITEM IN INVENTORY */

      /* DETERMINE WHETHER ITEM SHOULD BE REORDERED */

         /* IF QUANTITY ON ORDER IS ZERO AND        */
         /* QUANTITY ON HAND IS BELOW REORDER POINT */

            /* UPDATE INVENTORY FILE */

               /* QUANTITY ON ORDER=REORDER QUANTITY */

            /* PRINT ITEM ON REORDER LIST */
```

```
/* PRINT STATUS REPORT */

   /* REPEAT FOR EACH ITEM IN INVENTORY */

      /* PRINT ITEM NUMBER, QUANTITY ON HAND,  */
      /* NUMBER BACKORDERED, NUMBER SHIPPED,   */
      /* DOLLAR SALES OF SHIPMENTS             */

      /* IF QUANTITY ON ORDER IS LESS THAN */
      /* QUANTITY BACKORDERED, MARK THIS   */
      /* ITEM FOR EASY IDENTIFICATION      */
```

4.2 Payroll

All businesses require employees, and most employees require compensation for the work they perform. The procedure for computing and recording employee compensation is called payroll. In many businesses, payroll is a complex and time-consuming activity.

The first activity required is the collection of basic employee information. This includes items such as employee name, address, social security number, authorized deductions, and rate of pay. This information must be kept in a master file. Provisions must be made for adding information for new employees, changing information for current employees, and deleting information for employees leaving the organization.

Another activity is computing the payroll. Information identifying the employee and the number of hours worked must be obtained. Gross pay must be calculated, based on the hours worked and rate of pay. Overtime must be paid when required. Employee deductions must be calculated, including insurance, bonds, stock options, etc., as well as social security, withholding taxes, and city and state taxes.

Finally, we must write the payroll. This includes employee compensation (paycheck, transfer of funds to his account, recording of cash disbursements, etc.), record of earnings (pay stub), tax records for government agencies,

and management reports.

The problem we will deal with concerns computing and recording payroll information.

Problem Specification

Write a payroll program. The program should maintain the following information for each employee: social security number, name, address, year-to-date gross earnings, year-to-date federal taxes, year-to-date state taxes, year-to-date social security taxes, number of dependents, hourly rate, vacation time accrued, sick leave accrued, and a code indicating whether or not the employee makes a voluntary deduction of $1.25 per week for group health insurance.

All employees should be paid each week. A time card containing the social security number and hours worked will be provided for each employee. Time and a half will be paid for overtime (hours over 40). The payroll program should print a paycheck and stub for each employee.

Problem Clarification

The problem is concerned with computing and writing the payroll. We will assume that the master file has already been created and contains all the information we will need except for the number of hours worked. It is assumed to be sorted in ascending order on social security number and to contain a single record for each employee.

Input

The only input will be time cards. Each time card will contain identifying information (social security number) and the number of hours the employee worked during the week. Since we have to allow vacation and sick leave, the amount of time to be charged to each must be included on the time card. We will assume that there is one time card for each employee and that the time cards are sorted in ascending order on social security number.

Output

The two things our program must print are paychecks and pay stubs. We must print one of each for every employee. We will not print tax records for the government or reports for management, although these are a part of the output of most payroll programs.

Problem Refinement

The problem can be stated as

```
/* COMPUTE WEEKLY PAYROLL */.
```

This will be a repetitive process. We will have to compute payroll for each employee. The activities involved will be reading the information needed, computing pay, writing a paycheck, and writing a pay stub.

