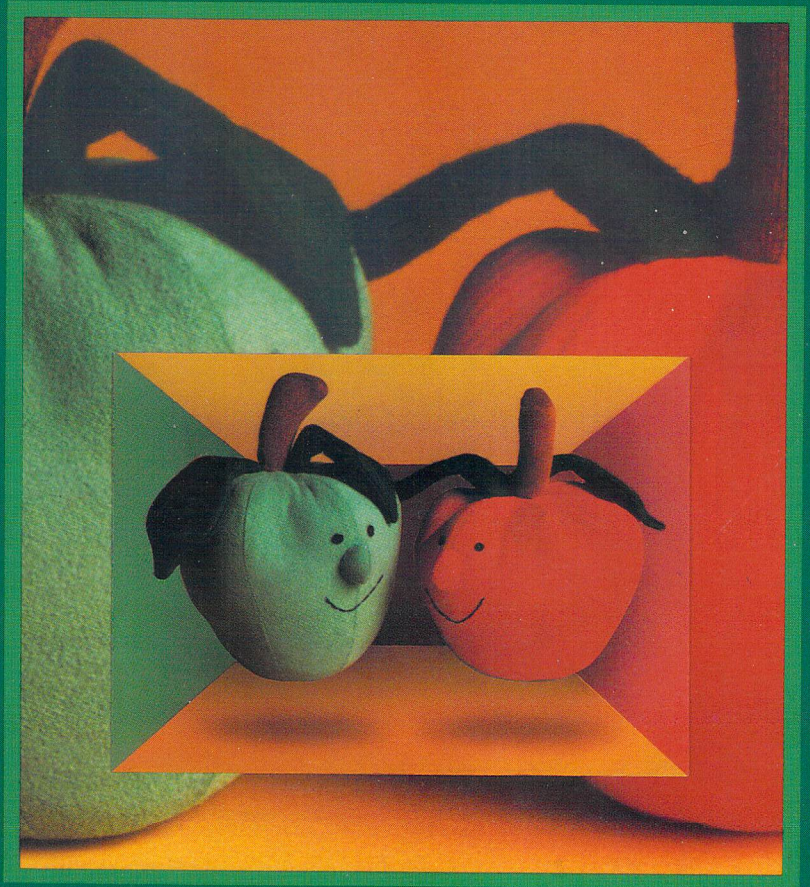
SAMS PRESENTS 22259

Applesoft® For The IIe®

Brian D. Blackwood and
George H. Blackwood



Applesoft for the IIe

Dr. George H. Blackwood is a retired Navy pilot and a former college professor with bachelor's, master's, education specialist, and D.D.S. degrees. He now devotes full time to writing.

Brian D. Blackwood has studied computer science and engineering at Michigan State University, and has a B.S. degree in computer science from Lamar University. He is presently employed as a programmer at a large data processing center that services banks and financial institutions.

Applesoft for the IIe

By
Brian D. Blackwood
and
George H. Blackwood

Howard W. Sams & Co., Inc.
4300 WEST 62ND ST. INDIANAPOLIS, INDIANA 46268 USA

FIRST EDITION
FIRST PRINTING — 1983

International Standard Book Number: 0-672-22259-0
Library of Congress Catalog Card Number:
83-50833

Edited by: *Lou Keglovits*
Illustrated by: *William D. Basham*

Printed in the United States of America.

Preface

The Apple computer is truly an amazing machine and programming the Apple borders on the realm of a mystical science. To explain how detailed and exacting programming is, let me bore you with a personal story about programming the Apple.

Once I took a course called Electrical Analysis (EE3301). It is a course designed to teach students to program mathematical and engineering problems using the FORTRAN language.

FORTRAN has several features such as single precision, double precision, and complex functions that are not available in the Applesoft language. One of the problems in the course dealt with converting complex impedances (total resistance and reactance in a circuit) to admittances, and using the admittance values to compute the current in each branch of the circuit. Capacitors and inductors have reactance, while resistors have resistance to current flow.

The class was divided into groups to develop a project to program. Naturally, being an Apple advocate, I wanted to simulate the complex function on the Apple computer using the Applesoft language.

The program was relatively easy for the group to write. The program ran and produced numbers, but the program had a bug in it. The bug was very difficult to find. The program ran beautifully, and crunched a bunch of numbers. The only thing was that the answers were incorrect. Not only were they incorrect, they changed in value each time the program was run. We searched for the bug, and had classmates search for the bug. The bug eluded us for about two weeks.

Finally, one of the members of the group had a friend who was an Apple expert. He looked at the program. At that time the program had three lines of code written in this order.

```
DEF FNA(X) = INT(X * 1000 + .5) / 1000
READ M,N
DIM A(M,N), B(N,N), C(M,N)
```

When the DEF FN statement was placed before the DIMension statement, each time the program was run, the "1000" inside the parentheses was changed to a different value. One time it was "1500," which produced an answer 50% greater than the actual answer. The next time it might be "1040," or "1020," or some different number.

The statements were placed in this order within the program.

```
READ M,N
DIM A(M,N), B(N,N), C(M,N)
DEF FNA(X) = INT(X * 1000 + .5) / 1000
```

Now each time the program ran, it produced the same answer.

Isn't it amazing that the computer operates in such an exact and precise mode?

At times programming can be the most frustrating and irritating thing that exists. But that's part of its beauty and challenge. For maximum performance and endurance on our journey, the mind must be continuously irritated, stimulated, and challenged. Is there anything worse than boredom?

Table of Contents

SECTION I — 40 COLUMN MODE

SECTION II — 80 COLUMN MODE

SECTION I

40 Column Mode

LESSON 1

Let's Get Started

OBJECTIVES

After completion of Lesson 1 you should be able to:

1. Turn on the Apple IIe computer and monitor.
2. Place the DOS 3.3 master disk in the disk drive and load the Integer BASIC language into memory. (The Apple IIe comes up with Applesoft language for use).
3. Be able to type in a simple program, RUN the program, and LIST the program on the screen.
4. Clear the computer's memory by typing NEW, and pressing RETURN.

VOCABULARY

Applesoft Language — An extended BASIC language that handles real numbers (instead of integers), and in most respects is an advance from the Integer BASIC language. (BASIC stands for Beginners All-purpose Symbolic Instruction Code, and was developed at Dartmouth College by John G. Kenney and Thomas E. Kurtz.)

Deferred Execution — This means that a line is to be executed at a later time. BASIC statements begin with a positive integer line number and are run in the deferred mode. (Example: 10 FOR J = 1 TO 5)

Immediate Execution — This means that a line is to be executed immediately. BASIC commands without a line number are run in the immediate execution. (Example: PRINT J, or PRINT 10)

Input/Output — Commonly referred to as I/O, and is a general term used to indicate communication with a computer. The process of transmitting data from an external source, such as keyboard, disk, or modem, to memory, and sending information from memory to an external device such as disk, printer, or modem.

Integer — An integer is any whole number, its negative, or zero. Integers never include decimal points, unless they are being expressed as real numbers.

Integer BASIC — A language that uses integers as the base and has limited string and array capabilities.

Interface Card — A card that is used as a common means of communicating between automatic data-processing parts of a single system. An interface card must be used to communicate between the disk drive and the computer's memory.

Interpreter — An interpreter translates program statements into a language that the computer can understand while the program is running. If there is an error in a program statement, the interpreter stops the program from running until the error is corrected. (Example: PUNT "J = ";J causes a SYNTAX ERROR)

Line Number — A line number is a positive integer that begins each program statement.

List — An immediate command that displays the entire program on the screen.

Logic — Logic is the science dealing with the formal principles or reasoning in electronic data processing. A program may run when there are no SYNTAX ERRORS, but the results may be incorrect because the logic is incorrect.

Memory — The term used to describe the internal storage locations within the computer.

Monitor – CRT – VDM – Screen — These are all words used to describe the television screen where the program and/or data is viewed. CRT stands for cathode ray tube. VDM stands for video display module.

NEW — NEW is the immediate execution command that erases the program stored in memory.

Program — A program is a set of instructions that allows the computer to solve a specific problem.

Program Statement — A program statement is an instruction to the computer preceded by a positive integer, called a line number.

Return — Pressing the RETURN key tells the computer that the action is finished, and to prepare for another action. When the RETURN key is pressed, two things happen, (1) the line just typed is entered as part of the program, and (2) the carriage moves to the beginning of the next line.

Slot — A slot is an opening into which an interface card is placed. The Apple IIe has seven (7) slots to accommodate interface cards.

40 Column — The Apple IIe screen is 24 rows by 40 columns. Many programs will be written using the 40 column mode.

80 Column — If an 80 column card is installed in the Apple IIe, the screen is 24 rows by 80 columns. The 80 column screen is used primarily for word processing using the Apple Writer II.

DISCUSSION

The Apple IIe is plugged into a three wire grounded socket. The Apple monitor is also plugged into a three wire grounded socket. The monitor is connected to the Apple IIe. The disk drive interface card is placed in input/output (I/O) slot #6, and the disk drive cable is attached to drive #1 on the interface card. If an 80 column card was purchased, it is placed in the auxiliary slot (related to slot #3). Specific instructions for assembling the Apple IIe configuration are found in the Apple IIe owners manual, Apple Computer Inc., 20525 Mariani Avenue, Cupertino, California, 95014.

TO ACTIVATE THE DISK OPERATING SYSTEM

The DOS 3.3 SYSTEM MASTER disk is placed in the disk drive, and the disk drive door is closed.

The monitor switch is placed in the "on" position. The Apple IIe switch, located at the left rear corner of the computer, is turned "on." The red light on the disk drive shines, the disk drive turns, and the read/write head loads the Integer BASIC language into the computer. This takes about four seconds and the screen information is shown in Fig. 1-1. When the information is loaded, the Applesoft prompt and cursor (]) appear on the screen, indicating that the Applesoft Language is the language to be used.

```
APPLE IIe
DOS 3.3 SYSTEM MASTER
JANUARY 1, 1983
COPYRIGHT APPLE COMPUTER, INC. 1980-1982
LOADING INTEGER BASIC
INTO MEMORY
```

Fig. 1-1. Screen information while loading integer BASIC into the computer memory.

If you are inexperienced with a computer, your first reaction is probably, now what do I do? Even an experienced computer operator has a lost feeling, the first time he or she turns on a new computer. The monitor, with its big eye, stares at you, and you stare back.

There are many details to learn. They can only be covered one step at a time, so let's start with a few basics.

Please type in the following program.

```
10 FOR J = 1 TO 5   PRESS RETURN
20 PRINT "J = ";J   PRESS RETURN
30 NEXT J           PRESS RETURN
40 END              PRESS RETURN
```

Now type in the word RUN, and press RETURN. If the program was typed correctly, these results appear on the screen.

```
RUN
J = 1
J = 2
J = 3
J = 4
J = 5
```

Very good, this programming is going to be easy for you.

Now let's make a mistake to see how the computer reacts. Please type in the following line.

```
20 PUNT "J = ";J PRESS RETURN
RUN
?SYNTAX ERROR IN LINE 20
```

The interpreter checked line 20, and found an error that would not allow the program to RUN. Anytime the interpreter finds an error (other than a logic error), the program stops running, and the computer prints out an error message (Lesson 3, Table 3-5).

Now retype line 20 correctly.

```
20 PRINT "J = ";J
```

Now type RUN, and press RETURN, to see if the program runs correctly. If the program does not run, make any corrections so that it does run. AT THIS STAGE, RETYPE THE ENTIRE LINE WHERE THE ERROR OCCURRED.

Now type LIST.

```
LIST
10 FOR J = 1 TO 5
20 PRINT "J = ";J
30 NEXT J
40 END
```

The program stored in memory appears on the screen when LIST (press RETURN) is typed on the screen.

Now type NEW (press RETURN).

Now type LIST and press RETURN. Nothing appears on the screen. The immediate execution command NEW clears the program from memory.

That's enough for the first sitting, however, let's summarize what we learned.

1. RETURN must be pressed after each immediate or deferred execution command is given to the computer. The RETURN tells the computer that one action is finished, and another one is about to begin.

2. The program statement must begin with a positive integer line number.
3. The program statement must be typed in the correct language before the computer will accept it to RUN in a program.
4. After a program is typed from the keyboard, typing RUN and pressing RETURN causes the program to execute.
5. To list all of a program stored in memory, type LIST, and then press RETURN.
6. To erase a program stored in memory, type NEW, and then press RETURN.

LESSON 2

Save and Load Programs on Disk

After completion of Lesson 2 you should be able to:

1. Initialize disks that are used to save and load programs.
2. Copy a disk.
3. Type a program on the screen, use the reserved word SAVE, to save the program on a disk.
4. Load a program stored on disk into the computer memory, by using the reserved word LOAD.
5. Use a limited number of disk operating commands such as CATALOG, RENAME, LOCK, and UNLOCK.

VOCABULARY

Booting DOS — The process of loading disk operating system commands into the Apple computer. Bootstrap is the technique of loading a program into a computer by means of certain preliminary instructions which in turn call in instructions to read programs, and/or data. The preliminary instructions are usually preset on a device (a disk, in this case), and called into action by the power "on" switch, or a special command from the keyboard, IN#6. Literally, the computer picks itself up "by its bootstraps."

Disk — A magnetic disk is a storage device that consists of a flat circular plate coated on both sides with some material (Mylar) that can be magnetized. A number of tracks (13 on DOS 3.2, and 16 on DOS 3.3) are available on the disk surface and data is read from or written to these tracks by means of a READ/WRITE head. The Apple IIe uses a single density, soft sectored, 5¼-inch disk as its virtual storage medium. The 5¼-inch floppy disk has a storage capacity of 118,000 bytes in the 3.2 disk operating system, and 146,000 bytes in the 3.3 disk operating system. The DOS 3.3 will store between 100 and 120 pages of normal text.

Directory — A directory is a translation table used to specify the size and format of files stored on the disk. Each record type and field type is

identified by a data file name. The disk is divided into thirty-five (35) tracks, three (3) of these tracks are used for the disk operating system, and one (1) track, #11, is used for the directory. The remaining thirty-one (31) tracks are for the programmer's use.

DOS — The disk operating system consists of a disk drive, or drives, an interface card that plugs into one of the eight input/output (I/O) slots in the Apple motherboard. The DOS interface card plugs into any slot numbered one (1) through seven (7). I/O slots #6, and #4 are primarily used for disk drives. When the disks are used in any manner, a request for use is made to the disk operating system. A software program handles the requests and is on the master disk, or any initialized disk.

Interface — Interface refers to the electronic connections between the computer and a peripheral unit such as a cathode ray tube (CRT, or screen), disk drives, modem, or printer. The interface is commonly referred to as an interface board that plugs into an input/output (I/O) slot. The cable from the peripheral unit plugs into the interface board.

Motherboard — A motherboard is a large insulating circuit board on which components, modules, or other electronic assemblies are mounted. Interconnections between board and components are made by welding, soldering, or other means.

ROM (Read Only Memory) — ROM is a fixed memory and is any type of memory which cannot be readily rewritten. The information in ROM is stored permanently and is used repeatedly. Such storage is useful for programs such as the disk operating system (DOS) boot program.

Write Protected — When a disk is write protected, it means that the disk can be read from, but it cannot be written to. (It has no cutout hole in the disk cover.) The Apple IIe DOS 3.3 MASTER is write protected.

The interface card between the computer and the disk drive is placed in input/output (I/O) slot #6 on the Apple motherboard. The interface card has two male plugs for attaching two disk drives. The boot disk drive cable is plugged into the plug marked "DRIVE #1." If there is a second disk drive, its cable is plugged into the plug marked "DRIVE #2." On a two disk drive system, the boot drive is referenced as DRIVE #1 (,D1, or SLOT #6, DRIVE #1 ,S6,D1). The other drive on a two disk drive system is referenced as "DRIVE #2" (DRIVE #2 ,D2, or SLOT #6,DRIVE #2 ,S6,D2)

BOOT THE SYSTEM

To boot (bring up) the disk operating system (DOS), the disk drive door is opened gently, the master disk (or initialized disk) is placed in the disk drive gently, and the disk drive door is closed gently.

Many operators prefer to open the disk drive door, insert the master disk, and LEAVE THE DISK DRIVE DOOR OPEN until after the power switch has

been turned "on." When the power is turned "on," the read/write head moves slightly and may damage the disk. After the power has been on for a second or more, the disk drive door is closed so the disk can be read to boot the system.

When the disk operating system is booting, the red light on the disk drive is turned "on," and the cursor disappears from the screen. When the disk operating system is loaded into memory, the red light on the disk drive turns "off," and the cursor reappears on the screen.

INITIALIZE A DISK

If this is the first time you have booted DOS from the master disk, it is important that you learn to initialize a disk for your own use. The master disk is WRITE protected, so you cannot write to it, only read from it. A WRITE protected disk has no square cutout hole on the right side when you are facing the label on the disk.

To initialize a disk on a one disk drive system:

1. Take the master disk out of the disk drive.
2. Place the disk to be initialized into the disk drive.
3. Type the phrase, INIT HELLO, so it appears on the screen. (INIT is a reserved word used to initialize a disk.)
4. Press RETURN.

To initialize a disk on a two disk drive system with the master disk in disk drive #1 (boot drive):

1. For precaution, open the door on disk drive #1 (boot drive). Even though the master disk is write protected, it is a good habit to open the door on the disk drive that is not being used. Many times when you are using DOS you will forget what you want to do. If the disk drive door is open and you send information to the wrong disk, the DOS system will print, I/O ERROR, on the screen. This I/O ERROR message makes you think and realize what action should be taken to perform the correct task.
2. Place the disk to be initialized in disk drive #2.
3. Type the phrase, INIT HELLO,D2, so it appears on the screen.
4. Press RETURN.

After RETURN is pressed, the red light on the #2 disk drive is turned "on," the cursor disappears from the screen, the stepper motor rotates the disk at about 360 revolutions per minute, and the disk is initialized to thirty-five tracks. Each track is broken into thirteen (13) sectors in DOS 3.2, and sixteen (16) sectors in DOS 3.3.

The initialized disk has a directory which holds all the information about programs or files that are stored on the disk. When a new program or file is

placed on the disk, the directory is updated to contain the information.

After the disk is initialized, the red light on the disk drive is turned "off," and the cursor reappears on the screen.

You can see what is on the disk by typing one of the following messages:

1. CATALOG — The command used to see what is listed on a disk (on a one disk drive system, or the disk drive that was last accessed on a two disk drive system).
2. CATALOG,D2 — The command used to see what is listed on a disk in disk drive #2.
3. CATALOG,D1 — The command used to see what is listed on a disk in disk drive #1.

CATALOG A DISK

When you type CATALOG (CATALOG,D2) and press RETURN, the following information is written on the screen.

```
CATALOG
DISK VOLUME 254
A 002 HELLO
■ (flashing cursor)
```

The "A" indicates the "HELLO" program is in the Applesoft language. The "002" means the program takes up two (2) sectors.

SLOT, DRIVE, AND VOLUME OPTIONS

When using the INIT command to initialize a disk, there are three options that can be used: slot number, drive number, and volume number. The volume number option is especially useful when you want to number your disks in a specific manner.

```
]INIT HELLO,S6,D2,V3
]CATALOG
DISK VOLUME 003
A 002 VOLUME 003
■ (flashing cursor)
```

On a one disk drive system, INIT HELLO,V3 produces volume #3, since the DOS system only sees slot #6, drive #1.

On a two disk drive system, INIT HELLO,D2,V3, produces the initialized volume #3 in disk drive #2, since the DOS system sees the interface card in I/O slot #6.

COPY A DISK

Apple Computer suggests that a copy be made of the DOS 3.3 MASTER DISK. The copy is used to boot DOS and the master is placed in a safe place

in case something happens to the copy of the master. It is always a good idea to make a copy of any important programs or data. If the original program or data is accidentally destroyed, the backup copy can be used.

On the DOS 3.3 MASTER DISK is a program to copy the contents of an entire disk. The program is called COPYA.

To copy the DOS 3.3 MASTER DISK, place the DOS 3.3 MASTER DISK in the disk drive and type RUN COPYA (press RETURN). The COPYA (A for Applesoft) is loaded into memory. The screen contains the following statements.

```
                    APPLE DISK DUPLICATION PROGRAM

ORIGINAL SLOT:   DEFAULT = 6  (PRESS RETURN)
        DRIVE:   DEFAULT = 1  (PRESS RETURN)
DUPLICATE SLOT:  DEFAULT = 6  (PRESS RETURN)
        DRIVE:   DEFAULT = 2  (if you have a one drive system
                              you must type in a 1 to change
                              the default value)
                              PRESS RETURN

 — PRESS 'RETURN' KEY TO BEGIN COPY —
```

When RETURN is pressed, the message, INSERT ORIGINAL DISK AND PRESS RETURN, appears on the screen. A portion of the DOS 3.3 MASTER DISK is loaded into memory.

When a portion of the DOS 3.3 MASTER DISK is loaded into memory (READ from disk), a message appears on the screen, INSERT DUPLICATE DISK AND PRESS RETURN. When a disk is placed in the disk drive, the COPYA program INITIALIZES the disk before writing to it. After the disk is initialized, a portion of the DOS 3.3 MASTER DISK is written to the copy. After the write procedure is completed, INSERT ORIGINAL DISK AND PRESS RETURN, is written to the screen. Alternating the original and the copy continues until all the original DOS 3.3 MASTER DISK is placed on the copy. The DOS 3.3 MASTER DISK must be read six times, and the copy must be written to six times, before the procedure is complete. With a two, or more, disk drive system, the MASTER and copy would have been inserted one time for the complete operation. After the copy is made, DO YOU WISH TO MAKE ANOTHER COPY?, appears on the screen. "N," for no, terminates the program.

FILE NAMES

In DOS the program must have a name that follows a certain pattern. A legal file name in DOS must be from one (1) to thirty (30) characters in length. The file name must begin with a letter from "A to Z," followed by any alphabetic character, a number from one (1) to zero (0), or any other character except a comma (,). A comma is the character reserved for the

slot, drive, and volume options. The command takes everything preceding the comma as a file name (even control characters).

LEGAL FILE NAMES

LOOP

LOOP1

JOHN DOE

M1000

ILLEGAL FILE NAMES

1001 — starts with a number.

JOHN DOE, PhD — contains a comma

A NAME LONGER THAN THIRTY CHARACTERS IS TRUNCATED TO THIRTY CHARACTERS

Now let's write a program, and SAVE it to disk.

```
10 FOR J = 1 TO 5
20 PRINT "J = ";J
30 NEXT J
40 END
```

Let's name the program LOOP.

SAVE A PROGRAM TO DISK

To save the program to disk, type SAVE LOOP (or SAVE LOOP,D1 – or SAVE LOOP,D2), and press RETURN. The cursor disappears, the red light on the disk drive goes "on," and the program is written to disk. When the program has been written to disk, the red light goes "off," and the cursor reappears on the screen. The DOS system requires that the program be named before it can be SAVEd. SAVE LOOP is the proper command when using DOS.

DOS remembers which disk drive was accessed last. If disk drive #2 was last read from or written to, then SAVE LOOP is directed to disk drive #2. To place LOOP on the disk in disk drive #1, use the command, SAVE LOOP,D1. The last disk accessed is the default drive. The default drive does not require that the drive option be placed on the command.

CLEAR COMPUTER MEMORY

If you want to clear memory to write another program, or to load a program from disk type NEW, and press RETURN. The NEW command clears memory, but does not disturb the DOS system. Memory should be cleared at any time a situation arises that requires a different memory use.

If you want to save a program, and then do more work on it, you do not need to clear memory. You can add to, change, or delete part of a program, and then save the program again. The last version SAVEd will be the program on disk.

LOAD A PROGRAM FROM DISK

Before loading a program type NEW, and press RETURN, so that memory is cleared.

Now, let's LOAD LOOP from disk. Type:

LOAD LOOP — if the LOOP program is on the disk in the default disk drive,
LOAD LOOP,D1 — if LOOP is on a disk in disk drive #1,
LOAD LOOP,D2 — if LOOP is on a disk in disk drive #2, and press RETURN.

Entering the command LOAD LOOP, and pressing RETURN loads the program into the computer memory. After the program is loaded, type LIST, and the program is listed to the screen. The LIST and EDIT functions are in Lesson 17.

RUN A PROGRAM FROM DISK

If you prefer to run the program directly from disk, type RUN LOOP, and press RETURN. The program is loaded into memory and run.

RENAME A PROGRAM ON DISK

To change the name of a program saved on disk, use the RENAME command, in this format.

RENAME LOOP, LOOP1 — if LOOP is on a disk in the default disk drive.
RENAME LOOP,LOOP1,D2 — to direct the command to the disk in disk drive #2.
RENAME LOOP,LOOP1,D1 — if LOOP is in a disk in disk drive #1.

If LOOP1 is already on the disk, and you use the command RENAME LOOP,LOOP1, you will have two files named LOOP1 on the disk. The RENAME command does not look through the directory to see which files are already on the disk.

DELETE A PROGRAM ON DISK

To delete the program named LOOP that has been saved on a disk, type, DELETE LOOP, and press RETURN. The cursor disappears, and the disk red light goes "on," and the program is deleted from the disk. When the deletion is complete, the red light on the disk drives goes "off," and the cursor returns to the screen.

LOCK A PROGRAM

To LOCK the program named LOOP, type LOCK LOOP, and press RETURN.

```
LOCK LOOP
CATALOG
DISK VOLUME 003
A 002 HELLO
*A 002 LOOP
```

When a CATALOG is commanded, an asterisk (*) appears to the left of the "A." This asterisk (*) indicates that the program LOOP is locked, and cannot be deleted, or renamed.

```
DELETE LOOP or (RENAME LOOP, LOOP1)
FILE LOCKED
```

If the disk is reinitialized, the program named LOOP will be lost. Initialization destroys all data on the disk.

UNLOCK A PROGRAM

To unlock the program named LOOP, so that it can be deleted or renamed, type UNLOCK LOOP.

```
UNLOCK LOOP
CATALOG
DISK VOLUME 003
A 002 HELLO
A 002 LOOP
```

When the disk is CATALOGed, the asterisk (*) has been removed, and the program named LOOP, can be deleted or renamed.

For more instructions on the DOS please refer to *Apple II, The DOS Manual, Disk Operating System,* 1980, 1981 by APPLE COMPUTER INC., 10260 BANDLEY DRIVE, CUPERTINO, CALIFORNIA, 95014.

LESSON 3

Reference Library of Editing Functions

DISCUSSION

The Apple IIe computer has special keys, CAPS LOCK, CONTROL, CURSOR ARROWS (←,→,↓,↑), DELETE, ESCAPE, OPEN APPLE (), SOLID APPLE (), AND FOUR SPECIAL CHARACTER KEYS ('~),([{),(]}),(\/). The use of these keys and characters will be explained in the alphabetical order of their first letter. The following library is only a partial list of the special functions of the Apple IIe. For a complete detailed list of all applications, Apple IIe software manual should be researched.

CAPS LOCK (40 column mode)

The CAPS LOCK key should be in the locked position when the DOS 3.3 MASTER DISK is booted. The CAPS LOCK key should be locked down when writing programs in Applesoft. An Applesoft program written in lower case (CAPS LOCK KEY UNLOCKED) will give a SYNTAX ERR when the offending program statement reaches the interpreter. In the unlocked position the CAPS LOCK key can be used to place lower case letters in PRINT and DATA statements.

```
10 PRINT "John Doe"
20 END
RUN
John Doe
DATA John Doe, (or "John Doe")
```

If the CAPS LOCK key is in the unlocked position, the shift key can be used to capitalize letters, and then downshifted to make lower case letters.

CAPS LOCK (80 column mode)

The 80 column mode (if the 80 column card is in the auxiliary slot) can be activated by typing PR#3, or IN#3. Once the 80 column card is activated, the mode can be changed from 40 to 80 column modes by either of two

ways. (1) ESCAPE 40 causes a return to the 40 column mode, and ESCAPE 80 returns to the 80 column mode. (2) CONTROL Q causes a return to the 40 column mode, and CONTROL R returns to the 80 column mode. The 80 column card is deactivated by pressing ESCAPE CONTROL Q. Another way to deactivate the 80 column card is to press CONTROL RESET. When CONTROL RESET is used to deactivate the 80 column card the program stored in memory is destroyed.

Programs written in the 80 column mode must be written in capital letters for the program to execute. Lower case can be used in PRINT and DATA statements.

CONTROL

RESET cannot be activated directly. CONTROL RESET must be pressed for the RESET to function. CONTROL functions are listed in Table 3-1 and Table 3-3.

Table 3-1. Control Functions 40 Column Mode (80 Column Card Inactive)

CONTROL CHARACTER	APPLE IIe NAME	WHAT FUNCTION IS EXECUTED
CONTROL C		Causes the program to stop running.
CONTROL G	BELL	Causes the bell to ring.
CONTROL H	DOWN ARROW	Causes the cursor to move one line down.
CONTROL I		Horizontal tab.
CONTROL J	DOWN ARROW	Line feed.
CONTROL K	UP ARROW	Vertical tab.
CONTROL L		Form feed.
CONTROL M	RETURN	The same as the RETURN character.
CONTROL S		Causes the program to stop running or listing.
CONTROL U	RIGHT ARROW	Negative acknowledge.
CONTROL X		Causes the line that is being typed to be deleted.
CONTROL [	ESC	Escape.

ARROW KEYS (40 and 80 column mode)

The Apple IIe has four arrow keys, LEFT, RIGHT, DOWN, and UP. The LEFT and RIGHT arrow keys move the cursor to the left or right simply by pressing the key. A built in REPEAT function can be activated by maintaining pressure on the arrow key. The DOWN arrow key will move down simply by pressing the down arrow key. The UP arrow key will move up after ESCAPE is pressed. ESCAPE functions are listed in Table 3-2 and Table 3-4.

Table 3-2. Escape Functions 40 Column Mode (80 Column Card Inactive)

CONTROL CHARACTER	APPLE IIe NAME	WHAT FUNCTION IS EXECUTED
ESCAPE SHIFT @		Clears the screen and places the cursor at row #1, column #1, and leaves the escape mode.
ESCAPE A		Moves cursor right one position, leaves escape mode.
ESCAPE B		Moves cursor left one position, leaves escape mode.
ESCAPE C		Moves cursor down one position, leaves escape mode.
ESCAPE D		Moves cursor up one position, leaves escape mode.
ESCAPE E		Clears to the end of the line.
ESCAPE F		Clears to the end of the screen.
ESCAPE I		Moves cursor one line up, remains in escape mode.
ESCAPE J		Moves cursor one space left, remains in the escape mode.
ESCAPE K		Moves cursor one space right, remains in the escape mode.
ESCAPE M		Moves cursor one line down, remains in the escape mode.
ESCAPE ARROW KEYS ARE THE SAME AS I, J, K, M		

DELETE KEY

The DELETE key is used with the Apple Writer II for word processing. When using the Apple Writer II, the delete key used by itself is a destructive delete key (text cannot be retrieved).

The delete key cannot be successfully used to delete program statements in the 40 or 80 column mode.

ERROR MESSAGES

A list of error messages can be viewed in Table 3-5.

OPEN APPLE KEY

OPEN APPLE CONTROL RESET is a power-on start. Using OPEN APPLE CONTROL RESET causes the program in memory to be lost.

The OPEN APPLE key is used with Apple Writer II in conjunction with the question mark (?) to get to the Apple Writer II help section.

The OPEN APPLE key is used in conjunction with Apple Writer II and the back arrow to place 128 characters in the text buffer, and the OPEN APPLE and the forward arrow are used to retrieve 128 characters from the text buffer and place the text on the screen.

Table 3-3. Control Functions 80 Column Card Active

CONTROL CHARACTER	APPLE IIe NAME	WHAT FUNCTION IS EXECUTED
CONTROL H	BACKSPACE	Moves the cursor one position to the left; from the left edge of the window, moves to the right end of the line above.
CONTROL J	LINE FEED	Moves cursor down to the next line in the window; scrolls down if needed.
CONTROL K	CLEAR EOS	Clears the cursor position to the end of the window.
CONTROL L	CLEAR	Moves the cursor position to the upper left corner of the window and clears the window.
CONTROL M	RETURN	Moves the cursor position to the left end of the text line in window, and scrolls.
CONTROL N	NORMAL	Sets the display to the normal mode.
CONTROL O	INVERSE	Sets the display to the inverse mode.
CONTROL Q	40 COLUMN MODE	Sets the screen to the 40 column mode after PR#3 has activated the 80 column card.
CONTROL R	80 COLUMN MODE	Sets the screen to the 80 column mode after PR#3 has turned the 80 column card on.
CONTROL S	STOPS LIST	Stops the program listing.
CONTROL U	QUIT	Deactivates the 80 column card, clears the screen, and the cursor goes to row 1, column 1.
CONTROL V	SCROLL	Scrolls the program down one line, but leaves the cursor in the same position.
CONTROL W	SCROLL UP	Scrolls the program up one line, but leaves the cursor in the same position.
CONTROL X		Causes the line being typed to be deleted.
CONTROL Y	HOME	Moves the cursor to the upper left corner of the window.
CONTROL Z	CLEARS LINE	Moves the cursor position to the right edge of the window.

REPEAT KEY

The REPEAT key is a built in function so all keys repeat themselves when slight extra pressure is placed on the key.

RESET

The SOLID APPLE key is used in combination with the RESET key (when pressed by itself, the RESET key performs no function) and the CONTROL key. When the CONTROL RESET key combination is pressed, the action taking place stops, and the cursor returns to the screen.

SOLID APPLE

The SOLID APPLE key is used in combination with CONTROL RESET for memory self test. When the SOLID APPLE CONTROL RESET combination

Table 3-4. Escape Functions 80 Column Card Active

ESCAPE FUNCTION	WHAT THE FUNCTION DOES
ESCAPE SHIFT @	Clears the screen window and moves the cursor to row 1, column 1.
ESCAPE A	Moves the cursor one position to the right.
ESCAPE B	Moves the cursor one position to the left.
ESCAPE C	Moves the cursor one position down.
ESCAPE D	Moves the cursor one position up.
ESCAPE E	Clears to the end of the line.
ESCAPE F	Clears to the bottom of the window.
ESCAPE I	Moves the cursor one line up.
ESCAPE J	Moves the cursor one space left.
ESCAPE K	Moves the cursor one space right.
ESCAPE M	Moves the cursor down one line.
ESCAPE R	Turns on the upper case restrict mode after PR#3 turns on the 80 column card.
ESCAPE T	Turns off the upper case restrict mode.
ESCAPE 40	Switches to a 40 column mode.
ESCAPE 80	Switches to an 80 column mode.
ESCAPE CONTROL Q	Deactivates the 80 column card.
ESCAPE ARROW KEYS ARE THE SAME AS I, J, K, M	

Table 3-5. Error Messages 40 Column or 80 Column Mode

TYPE OF MESSAGE	PROBABLE REASON
BAD RESPONSE TO INPUT	Out of range, string to array, etc.
BAD SUBSCRIPT	An attempt is made to reference an array element which is outside the dimensions of the array. This error can occur if the wrong number of dimensions are used in an array reference. For example, X(3,3) = 5.6 when the array was dimensioned DIM X(6).
CAN'T CONTINUE	An attempt was made to continue the program with the CONT command and no program was in memory. CONT was used after an error, or after the program had been changed, deleted from, or added to.
CONTROL C INTERRUPT	The program was stopped by pressing CONTROL C. The line number where the program stopped is displayed on the screen.
DIVISION BY ZERO	An attempt was made to divide by zero. This often occurs when a variable is used in an arithmetic expression before it is initialized to a value other than zero.

Table 3-5–cont. Error Messages 40 Column or 80 Column Mode

TYPE OF MESSAGE	PROBABLE REASON
FORMULA TOO COMPLEX	More than two statements of the form If "AA" THEN were executed where "AA" is a quoted string. Applesoft IF – THEN was not intended to be used with strings.
FORMAT ERROR	Immediate execution statement — ?SYNTAX ERROR Deferred execution statement — ?SYNTAX ERROR (in line) 30
ILLEGAL DIRECT	An attempt was made to use one of the following statements in immediate execution — DEF FN, GET, INPUT, ONERR GOTO, READ, RESUME.
ILLEGAL QUANTITY	The parameter passed to a math or string function was out of range, negative array subscript, using a LOG with a negative or zero number, using SQR with a negative number, raising a negative number to a power and not using an integer power, using a string with an improper argument.
NEXT WITHOUT FOR	The NEXT statement variable is missing or did not correspond with the FOR statement at the beginning of the loop.
OUT OF DATA	READ statement was executed but there was no data left to be read. A RESTORE statement will cause the DATA to be reset from the first item.
OUT OF MEMORY	Program too large, too many variables, FOR–NEXT loops nested over ten deep, GOSUBs nested more than 24 levels deep, expression too complicated, parentheses nested more than 36 levels deep, LOMEM and HIMEM errors.
OVERFLOW	Numbers are too large for the computer to handle.
REDIMENSIONED ARRAY	Dimensioned the same array more than once.
RETURN WITHOUT GOSUB	A RETURN statement was encountered without a corresponding GOSUB statement being executed.
STRING TOO LONG	The maximum for a string is 255 characters.
SYNTAX ERR (and a beep)	This syntax refers to the DOS.
?SYNTAX ERR	The "?" before the message refers to APPLESOFT.

Table 3-5–cont. Error Messages 40 Column or 80 Column Mode

TYPE OF MESSAGE	PROBABLE CAUSE
***SYNTAX ERR	The "***" refers to INTEGER BASIC. A SYNTAX ERROR is caused by a missing parentheses in an expression, an illegal character in a line, an incorrect punctuation, etc.
TYPE MISMATCH	The left side of an argument was a string variable and the left side was a numeric variable, or vice versa.
UNDEFINED STATEMENT	A DEF FN function was used, but it was never defined.
FOR A COMPLETE LIST OF ERROR MESSAGES AND NUMBERS REFER TO: Applesoft BASIC Programmer's Reference Manual — Volume 2 — For the Apple IIe Only	

is pressed in the correct sequence, memory is tested. If memory is OK, the message KERNEL OK is written on the screen. Any other message means the computer requires service.

After memory tests are completed, the computer must be restarted, either by using the off/on switch, or the ESCAPE CONTROL RESET combination.

When using the Apple Writer II, the SOLID APPLE key is used in conjunction with the TAB key. The combination causes the cursor to pass over the existing text to the next tab position. For more detail see the Apple Writer II manual.

TAB KEY

The TAB key is used with the Apple Writer II to tab from one tab position to the next, after the tab stops have been set.

LESSON 4

Print Rules

After completion of Lesson 4 you should be able to:

1. Write a program in Applesoft using PRINT statements.
2. Define and properly use the print rules pertaining to PRINT statements.

VOCABULARY

Applesoft BASIC — Applesoft IIe basic is a more extended, comprehensive, and flexible language than Integer BASIC.

Command — Command is synonymous with instruction. It is a word directing the computer to perform a specific action.

Documentation — Documentation is the total history of a program and its component parts from inception to completion. Documentation enables another programmer to understand the program.

Delimiters — Delimiters are the signals that tell the computer how closely the output is to be printed, i.e., the comma, and the semicolon.

Format — The format is the prearranged assignment of data. The format statements determine how the output will be printed.

Line Number — A line number is a positive integer, from zero (0) to 63999, used to begin a program statement. Each program statement must begin with a line number. The statements must be ordered from the least line number to the highest.

PRINT — The PRINT statement causes the data to be output.

REM — The REM statement allows comment within the program but produces no action in the program. In other words, "REM" reminds the programmer what the program does. You can type any comment in a REM statement.

Semicolon — The semicolon prevents the cursor from moving after output is completed. The semicolon leaves a PRINT line "open," and inhibits automatic repositioning of the cursor.

STATEMENT — A statement is an instruction that requires a line number, and it tells the computer what action to take.

DISCUSSION

The first objective of Lesson 4 is to write a program using a PRINT statement.

A program is a set of instructions developed to solve a specific problem. In this example, the problem is to print out the statement, "THIS IS THE USA."

```
5  REM — PROGRAM EXPLAINING 'PRINT' RULES
10 PRINT "THIS IS THE USA"
999 END
```

Now that was simple wasn't it? The number "5" is the first line number of the first program statement, in this program. The first line number doesn't have to start with number "0." You can use any beginning number up to "63999." Any positive integer greater than "63999" is out of the range of the machine, and will produce a "?SYNTAX ERR". Succeeding program statements must have higher line numbers than previous statements. The most practical way is to number every line in multiples of ten. This way you will be able to insert extra program statements if you want to expand your program. On the DOS 3.3 MASTER DISK, the RENUMBER program will renumber a program.

The REM statement helps the programmer to document the program.

The line **10 PRINT "THIS IS THE USA"** is a program statement that outputs the information enclosed in the quotation marks.

The line **999 END** is the end of the program. In most cases, a program will run successfully without an END statement, but it is recommended that an END statement always be used.

Now type in line 10 without quotation marks.

```
10 PRINT THIS IS THE USA (No quotation marks)
999 END
RUN
0 (Zero)
```

The output is zero (0) because the computer reads THIS IS THE USA as a variable that has zero (0) value. Variables are used to hold different values as the program run progresses.

Now type in line 10 using a beginning quote and no closing quote.

```
10 PRINT "THIS IS THE USA
999 END
RUN
THIS IS THE USA
```

THIS IS THE USA prints even though there is no closing quote. Applesoft is flexible enough to let you "get away" without a closing quote.

Now type line 10 with a no beginning quote but with a closing quote.

```
10  PRINT THIS IS THE USA''
999 END
RUN
0
```

The output is zero (0) because the computer recognizes THIS IS THE USA as an uninitialized variable and the ending quote is ignored.

Retype line 10 again, this time spelling PRINT incorrectly, and see what happens.

```
10  PUNT ''THIS IS THE USA''
20  PRINT (This print causes a line feed — a space between lines)
999 END
RUN
?SYNTAX ERROR IN LINE 10
```

Now that you have all the errors out of your system, on to the print rules. This program was written and tested line by line so the student can view the results produced by each program statement.

1. Anything in quotation marks is printed exactly as in the PRINT statement when the program is RUN.

```
10  PRINT ''THIS IS THE USA''
20  PRINT
999 END
RUN
THIS IS THE USA
```

2. PRINT statement with no punctuation following the closing quote, causes the output to be printed on one line and causes the computer to line feed (space down). Consecutive PRINT statements with no closing punctuation causes the output to be printed vertically, one output below the other.

```
30 PRINT ''THIS IS THE''
40 PRINT ''UNITED STATES''
50 PRINT ''OF AMERICA''
60 PRINT
RUN
THIS IS THE
UNITED STATES
OF AMERICA
```

3. A comma placed at the end of a print statement places the output in separate fields on the same line. Applesoft is designed to divide each line into 3 fields. The first field begins in column 1, and ends at column 16. The second field begins at column 17 and ends at column 32. The third field begins at column 33 and ends at column 40.

```
70 PRINT "THIS IS THE",
80 PRINT "USA"
90 PRINT
RUN
THIS IS THE       USA
```

4. A semicolon placed at the end of a PRINT statement causes the output to be packed (no space between characters).

```
100 PRINT "THIS IS THE";
110 PRINT "USA"
120 PRINT
RUN
THIS IS THEUSA
```

5. A comma between the two items in a PRINT statement places the output in the next available field.

```
130 PRINT "THIS IS THE","USA"
140 PRINT
RUN
THIS IS THE       USA
```

6. A semicolon between two items in a PRINT statement causes the output to be packed (no space).

```
150 PRINT "THIS IS THE";"USA"
160 PRINT
RUN
THIS IS THEUSA
```

(Note: Examples 3 and 5 give the same output, but are produced by different program statements. Examples 4 and 6 give the same output, but are produced by different program statements.)

7. Spaces placed between quotation marks and the item will be in the same relationship as in the output. (X's are placed in the PRINT statement to represent blank spaces).

```
170 PRINT "XXTHIS IS THEX";"USA"
190 PRINT
RUN
XXTHIS IS THEXUSA (X's represent blank spaces)
```

8. A PRINT following a PRINT statement closes out a line. (A PRINT following a PRINT statement with no punctuation causes a line feed, i.e., space between the lines. A PRINT following punctuation closes out the line, but does not cause a line feed).

```
200 PRINT "THIS IS THE";
210 PRINT
220 PRINT "USA"
RUN
```

```
THIS IS THE
USA
```

9. With the upper and lower case mode, output can be in the upper and lower case, when using PRINT statements.

```
230 PRINT : PRINT "John Doe" : PRINT
RUN
John Doe
```

10. To print variables with assigned values NO quotation marks are used and the assigned values are printed.

```
240 A = 5 : B = 10 : C = 15
250 PRINT A
260 PRINT B
270 PRINT C
280 PRINT
290 PRINT A,B,C
300 PRINT
310 PRINT A;B;C
RUN
5
10
15
5             10            15
51015
```

The same punctuation rules that apply to items enclosed in quotation marks apply to variables.

1. No punctuation prints vertically.
2. Commas after variables print in three columns.
3. Semicolons after variables pack the output leaving no space between numbers.

The complete program and RUN appears in Fig. 4-1.

```
5   REM — PROGRAM EXPLAINING 'PRINT" RULES
10  PRINT "THIS IS THE USA"
20  PRINT
30  PRINT "THIS IS THE"
40  PRINT "UNITED STATES"
50  PRINT "OF AMERICA"
60  PRINT
70  PRINT "THIS IS THE",
80  PRINT "USA"
90  PRINT
100 PRINT "THIS IS THE";
110 PRINT "USA"
120 PRINT
130 PRINT "THIS IS THE ","USA"
```

Fig. 4-1. Print rules program.

```
140 PRINT
150 PRINT "THIS IS THE";"USA"
160 PRINT
170 PRINT "XXTHIS IS THEX";"USA"

190 PRINT
200 PRINT "THIS IS THE";
210 PRINT
220 PRINT "USA"
230 PRINT : PRINT "John Doe": PRINT

240 A = 5:B = 10:C = 15
250 PRINT A
260 PRINT B
270 PRINT C
280 PRINT
290 PRINT A,B,C
300 PRINT
310 PRINT A;B;C
999 END
RUN
THIS IS THE USA

THIS IS THE
UNITED STATES
OF AMERICA

THIS IS THE    USA

THIS IS THEUSA

THIS IS THE    USA

THIS IS THEUSA

XXTHIS IS THEXUSA

THIS IS THE
USA

John Doe

 5
 10
 15

 5             10            15

 51015
```

Fig.4-1–cont. Print rules program.

LESSON 5

Variables

After completion of Lesson 5 you should be able to:

1. Define the variables used in Applesoft.
2. Distinguish between variables and reserved words.
3. Understand the relationship between integers and reals and how truncation affects mathematical calculations.
4. Use INT and DEF functions to round off calculations.

VOCABULARY

DEF FN — This allows the programmer to define functions within the program.

Deferred Execution — This means that a line is to be executed at a later time. BASIC statements with a line number are run in the deferred execution mode.

Immediate Execution — This means that a line is to be executed immediately. BASIC commands without a line number are run in the immediate execution mode.

Integer — This is any whole number, its negative, or a zero. Integers never include decimal points, unless they are being expressed as real numbers.

Literal — This is a sequence of characters enclosed in quotation marks. In A$ = "HELLO," A$ is a variable, HELLO is a string, and "HELLO" is a literal. See the definition of STRING.

Real — This is any number, including integers, that can be written with a decimal point.

Scientific Notation — This is the method of expressing numbers as a power of ten. In scientific notation, the number 1234 is 1.234×10^3, and the number 0.001234 is 1.234×10^{-3}. Applesoft uses the symbol "E" to indicate that the number before the "E" is to be multiplied by ten raised to power indicated after the "E". For instance, 1×10^{11} is expressed by Applesoft as 1E + 11.

String — This is any sequence of characters.

Truncate — This is to drop off the digits from a real number (1.3456) and produce an integer (1). In this case, the .3456 was truncated to form the integer (1). Truncation is different from rounding. If the real (1.3456) was rounded to two places, the result would be 1.35.

DISCUSSION

A variable, according to Webster is:

1. Something that is variable.
2. A quantity that may assume any one of a set of values.
3. A symbol representing a variable.

In Applesoft, a variable can be an alpha (alphabet) character (A through Z), two alpha characters (AA), or an alpha and a numeric (0 through 9) character, unless these characters are part of a word reserved specifically for the Applesoft language.

LEGAL APPLESOFT VARIABLES

A	BB	C1	Z2

LEGAL APPLESOFT VARIABLES — but only the first two characters are recognized by the Applesoft language.

DOE	S1M	SU3	PER

ILLEGAL APPLESOFT VARIABLES — these are reserved words.

ABS	AND	CALL	DEL
LET	LOAD	SAVE	VTAB

A complete list of reserved words is found on page 122 of the *APPLESOFT II BASIC PROGRAMMING REFERENCE MANUAL,* published by Apple Computer, Inc.

What happens when a reserved word is used as a variable?

```
10 FOR = 5
20 PRINT FOR
30 END
RUN
?SYNTAX ERROR IN 10
```

The syntax error message is produced and the program does not run. One deficiency of the Applesoft language is that it does not give an error message until the program is run. A better method would be to give an error message when the statements are input.

TYPES OF APPLESOFT VARIABLES

TYPE	EXAMPLE
INTEGER	A% = 1
REAL	A = 1.23
STRING	A$ = "B2.5"

An integer is any of the natural, or whole, numbers. The numbers 1, 2, 3, and 5 are integers. In Applesoft, integers must be in the range of −32767 to +32767 or the computer will give an ILLEGAL QUANTITY ERR because you are out of the range of its capabilities. Fig. 5-1 is a program written to demonstrate the limits of the Apple computer. Any number less than −32767, or greater than +32767 gives an ILLEGAL QUANTITY ERROR, in a specific line number. If A% had been given a value of −32768, the program would not run, and would have printed out the error message, ILLEGAL QUANTITY ERROR IN LINE 230.

```
ANY INTEGER LESS THAN -32767 OR GREATER THAN +32767 IS AN
ILLEGAL VALUE AND WILL PRINT THE ERROR MESSAGE — ?ILLEGAL
QUANTITY ERROR IN LINE???
230  A% = -32767
240  B% = 32767
250    PRINT
260    PRINT A%,B%
270    END

-32767          32767
```

Fig. 5-1. Program to demonstrate integer range.

Applesoft language uses the percent sign to indicate an integer variable. Fig. 5-2 demonstrates that A% = 10 is an integer quantity.

```
120  A% = 10
130    PRINT
140    PRINT "RESULTS ARE = ";A%
150    END

RESULTS ARE = 10
```

Fig. 5-2. Program to demonstrate integer function.

Fig. 5-3 is a program to demonstrate how an integer variable truncates a real number. B% is an integer variable. The value in B% equals 3.1416 and is a real (fractional) value. The B% is given the value 3.1416 in Fig. 5-3, line 120, but it truncates the real number and outputs it as an integer (3). An integer variable converts a positive real toward the lower value.

Fig. 5-4 is a program written to demonstrate how an integer variable truncates a negative real. C% = −0.843. When the program is run, the negative real (−0.8430) is truncated to a negative one (−1). Applesoft truncates

```
120  B% = 3.1416
130    PRINT
140    PRINT "RESULTS ARE = ";B%
150    END

RESULTS ARE = 3
```

Fig. 5-3. Program to demonstrate truncation by the integer (INT) function.

```
120  C% = -0.843
130    PRINT
140    PRINT "RESULTS ARE = ";C%
150    END

RESULTS ARE = -1
```

Fig. 5-4. Program showing how a negative real number is truncated by the INT function.

negative reals to negative integers by converting them down toward the next whole number.

Fig. 5-5 shows how the output results differ when the area of a circle is calculated with integers or reals. A% = PI% * R% ∧ 2 (R% = 3) calculates the area of a circle using integers. The output of this calculation is twenty-seven (27) square inches. A = PI * R ∧ 2 (R = 3) calculates the area of a circle using real variables, and produces an output of 28.2744 square inches. The correct type of variable must be used to produce accurate results.

```
130  PI% = 3.1416:R% = 3
140  A% = PI% * R% ∧ 2
150  PI = 3.1416:R = 3
160  A = PI * R ∧ 2: PRINT
170    PRINT "INTEGER AREA OF THE CIRCLE IS ";A%;" SQUARE INCHES":
       PRINT
180    PRINT "REAL AREA OF THE CIRCLE IS     ";A;" SQUARE INCHES":
       PRINT
190    END

INTEGER AREA OF THE CIRCLE IS 27 SQUARE INCHES

REAL AREA OF THE CIRCLE IS 28.2744 SQUARE INCHES
```

Fig. 5-5. Calculations with integers and reals.

Applesoft language outputs nine, or fewer, positive or negative digits as they are input (Fig. 5-6). When more than nine positive digits are input, the output is positive scientific notation. When more than nine negative digits are input, the output is negative scientific notation.

Fig. 5-7 shows a program written to demonstrate how the INT (integer) function is used to output real numbers with a specified number of deci-

```
130  A = 999999999
140  B = 9999999999
150  C = -999999999
160  D = -9999999999
170    PRINT
180    PRINT "APPLESOFT PRINTS = ";A;" FIGURES"
190    PRINT
200    PRINT "GREATER THAN NINE FIGURES = ";B
210    PRINT "APPLESOFT PRINTS IN SCIENTIFIC NOTATION"
220    PRINT
230    PRINT "NEGATIVE VALUE PRINTS = ";C;" IN"
240    PRINT "NEGATIVE NINE FIGURES"
250    PRINT
260    PRINT "NEGATIVE VALUE OUTPUT = ";D;" IN"
270    PRINT "NEGATIVE SCIENTIFIC NOTATION"
280    END

APPLESOFT PRINTS = 999999999 FIGURES

GREATER THAN NINE FIGURES = 1E+10
APPLESOFT PRINTS IN SCIENTIFIC NOTATION

NEGATIVE VALUE PRINTS = -999999999 IN
NEGATIVE NINE FIGURES

NEGATIVE VALUE OUTPUT = -1E+10 IN
NEGATIVE SCIENTIFIC NOTATION
```

Fig. 5-6. Positive and negative integers.

mals. For example, if the variable "A" whose value is 28.2743343 is to be rounded to two places, the rounding value is 100 (Q = 100). The formula INT(100*A + .5)/100 is used to round to two places. The computation follows the rules of precedence. Precedence of operations is discussed in Lesson 6.

```
100 × 28.2743343 = 2827.43343
2827.43343 + .5  = 2827.93343
INT(2827.93343)  = 2827
2827 / 100       = 28.27
```

Fig. 5-8 is a program written to demonstrate what happens when an expression or formula is divided by zero. In mathematics, dividing by zero gives an undefined result. The computer does not allow division by zero. When you attempt to divide by zero, accidentally or on purpose, the computer stops the program at the line number where the attempt to divide by zero was made and outputs an error message, ?DIVISION BY ZERO ERROR IN 200 (Fig. 5-8).

```
130  P = 1000
140  Q = 100
150  R = 10
160  S = 1
170  PI = 3.1415927:RA = 3
180  A = PI * RA ^ 2
190     PRINT
200     PRINT "AREA OUTPUT AS INPUT = ";A
210     PRINT "AREA OUTPUT TO 3 PLACES = "; INT (A * P + .5) / P
220     PRINT "AREA OUTPUT TO 2 PLACES = "; INT (A * Q + .5) / Q
230     PRINT "AREA OUTPUT TO 1 PLACE = "; INT (A * R + .5) / R
240     PRINT "AREA OUTPUT TO 0 PLACES = "; INT (A * S + .5) / S
250     PRINT "AREA OUTPUT TRUNCATED = "; INT (A)
260     END

AREA OUTPUT AS INPUT      = 28.2743343
AREA OUTPUT TO 3 PLACES = 28.274
AREA OUTPUT TO 2 PLACES = 28.27
AREA OUTPUT TO 1 PLACE   = 28.3
AREA OUTPUT TO 0 PLACES = 28
AREA OUTPUT TRUNCATED  = 28
```

Fig. 5-7. INT function used to output real numbers with a specified number of decimal places.

```
130  P = 0
140  PI = 3.1416:RA = 3
150  A = PI * RA ^ 2
160     PRINT
170     PRINT "WHEN THE AREA IS DIVIDED BY ZERO AN"
180     PRINT "ERROR MESSAGE IS PRINTED"
190     PRINT "P = 0 THEREFORE WHEN INT(A*P+.5)/P IS"
200     PRINT "CALCULATED "; INT (A * P + .5) / P;"THE VALUE IS ZERO":
        PRINT
210     END

WHEN THE AREA IS DIVIDED BY ZERO AN ERROR MESSAGE IS PRINTED
P = 0 THEREFORE WHEN INT(A*P+.5)/P IS CALCULATED
?DIVISION BY ZERO ERROR IN 200
```

Fig. 5-8. Divide-by-zero gives an error message.

Applesoft has a built in function that can be used to round to a specific number of decimal places. The results are the same as when using the INT function, but the DEF FN is more convenient when typing the output variable. The DEF FN, Fig. 5-9, line 130, is used in the following format to round to three decimal places.

```
130 DEF FNA(W) = INT(W*1000 + .5)/1000
```

The "A" variable is attached to the define function to identify it for later use. The variable "W" is placed in parentheses, DEF FNA(W), and the same

```
130    DEF FN A(W) = INT (W * 1000 + .5) / 1000
140    DEF FN B(X) = INT (X * 100 + .5) / 100
150    DEF FN C(Y) = INT (Y * 10 + .5) / 10
160    DEF FN D(Z) = INT (Z * 1 + .5) / 1
170  A = INT (A)
180  PI = 3.1415927:RA = 3
190  A = PI * RA ∧ 2
200    PRINT
210    PRINT "AREA OUTPUT AS INPUT = ";A
220    PRINT "AREA OUTPUT TO 3 PLACES = "; FN A(A)
230    PRINT "AREA OUTPUT TO 2 PLACES = "; FN B(A)
240    PRINT "AREA OUTPUT TO 1 PLACE = "; FN C(A)
250    PRINT "AREA OUTPUT TO 0 PLACES = "; FN D(A)
260    PRINT "AREA OUTPUT TRUNCATED = "; INT (A)
270    END

AREA OUTPUT AS INPUT     = 28.2743343
AREA OUTPUT TO 3 PLACES  = 28.274
AREA OUTPUT TO 2 PLACES  = 28.27
AREA OUTPUT TO 1 PLACE   = 28.3
AREA OUTPUT TO 0 PLACES  = 28
AREA OUTPUT TRUNCATED    = 28
```

Fig. 5-9. Decimal calculation using the DEF function.

```
130  PI = 3.1416
140    DEF FN C(X) = INT (X * 100 + .5) / 100
150    DEF FN B(A) = PI * R ∧ 2
160    FOR R = 1 TO 5
170    PRINT "AREA OF A CIRCLE = "; FN B(A)
180    NEXT R: PRINT
190    FOR R = 1 TO 5
200    PRINT "AREA OF A CIRCLE = "; FN C( FN B(A) )
210    NEXT R
220    END

AREA OF A CIRCLE = 3.1416
AREA OF A CIRCLE = 12.5664
AREA OF A CIRCLE = 28.2744
AREA OF A CIRCLE = 50.2656
AREA OF A CIRCLE = 78.54

AREA OF A CIRCLE = 3.14
AREA OF A CIRCLE = 12.57
AREA OF A CIRCLE = 28.27
AREA OF A CIRCLE = 50.27
AREA OF A CIRCLE = 78.54
```

Fig. 5-10. Using the DEF function to store formulas.

variable "W" is used in parentheses in the integer function, INT(W*1000 + .5)/1000.

When the computed area is to be output to three places, the output function is written, FNA(A). The "A" outside the parenthesis refers to the define

function, and the "(A)" inside the parenthesis refers to the area of the circle.

The define function can be used to store formulas. In Fig. 5-10, line 150, the formula for the area of a circle is stored in the form, DEF FNB(A) = PI * R ∧ 2. The value of PI was initialized in Fig. 5-10, line 130, as PI = 3.1416. The function FNB(A) was then used in Fig. 5-10, line 170, to print out the area of the circle each time the radius changed (FOR R = 1 TO 5). In Fig. 5-10, line 140, the define function was initialized to round a real number to two decimal places. This rounding function, FNC(X), was used to embrace the "AREA" function, FNB(A), to produce the area of a circle to two decimal places, FNC(FNB(A)). One function can be buried within another function to produce desired results.

A literal is a set of alphanumeric characters enclosed in quotation marks. The following are examples of literals.

"7-11 STORE"	"BILL"
"44-50"	"SUE"

String literals have been used in the PRINT statement. A string variable may consist of 256 characters (one row on the screen consists of 40 characters). The following are examples of string variables.

A$	Z1$	CC$	COB$
D2468$	H1$	MOLE$	HAIR$

A string variable must begin with an alphabetic character and may be followed by an alphabetic character or a numeric character, followed by a dollar sign ($). Only the first two characters of the string variables are recognized by Applesoft.

HA$ is equivalent to HAIR$
Z2$ is equivalent to Z2468$

```
130  A$ = "HI THERE SUE"
140    PRINT
150    PRINT "A$ PRINT = "A$
160    PRINT
170    PRINT "NUMBER OF CHARACTERS IN THE STRING = "; LEN (A$)
180    PRINT
190    PRINT "NUMBER OF CHARACTERS IN THE NAME SUE = "; LEN ("SUE")
200    END

A$ PRINT = HI THERE SUE

NUMBER OF CHARACTERS IN THE STRING = 12

NUMBER OF CHARACTERS IN THE NAME SUE = 3
```

Fig. 5-11. Alphanumeric strings.

A string is used in replacement statement form by placing the string variable on one side of an equals sign and the string literal on the other side of the equals sign, Fig. 5-11. The number of characters in the string can be determined by using the reserved word LEN. In Fig. 5-11, line 130, A$ = "HI THERE SUE." When LEN(A$) is used in line 170, the output shows the number of characters in the string is twelve (12). When LEN("SUE") is used in line 190, the output shows there are three (3) characters in the name "SUE."

The Apple has a LEN function that can be used in the immediate execution mode. After the program in Fig. 5-11 has been run, the immediate execution mode command can be used to determine the length of the string and the literal.

```
PRINT LEN (A$), LEN ("SUE") (press RETURN)
12              3
```

In this immediate execution mode, no line number is needed.

LESSON 6

HTAB, TAB, and VTAB Statements to Format Output

After completion of Lesson 6 you should be able to:

1. Use HTAB, TAB, and VTAB statements to format output on the CRT, similar to using tabulators and return on a typewriter.
2. Draw the location of rows and columns on the CRT.
3. Clear the CRT by the use of HOME and CALL statements.

VOCABULARY

CALL — A CALL causes the execution of a machine language subroutine at a memory location whose decimal address is specified in the call expression. CALL -936 clears the screen. CALL -936 causes the same result as HOME.

Colon — The colon separates multiple program statements that are on the same line. The colon is also called the program statement separator.

HOME — This command clears the screen of all data and moves the cursor to the left uppermost position of the screen. HOME produces the same results as CALL -936.

HTAB — This command moves the cursor from one to forty spaces over the current line and prints at the HTAB numeric expression. HTAB 20 prints data at column 20 on the current line.

Program Statement Separator — This is the colon in Applesoft. It allows multiple program statements at the same line number.

TAB — This command must be used in a PRINT statement, and prints data at the TAB numeric expression. PRINT TAB(20) causes a tab to column twenty, and prints the data following the tab statement in column twenty.

VTAB — This command moves the cursor to the line that is in the numeric expression. VTAB(20) moves the cursor to row twenty. The numeric expression of VTAB can range from 1 to 24.

DISCUSSION

HTAB is a function that allows the programmer to place information or data at a specific vertical column on the display screen. The screen has columns numbered from 1 to 40. (In the 80 column mode, the HTAB command will go to only 40 columns)

VTAB is a function that allows the programmer to place information or data on a specific horizontal row on the display screen. The screen has 24 rows, numbered from 1 to 24.