```
/* COMPUTE WEEKLY PAYROLL */
   /* REPEAT FOR EACH EMPLOYEE */
```

```
/* READ INFORMATION NEEDED TO COMPUTE PAY */
/* FOR THE NEXT EMPLOYEE                  */

/* COMPUTE PAY */

/* UPDATE MASTER FILE */

/* PRINT PAYCHECK */

/* PRINT PAY STUB */
```

The information needed to process one employee's weekly pay comes from two sources: the payroll master file and the employee's time card. We have assumed that the master file and time cards are sorted, and that there is one master file record and one time card for each employee. If this is true, we will only need to read the next master file record and the next time card in order to compute the next employee's pay. A close look at the problem, however, indicates that we should not accept the assumption that there will be one master file record and one time card for each employee.

If a new employee submits a time card before his payroll record is added to the master file, all of the time cards after his will be matched with the wrong master file record. This means that all the employees with social security numbers greater than his will receive incorrect paychecks. Their master file records will also be updated with the wrong information, destroying the integrity of the master file.

Even if we assume that this could never happen, we would have a similar problem if an employee's time card were not submitted. In this case we would have a master file record for an employee without a time card, and the master file and time cards would again be processed out of synchronization.

There are manual checks which can and should be implemented to prevent the above situations from occurring. But there is still one problem that is likely to arise. That is error in data entry. It is virtually certain that at some time an employee's social security number will be incorrectly entered on his time card. This would produce a master file record without a time card and a time card without a master file record.

In order to prevent incorrectly matching time cards with master file records, we will not compute pay for an employee unless the social security numbers on the master file record and the time card are the same. If we read a master file record and time card which do not match, we will print the one with the lower social security number on an exception report, and read a new time card or master file record. We will repeat this process until we find a matching time card and master file record.

```
/* READ INFORMATION NEEDED TO COMPUTE PAY */
/* FOR THE NEXT EMPLOYEE                  */

   /* READ NEXT MASTER FILE RECORD */
```

```
            /* READ NEXT TIME CARD */

            /* IF THE SOCIAL SECURITY NUMBERS DO NOT MATCH, */
            /* PRINT INVALID OR MISSING TIME CARD ON THE    */
            /* EXCEPTION REPORT AND FIND THE NEXT MATCHING  */
            /* TIME CARD AND MASTER FILE RECORD             */
```

The matching process is not inherent in the nature of the payroll program, so we will use a subroutine to match time cards and master file records.

```
/* SUBROUTINE TO MATCH TIME CARD WITH */
/*             MASTER FILE RECORD     */

   /* REPEAT UNTIL SOCIAL SECURITY NUMBERS MATCH */

      /* REPEAT WHILE TIME CARD IS INVALID */

         /* PRINT TIME CARD ON EXCEPTION REPORT */

         /* READ NEXT TIME CARD */

      /* REPEAT WHILE TIME CARDS ARE MISSING */

         /* PRINT MASTER FILE RECORD ON EXCEPTION REPORT */

         /* READ NEXT MASTER FILE RECORD */
```

We can now refine /* COMPUTE PAY */. We must compute gross pay, deductions, and net pay.

```
      /* COMPUTE PAY */

         /* COMPUTE GROSS PAY */

         /* COMPUTE DEDUCTIONS */

         /* COMPUTE NET PAY */
```

In computing gross pay we use the number of hours worked. We should test to make sure that the number of hours worked seems reasonable. If the employee works over

ten hours of overtime, we will print a message on the exception report.

We must compute regular pay, overtime pay, vacation pay, and sick leave. Note that the method of computing gross pay is different for employees who are paid salaries and those who are paid hourly wages. If we use a subroutine to compute gross pay, changes to the method of calculating pay will be internal to the subroutine and will not affect the structure of the main program.

```
/* SUBROUTINE TO COMPUTE GROSS PAY */
   /* COMPUTE PAY FOR SALARIED EMPLOYEES */
   /* COMPUTE PAY FOR HOURLY EMPLOYEES */
      /* REGULAR HOURS */
      /* OVERTIME HOURS */
         /* IF OVERTIME > 10 HOURS,            */
         /* PRINT MESSAGE ON EXCEPTION REPORT */
         /* COMPUTE OVERTIME PAY */
      /* VACATION PAY */
      /* SICK LEAVE */
```

We can now refine /* DEDUCTIONS */. This will include social security, taxes, and voluntary deductions. For modularity, we wll also use a subroutine here.