HTAB and VTAB are generally used together in the same program line and are separated by a colon. HTAB 5 : VTAB 10, causes the cursor to be placed at row 10, column 5 on the display screen. The colon is used as a separator between two or more program statements with the same line number.

TAB(26) is used only in a PRINT statement. (PRINT TAB(3);"JOHN" : HTAB 5 : VTAB 10) The same print rules apply as discussed in Lesson 4.

SPC(5) is a command that spaces from the last position printed on the screen. SPC is a relative command that moves the cursor a given number of positions away from a previously printed item. SPC must be used in a PRINT statement (PRINT SPC(5);"JOHN").

HOME is a command that moves the cursor and the prompt to the upper left corner of the screen, and clears the screen of all text. CALL -936 clears the screen in the same way.

The program and run in Fig. 6-1 demonstrates the use of HTAB, VTAB, and TAB functions.

```
10  HOME
20  VTAB 1: PRINT "A"
30  VTAB 1: HTAB 40: PRINT "B";
40  VTAB 24: HTAB 1: PRINT "C";
50  VTAB 24: HTAB 39: PRINT "D";: VTAB 10
60  VTAB 12: HTAB 13: PRINT "E"; TAB (26);"F"
999 END
```

(A) Program.

```
A                40                B


24          E           F


C                                  D
```

(B) Screen display.

Fig. 6-1. VTAB, HTAB, and TAB demonstration program.

```
10 HOME
20 VTAB 1 : HTAB 1 : PRINT "A"
```

Lines 10 and 20 clear the screen and print an "A" at VTAB 1 : HTAB 1. Since the cursor is placed at row 1, column 1 on the screen, the HTAB 1 statement is not necessary to be written in the program.

```
30 VTAB 1 : HTAB 40 : PRINT "B";
RUN
```

Line 30 prints the letter "B" at VTAB 1 : HTAB 40. The semicolon after the "B" is necessary to prevent a line feed.

```
40 VTAB 24 : HTAB 1 : PRINT "C"; : VTAB 10
```

Line 40 prints "C" at VTAB 24 : HTAB 1. The semicolon prevents line feed. VTAB 10 shifts the cursor to VTAB 10, because if we didn't shift the cursor, the computer would automatically shift the cursor to column #1 of the twenty-fourth (24th) line.

```
50 VTAB 24 : HTAB 39 : PRINT "D"; : VTAB 10
RUN
```

Line 50 prints "D" at VTAB 24 : HTAB 39. Even though the screen is 40 columns wide, it is not possible to print the "D" at VTAB 24 : HTAB 40 without shifting the "A" and the "B" characters off the screen. Immediately after print "D" at VTAB 24 : HTAB 40 the cursor jumps to the next line. This cursor jump causes the screen to scroll upward, and "A" and "B" would be shifted off the screen.

To clean up the program type LIST 40. Line 40 will be displayed on the screen. Now retype the line.

```
40 VTAB 24 : HTAB 1 : PRINT "C";
```

The new line 40 leaves out the last colon and the VTAB 10. The program still runs properly because the ":" and VTAB 10 are not necessary with the inclusion of line 50. (The EDIT function will be discussed in Lesson 17.)

```
60 VTAB 12 : HTAB 13 : PRINT "E"; TAB(26); "F"
```

Line 60 causes the letter "E" to be printed at VTAB 12 : HTAB 13. A TAB function is used in a PRINT statement and the numerical expression is contained in parentheses. The TAB(26) expression is separated from the PRINT by semicolons on each side. Notice that the "F" does not have the PRINT repeated but is enclosed in quotation marks.

The program and run in Fig. 6-2 further demonstrates the use of TAB and HTAB statements.

Line 70 clears the screen. Line 80 prints "H" in column 40 on the top line of the screen. After "H" is printed, the cursor moves to the second line first column to prepare for the next item. Since the print is complete in line 80,

```
70 HOME
80 PRINT "HERE WE GO"; TAB( 40); "H"
90 PRINT "A BLANK LINE??"
100 PRINT : PRINT
110 PRINT "ONE MORE TIME"; TAB( 40);"I";
120 PRINT "NO BLANK LINE HERE!!!"
130 PRINT "HERE WE GO!";:HTAB 40: PRINT "J"
140 PRINT "ONE MORE TIME!;: HTAB 40: PRINT "K"
150 PRINT "WHAT A DIFFERENCE!!!"
160 END
```

(A) Program.

(B) Screen display.

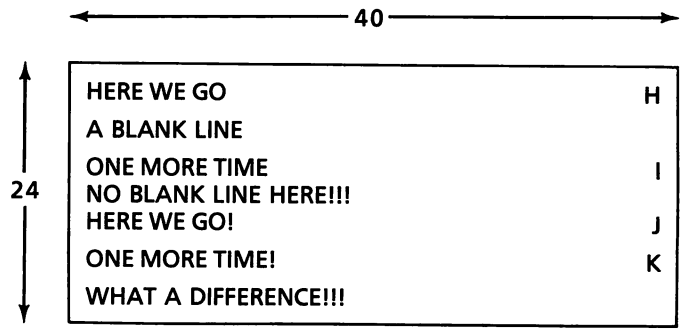


Fig. 6-2. Program to demonstrate PRINT results on the CRT.

the cursor moves to the second line, and closes out the second line. Line 90 prints on the third line, even though the second line is blank. Line 100 leaves two blank lines.

Line 110 is almost a duplicate of line 80, except for the semicolon after "I." The semicolon does not close out the line. Line 120 is printed immediately below "ONE MORE TIME" because the line was not closed out.

Lines 130 through 150 do essentially the same thing as lines 80 through 120 except the HTAB function is used instead of the TAB function. The HTAB prints in the place where the HTAB value is assigned. HTAB 40 prints at column 40.

You cannot print at VTAB 24 : HTAB 40 without pushing the top line of print off the screen.

LESSON 7

Precedence

After completion of Lesson 7 you should be able to:

1. Write the order of precedence of arithmetic operators and show how to modify precedence.
2. Demonstrate three methods to input data into a program.
3. Use constants to perform addition, subtraction, multiplication, division, and exponentiation.

VOCABULARY

Arithmetic Operators — Arithmetic operators are symbols that instruct the computer to do arithmetic operations, addition, subtraction, multiplication, division, and exponentiation.

ASC — This is the function that converts one string character to a numeric value. PRINT ASC("A") returns the ASCII (American Standard Code for Information Interchange) value of "A" which is 65.

*CHR$* — This is the function that converts a numeric value into a string character. PRINT CHR$(65) returns a character "A," which is the ASCII value of 65.

Constant — A constant is an item of data that remains unchanged after each program run.

Interactive Mode — The interactive mode is a method of operation in which the user is in direct communication with the computer and is able to obtain an immediate response to his input messages. A display where the user is allowed to input data in response to information displayed is said to be in the interactive mode. Conversational mode (display) is synonymous with interactive mode (display).

LET — LET is the replacement command that allows the value on the right side of the equals sign to be stored in the variable on the left side of the equals sign. LET may be a real, an integer, or a string. LET is an optional statement. LET A = 5 equates with A = 5.

Modem — Modem is a contraction of modulator and demodulator. A modem is a device that codes and decodes information to send and receive from a remote computer over telephone lines.

Operand — The operand is the item on which the operation is performed.

Operator — The operator is the action to be taken on the operand. In A = 5*2, times is the operand.

Precedence — This is the order in which things are done.

Replacement Statement — The replacement takes the value on the right side of the equals sign and stores it in the variable on the left side of the equals sign (i.e., A = 5 is a replacement).

Replacement Operator — In A = 5 the equals sign (=) is the replacement operator.

String — A string is a set of items which has been arranged in a sequence. The name "MARY" is a string.

Unary Operator — The unary operator is the sign preceding the first variable or constant in an expression. This is a processing operation performed on one operand. NOT, plus (+), and minus (−) are unary operators and apply to the sign of a number (−5, +3). If the unary is positive, the sign is implied. It is the same as a monadic operator.

DISCUSSION

The order of precedence is very important in mathematic calculations. Incorrect precedence produces incorrect answers. Correct precedence produces correct answers if all other procedures are correct. The order of precedence of arithmetic operators from highest to lowest is as follows:

1. ( ) items enclosed in parentheses are operated on first — highest priority.
2. NOT, +, −, NOT, POSITIVE, AND NEGATIVE are unary or monadic operators.
3. Exponentiation.
4. Multiplication and division from left to right.
5. Addition and subtraction from left to right — lowest priority.

Operators listed on the same line have equal priority and are executed starting from the left side of the expression and completed on the right side of the expression.

Integers and reals are classified as arithmetic variables. (Variables were discussed in Lesson 5.) Strings are classified as nonarithmetic variables. When integers and reals are used in an expression, the integers are converted to reals before the calculation takes place. The final result can be converted either to an integer or a real.

Examples of precedence follow.

1. Items enclosed in parentheses can either be variables or numeric values. If the variable is not given a value — a value of zero (0) is returned. The innermost set of parentheses are evaluated first.

```
15*(2+(3+2)*3) = 255
15*(2+ 3+2)*3  = 315
15* 2+ 3+2 *3  =  39
```

Precedence can be modified by using parentheses.

2. The monadic or unary operator is the sign (NOT, +, −) of the number.

```
+3+2 =  5 (number is positive when no sign is printed)
+3−2 =  1
−3+2 = −1
−3−2 = −5
```

3. Exponentiation.

```
3 ∧ 2 = 9
3 ∧ 80 = 1.47808831E+38
```

4. Multiplication and division.

```
(10 * 5)/(2 * 5) =   5
(10 * 5)/ 2 * 5  = 125
 10 * 5 / 2 * 5  = 125
```

5. Addition and subtraction.

```
(8 + 2) + (2 + 2) = 14
 8 + 2  +  2 + 2  = 14
(8 + 2) − (2 + 2) =  6
 8 + 2  −  2 + 2  = 10
(8 − 2) + (2 − 2) =  6
 8 − 2  +  2 − 2  =  6
(8 − 2) − (2 − 2) =  6
 8 − 2  −  2 − 2  =  2
```

There are several ways to get data or information into the computer. The replacement statement and the READ-DATA statement do not require any outside action or external peripherals. The INPUT statement is interactive between the user and the computer. Cassette tape, disks, and modem are external sources to place information or data into the computer.

```
10 LET A = 1 + 2 + 3
20 PRINT A
30 END
RUN
6
```

Line 10 is a replacement statement. The values on the right side of the equals sign are calculated and placed in a memory location that the com-

puter labels "A." The contents of memory location "A" are 1 + 2 + 3 or 6.

In this case, equals does not mean two equal values on opposite sides of the equals sign, but the value on the right side of the equals sign is transferred to the variable on the left side of the equals side. This is an operation (transfer) for the computer to perform and not an evaluation (decision). The equals is the replacement operator, and the LET is the replacement statement.

Line 20 PRINT A outputs 6, the value stored under the variable "A." The LET is optional. You get the same results with A = 5 as with LET A = 5. A = 5 saves memory and is easier to type.

The program in Fig. 7-1 was written to demonstrate the arithmetic operators, print rules, and replacement statements.

```
10  A = 5 : B = 10 : C = 20
20  D = C + B
30  E = C - B
40  F = A * B
50  G = C / A
60  H = A^2
70  PRINT D : PRINT E : PRINT F : PRINT
80  PRINT F, G, H : PRINT
90  PRINT D; E; F : PRINT
100 D = A : E = B : F = C
110 PRINT D, E, F
999 END
RUN
30          (D, no punctuation, line 70)
10          (E, no punctuation, line 70)
50          (F, no punctuation, line 70)
(LINE 70, PRINT SKIPS A LINE)
50          (F comma)          4          (G comma)          25          (H comma)
(line 80, PRINT skips a line)
301050      (D; E; F; Semicolons, line 90)
5           (D = A)            10         (E = B)            20          (F = C)
```

Fig. 7-1. Program to demonstrate arithmetic operators.

Line 100 D = A replaces the existing value of D (30) with the value of A (5). When "D" is printed, the replaced value of 5 is printed. E = B replaces the existing value of E (10) with the value of B. These values happened to be the same, so no difference is seen. F = C replaces the existing value of F (50) with the value of C (20).

INPUT is used to place values directly in the progam on an interactive basis. Type in the following lines, but leave the rest of the program as it is. The variables G and H were assigned values, but the values were not printed out.

```
6  INPUT "A = ";A
8  INPUT "B = ";B
10 INPUT "C = ";C
RUN
A = 5
B = 10
C = 20
```

The rest of the run is exactly the same as when A(5), B(10), and C(20) were used in replacement statements.

Now RUN the program using any values that you choose, but do not delete the program because the next step is to use the READ-DATA input method.

When you are through experimenting with different numbers using the INPUT statement, type in the following.

```
DEL 6,10 (PRESS RETURN)
```

This command deletes the INPUT statements at lines 6, and 8. Now type in the following statements.

```
10  READ A, B, C
120 DATA 5,10,20
```

The results of the run are the same whether a replacement statement, INPUT statement, or a READ-DATA statement combination was used.

A string is a set of items which has been arranged into a sequence. String variables (nonarithmetic) cannot be converted directly to integers or reals.

The program in Fig. 7-2 converts a string variable to a numeric variable and converts a numeric variable to a string variable.

Line 10 sets A$ = "A." The computer uses coded numbers to represent letters, as shown in Table 7-1. The letter "A" is converted to an ASCII number. Each letter, number, and symbol on the keyboard has an ASCII number.

Line 20 B = ASC(A$) places the ASCII number of "A" (65) into the variable "B." Line 30 PRINT B produces "B" = 65 which is the conversion of A$ (a string) into a real number.

```
10  A$ = "A"
20  B = ASC (A$)
30  PRINT B
40  D = 65
50  C$ = CHR$ (D)
60  PRINT C$
70  END
RUN
65
A
```

Fig. 7-2. Program to demonstrate ASC and CHR$.

Table 7-1. ASCII Character Codes

CODE		CHAR	CODE		CHAR	CODE		CHAR	CODE		CHAR
Dec	Hex		Dec	Hex		Dec	Hex		Dec	Hex	
0	00	NUL	32	20	SP	64	40	@	96	60	
1	01	SOH	33	21	!	65	41	A	97	61	a
2	02	STX	34	22	"	66	42	B	98	62	b
3	03	ETX	35	23	#	67	43	C	99	63	c
4	04	EOT	36	24	$	68	44	D	100	64	d
5	05	ENQ	37	25	%	69	45	E	101	65	e
6	06	ACK	38	26	&	70	46	F	102	66	f
7	07	BEL	39	27	'	71	47	G	103	67	g
8	08	BS	40	28	(	72	48	H	104	68	h
9	09	HT	41	29	)	73	49	I	105	69	i
10	0A	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	45	2D	-	77	4D	M	109	6D	m
14	0E	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	/
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	—	127	7F	DEL

ASC is the function that converts one string character into a number.

LINE 40 D = 65 places the ASCII value of the letter "A" into the variable "D." Line 50 C$ = CHR$(D) changes the value of the variable "D" into the string character C$. Line 60 PRINT C$ prints out the letter that was converted from the numeric equivalent (65) of the letter "A."

CHR$ is the function that converts a number into a string character.

In Applesoft the maximum length of a string is 255 characters.

LESSON 8

Loops

After completion of Lesson 8 you should be able to:

1. Write a program using a GOTO loop.
2. Write a program using a FOR–NEXT loop.
3. Write a program using nested loops.

VOCABULARY

Branch — A branch is a departure (or the act of departing) from a sequence of program steps to another part of the program. Branching is caused by a branch instruction that can be conditional (i.e., dependent on some previous state or condition in the program) or unconditional (i.e., always occurring). It is also known as a transfer or jump.

Conditional Transfer — See *Branch.*

FOR–NEXT — The FOR is the statement that is the beginning of a loop structure (FOR J = 1 TO 5). The NEXT statement is the foot of the loop structure (NEXT J).

GOTO — The GOTO statement is an unconditional branch (jump or transfer) to another part of the program. It may be executed in the immediate or deferred mode.

Increment — This is a fixed quantity that is added to another quantity.

Initialization — This is the process performed at the beginning of a program or program section or subroutine to ensure that all indicators and constants are set to prescribed conditions and value before the program or subroutine is run.

Loop — A loop is a set of instructions that is performed repeatedly until some specified condition is satisfied, whereupon a branch (jump or transfer) instruction is obeyed to exit from the loop. There are two types of loops, constructed (GOTO) and FOR NEXT.

Nested Loops — These are loops that exist within other loops.

STEP — FOR J = 1 TO 100 STEP 10. The STEP function causes the loop to increment by the value designated by the STEP. The STEP may be positive or negative. FOR J = 100 TO 1 STEP – 10.

Test — A test is a means to examine an element of data or an indicator to ascertain whether some predetermined condition is satisfied.
Unconditional Transfer — See *Branch.*

DISCUSSION

A loop is a series of instructions that are performed repeatedly until a specific condition is satisfied.

Suppose a program was written to count from one to five. One version of the program count could be as follows.

```
10 PRINT "1"
20 PRINT "2"
30 PRINT "3"
40 PRINT "4"
50 PRINT "5"
60 END
```

The program would not use the computer very efficiently. Writing a program to count to 1000 would take most of the day. A more efficient way to use the computer would be to write a program using a GOTO loop, Fig. 8-1.

```
10   X = 1
20   PRINT X
30   X = X + 1
40   IF X > 5 THEN 60
50   GOTO 20
60   PRINT : PRINT "I'M THRU COUNTING!"
999  END
RUN
1
2
3
4
5
I'M THRU COUNTING!
```

Fig. 8-1. A constructed GOTO loop.

Line 10 is the initializing statement, and is the top of the loop. The loop begins by initializing the variable "X" to the first value of the count, which is one (1). If a variable is not initialized before it is used, the computer may initialize the variable to zero (0). The variable "X" could be initialized to any number, such as 2, −40, or 308. The programmer must know the correct value to initialize the variable to produce the correct result.

Line 20 prints the value of "X" each time the loop is executed.

Line 30 is the incrementing statement that keeps track of the number of times the loop has executed. The loop variable "X" was initialized to one (1). Each time the loop is executed, the incrementing statement adds one to the

value stored in the variable "X". In this case, when the incrementing variable "X" is greater than five (5), the program (at line 40) jumps out of the loop and branches to line 60.

Line 40 is a testing statement. Each time the loop executes, line 40 tests to determine if "X" is greater than five (5). If "X" is less than five (5), then the program falls through to line 50, which is GOTO 20. Line 50 is an unconditional branch statement. When "X" is greater than five (5), the program jumps out of the loop, and branches to line 60, and outputs the statement, "I'M THRU COUNTING!"

The FOR–NEXT loop in Fig. 8-2 also counts from one to five.

```
10   FOR X = 1 TO 5          (replacement statement)
20   PRINT X
30   NEXT X
40   PRINT : PRINT "I'M THRU COUNTING!"
999  END
RUN
1
2
3
4
5

I'M THRU COUNTING!
```

Fig. 8-2. FOR–NEXT loop.

This is how the GOTO and the FOR–NEXT loops look when they are placed side by side.

GOTO Loop

```
10  X = 1
20  PRINT X
30  X = X + 1
40 IF X > 5 THEN 60

50  GOTO 20
60  PRINT : PRINT "I'M THRU
    COUNTING!
999 END
```

FOR–NEXT Loop

```
10  FOR X = 1 TO 5
20  PRINT X
30  NEXT X
40  PRINT : PRINT "I'M THRU
    COUNTING!"
999 END
```

The FOR–NEXT loop program is shorter and more efficient than the GOTO loop program. The GOTO loop is used in cases where the number of times of loop execution is not known beforehand. This will be explained more clearly when the GOTO loop is used with decision statements in Lesson 9.

The FOR–NEXT loop is used when the number of executions is known before the program begins. The FOR–NEXT loop can use loop variables to determine the number of times the loop is to be executed. In FOR X = 1 TO 5, the number of executions is going to be five (5). In FOR X = 1 TO N, the

variable "N" determines the number of times the loop is to be executed. The variable "N" can be entered as a replacement statement, an INPUT statement, or a READ DATA statement combination.

In the same FOR–NEXT loop program, type in these lines.

```
5  INPUT "COUNT TO #: ";N
10 FOR X = 1 TO N
RUN
COUNT TO #: 3
1
2
3
I'M THRU COUNTING!
```

Line 5 allows the user to input the highest number of the count. Line 10 causes X to start at the number "1," and go to "N," the highest number to be counted. In this case the loop is FOR X = 1 TO 3.

Now type these lines in the same program.

```
5  READ N
60 DATA 3
RUN
1
2
3
I'M THRU COUNTING!
```

Fig. 8-3 demonstrates how to use loops to print forward and backward by steps and how the HTAB function formats output from loops.

```
5    REM -HTAB IN LOOPS
10   FOR A = 1 TO 6
20   HTAB (A - 1) * 3 + 1: PRINT A;
30   NEXT A: PRINT
40   FOR B = 2 TO 6 STEP 2
50   HTAB (B - 1) * 3 + 1: PRINT B;
60   NEXT B: PRINT
70   FOR C = 6 TO 1 STEP - 1
80   HTAB (6 - C) * 3 + 1: PRINT C;
90    NEXT C
100  PRINT
110  FOR D = 6 TO 2 STEP - 2
120  HTAB (6 - D) * 3 + 1: PRINT D;
130  NEXT D: PRINT : PRINT
140  PRINT "A = ";A;" :B = ";B;" : C = ";C;" :D = ";D
RUN
1  2  3  4  5  6
   2     4     6
6  5  4  3  2  1
6     4     2
A = 7 :B = 8 :C = 0 :D = 0
```

Fig. 8-3. HTAB in loops.

```
10 FOR A = 1 TO 6
20 HTAB(A - 1)*3 + 1 : PRINT A;
30 NEXT A : PRINT
RUN
1  2  3  4  5  6
```

Line 10 designates that the loop executions will go from 1 to 6. In line 20, the HTAB function sets up the column in which the value of A is to be printed. The "*3" begins a field every three positions ("*4" would begin a field every four positions). The "+ 1" signifies column one of the screen. If the "+ 1" is not used, the HTAB tries to print in column zero. Since there is no column zero, the program does not run and prints out the error message, "ILLEGAL QUANTITY ERROR". A "+ 2" would signify column two on the screen. The "*3" controls the positions between numbers, while the "+ 1" signifies the number of columns from the left hand side of the screen. The value of A is printed horizontally because of the semicolon following the A (Table 8-1).

Line 30 completes the loop and the PRINT closes out the line of printed values of A.

Table 8-1. HTAB(A - 1)*3 + 1 : PRINT A;

LOOP EXECUTIONS	A	(A - 1)	(A - 1)*3	(A - 1)*3 + 1	PRINT A;
1	1	0	0	1	1
2	2	1	3	4	2
3	3	2	6	7	3
4	4	3	9	10	4
5	5	4	12	13	5
6	6	5	15	16	6

```
40 FOR B = 2 TO 6 STEP 2
50 HTAB(B - 1)*3 + 1 : PRINT B;
60 NEXT B : PRINT
RUN
   2  4  6
```

The loop in line 40 starts with a value of 2. STEP 2 causes the loop to be incremented by two (2) on each execution. The STEP can be any necessary value, positive or negative, to achieve the solution to the problem. Lines 50 and 60 are similar to lines 20 and 30.

The lines from 70 to 130 cause the loop to decrement and print the numbers backwards.

```
70  FOR C = 6 TO 1 STEP -1
80  HTAB(6 - D)*3 + 1 : PRINT C;
```

```
90  NEXT C
100 PRINT
110 FOR D = 6 TO 2 STEP -2
120 HTAB(6 - D)*3 + 1 :PRINT D;
130 NEXT D : PRINT : PRINT
140 PRINT "A = ";A;" :B = ";B;" :C = ";C" :D = ";D
RUN
6  5  4  3  2  1
6     4     2
A = 7    :B = 8    :C = 0    :D = 0
```

Line 70 sets loop C to go from 6 to 1 and is decremented in increments of −1.

In line 80, since the loop values are to be printed backwards in increments of −1, the value of 6 must be printed to the left side of the screen in column 1. To accomplish this, the value of C must be subtracted from the maximum value of the loop which is 6 (Table 8-2). Line 90 completes the "C" loop. Line 100 PRINT closes out the line.

Table 8-2. HTAB (6 − C)*3 + 1 : PRINT C;

LOOP EXECUTIONS	C	(6 − C)	(6 − C)*3	(6 − C)*3 + 1	PRINT C;
1	6	0	0	1	6
2	5	1	3	4	5
3	4	2	6	7	4
4	3	3	9	10	3
5	2	4	12	13	2
6	1	5	15	16	1

Lines 110 through 130 cause the values in the D loop to be printed out backwards in steps if −2.

The first PRINT in line 130 closes out the line of print of loop D, and the second PRINT skips a line before line 140 is printed.

The output from line 140 shows the next value of the variable after the loop has completed its executions. After a loop has completed its executions, the value of the loop variable is one unit more than the ending value of the loop (or one unit less, if the loop is STEPing backwards). In loop A the values go from 1 to 6, but the loop makes 7 the final value of A. In loop B the values go from 2 to 6, but the loop STEP makes the final value of B equal to eight (8). In loop C the values go from 6 to 1, but the loop makes the final value of C equal to zero (0). In loop D the values go from 6 to 2, but the loop makes the final value of D equal to zero (0). *These are very important facts to understand and remember.*

It is important to keep track of these final variable values because they can produce incorrect program results if the variables are used again and not correctly initialized. The program and RUN are shown in Fig. 8-3.

```
10  FOR S = 1 TO 3
20  FOR T = 1 TO 5
30  PRINT T; " ";                    INNER LOOP      OUTER LOOP
40  NEXT T : PRINT
50  NEXT S
60  END
RUN
1    2    3    4    5
1    2    3    4    5
1    2    3    4    5
```

Fig. 8-4. Double nested loops.

```
10  FOR S = 1 TO 10
20  HTAB (19 - S)
30  FOR T = 1 TO S
40  PRINT "*";
50  NEXT T : PRINT
60  NEXT S
70  END
RUN
                  *
                 **
                ***
               ****
              *****
             ******
            *******
           ********
          *********
         **********
```

Fig. 8-5. Graphics print using double nested loops.

Nested loops are a loop within a loop (Fig. 8-4). In Applesoft, loops can be nested 10 deep.

Line 20 sets the inner loop to execute 5 times. Line 30 causes the inner loop to print out the value of T on each execution of the loop, and the quotation marks cause one space to be placed between each of the values in the printout. If two spaces were left between the quotation marks, there would have been two spaces left between each value as it was printed. This is another method of spacing in a loop.

Line 10 and line 50 cause the outer loop to execute three times.

If loops are crossed, the program will not execute. Reverse lines 40 and 50 and observe that the NEXT variables are not in proper relation to the beginning of the loops.

```
10 FOR S = 1 TO 3
20 FOR T = 1 TO 5
30 PRINT T; " ";
40 NEXT S : PRINT
50 NEXT T
60 END
RUN
1 1 1
?NEXT WITHOUT FOR ERROR IN 50
```

Fig. 8-5 is a program using nested loops and the HTAB to print out half a triangle.

LESSON 9

Relational And Logical Operators

After Lesson 9 you should be able to:

1. Define and use relational and logical operators in writing a program.
2. Use decision statements in programming.

VOCABULARY

Bug — A bug is a mistake or malfunction in a computer program.

Debug — This means to remove a mistake, or mistakes, from a computer program or computer system.

Decision — This is an operation performed by a computer that enables it to choose between alternative courses of action. A decision is usually made by comparing the relative magnitude of two specified operands. A branch instruction is used to select the required path according to the results.

Default — The rule of default states that a computer program runs sequentially according to increasing line numbers unless a branch is executed.

Logical Operator — A logical operator is a word or symbol to be applied to two or more operands (NOT, AND, and OR are logical operators).

Relational Operator — A relational operator is a method of comparing quantities in order to make a decision.

DISCUSSION

Program statements have line numbers so the program can run sequentially from the lowest line number to the highest. The program runs sequentially until a program statement containing a relational or logical operator is reached. The program then must weigh the decision. If the decision is TRUE (1 or yes), then the program branches to a line number out of sequence. If the decision is FALSE (0 or no), then the program continues in its sequential run. That is, the program "falls through" or defaults to the next line number. The rule of default states that unless a branch is executed, the statement with the next highest line number is executed. With the

computer there are only two decision choices, true or false. There can be no other answer to the decision.

The following relational operators compare two quantities. Based on the result of the comparison, the computer can make a decision.

1. = Left expression "equals" the right expression (in this case, equals is not a replacement statement).
2. < > Left expression "does not equal" right expression.
3. > Left expression "is greater than" right expression.
4. < Left expression "is less than" right expression.
5. > = Left expression "is greater than or equal to" right expression.
6. < = Left expression "is less than or equal to" right expression.

Relational operators are related to logical operators.

1. NOT — NOT is the negation of an expression (IF NOT A GOTO 300)
2. AND — AND joins two OR MORE expressions together. For the statement to be true both expressions must be true. (IF A>B AND C>D GOTO 999.)
3. OR — OR joins two or more expressions. If either (or one) is true, the decision is true. (IF A>B OR C>D GOTO 999.)

The following program combines PRINT statements, loops, GOTO statements, decision statements, and program sections in one unit to further the learning experience. In Lesson 8 GOTO loops were discussed. This lesson details why a GOTO loop is used in cases where the total number of inputs is not known beforehand. The program deals with applicants who come into a drivers license bureau to apply for an operators permit. The office never knows how many applicants will present themselves on a given day. The GOTO loop accommodates the unknown number of applicants by using decision statements.

An applicant enters the drivers license office and the attendant asks the applicant's name and age. An operators license is issued or not issued on the basis of the age of the applicant. The total number of applicants by age groups and the total number of applicants for the day are printed out and the program terminates.

The program was intentionally written with REM statements in the program to demonstrate how programs can be written in sections to determine if each section runs properly. This is one method of debugging a program. These variables are used in the program.

AGE = age of the applicant
IS 18 = the applicant is 18
NA = number of applicants
UNDER 18 = the applicant is under 18
OVER 18 = the applicant is over 18

```
10 REM * PROGRAM TO DETERMINE
20 REM * LICENSE ELIGIBILITY AND
```

```
30  REM * COUNT THE NUMBER OF APPLICANTS
40  REM * INITIALIZE VARIABLES
50  INPUT "AGE = "; AGE
60  IF AGE < = 0 THEN 190
70  REM * COUNTING VARIABLE
80  IF AGE  18 THEN 130
90  IF AGE=18 THEN 160
100 REM * COUNTING VARIABLE
110 PRINT "OPERATORS LICENSE"
120 GOTO 50
130 REM * COUNTING VARIABLE
140 PRINT "NO OPERATORS LICENSE"
150 GOTO 50
160 REM * COUNTING VARIABLE
170 PRINT "JUNIOR OPERATORS LICENSE"
180 GOTO 50
190 REM * PRINT HEADINGS
200 PRINT
210 REM : PRINT TOTALS
220 END
RUN
AGE = 36
OPERATORS LICENSE
AGE = 15
NO OPERATORS LICENSE
AGE = 0
```

The program RUNs as planned. When age is input, the output shows the eligibility of the applicant. The first program revision counts the number of applicants. Change the program by typing the following line numbers and program statements.

```
40  NA = 0 (INITIALIZE SUMMING VARIABLE TO ZERO)
70  NA = NA + 1 (COUNTING STATEMENT)
190 PRINT "TOTAL APPLICANTS"
210 PRINT NA
RUN
AGE = 25
OPERATORS
AGE = 0
TOTAL APPLICANTS
1
```

It worked just as planned. Line 40 initializes the variable NA (number of applicants) to zero. A summing location must be initialized to the correct value before the variable is used. Many computers do not clear memory locations, and those locations could contain an undesired value.

Line 70 is a replacement statement that is also a counting statement which counts the number of applicants. The value of NA (originally zero) is incremented by the value of one (1) for each applicant. This incremented

value (NA + 1) is placed on the left side of the equals into NA. When processing the first applicant, the counter looks like this: NA = 0 + 1. With the second applicant the process is repeated, so the counter is NA = 1 + 1 and the results are placed on the left side of the equals sign into the variable NA. As each applicant's age is input, the counter is incremented by one (1). The incrementing continues until zero (0) is input, which causes the program to branch to line 190 to print out the totals (see line 60).

The second revision separates the applicants by age, counts and prints out the total number of applicants, and prints out the number of applicants that are UNDER 18, NOW 18, and OVER 18.

Type in the following line numbers and program statements.

```
40  NA = 0 : UNDER18 = 0 : NOW18 = 0 : OVER18 = 0
100 OVER18 = OVER18 + 1
130 UNDER18 = UNDER18 + 1
160 NOW18 = NOW18 + 1
190 PRINT "TOTAL NUMBER     UNDER 18     NOW 18     OVER 18"
210 HTAB 5 : PRINT NA; TAB(16); UNDER18; TAB(25); NOW18; TAB(33);
    OVER18
RUN
AGE = 15
NO OPERATORS LICENSE
AGE = 18
JUNIOR OPERATORS LICENSE
AGE = 25
OPERATORS LICENSE
AGE = 0
TOTAL NUMBER   UNDER 18   NOW 18   OVER 18
       3            1         1         1
```

The second revision initializes three more counting variables to zero.

```
40  NA = 0 : UNDER18 = 0 : NOW18 = 0 : OVER18 = 0
100 OVER18 = OVER18 + 1
130 UNDER18 = UNDER18 + 1
160 NOW18 = NOW18 + 1
```

Lines 100, 130, and 160 add counting statements to count the number of applicants in each age bracket. The counting statements are placed in the program sections that deal with the specific age of the applicant. The GOTO statements of lines 120, 150, and 180 are the ends of GOTO loops. The statements are unconditional jumps to line 50, the line that accepts the age of the next applicant.

The two program revisions complete the program and solve the problem of totaling the number of applicants and the total number of applicants by age.

The program section pertaining to applicants UNDER 18 is as follows:

```
80  IF AGE < 18 THEN 130
130 UNDER18 = UNDER18 + 1
140 PRINT "NO OPERATORS LICENSE"
180 GOTO 50
```

The program section pertaining to those applicants who are 18 years of age follows:

```
90  IF AGE=18 THEN 160
160 IS18 = IS18 + 1
170 PRINT "JUNIOR OPERATORS LICENSE"
180 GOTO 50
```

The program section dealing with those applicants OVER 18 is as follows:

```
90 IF AGE=18 THEN 160 (IF THE AGE IS OVER 18 THE STATEMENT IS FALSE AND THE
                       PROGRAM DEFAULTS TO LINE 100)

100 OVER18 = OVER18 + 1
110 PRINT "OPERATORS LICENSE"
```

The program section that deals with line 60. If the age is input as equal to zero in line 60, the program branches to line 160 to print out the results and end the program.

```
60  IF AGE < =0 THEN 190
190 PRINT "TOTAL NUMBER     UNDER 18     NOW 18     OVER 18"
200 PRINT
210 HTAB 5 : PRINT NA; TAB(16); UNDER18; TAB(25);
    NOW 18; TAB(33); OVER18
220 END
```

In the operators eligibility program, there are three age classifications, UNDER 18, NOW 18, and OVER 18. There are, however, only two decision statements to select the three age categories. Line 60 does not select an age category.

```
80 IF AGE < 18 THEN 130
90 IF AGE=18 THEN 160
```

The age groups start from the youngest group first. Line 80 selects off the youngest age group. Line 90 selects off the age group equal to 18. Thus, two of the three age groups are selected. This leaves the over 18 age group to follow the rule of default when that decision reaches line 90.

The complete operators license eligibility program is shown in Fig. 9-1.

In Applesoft, an IF–THEN statement that is TRUE executes all statements after the THEN. For example, all statements in line 20 are executed.

```
10 A = 5
20 IF A > 4 THEN A = 6 : B = A/12 : GOTO 90
```

In this case, since A>4 is TRUE, all statements are executed before the computer branches to line 90.

```
10  REM -PROGRAM TO DETERMINE
20  REM -LICENSE ELIGIBILITY AND
30  REM -COUNT THE NUMBER OF APPLICANTS
40  NA = 0:UNDER18 = 0:IS18 = 0:OVER18 = 0
50  INPUT "AGE = ";AGE
60  IF AGE < = 0 THEN 190
70  NA = NA + 1
80   IF AGE < 18 THEN 130
90   IF AGE = 18 THEN 160
100 OVER18 = OVER18 + 1
110 PRINT "OPERATORS LICENSE"
120 GOTO 50
130 UNDER18 = UNDER18 + 1
140 PRINT "NO OPERATORS LICENSE"
150 GOTO 50
160 IS18 = IS18 + 1
170 PRINT "JUNIOR OPERATORS LICENSE"
180 GOTO 50
190 PRINT "TOTAL NUMBER   UNDER 18   IS 18   OVER 18"
200 PRINT
210 HTAB 5: PRINT NA; TAB( 16);UNDER18; TAB( 25);IS18; TAB( 33);OVER18
220 END
RUN
AGE = 36
OPERATORS LICENSE
AGE = 21
OPERATORS LICENSE
AGE = 18
JUNIOR OPERATORS LICENSE
AGE = 15
NO OPERATORS LICENSE
AGE = -1
TOTAL NUMBER   UNDER 18   IS 18   OVER 18
      4              1         1         2
```

Fig. 9-1. Operators license eligibility program.

With the operators eligibility program completed, the following concepts have been reinforced.

1. PRINT statement rules.
2. HTAB and TAB rules.
3. GOTO loops
4. Operators and decision statements.
5. Initializing variables and counting statements have been introduced and will be discussed in greater detail in Lesson 12.

LESSON 10

Problem Solving and Flowcharting

After completing Lesson 10 you should be able to:

1. Begin using a logical method in problem solving.
2. Flowchart simple problems with flow chart symbols.

VOCABULARY

Code — Code is the representation of data or instructions in symbolic form; sometimes used as a synonym for instruction.

Hardware — This is the name for all physical units of a computer system. Hardware is made up of all the apparatus rather than the programs.

Logic — Logic is the science dealing with the formal principles of reasoning in electronic data processing. A program may run because there are no SYNTAX errors, but the results may be incorrect because the logic is incorrect.

Logic Flowchart — This is a chart representing a system of logical elements and their relationship within the overall design of the system or hardware unit. It is a representation of the various logical steps in any program or routine by means of a standard set of symbols. A flowchart is produced before detailed coding for the solution of a particular problem.

Software — In its most general form, software refers to all the programs that can be used on a particular computer system.

DISCUSSION

When you write a computer program, you solve a problem. The most basic approach to solving a problem is to first understand the problem. In the program to compute the area of a circle, the formula was discussed and thought out in high school math. The knowledge simplifies programming the output. A program to compute and compare the three types of depreciation somewhat changes the problem. The first approach to programming is to understand the problem and its ramifications.

Once the problem is understood, the solution must be placed in the

proper order. The exact output must be known. The precise formulas to output the correct answers must be used. The exact language that the computer understands must be programmed in the proper order, and the idiosyncrasies and normal operations of the computer must be understood. The computer can only output according to specific input. The excuse is often heard, "It's the computer's fault." Computers seldom (if ever) make errors; it's the human input that is in error. Computers are stupid, but exacting. Many programmers pray for a program statement DWIT (do what I think). The DWIT function is not yet available in Applesoft, so we'll do the best we can with what we have. Remember, the computer does exactly what you tell it to do, nothing more, nothing less.

Once the problem, the language, and the computer are understood, all other problems are relatively simple. The program can now be written to solve the problem.

The output must be tested for correct results with as many different inputs as possible. Simple inputs may produce correct results, but are there cases where the outputs are incorrect? The program should be tested and debugged to produce correct output under all circumstances. What if the program to put a man on the moon had a bug in it?

Has the program been documented with REM statements and all other written records been recorded so another person could RUN the program and understand the output? Have the variables been recorded so the computational formulas can be easily understood? Has the program been properly indexed so it can be easily located in the library? The answers to all these questions should be yes. It is easy to forget what problem the program solves, what the variables represent, and where the program is located.

Flowcharting, or logic flowcharting, is a technique representing a succession of events in symbolic form. Flowcharting is the first step in logical program development. It aids in thinking the program through from the problem stage to the computer stage.

In data processing, flowcharts may be divided into two types, system flowcharts, and program flowcharts.

System flowcharts, using symbols, show the logical relationship between successive events using hardware. Such symbols include data input (for example, magnetic tape, paper tape, disks, and punched cards), and data output (for example, magnetic tape, paper tape, disks, printers, and modem), Fig. 10-1.

Program flowcharts show diagrammatically the logical relationships between successive steps in a program. For most complicated programs, an outline flowchart precedes a detailed flowchart, before the program is written.

The purposes of outline or initial flowchart are to show:

1. All input and output functions.

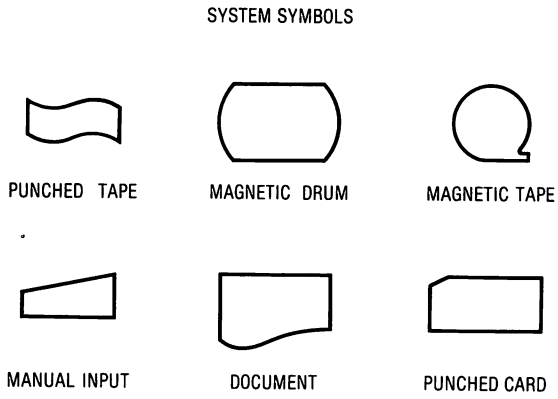


Fig. 10-1. Computer system flowchart symbols.

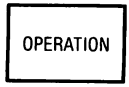
2. How input and output are to be processed.
3. How the program will be divided into routines and subroutines.

The purposes of a detailed or final flowchart are:

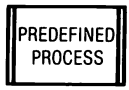
1. To interpret the detailed program specifications.
2. To determine the programming techniques to be used.
3. To provide direction for code and comment.
4. To fix the program style for ease of interpretation.

PROGRAM FLOWCHART SYMBOLS

TERMINAL	Used for the beginning and ending of a program. A symbol representing a terminal point in a flowchart.
GOTO	Flowlines show the transfer of control form one operation to another by default, conditional or unconditional branching.
INPUT OUTPUT	Input data entered into the computer and represents results returned from the computer. INPUT PRINT
DECISION	Indicates decision or switching type of operation that determines which of two alternate paths to follow. IF–THEN



Operation or process symbol that represents any kind of processing function such as initializing, summing, or computing.



Indicates a routine outside the main program, such as a subroutine. GOSUB

A symbol (pair) to represent the exit from or the entry to, another part of the flowchart. It is used to indicate transfer of control from one point to another point that cannot be conveniently shown on the flowchart because of the confusion of connector lines, or because the flowchart is continued on another page.

Figs. 10-2, 10-3, and 10-4 demonstrate flowcharts graphically and describe a program. Fig. 10-2 represents the Drivers License Program in Lesson 9. Figs. 10-3 and 10-4 represent the Sum of the Integers Program in Lesson 8. Fig. 10-4 shows how a FOR–NEXT loop is more efficient than a GOTO loop because it contains fewer statements.

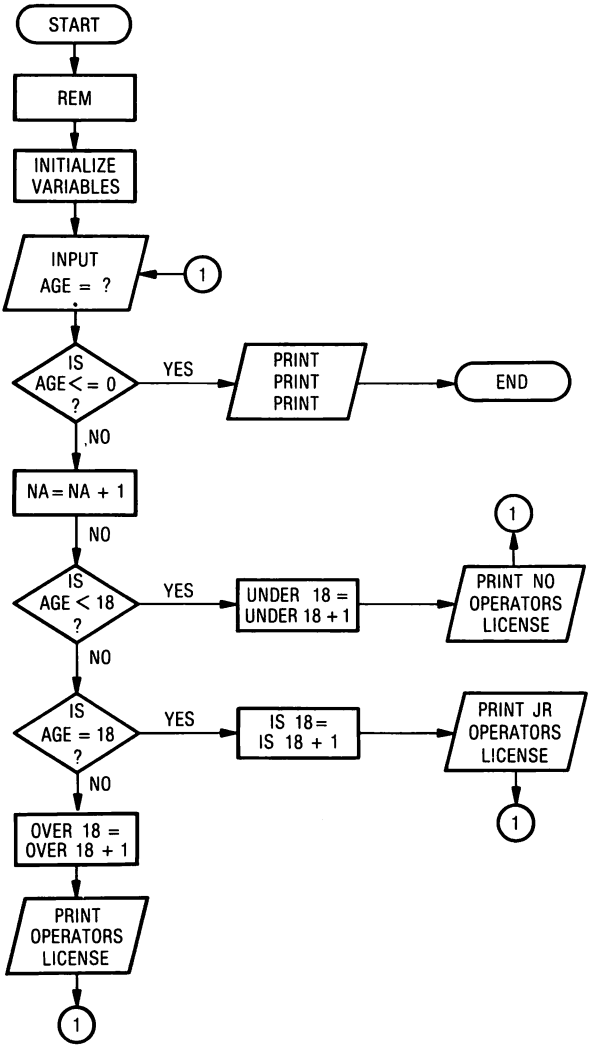
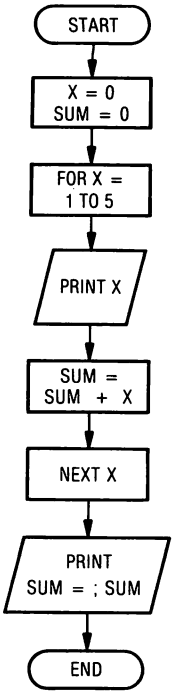
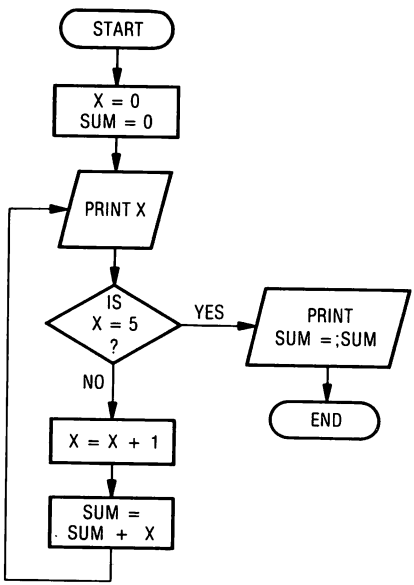


Fig. 10-2. Flowchart for license eligibility program.




```
10  X = 0
20  SUM = 0
30  FOR X = 0
40  PRINT X
50  SUM = SUM + X
60  NEXT X
70  PRINT "SUM = "; SUM
80  END
```

Fig. 10-3. Sum of the integers 1 through 5 using a FOR–NEXT loop.




```
10  X = 0
20  SUM = 0
30  PRINT X
40  IF X = 5
50  X = X + 1  THEN 80
60  SUM = SUM + X
70  GOTO 30
80  PRINT "SUM = "; SUM
99  END
```

Fig. 10-4. Sum of the integers 1 through 5 using a GOTO loop.

LESSON 11

Rules for Efficient Programming

After completion of Lesson 11 you should be able to:

1. Write three pairs of opposites to be used with decision statements.
2. Use three rules for efficient programming.
3. Understand how to save memory space and increase the speed at which a program runs.

DISCUSSION

This lesson deals with how to program more efficiently and how to make the program run faster. Efficiency and speed may be well and good, but to the average computer hobbyist speed is not that important. The important thing is to enjoy the hobby and write programs that are readable and can be deciphered six months from now. Place REM statements within the program that will help you understand and remember what the variable represented. Did that single "R" stand for RUN or RAIN? These points are very important if the program is to be reused at a later date. A couple of microseconds lost here and there isn't going to change the world. Write understandable programs. Write programs that jog your memory when you pick them out of your library four months from now. The variable SUM (even Applesoft recognizes only the first two letters) means something. The variable "S," now what did that stand for? Now back to speed and efficiency.

There are three pairs of opposites that are used to reverse the logic of the IF–THEN statement.

1. > is the opposite of < =
2. < is the opposite of > =
3. = is the opposite of < >

These pairs of opposites select a range. If the range is below age 18, the statement is AGE < 18. If the range includes age 18 and those ages below 18, the statement is AGE < = 18. If the over 18 group is to include the 18 year olds, the statement is written AGE > = 18.

Age less than 18	AGE < 18
Age less than 18 but includes 18	AGE < = 18
Age is equal to 18	AGE = 18
Age is not 18	AGE < > 18
Age greater than 18	AGE > 18
Age greater than 18 but includes 18	AGE > = 18

Decision statements (IF–THEN) operate on a TRUE (1 or YES), or FALSE (0 or NO) basis and are flowcharted as shown in Fig. 11-1.

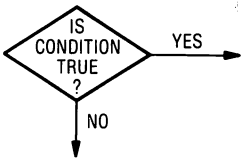


Fig. 11-1. Decision statement flowchart.

Decision statements should select the range from least to greatest by sequential line number for maximum programming efficiency. In other words, the ranges involving the smallest number should be selected first, and the increasing value of the range should be selected sequentially from the first value.

```
60 IF AGE < = 0 THEN 190
80 IF AGE < 18 THEN 130
90 IF AGE = 18 THEN 160 (Age < 18 defaults to line 100)

100 OVER18 = OVER18 + 1
```

Selecting the range from least (line 60) to greatest (line 90) makes programming an orderly endeavor, and thus, easier to perform and interpret. Fig. 11-2 demonstrates three rules for efficient programming.

Memory space can be saved and program speed increased by:

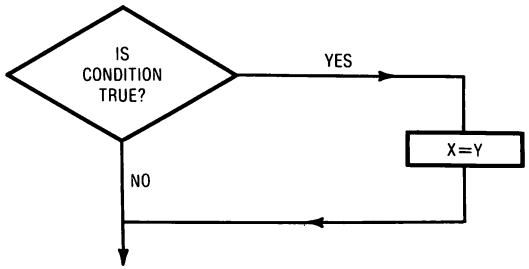
1. Using multiple statements for each line number.

```
100 UNDER18 = UNDER18 + 1 : PRINT "OPERATORS LICENSE" : GOTO 50
```

   This saves memory space, but also it sometimes helps keep track of program sections. In a long program, if short program sections are written with a single line number it eases readability.

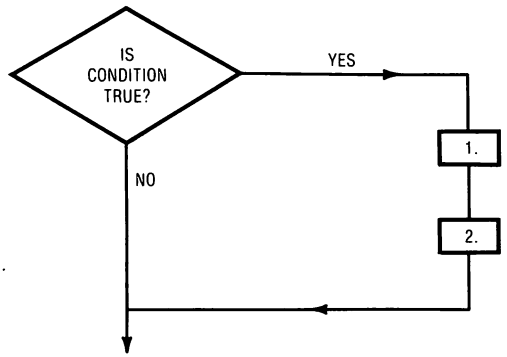
2. Using variables within the program instead of constants. PI = 3.1415926. If PI is used in the program (instead of 3.1415926), it runs faster. It takes more time to convert a constant to a real number than it does to fetch a variable. This is true in computations and in FOR–NEXT loops. Use FOR X = 1 TO PI, instead of FOR X = 1 TO 3.1415926.

1. IF CONDITION IS TRUE DO ONE THING.




```
10 X=6: Y=4
20 IF X > Y THEN X=Y
```

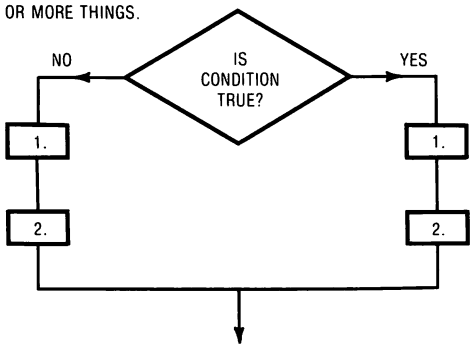
2. IF THE CONDITION IS TRUE DO TWO OR MORE THINGS.




```
30 IF X > Y THEN S=S+X+Y: C=C+5
```

WHEN X>Y THE STATEMENT IS TRUE AND BOTH S=S+X+Y AND C=C+5 WILL BE DONE.

3. IF THE CONDITION IS TRUE DO ONE OR MORE THINGS AND IF THE CONDITION IS FALSE DO ONE OR MORE THINGS.




```
40 IF X > Y THEN MAX = Y: GOTO 60
50 MAX = Y: T = T + 1
60 REM
```

Fig. 11-2. For efficient programming use these three rules for the decision statements.

3. Place items that are used frequently at the beginning of the program. When the program reaches an item that must be converted or fetched, the computer must search through the program sequentially until the item is found. If the item is on line 10, the search is much shorter than if the item is on line 10000.

There are other methods that save space and increase program speed but they do not increase efficiency greatly. Learn to program efficiently so the programs are usable, accurate, and understandable.

LESSON 12

Summing, Counting, and Flags

After completion of Lesson 12 you should be able to:

1. Write a program using summing and counting variables.
2. Initialize variables in the proper program location.
3. Use a flag to control all or part of a program.

VOCABULARY

Counting Variable — A counting variable is a variable used to count within a loop, e.g., C = C + 1. The variable is incremented by one (1) on each loop execution.

Flag — A flag is an additional piece of information added to a data item which gives information about the data. An error flag indicates that the data item has given rise to an error condition.

Illegal Value — In Applesoft, this means using a reserved word for a variable, i.e., using TO as a variable when it is a reserved word.

Legal Value — In Applesoft, this means using a variable that meets the requirements of the language, i.e., X = 5.

Summing Variable — A summing variable is used within a loop to sum the values of the loop variable. For example FOR X = 1 TO 5 : SUM = SUM + X. The summing statement SUM = SUM + X sums the values of X.

DISCUSSION

Counting variables are used to count some function within the program and are generally initialized to zero. The increment is one (1) if each execution of the loop is to be counted. C = C + 1 is the statement used to increment the count by one and store the count in the variable location "C."

Summing, also known as totaling, variables are used to sum or total within a loop. If "T" is the totaling variable, then "T" is usually initialized to zero. A program that totals daily and adds the daily total to the previous day's total would not be initialized to zero, but would be initialized to the previous day's total. If the variable is "X," then the totaling statement would

be T = T + X. The value of "X" is added to the total (T + X) and that value is placed in the variable "T" on the left side of the equals sign. The statement "T = T + X" is placed inside the loop and the total is output outside the loop, after the loop has made its final execution.

Variables are initialized at the beginning of the program. The statements that initialize the variables have no further function in the program. The GOTO statement should go to a line number below the line where the variables are initialized.

Fig. 12-1 is a program that demonstrates counting and totaling variables, and the location where these variables should be initialized.

```
10   C = 0 : T = 0 : REM * INITIALIZE VARIABLES
20   FOR X = 1 TO 5
30   PRINT X; " ";
40   C = C + 1 : REM * COUNTING STATEMENT
50   T = T + X : REM * TOTALING STATEMENT
60   NEXT X : PRINT : PRINT
70   PRINT "COUNT = "; C : PRINT
80   PRINT "TOTAL = "; T
999  END
RUN
1 2 3 4 5

COUNT = 5

TOTAL = 15
```

Fig. 12-1. Counting and totaling variables.

Note that the counting and totaling statements are within the body of the loop and the count and total change with every execution of the loop. When the loop has completed its last execution, the computer prints out the total count (line 70), prints out the total value of the variable "T," and the program ends.

A flag is a value stored in a variable. Flags are signals to the computer and are used to indicate the start of some programmed function. In the program in Fig. 12-2, the flag has a legal value of zero (0), one (1), and minus one (− 1). Flag = 0 causes the program to print out the number of additions, number of subtractions, final total, and the program ends. Flag = 1 causes the program to jump to the section of the program to input a number to add. Flag = − 1 causes the program to jump to a section of the program to input a number to be subtracted. After each addition and subtraction there is a GOTO 20 statement that is an unconditional jump to input another flag value.

If an illegal value (any value other than 0, 1, or − 1) is typed in the INPUT (20 for example) the program defaults to line 70. When Flag = 20, the decision at line 40 is FALSE, and the program defaults to line 50. In line 50, the decision is also FALSE and the program defaults to line 60. Line 60 is

FALSE and the program defaults to line 70. Line 70, GOTO 20, is an unconditional jump to line 20 to input another flag value.

The variables follow, and the program and RUN may be viewed in Fig. 12-2.

CA = count to be added
CS = count to be subtracted
N = number to be input
F = flag
T = total

Line 10 initializes the variables to zero. Line 20 prints out the flag values that control a specific part of the program. F = 0 outputs the results and causes the program to end. F = 1 adds a number that has been input and keeps a total. F = −1 subtracts a number that has been input and keeps a total. Line 30 allows the user to input the flag value. Fig. 12-3 is the flowchart of the FLAG VALUE program.

```
10   CA = 0 : CS = 0 : T = 0
20   PRINT "ENTER FLAG VALUE ( 0 TO QUIT : 1 TO ADD NUMBER :
     -1 TO SUBTRACT NUMBER)"
30   INPUT "?" ;F
40   IF F = 0 THEN 140
50   IF F = 1 THEN 80
60   IF F = -1 THEN 110
70   GOTO 20
80   INPUT "NUMBER TO BE ADDED "; N
90   CA = CA + 1 : T = T + N
100  GOTO 20
110  INPUT "NUMBER TO BE SUBTRACTED "; N
120  CS = CS + 1 : T = T - N
130  GOTO 20
140  PRINT : PRINT "# OF ADDS = "; CA
150  PRINT : PRINT "# OF SUBTRACTS = "; CS
160  PRINT : PRINT "FINAL TOTAL = "; T
170  END
RUN
ENTER FLAG VALUE ( 0 TO QUIT : 1 TO ADD NUMBER : -1 TO
SUBTRACT NUMBER)
? 1
NUMBER TO BE ADDED 34
ENTER FLAG VALUE (ALL OF LINE 20)
? -1
NUMBER TO BE SUBTRACTED 18
? 0
# OF ADDS = 1
# OF SUBTRACTS = 1
FINAL TOTAL = 16
```

Fig. 12-2. Program to demonstrate flag variables.

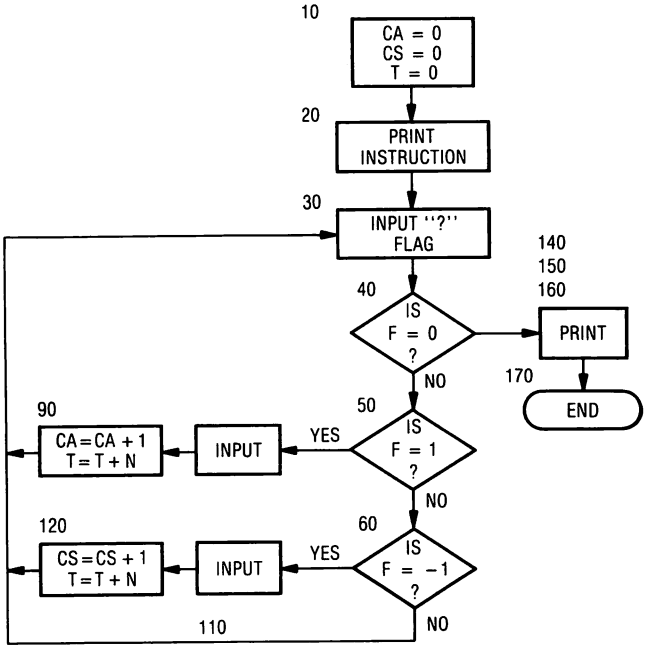


Fig. 12-3. Flowchart for flag program.

GOTO lines in Fig. 12-3 are represented by lines from one symbol to another but are not named in the flowchart. Unless F = 0, all GOTO lines return to input another flag value. Lines 140, 150, and 160 print out the results and the program ends at line 170.

LESSON 13

Single Subscripted Variables

After completion of Lesson 13 you should be able to:

1. Set DIMension limits using constants and variables.
2. Write programs using subscripted variables for numeric lists.
   a. To output the list as it is input.
   b. To output the list in reverse order.
   c. To operate on the numbers in the list.
   d. To total the numbers in the list.

VOCABULARY

Array (subscripted variable) — The array is an arrangement of items of data, each identified by a key or subscript. It is constructed in such a manner that a program can examine the array in order to extract data relevant to a particular key or subscript. The dimension of an array is the number of subscripts necessary to identify an item. Ticket reservations based on the day of the month would need a (DIM R(31)), and the array would be (R(DAY)). Ticket reservations based on the day and the month would need a double subscripted variable (DIM R(31,12)), and the array would be (R(DAY,MONTH)).

DIM — The dimension statement reserves memory locations for numeric or string arrays, such as A$(14) for string arrays, B(5) for real arrays, and C%(12) for integer arrays. DIM A(15) reserves 16 strings of 255 characters in length, starting from zero (0). DIM A$(N) must be placed in the program after the variable "N" has been input. The DIM A(N) must not be placed inside a loop.

List — A list is a series of data items.

Literal — A literal is a sequence of characters. (MARY is a literal---- "MARY" is a string)

Operate — This is a defined action by which a result is obtained from an operand.

String — A string is a sequence of characters. A string can be stored in a variable which is a letter followed by a dollar sign, for example A$, RE$, or H1$.

DISCUSSION

Lesson 13 introduces a new type of variable, the subscripted variable for numeric lists, or array.

		SUBSCRIPTED VARIABLE		
	C(A)		C(1)	
SIMPLE VARIABLES	CA	X9	CX4	G3H
SUBSCRIPTED VARIABLES	C(A)	X(9)	CX(4)	G3(H)

The variables A, A%, A(O), AB, and A(B) can all be used in the same program successfully. A subscripted variable, like a simple variable, reserves a memory location with a label and contents.

Table 13-1 shows five subscripted variables with the memory locations labeled C(A) with no values in the memory locations.

Table 13-1. Subscripted Variables

LABEL	CONTENTS
C(1)	
C(2)	
C(3)	
C(4)	
C(5)	

C(A) represents C(1), C(2), C(3), C(4), and C(5). Suppose the contents of the memory locations whose label is C(A) is filled with random numbers. LET C(1) = 8. LET C(2) = 5. LET C(3) = 19. LET C(4) = 1. LET C(5) = −8. Table 13-2 shows the five memory locations whose labels are C(A) and whose contents are shown above.

Table 13-2. Subscripted Variables

LABEL	CONTENTS
C(1)	8
C(2)	5
C(3)	19
C(4)	1
C(5)	−8

The subscripted variable, or array, is now filled with a list of numbers. When you remember that the subscript of a variable can be a variable (C(A)), or a constant (C(1)), you begin to realize what a powerful tool the single subscripted variable can be.

```
IF A = 1 THEN C(A) = C(1)
IF A = 2 THEN C(A) = C(2)
IF A = 3 THEN C(A) = C(3)
```

Subscripted variables give great processing power to the computer, and make the job of the programmer much easier.

To practice conversion of using constants and variables in the subscripted portion of the variable, complete the following exercise.

```
L(1) = 2          N(1) = 6          P = 1
L(2) = 5          N(2) = 3          Q = 2
M(3) = 7          O(3) = 1          R = 3
M(4) = -2         O(4) = 4          S = 4
L(1) =            P =               L(P) =
L(2) =            Q =               O(R) =
M(R) =            N(Q) =            N(Q) =
M(S) =            M(S) =            N(P) =
2                 1                 2
5                 2                 1
7                 3                 3
-2                -2                6
```

A DIM statement is necessary to reserve memory locations for a numeric array. The array range goes from zero (0) to 255 characters in length. An array can consist of up to 11 (0 to 10) elements before a DIM statement is required.

Integer arrays A%(N) will not be discussed separately because they are handled in a manner similar to real arrays A(N).