```
/* SUBROUTINE TO COMPUTE DEDUCTIONS */
   /* SOCIAL SECURITY */
   /* FEDERAL TAX */
```

```
/* STATE TAX */
/* GROUP HEALTH INSURANCE */
```

Because methods for computing social security and taxes change quite frequently, we will use subroutines to compute social security, federal taxes, and state taxes. To facilitate adding payroll deduction plans such as stock options, bonds, etc., we will use a subroutine to compute voluntary deductions. The final refinement necessary in /* COMPUTE PAY */ is the refinement of /* NET PAY */. To compute net pay, we subtract the deductions from gross pay. Due to the simplicity of this step, we will not use a subroutine here. The complete refinement of /* COMPUTE PAY */ is shown below.

```
      /* COMPUTE PAY */
         /* COMPUTE GROSS PAY */
            call subroutine to compute gross pay
         /* COMPUTE DEDUCTIONS */
            call subroutine to compute deductions
         /* COMPUTE NET PAY */
            /* SUBTRACT DEDUCTIONS FROM GROSS PAY */
/* SUBROUTINE TO COMPUTE GROSS PAY */
   /* COMPUTE PAY FOR SALARIED EMPLOYEES */
   /* COMPUTE PAY FOR HOURLY EMPLOYEES */
      /* REGULAR HOURS */
      /* OVERTIME HOURS */
```

```
            /* IF OVERTIME > 10 HOURS,              */
            /* PRINT MESSAGE ON EXCEPTION REPORT */

            /* COMPUTE OVERTIME PAY */

         /* VACATION HOURS */

         /* SICK LEAVE */

/* SUBROUTINE TO COMPUTE DEDUCTIONS */

   /* SOCIAL SECURITY */

      call subroutine to compute social security

   /* FEDERAL TAX */

      call subroutine to compute federal tax

   /* STATE TAX */

      call subroutine to compute state tax

   /* VOLUNTARY DEDUCTIONS */

      call subroutine to compute voluntary deductions

/* SUBROUTINE TO COMPUTE SOCIAL SECURITY */

/* SUBROUTINE TO COMPUTE FEDERAL TAX */

/* SUBROUTINE TO COMPUTE STATE TAX */

/* SUBROUTINE TO COMPUTE VOLUNTARY DEDUCTIONS */

   /* GROUP HEALTH INSURANCE */
```

After computing pay, we must update the master file. This requires adding this pay period's totals to the cumulative totals in the employee's record. We will use a subroutine for modularity.

```
/* SUBROUTINE TO UPDATE MASTER FILE */

   /* ADD THIS WEEK'S PAY TOTALS TO    */
   /* CUMULATIVE TOTALS IN EMPLOYEE'S */
   /* MASTER FILE RECORD               */

   /* UPDATE VACATION BALANCE AND */
   /* SICK LEAVE BALANCE          */
```

Next we will refine /* PRINT PAYCHECK */. We will assume that the paycheck information will be printed on pre-printed forms. Thus we will not have to print the bank name, company name, account number, etc.; however, we will have to print the employee's name and net pay. In order to reduce chances of fraud or incorrect hours being entered on the time card, we will print a message on an exception report whenever the pay for the week exceeds $1000.00. To facilitate testing of the program, we will use a subroutine to print the paycheck. Using a subroutine, we can easily test this part of the program for correct information before formating the information to be printed in the correct locations on the pre-printed check forms.

```
/* SUBROUTINE TO PRINT PAYCHECK */

   /* ON PRE-PRINTED FORM, PRINT */
   /* EMPLOYEE NAME AND NET PAY  */

   /* IF GROSS PAY EXCEEDS $1000, */
   /* PRINT EXCEPTION REPORT      */
```

Finally, we will refine /* PRINT PAY STUB */. As with the paycheck, we will assume that the pay stub information will be printed on a pre-printed form. We must print the

date, information identifying the employee, totals for this week's pay period, and cumulative totals. For reasons similar to those above, we will use a subroutine for printing the pay stub.

```
/* SUBROUTINE TO PRINT PAY STUB */

   /* ON PRE-PRINTED FORM, PRINT DATE,          */
   /* EMPLOYEE NAME, SOCIAL SECURITY NUMBER,    */
   /* CUMULATIVE TOTALS,                        */
   /* TOTALS FOR THIS PAY PERIOD                */
```

The complete refinement appears below.

```
/* COMPUTE WEEKLY PAYROLL */

   /* REPEAT FOR EACH EMPLOYEE */

      /* READ INFORMATION NEEDED TO COMPUTE PAY */
      /* FOR THE NEXT EMPLOYEE                  */

         /* READ EMPLOYEE'S MASTER FILE RECORD */

         /* READ EMPLOYEE'S TIME CARD */

         /* IF THE SOCIAL SECURITY NUMBERS DO NOT MATCH, */
         /* PRINT INVALID OR MISSING TIME CARD ON THE    */
         /* EXCEPTION REPORT AND FIND THE NEXT MATCHING  */
         /* TIME CARD AND MASTER FILE RECORD             */

            call subroutine to match time card with
            master file record

      /* COMPUTE PAY */

         /* COMPUTE GROSS PAY */

            call subroutine to compute gross pay

         /* COMPUTE DEDUCTIONS */

            call subroutine to compute deductions

         /* COMPUTE NET PAY */

            /* SUBTRACT DEDUCTIONS FROM GROSS PAY */
```

```
         /* UPDATE MASTER FILE */

            call subroutine to update master file

         /* PRINT PAYCHECK */

            call subroutine to print paycheck

         /* PRINT PAY STUB */

            call subroutine to print pay stub

/* SUBROUTINE TO MATCH TIME CARD WITH */
/*             MASTER FILE RECORD     */

   /* REPEAT UNTIL SOCIAL SECURITY NUMBERS MATCH */

      /* REPEAT WHILE TIME CARD IS INVALID */

         /* PRINT TIME CARD ON EXCEPTION REPORT */

         /* READ NEXT TIME CARD */

      /* REPEAT WHILE TIME CARDS ARE MISSING */

         /* PRINT MASTER FILE RECORD ON EXCEPTION REPORT */

         /* READ NEXT MASTER FILE RECORD */

/* SUBROUTINE TO COMPUTE GROSS PAY */

   /* COMPUTE PAY FOR SALARIED EMPLOYEES */

   /* COMPUTE PAY FOR HOURLY EMPLOYEES */

      /* REGULAR HOURS */

      /* OVERTIME HOURS */

         /* IF OVERTIME > 10 HOURS,           */
         /* PRINT MESSAGE ON EXCEPTION REPORT */

         /* COMPUTE OVERTIME PAY */

      /* VACATION HOURS */

      /* SICK LEAVE */

/* SUBROUTINE TO COMPUTE DEDUCTIONS */

   /* SOCIAL SECURITY */
```

```
      call subroutine to compute social security

   /* FEDERAL TAX */

      call subroutine to compute federal tax

   /* STATE TAX */

      call subroutine to compute state tax

   /* VOLUNTARY DEDUCTIONS */

      call subroutine to compute voluntary deductions

/* SUBROUTINE TO COMPUTE SOCIAL SECURITY */

/* SUBROUTINE TO COMPUTE FEDERAL TAX */

/* SUBROUTINE TO COMPUTE STATE TAX */

/* SUBROUTINE TO COMPUTE VOLUNTARY DEDUCTIONS */

   /* GROUP HEALTH INSURANCE */

/* SUBROUTINE TO UPDATE MASTER FILE */

   /* ADD THIS WEEK'S PAY TOTALS TO   */
   /* CUMULATIVE TOTALS IN EMPLOYEE'S */
   /* MASTER FILE RECORD              */

   /* UPDATE VACATION BALANCE AND */
   /* SICK LEAVE BALANCE          */

/* SUBROUTINE TO PRINT PAYCHECK */

   /* ON PRE-PRINTED FORM, PRINT */
   /* EMPLOYEE NAME AND NET PAY  */

   /* IF GROSS PAY EXCEEDS $1000, */
   /* PRINT EXCEPTION REPORT      */

/* SUBROUTINE TO PRINT PAY STUB */

   /* ON PRE-PRINTED FORM, PRINT DATE,        */
   /* EMPLOYEE NAME, SOCIAL SECURITY NUMBER,  */
   /* CUMULATIVE TOTALS,                      */
   /* TOTALS FOR THIS PAY PERIOD              */
```

BIBLIOGRAPHY

1. Aho, Alfred V., Hopcroft, John E., and Ullman, Jeffrey D., The Design and Analysis of Computer Algorithms, Reading, Massachusetts, Addison-Wesley Publishing Company, 1974.

2. Anthony, Robert N., Management Accounting: Text and Cases, Homewood, Illinois, Richard D. Irwin, Inc., 1964.

3. Breckner, David, and Abel, Peter, Principles of Business Computer Programming, Englewood Cliffs, New Jersey, Prentice-Hall, Inc., 1970.