In the program in Fig. 13-1, three numbers are input into a list, so the DIM statement is not necessary. However, it is a good procedure to always include a DIM statement in the program that contains subscripted variables.

The program in Fig. 13-1 allows the user to input "N" numbers into a list. The list of numbers is printed out as input, printed backwards, operated on, and totaled.

In the program, the variable "L" automatically references the list. To reference items in the list, the variable "L" must be used and it must be followed by a subscript value enclosed in parentheses (L(A), L(2), or L(N*2 − 3)). The subscript can either be a variable, constant, or an expression. This *flexibility* is what gives arrays their real power. Anything can be placed in the subscript parentheses to reference a single part of the list, as long as it is > = 0 and < = the DIM statement value.

```
90   HOME : VTAB 3
100  REM : ARRAYS – SINGLE SUBSCRIPTED
110  REM : VARIABLES – USED FOR LISTS
120  REM : 4 WAYS TO USE THEM
130  REM : INPUT 4 NUMBERS
140  INPUT "HOW MANY NUMBERS = ";N
150  DIM L(N)
160  FOR X = 1 TO N : L(X) = 0 : NEXT X
170  FOR K = 1 TO N
180  PRINT "NUMBER";K; "=";
190  INPUT L(K)
200  NEXT K
210  PRINT "PRINT LIST AS INPUT" : PRINT
220  FOR A = 1 TO N
230  PRINT L(A); " ";
240  NEXT A : PRINT : PRINT
250  PRINT "PRINT LIST BACKWARD" : PRINT
260  FOR B = N TO 1 STEP – 1
270  PRINT L(B); " ";
280  NEXT B : PRINT : PRINT
290  PRINT "TO OPERATE ON THE LIST" : PRINT
300  PRINT "C"; TAB (7); "L(C)"; TAB (14); "L(C)+5"; TAB (23);
     "C*L(C)"; TAB (33); "L(C)^2"
310  FOR C = 1 TO N
320  PRINT C; TAB (9); L(C); TAB (16); L(C) + 5;
     TAB (24); C*L(C); TAB (34); L(C)^2
330  NEXT C : PRINT
340  PRINT "THE LIST IS TOTALED" : PRINT
350  T = 0
360  FOR D = 1 TO N
370  T = T + L(D)
380  NEXT D
390  PRINT "TOTAL OF THE LIST = "; T
400  END
RUN
HOW MANY NUMBERS 3
NUMBER 1 = ? 4
NUMBER 2 = ? –5
NUMBER 3 = ? 2.5
PRINT LIST AS INPUT

4 –5 2.5

PRINT LIST BACKWARD

2.5 –5 4

TO OPERATE ON THE LIST

C          L(C)       L(C)+5     C*L(C)     L(C)^2
1          4          9          4          16
```

Fig. 13-1. Operating on lists.

```
2          -5          0          -10          25
3          2.5         7.5        7.5          6.25
THE LIST IS TOTALED

TOTAL OF THE LIST = 1.5
```

Fig. 13-1–cont. Operating on lists.

Line 90 clears the screen and places HOW MANY NUMBERS on line 3 of the screen. The screen holds the complete input and output of the program (if no more than three numbers are entered in the list). When viewed together, the input and output help the user determine if the output is correct in relation to the input. Lines 100 through 130 are REM statements that partially document the intent of the program.

Line 140 sets up the user request and prepares for the input of the number of numbers to be entered into the list. The user then types in a number.

Line 150 DIM L(N) reserves memory locations N + 1 numbers in the list "L." Line 160 initializes the memory locations of "N" numbers in the list to zero (0). L(X) could have been initialized to zero (0) by using L(1) = 0, L(2) = 0, L(3) = 0. The loop is much more efficient, especially for an array with many values in the list.

Lines 170 through 200 set up the input format for the numbers to be entered into the list.

```
170 FOR K = 1 TO N
180 PRINT "NUMBER";K;"=";
190 INPUT L(K)
200 NEXT K
```

In line 180, the variable "K" is placed between the NUMBER and the " = " to inform the user which of the numbers in the list is to be entered. The "K" in the print statement begins as number one (1) and is incremented by one (1) with each execution of the loop.

Lines 210 through 240 set up the list to be printed out as it was input. The loop structure is used and the loop variable is "A," so the subscripted variable is L(A).

```
210 PRINT "PRINT THE LIST AS INPUT" : PRINT
220 FOR A = 1 TO N
230 PRINT L(A);"  ";
240 NEXT A : PRINT : PRINT
```

Line 230 prints the numbers in the list by placing them in the subscripted variable (L(1) = 4) and prints one number on each loop execution. The " " represents 2 spaces enclosed between the quotation marks. The " " causes two spaces to be left between each printed number each time the loop executes.

Line 240 closes out the loop, and the first PRINT closes out the line after all the numbers in the list have been printed. The second PRINT statement leaves a blank line between program sections in the printout.

Lines 250 through 280 print out the list backwards.

```
250 PRINT "PRINT LIST BACKWARD" : PRINT
260 FOR B = N TO 1 STEP -1
270 PRINT L(B);" ";
280 NEXT B : PRINT : PRINT
```

Line 250 is the print statement that labels the output. Line 260 sets up to print the list backwards by increments of one (1). In line 270, the list is referenced by the letter "L," so the subscripted variable is L(B). The value of the number contained in L(B) is printed out on each execution of the loop. The " " leaves two spaces between each number as they are printed. Line 280 completes the loop. The first PRINT statement closes out the line after all numbers in the list are printed. The second PRINT statement leaves a blank line between the output sections of the program.

Lines 290 through 330 operate on the numbers in the list.

```
290 PRINT "TO OPERATE ON THE LIST : PRINT
300 PRINT "C"; TAB(7);"L(C)"; TAB(14);"L(C)+5";
    TAB(23);"C*L(C)";
    TAB(33);"L(C)^2"
310 FOR C = 1 TO N
320 PRINT C; TAB(9);L(C); TAB(16);L(C) + 5; TAB(24);C * L(C);
    TAB(34);L(C)  2
330 NEXT C : PRINT
```

Line 290 prints the output section header. Line 300 prints out the headings over each column of the output. Line 310 is the beginning of the loop statement. Line 320 is the statement that outputs the results of the operations. These operations are output each time the loop increments. Line 330 is the foot of the loop, and the PRINT statement leaves a blank line between output sections. Each line of print was closed out because there was no semicolon at the end of line 320 to leave the line open.

Lines 340 through 390 total the list and print out the results.

```
340 PRINT "THE LIST IS TOTALED" : PRINT
350 T = 0
360 FOR D = 1 TO N
370 T = T + L(D)
380 NEXT D
390 PRINT "TOTAL OF THE LIST = ";T
400 END
```

Line 350 initializes the totaling variable to zero (0). Line 370 is the totaling statement that places the list values in the subscripted variable L(D) and adds one list number on each loop execution. Line 380 is the foot of the

loop statement. When loop "D" has completed its last execution, the program defaults to line 390 to print out the total of the numbers in the list, and line 400 ENDs the program.

The program in Fig. 13-2 is similar to the list of numbers program but it has the READ DATA statement to read the number of numbers in the list, and then reads the list.

```
160 RESTORE
170 READ N
180 PRINT "NUMBER OF NUMBERS =";N
190 FOR K = 1 TO N
200 READ L(K) : NEXT K
210 PRINT "PRINT LIST AS INPUT":PRINT
220 FOR A = 1 TO N
230 PRINT L(A);" ";
240 NEXT A: PRINT : PRINT
250 PRINT "PRINT LIST BACKWARD":PRINT
260 FOR B = N TO 1 STEP - 1
270 PRINT L(B);" ";
280 NEXT B: PRINT : PRINT
290 PRINT "TO OPERATE ON THE LIST": PRINT
300 PRINT "C"; TAB ( 7);"L(C)"; TAB( 14);"L(C)+5"; TAB( 23);"C*L( C)"; TAB(
    33);"L(C)^2"
310 FOR C = 1 TO N
320 PRINT C; TAB( 9);L(C); TAB( 16);L(C) + 5; TAB( 24);C * L(C); TAB( 34);L(C)^2
330 NEXT C: PRINT
340 PRINT "THE LIST IS TOTALED": PRINT
350 T = 0
360 FOR D = 1 TO N
370 T = T + L(D)
380 NEXT D
390 PRINT "TOTAL OF THE LIST =";T
400 DATA 3,4,-5,2.5
410 END
RUN
NUMBER OF NUMBERS = 3
PRINT LIST AS INPUT

4 -5 2.5

PRINT LIST BACKWARD

2.5 -5 4

TO OPERATE ON THE LIST

C             L(C)          L(C)+5        C*L(C)        L(C)^2
1             4             9             4             16
2             -5            0             -10           25
3             2.5           7.5           7.5           6.25

THE LIST IS TOTALED
TOTAL OF THE LIST = 1.5
```

Fig. 13-2. Using DATA statements for lists.

```
160 RESTORE
170 READ N (IN LINE 400--N = 3--THE FIRST DATA ITEM IN LINE 400)
180 PRINT "NUMBER OF NUMBERS = ";N : PRINT
190 FOR K = 1 TO N
200 READ L(K) : NEXT K
--------------------------------
400 DATA 3,4,-5,2.5
410 END
```

Line 160 RESTORE is not applicable in this program but it should be introduced at this point. If the data statement was to be read two or more times, the RESTORE statement would reset the data values and make it available to be reread for other program executions.

Line 180 prints out the number of numbers in the list after it has been READ from the DATA statement. Lines 190 and 200 include the loop and the READ statement to read the values in the list through the subscripted variable L(K).

When the last item in the list is read, the program defaults to line 210 to print out the section heading and continue with the program.

Line 400 DATA 3(N), 4(K), −5(K), 2.5(K) sets up the data items to be read in the proper order. READ N (line 170) reads the first item in the DATA statement, and the other three items in the data statement are read by the "K" loop and the READ L(K).

Fig. 13-3 is a program to demonstrate how integers, reals, and strings are processed in the READ DATA statements. To read the correct item, the variables in the READ statement must be aligned with the correct item in the DATA statement.

```
5  REM *DATA STATEMENTS
10 READ A%,A,A$,B$
20 PRINT A%,A,A$,B$
30 DATA 4.5,2.5,HELLO,"BYE"
40 END
RUN
4             2.5             HELLO
BYE
```

Fig. 13-3. DATA statements.

```
10 READ A%, A, A$
20 DATA 4, 2.5, HELLO (or "HELLO")
```

A$ will read either HELLO or "HELLO" in the DATA statement. A string variable in a READ statement will read a literal or a string, but outputs only in the string form.

LESSON 14

Double Subscripted Variables

After completion of Lesson 14 you should be able to:

1. Discuss double subscripted arrays.
2. Write programs using double subscripted arrays.

DISCUSSION

Double subscripted arrays are arrays that have two subscripts. Double subscripted arrays are used for outputting data or information in table form. The following variables are examples of double subscripted arrays.

CF(1,4) X(5,10) JANE(R,C) FOB(6,S)

Tables and arrays have rows and columns. Columns are positioned vertically on the screen. Rows are positioned horizontally on the screen. Table 14-1 shows how an array of three rows and three columns is arranged.

Table 14-1. CF(R,C) Array

CF(R,C)	COLUMN 0	COLUMN 1	COLUMN 2
ROW 0	CF(0,0)	CF(0,1)	CF(0,2)
ROW 1	CF(1,0)	CF(1,1)	CF(1,2)
ROW 2	CF(2,0)	CF(2,1)	CF(2,2)

CF = array name
R = row subscript
C = column subscript

The double subscripted array references a memory location that holds a value, the same as a simple variable. Values for CF(R,C) could be assigned as shown in Table 14-2.

An array can use variables or constants as subscripts. For example if R = 1 and C = 3 then CF(R,C) = CF(1,3). Subscripts can also combine with arithmetic operators and have an expression as a subscript (CF(R + 1,C – 1) or CF(R*2,C/3)).

Table 14-2. CF(R,C) Array With Values Entered

CF(R,C)	COLUMN 1	COLUMN 2	COLUMN 3
ROW 1	CF(1,1)=20	CF(1,2)=30	CF(1,3)=40
ROW 2	CF(2,1)=25	CF(2,2)=35	CF(2,3)=45
ROW 3	CF(3,1)=30	CF(3,2)=40	CF(3,3)=50

CF = array name
F = row subscript
C = column subscript

Values entered into a program as constants can be stored in subscripted arrays.

```
10 INPUT "GROSS INCOME = ";GI
20 INPUT "EXPENSES     = ";EX (EXP is a reserved word)
30 CF(C,1) = GI
40 CF(C,2) = EX
```

Whole columns or rows in an array can be added or subtracted the same as simple variables. As a matter of fact, any arithmetic operator that can be used on a simple variable can be used on an array. In the following example, column 2 is subtracted from column 1 to produce column 3 in the array CF.

```
CF(C,3) = (CF(C,1) - CF(C,2)
```

Variables, single subscripted arrays, and double subscripted arrays can be handled in a similar fashion.

The program written for Lesson 14 is a very elementary business program (Fig. 14-1). The user inputs the amount of his or her gross income, expenses, and years to operate. The expenses are subtracted from the gross income to produce the net income. The gross income, expenses, and the net income are the same for each year of the output. A line is drawn under the matrix, and the totals are output. The program is for teaching purposes. A commercial grade program would allow for the income and expenses to change each year.

A FOR–NEXT loop is used to compute and print out the information on a yearly basis. After the yearly figures are computed and printed, doubly nested loops are used to print out the totals of each column.

The variables used in Fig. 14-1 are as follows.

CF = cash flow	R = row
GI = gross income	C = column
EX = expenses	YRS = years to operate
H1$ = print heading	H2$ = print heading

The program was designed so the input and output would remain on the screen at the same time.

```
100  REM:PROGRAM TO DEMONSTRATE
110  REM:DOUBLE SUBSCRIPTED VARIABLES
120  HOME: VTAB 2:HTAB 6:PRINT "DOUBLE SUBSCRIPTED VARIABLES"
130  H1$ = "GROSS INCOME EXPENSES YEARS TO OPERATE"
140  VTAB 4 : PRINT H1$
150  VTAB 5 : HTAB 3 : INPUT " ";GI : VTAB 5 :
     HTAB 17 : INPUT " "; EX
160  VTAB 5 : HTAB 30 : INPUT " "; YRS : PRINT : PRINT
170  DIM CF(YRS,3)
180  FOR R = 1 TO YRS
190  FOR C = 1 TO 3
200  CF(R,C) = 0
210  NEXT C, R
220  H2$ = "YEAR GROSS INCOME EXPENSES NET INCOME"
230  PRINT H2$
240  FOR C = 1 TO YRS
250  CF(C,1) = GI : CF(C,2) = EX
260  CF(C,3) = CF(C,1) - CF(C,2)
270  HTAB 2 : PRINT C; TAB (8); CF(C,1); TAB (21);
     CF(C,2); TAB (30); CF(C,3)
280  TYRS = TYRS + C : NEXT C
290  FOR R = 1 TO YRS
300  FOR C = 1 TO 3
310  CF(0,C) = CF(0,C) + CF(R,C)
320  NEXT C,R
330  PRINT "____   _________   _________   _________"
340  HTAB 1 : PRINT TYRS; TAB (8); CF(0,1);
     TAB (21); CF(0,2) TAB (29) CF(0,3)   (There are no semicolons between the
                                           last three items)
350  END
RUN

          DOUBLE SUBSCRIPTED ARRAYS
GROSS INCOME   EXPENSES   YEARS TO OPERATE
   1500.21       875.35          4

YEAR  GROSS INC.  EXPENSES  NET INCOME
  1     1500.21     875.35     624.86
  2     1500.21     875.35     624.86
  3     1500.21     875.35     624.86
  4     1500.21     875.35     624.86
 10     6000.84     3501.4     2499.44
```

Fig. 14-1. Double subscripted arrays.

Lines 100 and 110 are REM statements used to partially document the program, which is a demonstration of doubly subscripted variables.

Line 120 HOME clears the screen and prints on line 2, DOUBLE SUBSCRIPTED VARIABLES.

Line 130 places the literal "GROSS INCOME EXPENSES YEARS TO OPERATE," into H1$ by using an assignment statement. Placing the literal into the string is useful if the same heading is printed out several times dur-

ing the program. Line 140 tabs to line 4 on the screen and prints out the heading.

Lines 150 and 160 ask for input below the subject heading, thereby letting the user know what input is requested. In Applesoft, the statement INPUT GI could be used, but it leaves a question mark on the screen in front of the data. INPUT " ";GI is used because it leaves no question mark on the screen in front of the data.

The DIMension statement is not necessary in the program if fewer than 11 array elements are used. Applesoft automatically sets up 10 array element memory locations. The DIM statement that allows a variable number of rows is CF(YRS,3). If we knew that the number of rows needed would never be more than 5, the DIM statement could be written DIM CF(4,3). This version allows 4 + 1 rows and 3 + 1 columns, or 20 real number elements in the array. Don't forget the computer uses zero as a counting number, so when you tell it four (from 1 to 4), the computer count reserves five locations.

Lines 170 through 200 initialize the locations in the table to zero.

```
170 FOR R = 1 TO YRS
180 FOR C = 1 TO 3
190 CH(R,C) = 0
200 NEXT C,R
```

The double nested loop is the most efficient method to initialize the locations in the table to zero. The table locations could have been initializing every element in the array and setting them to zero (0).

```
CF(1,1) = 0
CF(1,2) = 0
CF(1,3) = 0
CF(2,1) = 0
CF(2,2) = 0
CF(2,3) = 0
CF(3,1) = 0
CF(3,2) = 0
CF(3,3) = 0 etc.
```

A single FOR–NEXT loop could also be used.

```
FOR C = 1 TO 3
CF(1,C) = 0
CF(2,C) = 0
CF(3,C) = 0
CF(4,C) = 0 etc.
```

Line 210 is the header to be printed before the output of data. The header must be printed before the loop executes. If the header is within the loop, it will be printed each time the loop is executed. Line 220 prints the header. There is no VTAB statement because the table is printed below the input

information. Two blank lines separate the input data from the output information. The two PRINT statements in line 160 cause the two blank lines below the input.

In line 160, the number of years on which to compute the table was entered. In line 240, CF(C,1) = GI is a replacement statement that stores the input variable on the right side of the equals sign into the array element CF(C,1) on the left side of the equals sign. CF(C,2) = EX is a replacement statement to place the expense item into the cash flow array CF(C,2).

Line 250 is a replacement statement that sets up the third column of the table. The column (C,3) is to hold the net income, which is the gross income (C,1) less the expenses (C,2).

Line 260 places the results of each execution of the loop in the proper location under the header. The CF(C,1) is used directly in the PRINT statement to print out the results of that column.

Line 270 totals the number of years in the period and NEXT C is the ending statement in the loop.

Lines 280 through 310 compute the totals for each column and place them in row number zero (0).

```
280 FOR R = 1 TO YRS
290 FOR C = 1 TO 3
300 CF(0,C) = CF(0,C) + CF(R,C)
310 NEXT C, R
```

The elements CF(0,1), CF(0,2), and CF(0,3) have not been used in the table, but are available. Each of the loops started at 1, so the zero (0) was unused. The locations CF(0,1), CF(0,2), and CF(0,3) are used to place the totals of each column. Line 300 inside the doubly nested loops is the statement that totals each of the columns. Line 310 is the ending statement of the doubly nested loops.

Line 320 draws a line under the columns and the totals are placed below the lines. Line 340 prints out the totals in the proper positions below the lines. Notice that some of the semicolons between the TAB statements are missing. Applesoft is flexible enough to accept statements without semicolons and still operate properly. Line 340 ends the program.

LESSON 15

String Arrays

After completion of Lesson 15 you should be able to:

1. Use LEFT$, MID$, and RIGHT$.
2. Use string arrays (or string subscripted variables) in loops.
3. Output alphanumeric lists using string arrays.
4. Write programs using string arrays to print lists of names and addresses.

VOCABULARY

Concatenate — This means to link together in a set, series, or chain.

GOSUB — This is a statement that causes a branch to a subroutine at a specific line number (GOSUB 1000).

Null String — A null string is a string which contains no characters. Strings are initialized with zero characters. A$ = " " is a null string and PRINT A$ will not print any characters on the screen, nor will the cursor advance on the screen.

ON ERR GOTO 430, 440, etc. — This is a statement that causes an unconditional GOTO branch when a particular error is encountered. Error #1 jumps to line 430, error #2 jumps to line 440, etc. The equivalent statement for a GOSUB branch is ON ERR GOSUB 430, 440, etc.

String Array — This is a complex variable used to manipulate all or part of a string.

Subroutine — A subroutine is a discrete part of a program that performs a logical section of the overall function of the program. It is available whenever a particular set of instructions is required. The instructions forming a subroutine do not need to be repeated every time they are needed, but can be entered by means of a branch from the main program. Subroutines may be written in general form to perform operations common to several programs. In Applesoft, the statements GOSUB or ON (ERR) GOSUB direct the program to the subroutines. The last line of a subroutine is a RETURN statement, that jumps to the line in the main pro-

gram directly below the GOSUB or ON (ERR) GOSUB. Subroutines can be called from the main body of the program or from other subroutines. GOSUBs can be nested 25 deep in Applesoft.

DISCUSSION

String variables were introduced in Lesson 5 with simple variables, integers and reals. The complex variables, integer and real arrays were discussed in Lessons 13 and 14.

The complex variable (also known as the string array, or string subscripted variable) is a variable with a subscript. The string array is used to output alphanumeric information, such as lists of names and addresses.

A$ = a string variable.
HI SUE = a literal.
"HI SUE" = a literal enclosed in quotation marks (a string).
LEFT$(A$,J) = a string function having two arguments.
RIGHT$(A$,J) = a string function having two arguments.
MID$(A$,J,1) = a string function having three arguments.

In a string array, the dollar sign ($) follows the name of the array. The Applesoft language uses three functions to retrieve all or part of a string, or to print all or part of a string. A function is that part of a computer instruction that specifies the operation to be performed. An argument is a variable factor, the value of which determines the value of the function. The three functions used to manipulate strings are LEFT$, MID$, and RIGHT$. When string arrays are manipulated as single entities, they are handled as illustrated in Fig. 15-1.

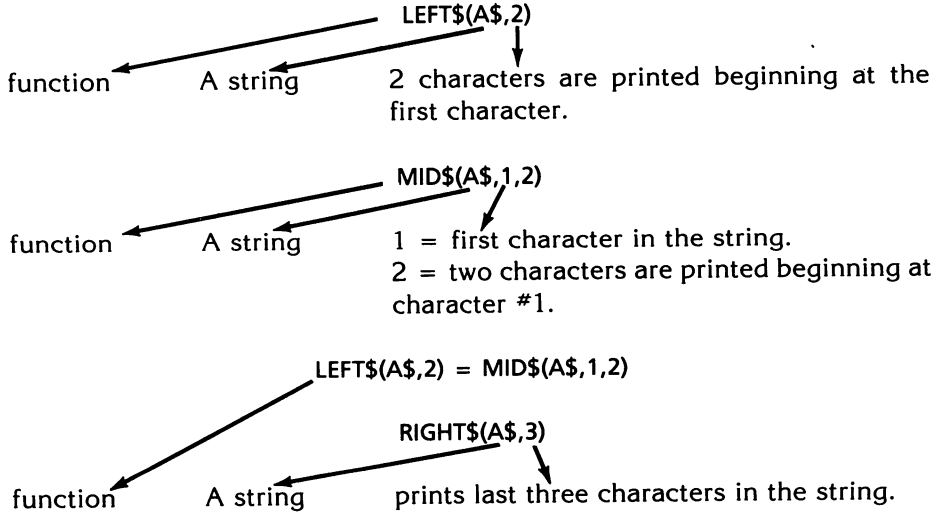


Fig. 15-1. String functions.

Fig. 15-2 is a program written using constants in the LEFT$, MID$, and RIGHT$ functions to demonstrate the output of A$ = "HI SUE."

```
100 A$ = "HI SUE"
110 PRINT LEFT$ (A$,2)
120 PRINT RIGHT$ (A$,3)
130 PRINT MID$ (A$,1)
140 PRINT MID$ (A$,2)
150 PRINT MID$ (A$,3)
160 PRINT MID$ (A$,4)
170 PRINT MID$ (A$,5)
180 PRINT MID$ (A$,6)
190 PRINT MID$ (A$,1,1)
200 PRINT MID$ (A$,2,1)
210 PRINT MID$ (A$,3,1)
220 PRINT MID$ (A$,4,1)
230 PRINT MID$ (A$,5,1)
240 PRINT MID$ (A$,6,1)
250 PRINT MID$ (A$,4,1)
260 PRINT MID$ (A$,4,3)
999 END
RUN
HI
SUE
HI SUE
I SUE
 SUE
SUE
UE
E
H
I

S
U
E
S
SUE
```

Fig. 15-2. Program of string functions.

The program also demonstrates that the function MID$ can output the same characters as LEFT$ and RIGHT$. With proper manipulation the programmer does not need LEFT$ or RIGHT$. MID$ alone will do the job.

LEFT$, MID$ and RIGHT$ functions can be used in loops to output all or parts of a string. A loop variable J is used in this example (Fig. 15-3), and the program is shown in Fig. 15-4. The variable L holds the value of the length of the string.

To concatenate is to link together in a series, set, or chain. Applesoft has the ability to concatenate. Strings can be altered to produce desired output (Fig. 15-5).

FOR J = 1 TO L (L = LEN(A$))

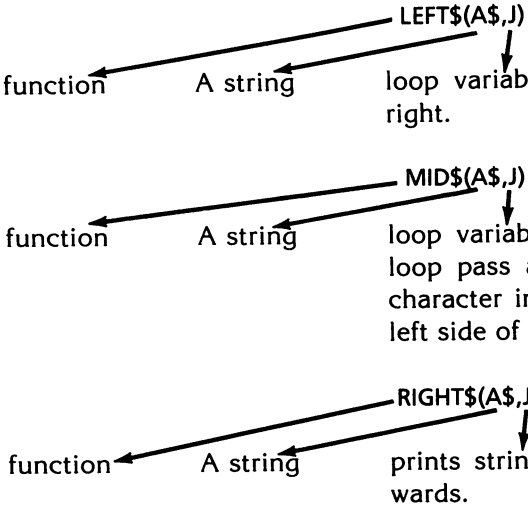


Fig. 15-3. String functions in a loop.

```
100 A$ = "HI SUE"
110 PRINT LEN (A$), LEN ("HI SUE"): PRINT
120 FOR J = 1 TO LEN (A$)
130 PRINT LEFT$ (A$,J)
140 NEXT J: PRINT :L = LEN (A$)
150 FOR J = 1 TO L
160 PRINT RIGHT$ (A$,J)
170 NEXT J: PRINT
180 FOR J = 1 TO L
190 PRINT MID$ (A$,J)
200 NEXT J: PRINT
210 FOR J = 1 TO L: PRINT MID$
    (A$,4): NEXT J: PRINT
220 FOR J = 1 TO L: PRINT MID$
    (A$,J,2): NEXT J: PRINT
230 FOR J = 1 TO L: PRINT MID$
    (A1,J,1);: NEXT J
999 END
RUN
6             6

H
HI
HI
HI S
HI SU
HI SUE
```

Fig. 15-4. Program of string functions in a loop.

```
E
UE
SUE
 SUE
I SUE
HI SUE

HI SUE
I SUE
 SUE
SUE
UE
E

SUE
SUE
SUE
SUE
SUE
SUE

HI
I

 S
SU
UE
E

HI SUE
```

Fig.15-4–cont. Program of string functions in a loop.

```
10   A$ = "HI SUE"
20   B$ = A$ + " " + "AND JIM"
30   PRINT A$
40   PRINT B$
50   C$ = LEFT$(A$,3) + RIGHT$(B$,3) + "!!!"
60   PRINT C$
999  END
RUN
HI SUE
HI SUE AND JIM
HI JIM!!!
```

(3 includes the space after HI — if 2 was used the output would be HIJIM!!!)

Fig. 15-5. Concatenation.

The rest of Lesson 15 is on program development. The objective is to produce a program that accepts a person's name and address for the purpose of compiling a mailing list. The program has error checking to notify the input operator if the input is incorrect. Once the input is correct, the name and address is output in the proper format.

A correct program is not usually written on the first attempt. The program is written and then revised. This lesson presents an outline for program development and shows some of the steps you should follow when writing usable programs.

The original program, Fig. 15-6, is inflexible. A$ holds the name and address of an individual. If an individual with another name and address is placed in A$, the output is not correctly formatted. As an extra learning experience, the program demonstrates the use of MID$ to replace LEFT$, and RIGHT$.

```
15   REM: NEXT PROGRAM CHANGE — INPUT A$ AT LINE 20
20   A$ = "JOHN DOE 2200 MAIN ST. ANYTOWN  USA  00000"
30   PRINT A$
40   PRINT LEFT$(A$,8)
50   PRINT MID$(A$,10,13)
60   PRINT RIGHT$(A$,18) : PRINT
70   PRINT MID$(A$,1,8)
80   PRINT MID$(A$,10,13)
90   PRINT MID$(A$,24,19)
999  END
RUN
JOHN DOE 2200 MAIN ST. ANYTOWN  USA  00000
JOHN DOE
2200 MAIN ST.
ANYTOWN  USA  00000

JOHN DOE
2200 MAIN ST.
ANYTOWN  USA  00000
```

Fig. 15-6. First version of name and address program.

When the program is typed and RUN, it can be readily understood that the program is useful only if the name and the address of the individual input is JOHN DOE 2200 MAIN ST. ANYTOWN USA 00000.

When line 20 is changed to "20 INPUT A$," the input must be the same number of letters and characters as in the original JOHN DOE name and address to be output in the correct format.

For a program to be valuable, it must be flexible. The name line of the program must be able to accept any reasonable name, no matter if it has three characters or 255 characters. The address field must be able to accept different numbers of characters for different street numbers and street names. The city, state, and zip code line must also be able to accept a different number of characters. To achieve flexibility, a delimiter (;) is used after each line of the name and address.

```
JOHN DOE;2200 MAIN ST.;ANYTOWN USA 00000
```

As a step toward developing flexibility, an inflexible formula is first demonstrated (Fig. 15-7). The semicolon (;) is used as a delimiter at the end of each field. No spaces are left between the contents of the field and the delimiter. Table 15-1 is an explanation of the line positions.

The following is an explanation of the expressions used in this program.

N = 9 N = variable of the delimiter at the end of the 1st field. Nine (9) is the column the delimiter occupies.

A1 = 23 A1 = variable of the delimiter at the end of the 2nd field. Twenty-three (23) is the column the delimiter occupies.

L = LEN(A$) L points to the end of the 3rd field. The column L occupies is determined by the length of the city, state, and zip code.

```
10 DIM A$(60)
20 INPUT A$
30 N = 9:A1 = 23:L = LEN (A$)
40 PRINT A$
50 PRINT LEFT$ (A$,N - 1)
60 PRINT MID$ (A$,N + 1,A1 - (N + 1))
70 PRINT RIGHT$ (A$,L -A1)
999 END
RUN
?JOHN DOE 2200 MAIN ST. ANYTOWN USA 00000
JOHN DOE 2200 MAIN ST. ANYTOWN USA 00000
JOHN DOE
2200 MAIN ST.
ANYTOWN USA 00000
```

Fig. 15-7. Second version of name and address program.

Table 15-1. Line positions

LINE	START	SYMBOL	END	SYMBOL
1	Position #1	LEFT$(A$,	Before 1st del	N-1
2	After 1st del	N+1	Before 2nd del	A1-1
3	After 2nd del	A1+1	Length of string	L

Line 1 LEFT$ (A$,N-1)

Line 2 MID$ (A$,N+1,A1-(N+1))

Line 3 RIGHT$ (A$,L-A1) the closing delimiter should be A1+1, but Applesoft picks up the delimiter position as A1.

In this sequence of learning events, the first string function used, LEFT$(A$,8), had constants within the parentheses that printed out the first through the eighth characters in the string. The second type of string function used, LEFT$(A$, N - 1), had a constant and a variable with a fixed value in relation to the delimiter and a fixed A$ input. The program to be studied next has a variable length input and a delimiter that is determined by the variable length input, LEFT$(A$, D1C - 1). The variable length input and input error checks produce a useful program.