4. Conway, Richard, and Gries, David, An Introduction to Programming: A Structured Approach Using PL/1 and PL/C-7, Cambridge, Massachusetts, Winthrop Publishers, Inc., 1975.

5. Hughes, Joan K., PL/1 Programming, New York, John Wiley & Sons, Inc., 1973.

6. Kennedy, Michael, and Solomon, Martin B., eight statement pl/c (pl/zero) plus pl/one, Englewood Cliffs, New Jersey, Prentice-Hall, Inc., 1972.

7. LaFave, L. J., Milbrandt, G. D., and Garth, D. W., Problem Solving: The Computer Approach, New York, McGraw-Hill Ryerson Limited, 1972.

8. Neuhold, Erich J., and Lawson, Harold W., Jr., The PL/1 Machine: An Introduction to Programming, Reading, Massachusetts, Addison-Wesley Publishing Co., Inc., 1971.

9. Polya, G., How to Solve It, Princeton, New Jersey, Princeton University Press, 1945.

10. Shelly, Gary B., and Cashman, Thomas J., Introduction to Computer Programming: ANSI Cobol, Fullerton, California, Anaheim Publishing Co., 1973.

11. Sprowls, R. Clay, PL/C: a processor for PL/1, San Francisco, California, Canfield Press, 1972.

12. Van Tassel, Dennie, Program Style, Design, Efficiency, Debugging, and Testing, Englewood Cliffs, New Jersey, Prentice-Hall, Inc., 1974.

13. Weinberg, Gerald M., PL/1 Programming Primer, New York, McGraw-Hill Book Company, 1966.

14. _____, PL/1 Programming: A Manual of Style, New York, McGraw-Hill Book Company, 1970.

15. Weinberg, Gerald, Yasukawa, Norie, and Marcus, Robert, Structured Programming in PL/C: An Abecedarian, New York, John Wiley & Sons, Inc., 1973.

16. Weston, J. Fred, and Brigham, Eugene F., Essentials of Managerial Finance, Hinsdale, Illinois, The Dryden Press, 1974.

Appendix A

Further Problems for Refinement

Business Problems

1. Program a Point of Sale terminal, such as that used in a fast food restaurant. The program should accept input such as HAMBURGER, CHEESEBURGER, FRIES, LARGE COKE, SMALL COKE, etc., and determine the cost of each item, the subtotal, the sales tax, and the total cost. The program should also keep count of the total number of each item sold over a period of time. In addition, the program should determine the amount of change to be returned to the customer and the minimum number of coins and bills of each denomination required to make the change.
2. Write a program to maintain savings accounts for a bank. The program should process deposits, withdrawals, interest accumulation, and service charges. Allow for at least two types of savings accounts, one which pays the minimum interest rate but allows immediate withdrawals with no service charge, and another which pays a higher interest rate but requires a minimum balance and six months notice for withdrawals without penalty.
3. Write a program to compute salesmen's commissions.

There should be different commissions for the different types of items sold and higher commission rates for larger sales volumes.

4. Write a program to verify employee time cards used in a payroll program. The program should verify that the employee identification number field contains only numeric data, the employee name field contains only alphabetic data and valid punctuation, and the hours worked field contains only numeric data. The program should also verify that cards are arranged in ascending order by employee identification number.
5. Write a program to maintain a file containing subscribers to magazines. Each record should contain the subscriber's name and address, names of magazines ordered, and number of issues ordered. The program should print mailing labels for each customer subscription on a weekly or monthly basis, depending upon the magazine. It should also send renewal notices prior to the subscription expiration date.