Developing a complicated program is a detailed, exacting and thought provoking experience. Not all the programmer's thoughts can be written on

paper. The following program is presented in the detailed manner in which it was developed. The final program varies from the outline flowchart and this is a feature of progressive thought. For the benefit of the learning programmer, the initial flowcharts were not changed to conform to the finished program. The differences from one step to the next emphasize how development occurs.

GENERAL OUTLINE FOR PROGRAM DEVELOPMENT

A. What is the problem?
B. Detailed input format
C. Detailed output format
D. Outline flowchart
E. Assignment of variables
F. Start and end of lines
G. Basic flowchart
H. Error checking
    1. Number of delimiters
    2. Length of lines
        a. Length of line 1
        b. Length of line 2
        c. Length of line 3
I. Write error checking section of the outline flowchart
J. Write final flowchart
K. Write program
L. Debug and modify the program
M. Code the final program

The explanation and details of the logic use the same code and headings as the general outline for program development.

A. What is the problem? The problem is to input three lines of variable length, each separated by a delimiter (;) to allow any length of name, any length of street number and address, and any length of city, state, and zip code up to 255 characters each.

B. Detailed input format. Line = A$.

JOHN DOE;2200 MAIN ST.;ANYTOWN USA 00000

C. Detailed output format.

JOHN DOE
2200 MAIN ST.
ANYTOWN USA 00000

D. Outline flowchart (shown in Fig. 15-8).

E. Assignment of variables and expressions, Tables 15-1 and 15-2. The variables use the logic that a delimiter (D) is used after line 1 to close the line, hence D1C, means the delimiter that closes line one (1). A delimiter (D)

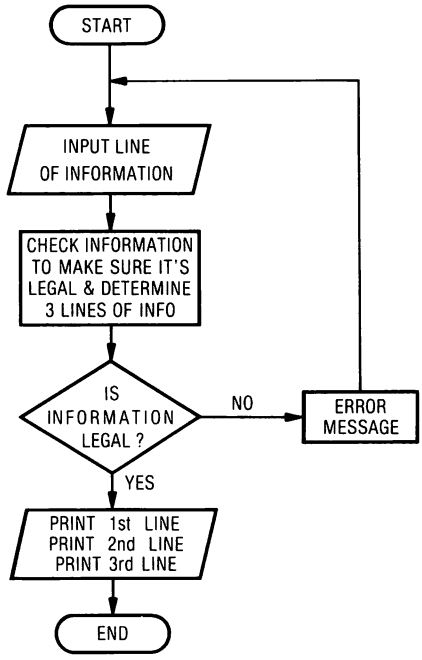


Fig. 15-8. Outline flowchart.

is used after line 2 to close out the line, hence D2C. L = LEN(A$) is the delimiter at the end of line three. The list of variables is shown in Table 15-2.

F. Start and end of lines. Once the delimiter variables are assigned, they are placed at the start and end of the lines (Table 15-3).

G. Basic flowchart. The basic flowchart is shown in Fig. 15-9. The basic flowchart shows the beginning pattern to develop the program. A$ is input by the user. The length of A$ is stored in the variable L. The first delimiter D1C is initialized to zero. A$ starts at the first character and ends at L (L = LEN(A$)). The statements D1C = D1C + 1 and IF MID$(A$,D1C,1) = ";" THEN — branch to D1C = D1C + 1, counts the number of characters in the first line of A$. This looping continues until the first delimiter (;) is found. The numeric value stored in D1C is then transferred to D2C. If after the end of line 1, D1C has a value of 10, then 10 is transferred to D2C. D2C = D1C. D2C is then initialized to a value of 10. The second line starts after the first delimiter (11). The same logic is applied to count the number of characters in line 2. D2C = D2C + 1 and IF MID$(A$,D2C,1) = ";" THEN — branch to D2C = D2C + 1 and count the number of characters in line 2 until the delimiter is reached. These two delimiters set the end of the first and second lines. The third line is composed of all the characters between D2C and L.

Table 15-2. Assignment of Expressions

1 (st column)	Beginning of line 1 — LEFT $(A$,
D1C	Points to the delimiter at the end of line 1.
D1C−1	End of line 1.
D1C+1	Beginning of line 2.
D2C	Points to the delimiter at the end of line 2.
D2C−1	End of line 2.
D2C+1	Beginning of line 3.
L	End of line 3.
ERR	Variable that is assigned a value when an input error has occurred, and the value is used to print the type of error.
J - (J,J)	Loop variable or subscript variable.

Table 15-3. Delimiters and Line Positions

(D1C−1) (D1C+1) D2C−1) (D2C+1)

INPUT JOHN DOE;2200 MAIN ST.;ANYTOWN USA 00000(L)

1st DEL (D1C) 2nd DEL (D2C) L = LEN (A$)

LINE	START	SYMBOL	END	SYMBOL
1	Beginning col. 1	LEFT$(A$,	Before 1st del.	D1C−1
2	After 1st del.	D1C+1	Before 2nd del.	D2C−1
3	After 2nd del.	D2C+1	Length of String	L

Line 1 LEFT$(A$,D1C−1)
Line 2 MID$(A$,D1C+1,D2C−(D1C+1))
Line 3 RIGHT$(A$,L−D2C)

H. Error checking (Table 15-4). For a program to be effective and efficient, those sections that interrupt the flow of the program must be eliminated. The algorithm from Section G. specifies that the input section (A$) must have two delimiters separating three lines. The first error check is to determine if D1C is greater than the length of the line. If D1C > L THEN. IF D1C > L, the loop has searched through A$ completely, and has not found a delimiter. Fig. 15-10 shows input with no error check. Compare Figs. 15-11 and 15-12 with Fig. 15-10 to see how this process looks logically. If the first loop finds a delimiter, the flowchart goes to the second loop to search for the second delimiter. If D2C > L, the second delimiter is not found. This is another error. If two delimiters are found, there is no error. Another error would occur if there are three semicolon delimiters. Another error would occur if the first delimiter was the first character in line 1. Line 1 would be zero length. If the second delimiter is the next character after the

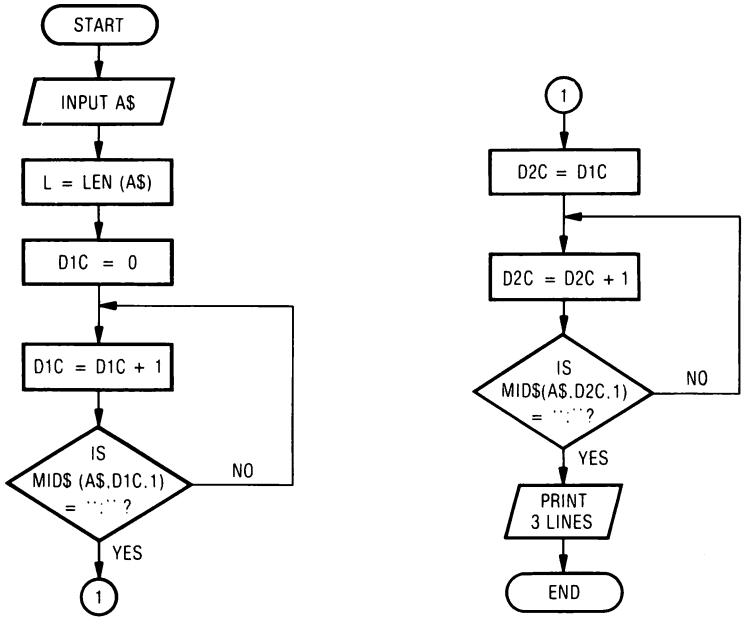


Fig. 15-9. Basic flowchart.

Table 15-4. Errors

ERROR #	CONDITION
1	IS D1C > L ?
2	IS D1C = 1 ?
3	IS D2C > L ?
4	IS D1C + 1 = D2C ?
5	IS D2C = L ?
6	IS A 3rd DELIMITER FOUND IN THE LAST PART OF THE INPUT ?

first delimiter, line 2 would be zero length. If line 3 had no characters between D2C and L, another error would occur. This gives a possibility of six errors, three delimiter errors, and three line length errors.

ILLEGAL CONDITIONS

1. Delimiters — not exactly 2
2. Length of lines
   a. Length of line 1 = 0 characters
   b. Length of line 2 = 0 characters
   c. Length of line 3 = 0 characters

DOE MAIN USA

D1C	MID$(A$,D1C,1)
1	D
2	O
3	E
4	
5	M
6	A
7	I
8	N
9	
10	U
11	S
12	A
13	ILLEGAL QUANTITY ERR (when D1C = 256)

Fig. 15-10. Input with no error check.

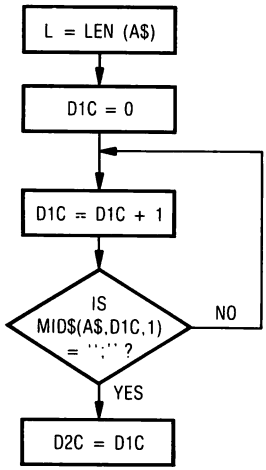


Fig. 15-11. Case No. 1 — flowchart with no error checking.

H.1 Number of delimiters. The input format has two, and only two delimiters (the L delimiter is not input). If there are any more or any less than two delimiters, the input is illegal, Fig. 15-13.

H.2 Length of lines. There are three lines of input separated by two delimiters. If any, or all, of these lines are zero length, the input is illegal (Fig. 15-14).

```
LINE 1 = 0 ;2200 MAIN ST.;ANYTOWN USA 00000(L)
LINE 2 = 0 JOHN DOE;;ANYTOWN USA 00000(L)
LINE 3 = 0 JOHN DOE;2200 MAIN ST.;(L)
```

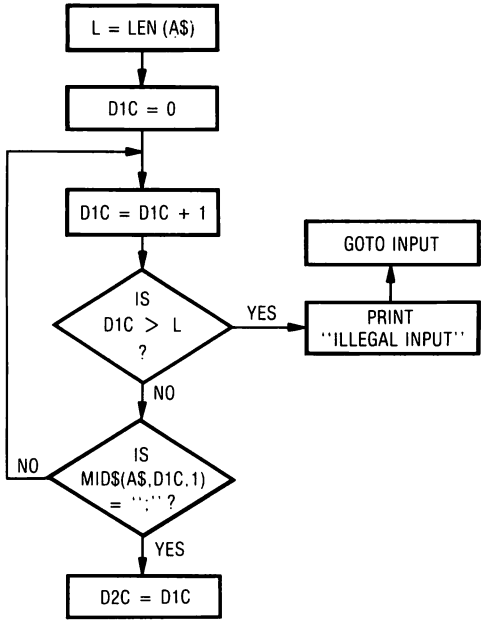


Fig. 15-12. Case No. 2 — flowchart with error checking statement.

# of Delimiters	Test	Decision Statement
0	ILLEGAL	D1C>L
1	ILLEGAL	D2C>L
2	LEGAL	
3	ILLEGAL	FOR J = D2C+1 TO L IF MID$(A$,J,1) = ";" NEXT J

Fig. 15-13. Check for the number of delimiters.

Line	Condition	Decision Statement	
1	;MAIN;USA	D1C = 1	ILLEGAL
2	DOE;;USA	D1C+1=D2C	ILLEGAL
3	DOE;MAIN;(L)	L = D2C	ILLEGAL

Fig. 15-14. Error check for line length.

I. Write error checking routine section of the flowchart. Fig. 15-15 shows the error checking aspects of the flowchart. Examine Fig. 15-15 carefully to determine where each error case is checked. Fig. 15-10 shows what happens with no error checking. If there are no delimiters separating the name and address fields in "DOE MAIN USA" (A$), the D1C = D1C + 1 loop executes until D1C is greater than L. When D1C > L, the computer prints

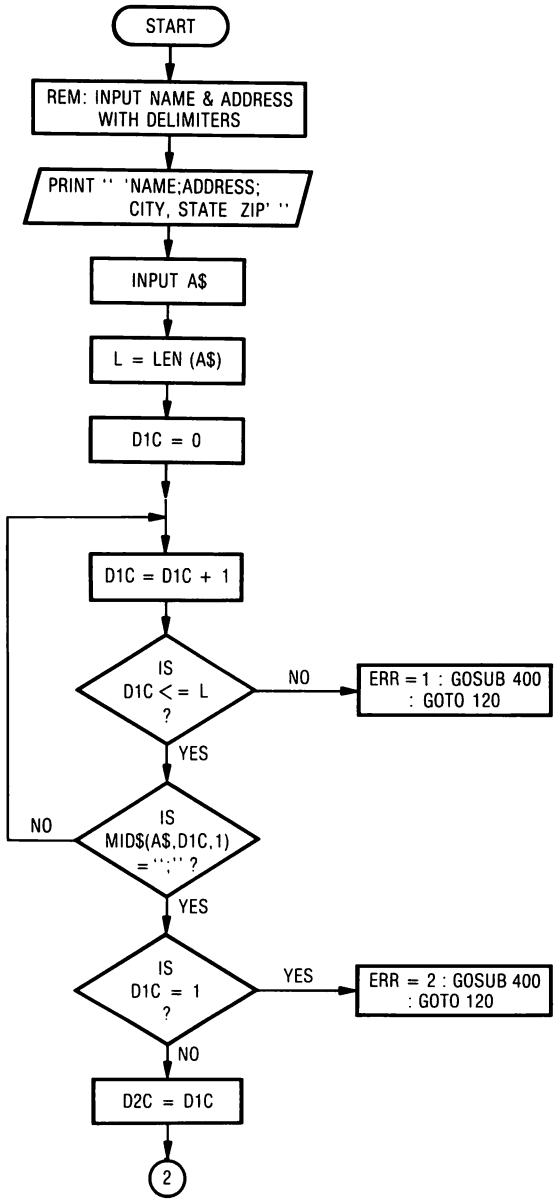


Fig. 15-15. Final flowchart.

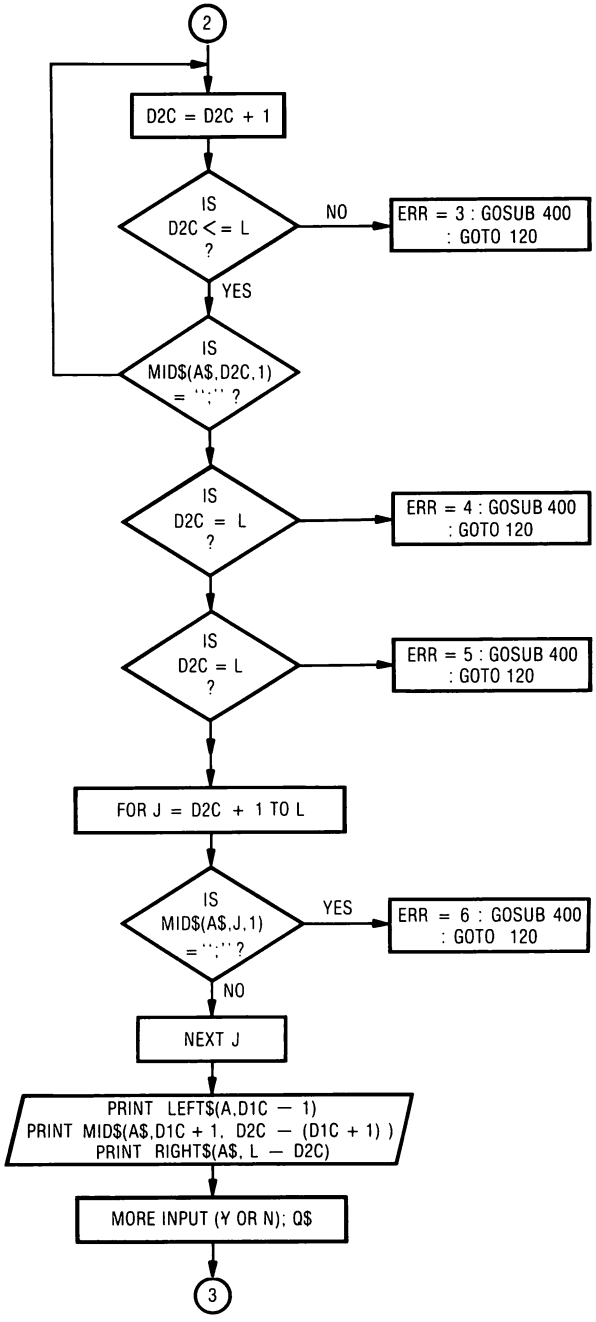


Fig. 15-15 — cont. Final flowchart.

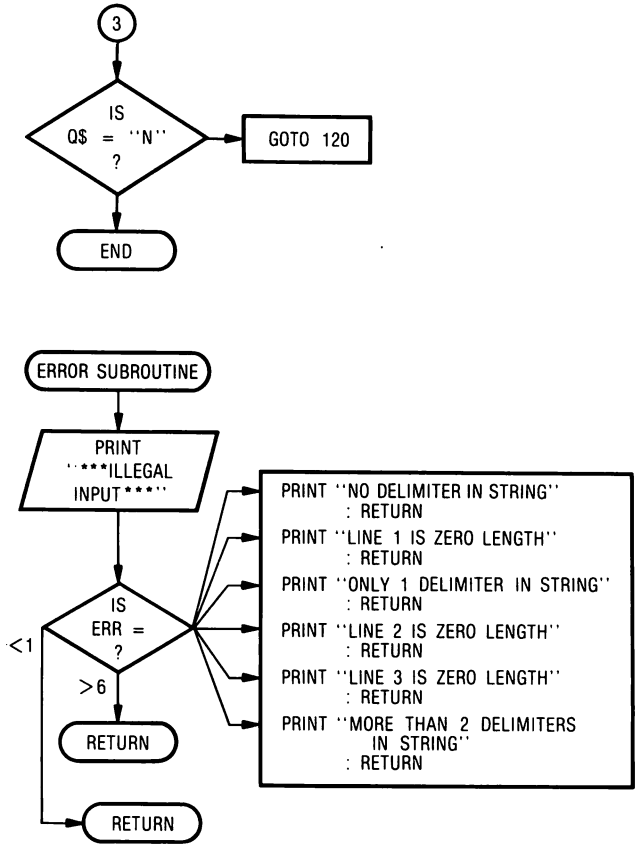


Fig. 15-15 — cont. Final flowchart.

ILLEGAL QUANTITY ERROR because it is telling the machine to compare a nonexistent character with ";".

J. Final flowchart. The final flowchart is an incorporation of all the details, charts, ideas, and logic to this point. The final flowchart should be written so very few changes are needed to code the program. The final flowchart is shown in Fig. 15-15.

K.L.M. Write, debug and modify the program (Fig. 15-16). Most programmers are perpetual students, tinkerers, and perfectionists. They will usually seek modifications to do the job better. This is the real idea of programming and life.

This is how the logic developed from conception to completion. The fol-

```
100  REM : PRINT NAME AND ADDRESS
110  REM : CHECK FOR INPUT ERRORS
120  PRINT : PRINT "INPUT 'NAME;ADDRESS;CITY STATE ZIP' "
130  INPUT "?";A$
140  L = LEN (A$) : D1C = 0
150  D1C = D1C + 1
160  IF D1C>L THEN ERR = 1 : GOSUB 400 : GOTO 120
170  IF MID$(A$,D1C,1) <> ";" THEN 150
180  IF D1C = 1 THEN ERR = 2 : GOSUB 400 : GOTO 120
190  D2C = D1C
200  D2C = D2C + 1
210  IF D2C>L THEN ERR = 3 : GOSUB 400 : GOTO 120
220  IF MID$(A$,D2C,1) <> ";" THEN 200
230  IF D1C + 1 = D2C THEN ERR = 4 : GOSUB 400 : GOTO 120
240  IF D2C = L THEN ERR = 5 : GOSUB 400 : GOTO 120
250  FOR J = D2C + 1 TO L
260  IF MID$(A$,J,1) = ";" THEN ERR = 6 : GOSUB 400 : GOTO 120
270  NEXT J
280  PRINT : PRINT LEFT$(A$,D1C - 1)
290  PRINT : PRINT MID$(A$, D1C + 1, D2C - (D1C + 1))
300  PRINT : PRINT RIGHT$(A$, L - D2C)
301  REM : LEFT$(A$,D1C - 1) = = MID$(A$,1,D1C - 1)
302  REM : RIGHT$(A$,L - D2C)= = MID$(A$,D2C + 1, L - D2C)
310  PRINT : INPUT "MORE INPUT (Y OR N)" ; Q$
320  IF Q$<>"N" THEN 120
330  END
400  PRINT "***ILLEGAL INPUT***"
410  ON (ERR) GOTO 430, 440, 450, 460, 470, 480
420  RETURN
430  PRINT "NO DELIMITER IN STRING" : RETURN
440  PRINT "LINE 1 IS ZERO LENGTH" : RETURN
450  PRINT "ONLY ONE DELIMITER IN STRING" : RETURN
460  PRINT "LINE 2 IS ZERO LENGTH" : RETURN
470  PRINT "LINE 3 IS ZERO LENGTH" : RETURN
480  PRINT "MORE THAN 2 DELIMITERS IN STRING" : RETURN
```

```
RUN

INPUT 'NAME;ADDRESS;CITY STATE  ZIP'
?JOHN DOE 2200 MAIN ST. ANYTOWN USA  00000
***ILLEGAL INPUT***
NO DELIMITER IN STRING

INPUT 'NAME;ADDRESS;CITY STATE  ZIP'
?;2200 MAIN ST.;ANYTOWN USA  00000
***ILLEGAL INPUT***
LINE 1 IS ZERO LENGTH
```

Fig. 15-16. Final flowchart for name and address program.

```
INPUT 'NAME;ADDRESS;CITY STATE  ZIP'
?JOHN DOE:2200 MAIN ST. ANYTOWN USA  00000
***ILLEGAL INPUT***
ONLY ONE DELIMITER IN STRING

INPUT 'NAME;ADDRESS;CITY STATE  ZIP'
?JOHN DOE;;ANYTOWN USA  00000
***ILLEGAL INPUT***
LINE 2 IS ZERO LENGTH

INPUT 'NAME;ADDRESS;CITY STATE  ZIP'
?JOHN DOE;2200 MAIN ST.;
***ILLEGAL INPUT***
LINE 3 IS ZERO LENGTH

INPUT 'NAME;ADDRESS;CITY STATE  ZIP'
?JOHN DOE;2200 MAIN ST.;ANYTOWN ;USA  00000
***ILLEGAL INPUT***
MORE THAN 2 DELIMITERS IN STRING

INPUT 'NAME;ADDRESS;CITY STATE  ZIP'
?JOHN DOE;2200 MAIN ST.;ANYTOWN USA  00000

JOHN DOE
2200 MAIN ST.
ANYTOWN USA 00000
MORE INPUT (Y OR N) N
```

Fig. 15-16 — cont. Final flowchart for name and address program.

lowing explanation of statements may be repetitious but it may also be helpful.

FLOWCHART NORMAL LOGIC	PROGRAM EXPEDIENT LOGIC
D1C < = L	D1C > L
MID$(A$,D1C,1) = ";"	MID$(A$,D1C,1) < > ";"
MID$(A$,D2C,1) = ";"	MID$(A$,D2C,1) < > ";"
D2C < = L	D2C > L
Q$ = "N"	Q$ < > "N"

The flowchart is written with normal logic. The program was coded with expedient logic. Once the flowchart is developed, the sections are broken down to determine the most efficient and fastest way for the program to run. A flowchart is simply a tool to help clarify the logic involved in solving a problem. When converting a flowchart to a program, sometimes it is useful to reverse the "IF" check to save memory. The common practice is to use MID$(A$,D1C,1) = ";". The program must search for ";" until it is found. When = ";" is not true, the program must unconditionally branch

(GOTO) backwards to increment D1C. Expedient logic MID$(A$,D1C,1) < > ";" causes the program to search for < > ";". If it is not found, the program conditionally branches to increment D1C, thus saving a GOTO statement with each decision. This not only saves program statements, but makes the program more efficient, run faster, and uses less memory. Expedient logic makes the program simpler and more efficient. It gets the job done better and faster.

Lines 100 and 110 are REM statements that document that the intent of the program is to print the name and address of an individual and to check for input errors.

Line 120 prints out the format to enter the name and the address of the individual. Line 130 is the statement that allows A$ to be input. The user is soon aware that the name and address must be entered exactly the same way as the prompt header. If the input is different from the prompt header, the program prints out an error message why the input is incorrect. The program will not accept the name and address unless it is entered correctly.

Line 140 sets the value of the end of the third line delimiter as L = LEN(A$). The D1C delimiter is initialized to zero. The D2C and L delimiters are not initialized to zero, because in line 190 D2C = D1C takes the value of D1C at the end of line 1 and stores it in D2C. This value assignment maintains continuity of the program in checking for the relationship with L = LEN(A$).

Line 150 is a counting statement that is incremented on each character of line 1 of A$, until it detects the delimiter (;).

In line 160, if the counting statement increments until the value of D1C is greater than L, the THEN is executed to send the program to ERR = 1. In Applesoft, when the statement (IF D1C > L THEN ERR = 1) is TRUE, all statements at the line number are executed. GOSUB 400 branches to the subroutine beginning at line 400. ***ILLEGAL INPUT*** is printed. The ERR = 1 sends the program to line 430. The error-line relationship is shown in Fig. 15-17.

ERROR #	ON (ERR) GOTO
1	430
2	440
3	450
4	460
5	470
6	480

Fig. 15-17. Error-line number relationship.

At the end of line 430 is a RETURN statement. A RETURN statement must be placed at the end of a subroutine. The RETURN causes the program to branch to the program statement immediately after the GOSUB.

The sequence of events that occur after ERR = 1 is as follows.

1. GOSUB 400 Branch to line 400.
2. ERR = 1 ON (1) GOTO 430 to print the input error. RETURN follows the error printout.
3. RETURN GOTO 120 — line immediately following GOSUB 400.

The ON-GOTO (ON-GOSUB) is a relationship programmed in Applesoft. A specific ERROR number relates to a specific error condition in the program.

In line 170, the decision statement checks to see if the delimiter (;) at the end of the first line has been reached. If the character is not the delimiter, the program branches to line 150 to increment the value stored in D1C.

In line 180, if there is only one delimiter in A$, then ERROR = 2. The statement GOSUB 400 sends the program to line 400 to print out ***ILLEGAL INPUT*** and then to line 440 to print out the input error, LINE 1 IS ZERO LENGTH.

In line 190, the value stored in D1C is assigned to D2C. This statement maintains the relationship from one delimiter to the next delimiter. The value relationship is continued as the program moves to L, the delimiter at the end of line 3.

In line 210, if the present value stored in D2C is greater than L, THEN input error #3 is printed on the screen. The program executes GOSUB 400, ON (ERR = 2) GOTO 450, to print ONLY ONE DELIMITER IN STRING. The RETURN statement branches to the line immediately after the GOSUB 400. The statement is GOTO 120, and immediately branches to line 120, for more input.

Line 220, if the decision statement does not find the D2C delimiter, it branches to line 200 to increment D2C.

In line 230, if the two semicolon delimiters are together (;;) with no characters between them, line #2 is missing. ERR = 4. GOSUB 400 executes to line 400 to print out ***ILLEGAL INPUT***, ON (ERR = 4) GOTO 460, prints out, LINE 2 IS ZERO LENGTH. RETURN branches to the third statement in line 230 (GOTO 120) to input the correct information.

In line 240, if D2C = L there are no characters in line #3, and ERR = 5, causes a branch to the subroutine to print ***ILLEGAL INPUT***, ON(ERR = 5) GOTO 470 prints, LINE 3 IS ZERO LENGTH.

Lines 250 through 270 is a loop to check the number of delimiters from D2C to L. The program has checked that there are 2 delimiters to this point. Line 170 checks the delimiter at the end of line #1. Line 220 checks for the delimiter at the end of line #2. If the loop FOR J = D2C + 1 TO L executes and finds another semicolon delimiter then ERR = 6. The program branches to line 400, prints out ***ILLEGAL INPUT***, IN(ERR = 6)

GOTO 480, MORE THAN 2 DELIMITERS IN STRING. The RETURN is executed to GOTO 120, and GOTO 120 branches to input information.

Lines 280, 290, and 300 print out the name and address of the individual in the correct format.

Lines 301 and 302 are inserted to inform the user of the correct MID$ functions to replace the LEFT$ and RIGHT$ to print out lines #1 and #3.

Line 310 PRINT : INPUT "MORE INPUT (Y OR N)";Q$ queries the user, is another name and address to be input, or does the user wish to end the program?

Line 320 IF Q$ < > "N" THEN 120 is the decision statement to branch to line 120 for more input, or to end the program. This line could be flowcharted in either of two ways for equal efficiency (Fig. 15-18).

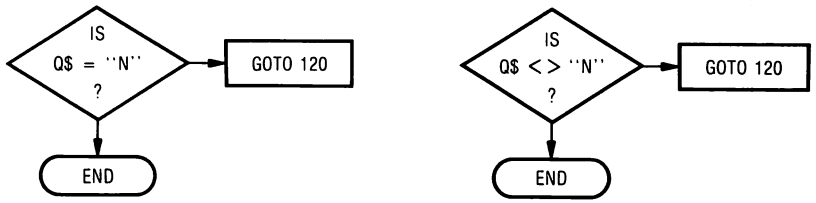


Fig. 15-18. Decision flowchart.

Lines 400 through 480 are the subroutines and the ON (ERR) GOTO statements. Subroutines ae placed after the main body of the program. Applesoft indicates an END statement is optional and need not be used. Experience shows that complicated programs will not always run properly without an end statement. The subroutines perform better when they branch past the end statement of the program. The subroutine runs and the RETURN branches to the statement immediately after the GOSUB in the main body of the program. Use an END statement with all programs.

LESSON 16

Functions

After completion of Lesson 16 you should be able to:

1. Use arithmetic functions in programming.
2. Convert radians to degrees using the DEF FN.

VOCABULARY

Argument — An argument is a variable factor, the value of which is determined by the function.

Degree — A degree is defined as 1/360 of a complete circle. Conversion from degrees to radians is — Degree / (180/PI) = Radian. The conversion factor (180/PI) = 57.29578.

Function — A function is that part of a computer instruction that specifies the operation to be performed.

Radian — A radian is the unit of plane angular measurement that is equal to the angle at the center of a circle subtended by an arc equal to the length of the radius. Conversion from radians to degrees is — Radians * (180/PI) = Degrees. The conversion factor (180/PI) = 57.29578.

DISCUSSION

In Lesson 16, the arithmetic functions are placed in alphabetical order in a program to demonstrate their use (Fig. 16-1).

A function is that part of the computer instruction that specifies the operation to be performed. Functions act upon the input to the function. The function then performs some operation on the argument, and outputs the result. The operation may involve many steps in a program stored in memory. Calling a function automatically makes use of these programmed steps.

Fig. 16-1 is a program to demonstrate what each function outputs. The output is defined in the RUN by using a PRINT statement with the function type placed in the output.