6. Write a program to maintain a list of activities on an executive's schedule, (e. g., MARCH 5, 9:30 AM, CONFERENCE WITH CARL). The program should print the activities in calendar form upon request, (e. g., CALENDAR,MARCH would request that the program print all activities scheduled for March).
7. Write a program to produce personalized form letters.

The letter to be printed should be entered with special characters denoting parts of the text that vary for each person on the mailing list (for example, name, company name, address, etc.). The name, company name, and address should be included for every person on the mailing list. The letters should have margins right justified.

8. Write a program to maintain a list of a person's stock holdings. The list should contain the name of the company, number of shares purchased, date of purchase, and purchase price. Given the present market value of the stocks, the program should print the gain or loss for each stock held, and the overall gain or loss.

9. Write a program to keep track of accounts for a credit card company. The program should process charges, payments, cash advances, and finance charges. In addition, each customer should be sent a monthly statement.

10. Write a program to convert from one currency to another. The program should accept an amount in any currency and convert it to the currency requested.

11. Write a program to determine whether it is better to buy or lease an automobile. The program should consider factors such as purchase cost, number of years to be driven, cost of maintenance, leasing cost, and cost of insurance.

Statistics Problems

12. Write a program to process inquiries about census information. The census information should include items such as district, type of dwelling, and name, age, sex, and race of persons in the household. The program should be able to answer inquiries such as NUMBER OF FEMALES, AGE < 18, DISTRICT 1.

13. Write a program to compute grade averages for students in a class. The program should keep the names and homework grades for each student in the class. It should print the class average and median for each assignment, the final average for each student, the class rank for each student, and a graph of the distribution of final averages.

14. Write a program to compute statistics on a collection of data requested by a user. The user should be allowed to enter the test data and the names of the statistics he wants printed (e. g., MEAN, MEDIAN, STANDARD DEVIATION, etc.).

Miscellaneous Problems

15. Write a program to simulate the game of blackjack. The dealer should be allowed to use multiple decks of cards. The dealer must take another card if his point total is 16 or less and may not take another card if his point total is 17 or more. Devise a strategy for

the player (e. g., stand pat with 12 or more points) and determine the player's gain or loss after many games have been played.

16. Write a program to deal a bridge hand and bid the first round.

17. Write a program to store recipes. The program should print recipes requested and the amount of each ingredient needed to serve a specified number of people.

18. Each senior highway patrolman is assigned a junior highway patrolman as a partner. To minimize dissatisfaction, the highway patrol tries to assign each man his preferred partner. Each senior patrolman ranks the junior patrolmen according to his preference, and vice versa. Write a program to assign partners based on the rankings. The pairings are optimal if for each two senior patrolmen S1 and S2, and their paired junior patrolmen J1 and J2,

    1. either S1 ranks J1 higher than J2, or J2 ranks S2 higher than S1, and
    2. either S2 ranks J2 higher than J1, or J1 ranks S1 higher than S2.

New address:

Alan D. Bernard

General Robotics Corporation

57 N. Main Street

Hartford, Wisconsin 53027