ABS — returns the absolute, or positive value of a number.

```
100 M = - 4:N = 2.5:P = 3:Q = 0
110 PRINT
120 PRINT "M = ";M;" N = ";N;" P = ";P;" Q = ";Q: PRINT
130 PRINT "ABS(M) = "; ABS (M)
140 PRINT "EXP(P)  = "; EXP (P)
150 PRINT "LOG(P)  = "; LOG (P)
160 PRINT "RND(P)  = "; RND (P)
170 PRINT "SGN(M) = "; SGN (M)
180 PRINT "SGN(Q) = "; SGN (Q)
190 PRINT "SGN(N) = "; SGN (N)
200 PRINT "SQR(P)  = "; SQR (P)
210 REM :GEOMETRIC FUNCTIONS GIVEN IN RADIANS
220 REM :USE DEF FN TO CONVERT RADIANS TO DEGREES
230 PRINT "SIN(P)   = "; SIN (P);" RADIANS"
240 DEF FN SD(X)    = SIN (X / 57.2958)
250 PRINT "SIN (P) "; FN SD(P);"DEGREES"
260 PRINT "COS(P)  = "; COS (P);"RADIANS"
270 DEF FN CD(X)    = COS (X / 57.2985)
280 PRINT "COS(P)  ="; FN CD(P); " DEGREES
290 PRINT "TAN(P)  = "; TAN (P);" RADIANS
300 DEF FN TD(X)    = TAN (X / 57.2958)
310 PRINT "TAN(P)  = " FN TD(P);"DEGREES"
320 PRINT "ATN(P)  = "; ATN (P);" RADIANS"
330 DEF FN AD(X)    = ATN (X / 57.2958)
340 PRINT "ATN(P)  = "; FN AD(P); "DEGREES"
350 END
RUN
M = -4 N = 2.5 P = 3 Q = 0

ABS(M) = 4
EXP(P)  = 20.0855369
LOG(P)  = 1.09861229
RND(P)  = .0430616123
SGN(M) = -1
SGN(Q) = 0
SGN(N) = 1
SQR(P)  = 1.73205081
SIN(P)   = .141120008 RADIANS
SIN (P) .0523359375 DEGREES
COS(P)  = -.989992497 RADIANS
COS(P)  = .998629665 DEGREES
TAN(P)  = -.142546543 RADIANS
TAN(P)  = .0524077605 DEGREES
ATN(P)  = 1.24904577 RADIANS
ATN(P)  = .0523120883 DEGREES
```

Fig. 16-1. Functions program.

EXP — raises the value to six places to the indicated power of 2.718289.
LOG — this is the natural logarithm function. The conversion from the natural log, log base 10 is to divide by 2.302585093.

RND — returns a real number greater than zero and less than one. The random number returned should be different each time the program is RUN.

SGN — if the expression is positive, a value of +1 is returned. If the expression is zero, a zero is returned. If the expression is negative, −1 is returned.

SQR — returns the square root of a number.

SIN, COS, TAN, ATN — are trigonometric functions. Their values are returned in radians. The conversion from radians to degrees, and vice versa, is discussed in the vocabulary section of Lesson 16. Many people work in degrees, so the DEF FN is a simple way to convert radians to degrees.

Table 16-1. ASCII Character Codes

CODE		CHAR	CODE		CHAR	CODE		CHAR	CODE		CHAR
Dec	Hex		Dec	Hex		Dec	Hex		Dec	Hex	
0	00	NUL	32	20	SP	64	40	@	96	60	
1	01	SOH	33	21	!	65	41	A	97	61	a
2	02	STX	34	22	"	66	42	B	98	62	b
3	03	ETX	35	23	#	67	43	C	99	63	c
4	04	EOT	36	24	$	68	44	D	100	64	d
5	05	ENQ	37	25	%	69	45	E	101	65	e
6	06	ACK	38	26	&	70	46	F	102	66	f
7	07	BEL	39	27	'	71	47	G	103	67	g
8	08	BS	40	28	(	72	48	H	104	68	h
9	09	HT	41	29	)	73	49	I	105	69	i
10	0A	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	45	2D	−	77	4D	M	109	6D	m
14	0E	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	/
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	DEL

Other language, string, and numeric functions are listed and defined below.

ASC("A") — returns the ASCII code for the character in the argument. The ASCII character codes are shown in Table 16-1. Strings cannot be converted directly to numerics. ASC("A") is used to convert the character in the string to an ASCII numeric value, which is 65.

CHR$(65) — numerics cannot be converted directly to strings. CHR$(65) converts the ASCII value 65 to the string character "A."

FRE (0) — returns the number of bytes of free memory available. PRINT FRE(0) is the immediate execution command. Changing the argument from 0 to 10 has no effect on the amount of memory returned.

INT — returns the largest integer value of a real number. PRINT INT(3.14) returns the integer 3.

LEFT$ — discussed in detail in Lesson 15.

LEN — returns the length of the string. A$ = "HI SUE". PRINT LEN(A$), LEN("HI SUE") returns the number 6 for each LEN, the number of characters and spaces in the string.

MID$ — discussed in detail in Lesson 15.

RIGHT$ — discussed in detail in Lesson 15.

STR$ — returns a string that represents the value of the argument. PRINT STR$(3.14) returns a string value of 3.14. The STR$ function converts a real to a string.

VAL — interprets a string value. PRINT VAL("3.14") returns 3.14. The VAL function converts a string to a real.

LESSON 17

List, Delete, and Edit

After completion of Lesson 17 you should be able to:

1. List programs or parts of programs, and delete a program or delete part of a program.
2. Edit using ESCAPE and I, J, K, or M, or ESCAPE and the direction arrow keys.
3. Comprehend four types of program statements to edit.
4. Delete text and/or spaces from an existing line of text.
5. Insert text into an existing line of text.

VOCABULARY

Cursor — The cursor is the symbol next to the prompt on the screen. In the 40 column mode (80 column card inactive) the cursor is the checkerboard green on black symbol. In the 80 column mode (80 column card active) the cursor is a solid symbol green on black.

DEL — This is the immediate execution command that deletes a line or lines from a program in Applesoft. DELETE is the immediate command that deletes a program from DOS 3.3.

Edit — This means to arrange data into a format required for subsequent processing. Editing may involve deletion of data not required, conversion of fields into machine language (i.e., Applesoft language statements converted to binary), and preparation of data for subsequent output (i.e., zero printing).

LIST — This command lists all the line numbers and program statements in a program.

Prompt — The prompt is the symbol that designates the language the computer is using. In Applesoft, the prompt is (]), and in Integer Basic the prompt is (>).

DISCUSSION

If a program is in memory, the entire program can be listed at one time, or parts of it can be listed. The following examples show how the LIST func-

tion can be used. In this example, the line numbers of the program run from 10 through 500.

LIST — Lists the entire program.
LIST 10,10 — Lists line number 10.
LIST 10,100 — List lines 10 through 100.
LIST –100 — Lists all lines above 100.
LIST 100– — Lists all lines below 100.

The DEL function deletes portions of Applesoft programs. The DEL function is different from the DELETE function. The DELETE command relates to the disk operating system. With a proper file name, DELETE will clear the file off the disk. See Lesson 4, SAVE AND LOAD PROGRAMS TO DISKS.

NEW — Deletes an entire program stored in memory.
DEL 10,10 — Deletes line 10 of the program.
DEL 10 — Causes the message ?SYNTAX ERROR to be printed on the screen. The proper form for a one line deletion is DEL 10,10.
10 — Press RETURN causes line 10 to be deleted from the program.
DEL 10,100 — Causes lines 10 through 100 to be deleted from the program.
DEL 10-100 Causes a message ?SYNTAX ERROR to be printed on the screen.

In the 40 column mode, the Apple IIe has six edit methods.

1. 80 column card inactive — checkerboard cursor.

Press ESCAPE once.

1. I — moves cursor up
2. J — moves cursor left
3. K — moves cursor right
4. M — moves cursor down

2. 80 column card inactive — checkerboard cursor.

Press ESCAPE once.

1. ← moves cursor left
2. → moves cursor right
3. ↓ moves cursor down
4. ↑ moves cursor up

3. 80 column card inactive — checkerboard cursor

Press ESCAPE before each cursor key is pressed.

1. B — moves cursor left
2. A — moves cursor right
3. D — moves cursor up
4. C — moves cursor down

4. 80 column card active — solid cursor

Press ESCAPE — produces a "+" in the solid cursor to indicate the edit mode.

1. I — moves cursor up
2. J — moves cursor left

3. K — moves cursor right
4. M — moves cursor down

5. 80 column card active — solid cursor.

Press ESCAPE — produces a "+" inside the solid cursor.

1. ← moves cursor left
2. → moves cursor right
3. ↓ moves cursor down
4. ↑ moves cursor up

6. 80 column card active — solid cursor

Press ESCAPE before each cursor move key is pressed. The "+" sign is placed inside the solid cursor.

1. B — moves cursor left
2. A — moves cursor right
3. D — moves cursor up
4. C — moves cursor down

(Only the cursor-edit moves in items #1, #2, #4, and #5 will be discussed. The edit moves in items #3, and #6 are not discussed because the cursor-edit moves are so slow and tedious to perform.)

In the 40 column mode, 80 column card inactive, the cursor looks like a checkerboard. The cursor configuration does not change when going from the nonedit mode to the edit mode (Fig. 17-1).

In the 40 column mode with the 80 column card active, the cursor is solid and inverse of the screen (i.e., green on black). In the edit mode, the cursor contains a "+" sign (Fig. 17-2).

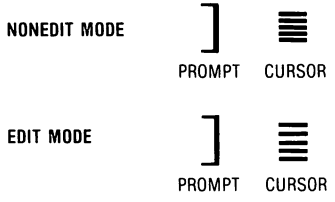


Fig. 17-1. 40 column mode with 80 column card inactive.

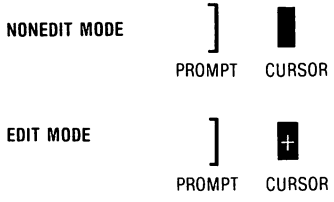


Fig. 17-2. 80 column mode with 80 column card active.

The editing technique is the same in the 80 column card inactive mode as it is in the 80 column card active mode.

To use the edit mode, press ESCAPE and then move the cursor to the desired edit position by using I, J, K, M, or the four direction arrow keys.

To leave the edit mode, press any alpha, numeric, or special character key, except I, J, K, or M. For simplicity of discussion, the SPACE BAR will be the special character key that is pressed to change from the edit mode to the nonedit mode. There are four types of statements that are edited.

1. Characters enclosed in quotation marks that occupy one line, or less, on the 40 column screen.
2. Characters that are not enclosed in quotation marks that occupy one line, or less, on the 40 column screen.
3. Characters enclosed in quotation marks that occupy more than one line on the 40 column screen.
4. Characters not enclosed in quotation marks that occupy more than one line on the 40 column screen.

To edit characters enclosed in quotation marks occupying one line, or less in the 40 column mode, do the following steps.

1. Press ESCAPE.
2. Move the cursor to the first integer in the line number of the program statement using I, J, K, M, or the direction arrow keys.
3. Press the SPACE BAR to change from the edit mode to the nonedit mode.
4. Use the forward arrow (repeat key) key to position the cursor past the closing quote in the line.
5. Press RETURN.
6. If the cursor is stopped before the last character in the line, and RETURN is pressed, all characters from the cursor position to the end of the statement will be lost.

To edit characters not enclosed in quotation marks and occupying one line, or less, in the 40 column mode, do the following:

```
10 X = INT(4000*2.948E+21/SIN(B))
```

1. Press ESCAPE.
2. Move the cursor to the first integer in the line number of the program statement to be edited, using I, J, K, M, or the direction arrow keys.
3. Press the SPACE BAR to change from the edit mode to the nonedit mode.
4. Use the forward arrow (repeat key) key to place the cursor past the last character in the line.
5. Press RETURN.
6. If the cursor is stopped before the last character in the line, and RETURN is pressed, all characters after the cursor position will be lost.

To edit characters enclosed in quotation marks occupying more than one line, in the 40 column mode, follow this procedure.

1. Press ESCAPE.
2. Place the cursor over the first integer in the line number of the program statement to be edited.

3. Press the SPACE BAR to change from the edit mode to the nonedit mode.
4. Move the cursor past the last character in the first line.

```
10 PRINT "THIS IS THE UNITED STA
   TES OF AMERICA"XX(XX REPRESENTS CURSOR)
```

5. Press ESCAPE to change to the edit mode.
6. Press "K," or the forward arrow (repeat key) key until the cursor is over the "T" in TES OF AMERICA.

```
10 PRINT "THIS IS THE UNITED STA
   TES OF AMERICA"
```

7. Press the SPACE BAR to change from the edit mode to the nonedit mode.
8. Press the forward arrow (repeat key) key until the cursor is past the closing quotation mark in the second line.

```
10 PRINT "THIS IS THE UNITED STAXX
   TES OF AMERICA"XX (POSITION OF THE CURSOR)
```

9. Press RETURN.

If the cursor (in the nonedit mode) is run from the first integer in the line number of the statement to be edited over both of the lines in quotation marks, spaces will be left between the STA and the TES OF AMERICA.

```
10 PRINT "THIS IS THE UNITED STA
                TES OF AMERICA"
20 END
RUN
THIS IS THE UNITED STA                  TES OF
 AMERICA
]
```

To edit characters not enclosed in quotation marks that occupy more than one line in the 40 column mode, follow this procedure.

```
10 X = INT(10000.87694*1E+23/SIN(B))
   * COS(B +W)/EXP(B+G)
```

1. Press ESCAPE.
2. Move the cursor to the first integer in the line number of the statement to be edited.
3. Press the SPACE BAR to change from the edit mode to the nonedit mode.
4. Use the forward arrow (repeat key) key to place the cursor past the last character in the second line. (Text leaves spaces when the cursor passes over the entire two lines. Formulas do not leave spaces when the cursor passes over both lines.)

5. Press RETURN.
6. If the cursor is stopped before the last character in the second line, and RETURN is pressed, all characters from the cursor to the end of the statement will be lost.

To take words and/or spaces out of a line of text without retyping the line, follow this procedure.

```
10 PRINT "THIS IS THE USA"
20 END
```

1. Press ESCAPE.
2. Move the cursor over the one (1) in the ten (10) of the line number of the program statement to be edited, by using I, J, K, M, or the direction arrow keys.
3. Press the SPACE BAR to change from the edit mode to the nonedit mode. The IS (and following space) is to be removed without leaving blanks and unusual spaces when the program is run.
4. Use the forward arrow key to move the cursor to the "I" in IS.

```
10 PRINT "THIS IS THE USA"
20 END
```

5. Press ESCAPE.
6. Use the "K" or forward arrow to move the cursor over the "T" in THE.

```
10 PRINT "THIS IS THE USA"
20 END
```

7. Press the SPACE BAR to change from the edit mode to the nonedit mode.
8. Use the forward arrow key to move the cursor past the closing quotation mark.
9. Press RETURN.

```
10 PRINT "THIS THE USA"
20 END
RUN
THIS THE USA
```

To add text into a line of text enclosed in quotation marks, follow this procedure.

```
10 PRINT "THIS IS THE USA" (The object is to place the words GOOD OLDX (X REPRESENTS A SPACE) between THE and USA)
```

1. Press ESCAPE.
2. Place the cursor over the one (1) in the line number ten (10) in the statement to be edited by using I, J, K, M, or the direction arrow keys.
3. Press the SPACE BAR to change from the edit mode to the nonedit mode.

4. Use the forward arrow key to place the cursor over the "U" in USA.

```
10 PRINT "THIS IS THE USA"
20 END
```

5. Press ESCAPE.
6. Press the "I" key, or the up arrow key, one time to raise the cursor one line above the line of text.
7. Press the SPACE BAR to change from the edit mode to the nonedit mode.
8. Type in the words GOOD OLDX (X represents a space).

```
                GOOD OLDX
10 PRINT "THIS IS THE USA"
```

9. Press ESCAPE.
10. Press the "J" key or the back arrow key until the cursor is over the "G" in GOOD, and above the "U" in USA.
11. Press the "M" key, or the down arrow key one time so the cursor is over the "U" in USA.
12. Press the SPACE BAR to change from the edit mode to the nonedit mode.
13. Press the forward arrow key until the cursor is past the closing quotation mark.
14. Press RETURN.

```
10 PRINT "THIS IS THE GOOD OLDXUSA"
20 END

RUN
THIS IS THE GOOD OLDXUSA
```

LESSON 18

Play Computer

After completion of Lesson 18 you should be able to:

1. Play computer and RUN a program manually, or mentally, to determine the output.
2. Play computer to determine why a program doesn't RUN or why a program doesn't run correctly (debug).
3. Use the TRACE function to aid in debugging programs.
4. Use NOTRACE function to counter the TRACE function.

VOCABULARY

NOTRACE — This command turns off the TRACE mode (see TRACE below).

Pass — A pass is the single execution of a loop, or the passage of magnetic tape or disks under the read/write head of a device.

TRACE — The TRACE function is an aid in following the sequence of execution of a program. It is used as an aid in debugging programs. TRACE causes the line number to be printed as the program executes. The data or information is also printed out in relation to the line number. TRACE is turned off by NOTRACE. TRACE and NOTRACE are immediate or deferred commands.

DISCUSSION

The primary purpose of this lesson is to "think" like a computer and run the program mentally and manually. Will you be able to determine exactly what the program will output, instead of what you think it will output? You must learn to think like a computer if you are going to understand and outsmart this exacting machine. When you play computer, the program is RUN exactly as it is written. If the rule of default applies, use the rule of default. If the program should branch, follow the branch. Chart 18-1 is designed to allow you to write the value of each variable as the program progresses. RUN the program mentally several times to get the feel of the program. Complete Chart 18-1 to see if you think like a computer. The program, RUN, and TRACE of the program are shown in Fig. 18-1.

```
10   REM : PROGRAM TO PLAY COMPUTER
20   A = 5 : B = 10 : C = -10
30   IF C > 0 THEN 130
40   IF ( B > A) THEN 90
```

(without parentheses: IF B>A THEN 90, produces a SYNTAX ERR because AT is a reserved word — to correct this, use parentheses around B > A)

```
50   IF C < = 0 THEN C = C + 1
60   B = B - 2
70   PRINT A, B, C
80   GOTO 30
90   A = A + 1
100  C = C + 2
110  PRINT A, B, C
120  GOTO 30
130  C = C - 10
140  PRINT A, B, C
150  END

RUN
 6          10          -8
 7          10          -6
 8          10          -4
 9          10          -2
10          10           0
10           8           1
10           8          -9
TRACE
RUN
#10  #20  #20  #20  #30  #40  #90  #100  #110  6
10                   -8
#120  #30  #40  #90  #100  #110  7
10                   -6
#120  #30  #40  #90  #100  #110  8
10                   -4
#120  #30  #40  #90  #100  #110  9
10                   -2
#120  #30  #40  #90  #100  #110  10
10                    0
#120  #30  #40  #50  #60  #70  10
8                     1
#80  #30  #130  #140  10          8
-9
#150
```

Fig. 18-1. Program Play Computer.

CHART 18-1. Variable Chart

ASSIGN VALUE OF VARIABLES	A	B	C
From line 20	5	10	−10
Values-1st pass			
Values-2nd pass			
Values-3rd pass			
Values-4th pass			
Values-5th pass			
Values-6th pass			
Values-7th pass			

After Chart 18-1 has been completed and you are satisfied you understand how and why to program functions, RUN the program to get the correct results. Did you do as well as the computer?

Type in the immediate command TRACE, and press RETURN. Now RUN the program. When the program runs, the line numbers, and variable values, are printed on the screen. If the program does not RUN, TRACE can aid in debugging the program, to determine why it doesn't run. The error messages built into the language can also aid in debugging a program. If the program stops at line 120, and no error message is given, many times the TRACE mode can aid in correcting the problem.

TRACE function can be removed by typing NOTRACE on the screen. The next program RUN will be without line numbers on the screen. The sequence of commands to run the program and use the TRACE and NOTRACE functions should follow this routine.

```
RUN
```

```
TRACE
RUN
```

```
NOTRACE
```

LESSON 19

Reserved Words

After completion of Lesson 19 you should be able to:

1. Use reserved words in programs in their proper relationship.
2. Use parentheses to separate characters that the computer interprets as reserved words.

VOCABULARY

Reserved Words — Reserved words are words programmed into the language to aid in carrying out the programming functions.

DISCUSSION

Reserved words are an aid in programming. These words cannot be used as variables. Applesoft tokenizes reserved words to a decimal number similar to the decimal number that represents an ASCII symbol. For example, there is a ?SYNTAX ERROR in the following statement.

```
40 IF B = A THEN 90
```

When the program is RUN, the program is stopped at line 40 and ?SYNTAX ERROR is printed on the screen. When line 40 is LISTed, the variable A and the reserved word THEN have been changed to the reserved word AT and a new variable HEN90.

```
40 IF B = AT HEN90
```

To use the same variable A in relation to the reserved word THEN place the variables and the equals sign within parentheses.

```
40 IF (B = A) THEN 90
```

The list of reserved words is taken directly from Applesoft *Basic Programmer's Reference Manual, Volume 2 For the IIe Only,* by the Apple Computer Inc., 10260 Bandley Dr., Cupertino, California 95014

RESERVED WORDS IN APPLESOFT

&
ABS AND ASC AT ATN
CALL CHR$ CLEAR COLOR= CONT COS
DATA DEF DEL DIM DRAW
END EXP
FLASH FN FOR FRE
GET GOSUB GOTO GR
HCOLOR= HGR HGR 2 HIMEM: HLIN HOME HPLOT HTAB
IF IN# INPUT INT INVERSE
LEFT$ LEN LET LIST LOAD LOG LOMEM:
MID$
NEW NEXT NORMAL NOT NOTRACE
ON ONERR OR
PDL PEEK PLOT POKE POP POS PRINT PR#
READ RECALL REM RESTORE RESUME RETURN RIGHT$ RND ROT=
RUN
SAVE SCALE= SCRN(SGN SHLOAD SIN SPC(SPEED= SQR STEP
STOP STORE STR$
TAB(TAN TEXT THEN TO TRACE
USR
VAL VLIN VTAB
WAIT
XPLOT XDRAW

LESSON 20

Menu Selection and Coding Formulas

After completion of Lesson 20 you should be able to:

1. Write programs using a menu selection.
2. Translate formulas to computer code for computational purposes.

VOCABULARY

Code — Code is the representation of data or instructions in symbolic form. It is sometimes synonymous with instruction. Coding is the act of converting data or instructions into program statements.

Comment — Comment is the written note that can be included in the coding of a computer program to clarify the procedures and variables, but has little effect on the computer itself. The REM statement is used for comment, or to document a program.

*Get A$* — GET A$ stops the program in order to view the output. The program will resume when any key is pressed.

Menu Selection — A menu selection is a method of using a terminal to display a list of options that can be chosen by the user. The menu asks a simple question. The user can respond to the question by entering a number value, or an alphabetic character. This is referred to as "user friendly."

DISCUSSION

Many people have little knowledge of computers. For those people, the programs must be written to tell them what input is required, and in what format. One way to aid these people with the correct selection is to use a menu. A menu selection is a method to display a list of optional choices. The user can select his or her choice by entering a number, or a letter.

Lesson 20 deals with the menu selection process. The program written for this lesson deals with three types of depreciation, (1) straight line depreciation, (2) double declining balance depreciation, and (3) sum of the years digits depreciation. The variables used in the program are listed in Fig. 20-1.

STRAIGHT LINE DEPRECIATION

BV	Book value
DY	Depreciation per year
GA	Gross amount or gross cost of the asset
GET A$	Stops the program to allow the user to view the output
LA	Life of the asset
P	Rounded to 2 places (100)
S	Selection
SV	Salvage value
TD	Total depreciation

DOUBLE DECLINING BALANCE

BV	Book value
DY	Depreciation per year
GA	Gross amount or gross cost of the asset
GET A$	Stops the program to allow the user to view the output
K	Constant
LA	Life of the asset
SV	Salvage value
TD	Total depreciation

SUM OF THE YEARS DIGITS

BV	Book value
DY	Depreciation per year
GA	Gross amount or gross cost of the asset
GET A$	Stops the program to allow the user to view the output
K	Constant
LA	Life of the asset
SV	Salvage value
TD	Total depreciation
Z	Variable to hold (LA − Y). Z = (LA − Y) + 1. A method to compute and print the years forward, after they were computed backwards

Fig. 20-1. Variables.

```
500 HOME : VTAB 3 : HTAB 8 : PRINT "***DEPRECIATION***" :
    PRINT : PRINT
510 HTAB 5 : PRINT "1. STRAIGHT LINE DEPRECIATION" : PRINT
520 HTAB 5 : PRINT "2. DOUBLE DECLINING BALANCE" : PRINT
530 HTAB 5 : PRINT "3. SUM OF THE YEARS DIGITS" : PRINT
540 HTAB 8 : INPUT "SELECTION PLEASE!" : PRINT
550 IF S < 1 OR S > 3 THEN 500
560 ON S GOTO 1500, 2500, 3500
```

Fig. 20-2. Menu section of depreciation program.

The menu selection of the program is listed in Fig. 20-2, and consists of lines 500 through 560.

Line 500 clears the screen and sets the position at which ***DEPRECIATION PROGRAM*** is printed. The two PRINT statements after

DEPRECIATION PROGRAM leave two blank lines before the first menu selection.

Lines 510 through 530 print out the three types of depreciation. The user must determine which type of depreciation to select (Table 20-1).

Table 20-1. Formulas and Computations

STRAIGHT LINE DEPRECIATION

$$\text{DEPRECIATION/YEAR} = \frac{\text{COST OF ASSET} - \text{SALVAGE VALUE}}{\text{NUMBER OF YEARS}}$$

YEAR	DEP/YR	TOTAL DEP.
1	200	200
2	200	400
3	200	600

DOUBLE DECLINING BALANCE (the numbers have been rounded)

3 year straight line = 1/3 .333 (K)
200% double declining balance = 2 * 1/3 .667 (K)

YEAR	CONSTANT		COST	DEP/YR	BOOK VL.	TOTAL DEPRECIATION
1	.667	*	625	417	208	417
2	.667	*	208	139	69	554
3	.667	*	69	46	25	600

SUM OF THE YEARS DIGITS (the numbers have been rounded)

$$\text{SYD} = \frac{n\,(n+1)}{2} \qquad \text{SYD} = \frac{3 * (3+1)}{2} = 6$$

YEAR	CONSTANT		GA -SV	DEP/YR	BOOK VL.	TOTAL DEPRECIATION
1	.500	*	600	300	325	300
2	.333	*	600	198	125	500
3	.167	*	600	100	25	600

PRINT YEARS
1
2
3

Z = (LA-Y) + 1
3 – 3 = 0 + 1 = 1
3 – 2 = 1 + 1 = 2
3 – 1 = 2 + 1 = 3

Line 3560 uses the INT function to round off the results to 2 places. DEF FN was used to round off to 2 places in line 2560. The two different methods expose the student to the fact that either method will accomplish the same rounding results.

Line 540 allows the user to select any number. The choice of the number is not limited to 1, 2, or 3.

Line 550 is a decision statement that causes a branch to line 500 if a number other than 1, 2, or 3 is entered. This is a form of error checking that limits the input choices that will be accepted by the program. Since there

are three choices, the computer is programmed so only an input of 1, 2, or 3 will allow the program to continue.

In line 560, the input value is placed in the variable. When S = 1, the program branches to line 1500. When S = 2, the program branches to line 2500. When S = 3, the program branches to line 3500. The ON S GOTO statement was discussed in Lesson 15 in conjunction with input errors.

When a correct selection is entered, the program branches to either line 1500, line 2500, or to line 3500. At each of these lines is a GOSUB 100. The GOSUB 100 causes a branch to line 100 of the program. Line 100 is the beginning of the input subroutine. This subroutine is placed at the beginning of the program because it is used with each type of depreciation. The program is more efficient with the input subroutine at the beginning of the program because fewer lines are searched before the subroutine is found. The headings are printed and the input is requested (Fig. 20-3).

```
100  INPUT "GROSS AMOUNT = $" ;GA : PRINT
105  IF GA < 1 THEN 100
110  INPUT "SALVAGE VALUE = $" ;SV : PRINT
115  IF SV < 0 THEN 110
120  IF GA < SV THEN 100
130  INPUT "LIFE OF ASSET = " ;LA : PRINT
135  IF LA < 1 THEN 130
140  RETURN
```

Fig. 20-3. Input section of depreciation program.

The gross amount of the asset is entered at line 100. Line 105 checks to determine that the gross amount is not a negative value. A negative gross amount would have no meaning in a depreciation schedule.

In line 110, the salvage value of the asset is entered. Line 115 checks to determine if the salvage value is less than zero. The salvage value could be zero at the end of the depreciation term, but it could not be less than zero.

In line 120, if the gross cost of the asset is less than the salvage value, the depreciation formula will not compute the depreciation properly.

In line 135, the asset must have a useful life of at least one year, or some period of time greater than zero.

In line 140, RETURN causes a branch to the section of the program that requested the information.

If 1 is entered from the menu section, the program branches to line 1500 which is the section to compute straight line depreciation (Fig. 20-4).

Line 1500 branches to the input routine. After the input values are entered, line 140 causes a RETURN to line 1510 to begin calculation of the straight line depreciation. The values used for all three depreciation examples are: cost of asset (GA) = $625, salvage value (SV) = $25, and life of the asset (LA) = 3 years. The straight line depreciation formulas are converted to the program statements.

```
1500 GOSUB 100
1510 TD = 0
1520 PRINT "YEAR DEP/YR TOTAL DEP.": PRINT :P = 100
1530 FOR X = 1 TO LA
1540 DY = (GA - SV) / LA
1550 TD = TD + DY
1560 PRINT X; TAB( 10); INT (DY * P + .5) / P; TAB( 25); INT (TD * P + .5) / P
1570 IF X = INT (X / 8) * 8 THEN GET A$
1580 NEXT
2400 GOTO 9990
RUN
                    ***DEPRECIATION PROGRAM***

  1. STRAIGHT LINE DEPRECIATION
  2. DOUBLE DECLINING BALANCE
  3. SUM OF THE DIGITS

                        SELECTION PLEASE!1
GROSS AMOUNT  = $625
SALVAGE VALUE =  $25
LIFE OF ASSET =    3

YEAR            DEP/YR          TOTAL DEP.
1               200             200
2               200             400
3               200             600

ANOTHER PROBLEM ? (Y OR N)N
```

Fig. 20-4. Section of program to compute straight line depreciation.

STRAIGHT LINE DEPRECIATION

$$\text{DEPRECIATION/YEAR} = \frac{\text{COST OF ASSET} - \text{SALVAGE VALUE}}{\text{NUMBER OF YEARS}}$$

DEPRECIATION INFORMATION	PROGRAM STATEMENT
LIFE OF THE ASSET = 3	FOR X = 1 TO LA
DEP/YR = (GA - SV)/LA	DY = (GA - SV)/LA
DEP/YEAR = $200	TD = TD + DY
TOTAL DEPRECIATION = $600	1580 NEXT X

Line 1560 prints the year, the amount of each year's depreciation, and the total depreciation. Line 1560 is included within the loop, so the results will be printed for each year's computation.

Line 1570 performs the same function in lines 1570, 2570, and 3570. It causes the loop to stop after every eight executions (as shown in Table 20-2). This is useful when the life of the asset is a long period (over ten years) of time and the user needs to study the sections of the printout. The integers could be any number such as 10, 12, or 20, as long as the number of lines is less than the number of lines on the screen.

Table 20-2 IF X = INT(X/8) * 8 THEN GET A$

X	INT(X/8)	INT(X/8) * 8
1	0	0
2	0	0
7	0	0
8	1	8
9	1	8
10	1	8
15	1	8
16	2	16
17	2	16

GET A$, from line 1570, is a function to stop the program to allow the user to press any key on the keyboard to continue the program. RETURN does not have to be pressed after the GET A$.

DOUBLE DECLINING BALANCE

The double declining balance method of depreciation applies a constant depreciation rate to a reducing book value. This method charges off high depreciation in the early years and lower amounts in later years. The rate used in this program is 200%, or twice the straight line depreciation. This fact is expressed in the formula K = (1/LA) * 2. The formula for the 150% rate is K =(1/LA) * 1.5. The formula for the 125% rate is K = (1/LA) * 1.25

Please be aware that this is a programming manual, not an accounting text. The double declining balance formula is taken from an accounting text and has been checked by a qualified accountant. The computed figures in the final year of the depreciation may cause questions. The final book value does not equate with the salvage value, nor does the total depreciation equate with the allowable depreciation. Adjustments must be made to the figures computed in the final year of the life of the asset.

In the double declining balance depreciation method, the constant "K" is multiplied by the initial cost of the asset to determine the amount of yearly depreciation. The cost (Book Value) is reduced by the depreciation amount each year, but the book value cannot be reduced below the salvage value. The total depreciation cannot be greater than the cost of the asset less the salvage value.

This example uses the same input as does the straight line depreciation. The cost of the asset (Book Value) = $625, salvage value = $25, and the life of the asset = 3 years.

The double declining balance routine begins at line 2500 and ends at line 3400 (Fig. 20-5).

Line 2500 branches to the input routine. The input routine RETURNs to line 2510 (K = (1/LA) * 2 : BV = GA). The constant "K" is computed by the

```
2500 GOSUB 100
2510 K = (1 / LA) * 2:BV = GA
2512 TD = 0
2515 PRINT "YR. CONST. DEP/YR BK. VAL. TOT DEP"
2520 FOR X = 1 TO LA
2530 DY = BV * K
2540 BV = BV - DY
2545 TD = TD + DY
2550 DEF FN A(X) = INT (X * 100 + .5) / 100
2560 PRINT X; TAB( 5); FN A(K); TAB( 14); FN A (DY); TAB( 22); FN A(BV); TAB( 31);
     FN A(TD)
2570 IF X = INT (X / 8) * 8 THEN GET A$: REM -CAUSES LOOP TO STOP EVERY
     8 TIMES
2580 NEXT
3400 GOTO 9990
RUN
                    ***DEPRECIATION PROGRAM***

  1. STRAIGHT LINE DEPRECIATION
  2. DOUBLE DECLINING BALANCE
  3. SUM OF THE DIGITS

                    SELECTION PLEASE !2

GROSS AMOUNT = $625
SALVAGE VALUE = $25
LIFE OF ASSET = 3

YR.         CONST.      DEP/YR      BK. VAL.    TOT DEP
1           .67         416.67      208.33      416.67
2           .67         138.89      69.44       555.56
3           .67         46.3        23.15       601.85

ANOTHER PROBLEM ? (Y OR N)N
```

Fig. 20-5. Double declining balance routine.

formula (1/LA) * 2) for 200% of straight line depreciation. BV = GA is the gross cost of the asset stored in the variable BV. The constant times the reducing book value will give the yearly depreciation.

Line 2512 initializes the total depreciation to zero, and line 2515 causes the headings to be printed.

Line 2520 is the beginning statement of the loop for the computations.

Line 2530 computes the depreciation for one year and stores that value in the variable DY (depreciation per year).

Line 2545 is the summing statement that adds each year's depreciation to the previous year's depreciation and stores the total depreciation in the total depreciation variable TD.

Line 2550 is a DEF FN statement that rounds the printed calculations to two decimal places.

Line 2560 causes the information to be printed in table form on each loop execution. Line 2570 is the same as 1570; results are seen in Table 20-2.

Line 3400 branches to line 9990, which cues the user for more input.

```
3500 GOSUB 100
3510 T = 0:P = 100:BV = GA - SV:T D = 0
3520 PRINT "YEAR CONSTANT DEP/YR TOTAL DEP."
3522 PRINT
3530 FOR X = 1 TO LA:T = T + X: NEXT
3535 REM :T=LA*(LA+1)/2
3540 FOR Y = LA TO 1 STEP - 1:K = Y / T
3550 DY = K * BV
3555  TD = TD + DY
3558 Z = (LA - Y) + 1
3560 PRINT Z; TAB( 7); INT (K * P + .5) / P; TAB( 18); INT (DY * P + .5) / P;
     TAB( 26 + (TD < 100)); INT (TD * P + .5) / P
3570 IF Z = INT (Z / 8) * 8 THEN GET A$
3600 NEXT
RUN
```

```
                    ***DEPRECIATION PROGRAM***

  1. STRAIGHT LINE DEPRECIATION
  2. DOUBLE DECLINING BALANCE
  3. SUM OF THE DIGITS

                    SELECTION PLEASE !3

GROSS AMOUNT = $625
SALVAGE VALUE = $25
LIFE OF ASSET = 3

YEAR          CONSTANT        DEP/YR          TOTAL DEP.
1             .5              300             300
2             .33             200             500
3             .17             100             600
ANOTHER PROBLEM ? (Y OR N)N
```

Fig. 20-6. Sum of the years digits section.

```
50    GOTO 500
100   INPUT " GROSS AMOUNT = $" ; GA : PRINT
105   IF GA < 1 THEN 100
110   INPUT "SALVAGE VALUE = $" ;SV : PRINT
115   IF SV < 0 THEN 110
120   IF GA < SV THEN 100
130   INPUT "LIFE OF ASSET = " ;LA : PRINT
135   IF LA < 1 THEN 130
140   RETURN
500   HOME : VTAB 3 : HTAB 8 : PRINT "***DEPRECIATION***" :
      PRINT : PRINT
510   HTAB 5 : PRINT "1. STRAIGHT LINE DEPRECIATION" : PRINT
520   HTAB 5 : PRINT "2. DOUBLE DECLINING BALANCE" : PRINT
530   HTAB 5 : PRINT "3.SUM OF THE YEARS DIGITS" : PRINT : PRINT
540   HTAB 8 : INPUT "SELECTION PLEASE!" ; S : PRINT
550   IF S < 1 OR S > 3 THEN 500
```

Fig. 20-7. Depreciation program.

```
560   ON S GOTO 1500, 2500, 3500
1500  GOSUB 100
1510  TD = 0
1520  PRINT ''YEAR        DEP/YR        TOTAL DEP.'' : PRINT :
      P = 100
1530  FOR X = 1 TO LA
1540  DY = (GA - SV)/LA
1550  TD = TD + DY
1560  PRINT X; TAB (10); INT (DY*P + .5)/P; TAB (25);
      INT(TD*P + .5)/P
1570  IF X = INT (X/8) * THEN GET A$
1580  NEXT
2400  GOTO 9990
2500  GOSUB 100
2510  K = (1/LA) * 2 : BV = GA
2512  TD = 0
2515  PRINT '' YR       CONST.       DEP/YR       BK.VAL.       TOT DEP''
2520  FOR X = 1 TO LA
2530  DY = BV * K
2540  BV = BV - DY
2545  TD = TD + DY
2550  DEF FN A(X) = INT (X*100 + .5)/100
2560  PRINT X; TAB (5); FNA (K); TAB (14); FNA (DY);
      TAB (22); FNA (BV); TAB(31); FNA (TD)
2570  IF X = INT (X/8) * 8 THEN GET A$
2580  NEXT
3400  GOTO 9990
3500  GOSUB 100
3510  T = 0 : P = 100 : BV = GA - SV : TD = 0
3520  PRINT ''YEAR       CONSTANT       DEP/YR       TOTAL DEP.'' : PRINT
3530  FOR X = 1 TO LA : T = T + X : NEXT X
3535  REM : T = LA * (LA+1)/2
3540  FOR Y = LA TO 1 STEP - 1 : K = Y/T
3550  DY = K * BV
3555  TD = TD + DY
3558  Z = (LA - Y) + 1
3560  PRINT Z; TAB (7); INT (K*P + .5)/P; TAB (18);
      INT (DY*P + .5)/P; TAB (26 + (TD < 100));
      INT (TD*P + .5)/P
3570  IF Z = INT (Z/8) * 8 THEN GET A$
3600  NEXT
9990  PRINT
9991  INPUT ''ANOTHER PROBLEM? (Y OR N)'' ; A$ :
      IF A$ = ''Y'' THEN 500
9999  END
```

Fig.20-7–cont. Depreciation program.

SUM OF THE YEARS DIGITS DEPRECIATION

The third type of depreciation is the sum of the years digits (Fig. 20-6) and consists of lines 3500 through 3600. In this method of depreciation, the years of the asset life are listed numerically and totaled. The highest year in the life of the asset is then divided by the total to compute the depreciation constant for the first year. The changing yearly constant is multiplied by a fixed gross amount of the asset less the salvage value.

SUM OF THE YEARS DEPRECIATION

DEPRECIATION INFORMATION	PROGRAM STATEMENTS
Total years 3 + 2 + 1 = 6	FOR X = 1 TO LA
	T = T + X : NEXT X
	T = LA * (LA + 1)/2
3/6 = 1st year's maximum depreciation	FOR Y = LA TO 1 STEP – 1
	K = Y/T
2/6 = 2nd year's depreciation	
1/6 = 3rd year's depreciation	
	DY = K * BV
	TD = TD + DY
	Z = (LA-Y) + 1
	NEXT Y

The entire program is shown in Fig. 20-7.

LESSON 21

Program Outline

After completion of Lesson 21 you should be able to:

1. Comprehend that all computer programs are written according to a general outline, but no program will exactly follow such an outline.

VOCABULARY

DATA — Data is a general expression used to describe any group of operands that denote any conditions, values, or states (i.e., all values and descriptive data operated on by a computer program but not part of the program itself). The word data is used as a collective noun and is usually accompanied by a singular verb: "data are" may be pedantically correct but is awkward syntax. Data is sometimes contrasted to information, which is said to be the result of processing data. Information is derived from the assembly, analysis, or summarizing of data into meaningful form.

DATA Statement — A DATA statement contains a list of items that can be used by the READ statement. DATA statements can contain integers, reals, variables, literals, or strings. The item in the DATA statement must be in the same relationship and position as the item in the READ statement.

READ Statement — The READ statement is used by the program to read data into memory.

DISCUSSION

The outline for program structure must be considered very general and no program will rigidly comply with the outline. There must be a starting point to writing a program. The logical start to writing a program must exist within the framework of an outline.

COMPUTER PROGRAM GENERAL OUTLINE

A. Beginning of the program.

B. Initialize the variables
   1. C = 0 counting variable.
   2. S = 0 summing variable.
   3. F = 0 flag variable.
   4. DEF FN define functions for formulas or rounding.
   5. DIM (2,3) DIM statements when constants are used.
   6. DIM (R,C) DIM (R,C) statements are used after the variables have been entered either by READ, INPUT, or assignment statements.
   7. RESTORE RESTORE resets the "data list pointer" to the first element of data. RESTORE causes the next READ statement encountered to re-READ the DATA statements from the first one.

C. Print the general program headings.
D. Menu selection.
E. INPUT–READ statements.
F. Beginning of the FOR–NEXT loop, or GOTO loop.
G. Decision statements.
H. Computation statements.
I. Incrementing statements.
   1. C = C + 1 counting statement.
   2. S = S + X summing statement.

J. PRINT — A PRINT statement inside a loop prints each time the loop executes.
K. End of the loop NEXT for a FOR–NEXT loop, or GOTO for a GOTO loop.
L. PRINT — A PRINT statement in this position outputs information after the last execution of the loop.
M. DATA — The DATA statement(s) can be placed anywhere in the program. Generally they are placed just above the END statement.
N. END — The END statement denotes the end of the main body of the program.
O. SUBROUTINES — SUBROUTINES are placed after the END statement of a program.

The FOR–NEXT loop will be used to demonstrate the effects of statements inside the loop, and outside the loop. The program will follow the general outline, where possible.

```
10 SUM = 0                      initialize variables
20 FOR X = 1 TO 5               head of the for next loop
30 PRINT X                      print a variable inside the loop
40 SUM = SUM + X                summing statement inside the loop
50 NEXT X                       foot of the loop statement
60 PRINT "SUM = "; SUM          print statement outside the loop for final values
```

```
RUN
1
2
3
4
5
SUM = 15
```

The program generally conforms to the outline. Lines 30 and 40 should be reversed according to the outline. Does it make a difference if they are reversed?

```
30 SUM = SUM + X
40 PRINT X
RUN
1
2
3
4
5
SUM = 15
```

No, the reversal of lines 30 and 40 within the loop make no difference. Generally, the PRINT X statement comes after the FOR statement at the beginning of the loop unless there are computation statements inside the loop.

Now reverse lines 30 and 40 to return to the original program. From the original program, make these changes.

```
DEL 30,30
55   PRINT X
```

The complete program looks like this.

```
10   SUM = 0
20   FOR X = 1 TO 5
40   SUM = SUM + X
50   NEXT X
55   PRINT X
60   PRINT "SUM = ";SUM
70   END
RUN
6
SUM = 15
```

The loop executes from 1 to 5 and prints out the next number in the series. There are two important points in this example, (1) because the general outline was violated, the program did not produce the correct results, and (2) the loop variable value is one more than the final index value (5) after the last execution of the loop.

To the original program, make these changes.

```
DEL 60,60
45   PRINT "SUM = ";SUM
```

The entire program follows.

```
10   SUM = 0
20   FOR X = 1 TO 5
30   PRINT X
40   SUM = SUM + X
45   PRINT "SUM = ;SUM
50   NEXT X
70   END
RUN
1
SUM = 1
2
SUM = 3
3
SUM = 6
4
SUM = 10
5
SUM = 15
```

Since the SUM was within the loop, SUM = was printed with each execution of the loop. Make the following changes to the original program.

```
DEL 10,10
25   SUM = 0
```

The complete program follows.

```
20   FOR X = 1 TO 5
25   SUM = 0
30   PRINT X
40   SUM = SUM +X
50   NEXT X
60   PRINT "SUM = ";SUM
70   END
RUN
1
2
3
4
5
SUM = 5
```

In this case, the summing variable was initialized to zero each time the loop executed, so the final value is SUM = 5. The initialized variable must be outside the loop, or it will be reset to zero each time the loop executes.

LESSON 22

Cleanup

After completion of Lesson 22 you should be able to:

1. Open the closet door and have all the final tidbits of the Applesoft language fall out for your inspection and pleasure.

VOCABULARY

Center Justify — This means the text is centered with no well defined right or left margins.

Fill Justify — This means that the right and left margins are aligned.

Left Justify — This means to format the output so the printed field is aligned on the left hand boundary. Most computers left justify alphabetic data.

Print Field Definition — This is the technique of print output to fit a standard form, thus making it more readable.

Right Justify — This means to format the output so the printed field is aligned on a right hand boundary. Most computers right justify numeric data.

Zero Printing — This is the printing of zeros to fill decimal places. Applesoft does not zero print. A special routine must be written in the program to cause zero printing. For example, if the number should be $123.10, Applesoft only prints $123.1.

Zero Suppression — This is the elimination of nonsignificant zeros before printing, i. e., those zeros to the left of the significant digits. It is also known as zero elimination.

DISCUSSION

This lesson "cleans up" many parts of the language not covered in the first 21 lessons.

Applesoft does not align columns of different numbers to the power of ten. All numbers are printed starting from one selected space and are printed to the right.

284.6
98.3
2
3.47

Fig. 22-1 is a program that causes numbers to be aligned in the proper columns, so the units are aligned, the tens are aligned, etc. The program causes the printout to right justify by two methods, (1) by aligning the right hand number, and (2) by aligning the decimal point. Either justification can be used in a program as a routine or a subroutine. Lines 60 through 80 right justify any number. Lines 90 through 130 right justify the decimal of numbers that are greater than or equal to one-tenth (.1) but less than 1E+9.

```
5    D = LOG (10)
6    DEF FN MA (X) = INT (LOG (M)/ D)
10   HOME
20   INPUT "ENTER?" ;M
30   IF M = 0 THEN END
40   GOSUB 60
50   GOTO 20
60   L = LEN (STR$ (M)) : PRINT "R JUST =";
70   FOR J = L to 12 : PRINT ":" ; : NEXT J
80   PRINT M;
90   L = FN MA (M)
100  FOR J = L + (L < - 1) TO 8 : PRINT "$" ; : NEXT J
120  PRINT M;
130  PRINT : RETURN
]RUN
ENTER?2.34
R JUST = ::::::::::::::2.34$$$$$$$$$2.34
ENTER?234.69
R JUST = ::::::::::::234.69$$$$$$$234.69
ENTER?.5678
R JUST = ::::::::::::.5678$$$$$$$$$$.5678
ENTER?3456789
R JUST = :::::::::3456789$$$3456789
ENTER?2123.5678
R JUST = ::::::2123.5678$$$$$$2123.5678
ENTER?0
```

Fig. 22-1. Justification.

Lines 5 and 6 are used to compute the magnitude of the number to be printed. D holds a constant value which is 2.30258509. This is the value used to compute the power to which base 10 is raised. The power is used to right justify on the decimal point in lines 90 through 130.

DEF FN MA(X) = INT(LOG(M)/D) is the function that computes the magnitude of the number.

Line 10 clears the screen. Line 20 asks the user to input a number. Line 30 is a decision statement that ends the program.

In this example, the number 2.34 will be input and the results will be discussed as it is justified.

In line 60, L = LEN(STR$(M)) converts the number into a string. This allows the string "2.34" to be evaluated. The number of characters (4) is stored in the variable L. PRINT "R JUST = " prints out the header that is printed in the first eight columns of the screen.

Line 70 computes the number of colons to be printed out before the number (M) is printed. FOR J = L TO 12 outputs 13 places (0 – 12) and subtracts off the length of the string. R JUST = occupies the first eight columns. The string contains four characters, so FOR J = 4 TO 12 will print nine colons before printing the number.

```
COLUMN 8,9 . . . . . 17. . . 21
R JUST =: : : : : : : : :2.34
```

Line 80 prints the number after the colons so all numbers are right justified. The semicolon after the M keeps the line open to print the next section that is to justify on the decimal point.

The right justification is based on adding the number of columns occupied by R JUST = , less the number of columns occupied by the string. All numbers are right justified to column 21 on the screen.

Lines 90 through 130 justify by placing the decimal in column 32 and aligning the numbers in the proper columns in relation to the decimal point.

In line 90, the power of the number is determined by the formulas in lines 5 and 6. The value of the power is stored in the variable L. The value stored in L, shown in Table 22-1, is used to determine how many places to print.

Line 10 is a loop to print dollar signs ($) from column 21 to nine places (0 – 8) (L + , the number of $ signs, = 9). The power of the number is placed in the variable L, and L TO 8, determines the number of dollar signs to be printed before the number is printed. The number input is 2.34.

```
COLUMN 8,9 . . . . . 17. . . 21. . . . . 32
R JUST =: : : : : : : : :2.34$$$$$$$$$2.34
```

In this case, 2.34 has a power of zero (see Table 22-1). FOR J = 0 TO 8 prints out nine (9) dollar signs.

Line 120 prints 2.34 immediately after the ninth dollar sign so the decimal falls in column 32.

Line 100 must handle a special case when the decimal input is less than .01 and greater than .099 (see special case, Table 22-1). This special case is handled by (L < – 1)). FOR J = L TO 8 works in all cases where "L" is greater than or equal to one-tenth (.1). When "L" is less than one-tenth (.1), a one (1) is added back to align the decimal points.

Table 22-1. L = FN MA (M)

CASE	M	FN MA (M)	
0	0 ILLEGAL VALUE		This case causes special handling by (L < −1)
1	>0 to ⩽.01	−2	
2	>.01 to ⩽1	−1	
3	>1.0 to ⩽10	0	
4	>10 to ⩽100	1	
5	>100 to ⩽1000	2	
6	>1000 to ⩽10000	3	
	1 E +8 to ⩽1 E −9		

A simpler method of controlling the column printout is to use decision statements. Using "N" as a variable to hold the value of the number, the decision statements are shown in Table 22-2.

With these four statements, the number to be placed in columns must be less than 1000. Decision statemets can cause any number to be printed in specific columns as long as the number is in the range of the computer.

Table 22-2. HTAB Using Decision Statements

	COLUMN	32	33	34	35	36	37
N = .78	IF N < 1.00 THEN HTAB 35				.	7	8
N = 1.78	IF N > .99 THEN HTAB 34			1	.	7	8
N = 11.78	IF N > = 10.00 THEN HTAB 33		1	1	.	7	8
N = 111.78	IF N > = 100 THEN HTAB 32	1	1	1	.	7	8

CONTINUE STATEMENT

The CONT (Fig. 22-2) is an immediate execution command that causes the program to continue running after it has been stopped by a STOP, END, or Control C. If the program was stopped at line 40, and a CONT command was typed, the CONT starts the program at line 100. CONTinue will not be successful in continuing the program if a program line has been modified. A ?CAN'T CONTINUE ERROR will be printed on the screen when no further instructions exist, after an error has occurred, or after a line has been changed or deleted in the existing program. If a GOTO 100 was typed, this immediate execution command causes the program to start running at line 100.

FLASH, INVERSE, and NORMAL functions are demonstrated in the FLASH SCREEN program (Fig. 22-3). (FLASH IS NOT A COMMAND IN

THE 80 COLUMN CARD ACTIVE MODE). The program combines FLASH, NORMAL, INVERSE, and RND functions to randomly print the letters of the alphabet, and changes the video mode surrounding the character.

Line 10 is the beginning of a loop that sets up the number of times the program is executed. The number 653 was selected at random. Any number could be used.

```
5   REM -PROGRAM STRUCTURE
10  SUM = 0
20  FOR X = 1 TO.5
40  SUM = SUM + X
50  NEXT X
55  PRINT X
60  PRINT "SUM = ";SUM
70  END
100 REM :TO RUN TYPE GOTO 100
110 SUM = 0
120 FOR X = 1 TO 5
130 PRINT X
140 SUM = SUM + X
145 PRINT "SUM = ";SUM
150 NEXT X
170 END
RUN
6
SUM = 15
]CONT
1
SUM = 1
2
SUM = 3
3
SUM = 6
4
SUM = 10
5
SUM = 15
]GOTO100
1
SUM = 1
2
SUM = 3
3
SUM = 6
4
SUM = 10
5
SUM = 15
```

Fig. 22-2. CONTinue and GOTO 100.

```
4    REM FLASH SCREEN
5    HOME
10   FOR J = 1 TO 653
20   I = INT (RND (1)*39) + 1
30   K = INT (RND (1.) * 23) + 1
40   L = INT (RND (1.) * 3) + 1
50   ON L GOTO 60, 70, 80
60   INVERSE : GOTO 100
70   NORMAL : GOTO 100
80   FLASH
100  HTAB I : VTAB K : PRINT CHR$ (RND (1.) * 26 + 65);
110  N = RND (1.) : NEXT J : NORMAL
120  END
```

Fig. 22-3. Random numbers program.

Line 20 randomly selects the column in which the character is to be printed. The screen has a line 40 characters long. RND returns a number from zero to less than one. RND(1) * 39 always returns a number less than 39, from zero to 38. RND(1) * 39 + 1 always returns a number greater than zero but less than 40. The "+ 1" is used to prevent the illegal value zero from being generated. The screen has 40 columns, from one to 40. If the RND function generated a zero, the program would stop and the error message ?ILLEGAL QUANTITY would be printed on the screen. It is important to remember the parameters and limits of each function.

Line 30 randomly sets the limit of the rows on the screen. The screen has 24 rows, from one to 24. RND(1.) * 23 + 1 sets the limit to 22 + 1, which prevents the screen from scrolling while the program is running. In line 20, RND(1) is used. In line 30, RND(1.) is used. Either 1 or 1. can be used without changing the function.

The program is designed so that no character is printed in column 40, nor is any character printed in row 24. A print at column 40, row 24 causes the screen to scroll.

In line 40, the positive argument of RND returns a different sequence of numbers each time. RND(1.) * 3 returns the numbers 0, 1, 2. The "+ 1" changes the sequence of numbers to 1, 2, 3.

The 1, 2, 3 generated by the program in line 40 is used by line 50 to cause the program to jump to line 60 for L = 1, line 70 for L = 2, and line 80 for L = 3.

Line 100 prints the characters and the video mode randomly according to VTAB I (line 20), and HTAB K (line 30). The PRINT CHR$(RND(1.)*25 + 65) changes the 26 values (0 to 25) + 65, from the ASCII numeric code to the alphabetic characters (Table 16-1).

A = 65
B = 66

C = 67
D = 68
etc.

In line 110, N = RND(1.) seems to serve no useful purpose. It is included because the random function that the Apple uses to generate random numbers may get into an endless loop that generates the same series continuously. N = RND(1.) prevents the random number generator from continued repetition of the same series. This knowledge comes from two sources, (1) previous programming experience with the theory of algorithms that generate random numbers, and (2) from the program not printing any new positions, but printing the same position over and over. NEXT J is the foot of the loop statement. NORMAL causes the screen to return to the normal mode (green letters on a black background), and HOME clears the screen. The program ENDs at line 120.

There is no printed RUN on this hypnotic eye blinker. You have to RUN it to see and believe.

The program in Fig. 22-4 introduces ONERR GOTO, POKE, PEEK, and RESUME.

```
10    ONERR GOTO 8000
20    PRINT "DISCO KID" : STRIKESAGAIN
30    READ D, A, B
40    DATA 1007, 34.5
50    INPUT "LETTER?" ;A
60    POKE 216, 0
70    NEXT J
7999  END
8000  Y = PEEK (222) : L = PEEK (218) + PEEK (219) * 256
8010  IF Y = 16 THEN PRINT "SYNTAX ERROR IN LINE" ;L : PRINT : GOTO 30
8020  IF Y = 42 THEN PRINT "OUT OF DATA IN LINE" ;L : PRINT : GOTO 50
8030  IF Y = 254 THEN PRINT "ANSWER THE CORRECT TYPE IN LINE" ;L : PRINT :
      RESUME
RUN
DISCO KID
SYNTAX ERROR IN LINE 20
OUT OF DATA IN LINE 30
LETTER?A          (letter E — reserved for exponentiation)
ANSWER THE CORRECT TYPE IN LINE 50
LETTER?5
NEXT WITHOUT ERROR IN LINE 70
```

Fig. 22-4. ONERR GOTO.

Line 10 is a declarative statement that tells the computer what to do when an error is detected. The computer handles errors in normal fashion until it executes an ONERR GOTO statement. The ONERR GOTO statement is similar to TRACE in that it affects the entire program during its execution.

After an ONERR GOTO statement has been executed, anytime an error is detected, the program branches to the line specified (ONERR GOTO 8000). The computer remembers line 10 for all errors. The program handles three error conditions. With more detailed programming, all seventeen possible error conditions could be placed in the program.

Line 60 places zero into memory location 216. POKE 216,0 is a statement that clears the error flag so that normal error messages may occur. When information is to be placed in a specific memory location, the POKE command is used.

In line 8000, Y = PEEK(222) returns the contents of memory location 222. The value of "Y" is stored in a variable to make it easier to use. L = PEEK(218) + PEEK(219) * 256 sets the value of the line number in the program where the error occurred.

In line 8010, the number 16 is the value for the error code ?SYNTAX ERROR.

In line 8020, the number 42 is the value that prints out OUT OF DATA error.

In line 8030, the number 245 is the "Y" value that gives a bad response to an INPUT statement.

PAUSE LOOPS

Pause loops (Fig. 22-5) cause a delay in a program to allow the user time to view the information. GET A$ (line 120) is a form of pause loop that stops the program after every ten loop executions. A simple pause loop that allows the program to continue without user participation is:

```
FOR P = 1 TO 1000 : NEXT P.
```

A nested pause loop that allows the program to continue without user participation is:

```
FOR N = 1 TO 1000
FOR P = 1 TO 100
NEXT P, N
```

A pause loop (similar to GET A$) that stops the program after a certain number of printouts (fifty in this example), and requires the user to press RETURN is:

```
FOR X = 1 TO 1000
IF X = INT(X/50) * 50 THEN INPUT Q$
NEXT X
```

HTAB AND VTAB SPACING IN LOOPS

HTAB, TAB, and VTAB tab from the #1 position of the column or row. SPC(6) leaves six spaces between the previous item and the next item.

HTAB(I*2) + 1 leaves two (2) spaces between items printed and the +1 starts in column #1. Zero is an illegal column value.

```
10  REM -PAUSE LOOPS
20  FOR A = 1 TO 1000
30  NEXT A
40 PRINT END OF 1ST PAUSE LOOP" : PRINT
50  FOR B = 1 TO 10
60  FOR C = 1 TO 100
70  NEXT C,B
80  PRINT "END OF 2ND PAUSE LOOP" : PRINT
90  FOR D = 1 TO 20
100 PRINT D
120 IF D = INT (D / 10) * 10 THEN GET A$
130 NEXT D
140 PRINT "END OF 3RD PAUSE LOOP"
150 END
RUN
END OF 1ST PAUSE LOOP

END OF 2ND PAUSE LOOP

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
END OF 3RD PAUSE LOOP
```

Fig. 22-5. Pause loops.

The use of VTAB and HTAB in loops is illustrated in the program in Fig. 22-6. Line 40 prints the numbers one (1) through five (5) beginning in column #1 of the screen, with two spaces between each number. This printout is used as a reference with which to compare the spacing in line 70. Line 70 produces similar output and spacing as line 40. Line 72 causes the first

print in column five (5) rather than in column one (1), with two spaces between each number. Line 74 produces the first print in column one (1) with five (5) spaces between each number. The program RUN demonstrates that "*5" produces the number of spaces between the items, while " + 1" determines in which column the first item is to be printed.

```
10  HOME
20  FOR X = 1 TO 5
30  FOR I = 1 TO 5
40  VTAB 10: HTAB (I - 1) * 2 + 1
50  PRINT I;
60  NEXT I: VTAB 12
70  HTAB (X - 1) * 2 + 1
72  REM :HTAB (X-1)*2+5
74  REM :HTAB (X-1)*5+1
80  PRINT X;
90  NEXT X
100 END
RUN
1  2  3  4  5
1  2  3  4  5

RUN — LINE 70
(I) 1 2 3 4 5

(X) 1 2 3 4 5

RUN — LINE 72
(I) 1 2 3 4 5

(X)    1 2 3 4 5

RUN — LINE 74
(I) 1 2 3 4 5

(X) 1   2   3   4   5
```

Fig. 22-6. HTAB spacing.

The programs in Fig. 22-7, 22-8, and 22-9 produce identical spacing results. These results are produced by three different methods, (1) decision statements, (2) loops, and (3) HTAB formula.

Applesoft suppresses leading and trailing zeros. Suppressing trailing zeros leaves a blank column where the trailing zero is supposed to be. It leaves the same feeling as reading a suspense story without knowing how it ended, you know, an empty feeling in the pit of your stomach.

The final program in Lesson 22 (Fig. 22-10) was initially written to print out a zero in the position where the zero had been suppressed. As it turned out, the program not only demonstrated printing the zero in the trailing

position, but it also reenforced HTABing by decision statements. Fig. 22-10, and Tables 22-3 and 22-4 are used to explain the zero printing. The important line in the program for overcoming zero suppression and printing the trailing zero is line 110.

```
110 IF (INT(A*100 + .5) - INT(A*10 + .5)*10 = 0 THEN PRINT "0";
```

Case No. 6, Table 22-3.

Without	line 100 the output is	8171.3
With	line 110 the output is	8171.30

The key to line 100 is to subtract the integer A from itself in two different ways, (1) (INT(A*100 + .5)), and (2) (INT(A*10 + .5)*10). If the result of these subtractions equals zero (0), then a zero (0) is printed in the last column of the output.

```
10  FOR X = 1 TO 3
20  FOR Y = 1 TO 5
30  PRINT Y;" ";
40  IF X > 1 THEN PRINT " ";
50  IF X > 2 THEN PRINT " ";
60  NEXT Y: PRINT : PRINT
70  NEXT X
80  END
RUN
1 2 3 4 5

1  2  3  4  5

1    2    3    4    5
```

Fig. 22-7. Decision statement spacing.

```
10  FOR X = 1 TO 3
20  FOR Y = 1 TO 5
30  PRINT Y;
40  FOR M = 1 TO X
50  PRINT " ";
60  NEXT M
70  NEXT Y
80  PRINT : PRINT
90  NEXT X
100 END
RUN
1 2 3 4 5

1  2  3  4  5

1    2    3    4    5
```

Fig. 22-8. Loop spacing.

```
10  FOR X = 1 TO 3
20  FOR Y = 1 TO 5
30  HTAB (X + 1) * Y - X
40  PRINT Y;
50  NEXT Y
60  PRINT : PRINT
70  NEXT X
80  END
RUN
1 2 3 4 5

1  2  3  4  5

1    2    3    4    5
```

Fig. 22-9. Loop and HTAB spacing.

Line 10 sets up a formula to round the printout to two places. The rudiments of print field definition involve printing a suppressed trailing zero so the printout looks normal. The technique of printing the trailing zero reenforces the print rules of Lesson 4. To print the zero and not disrupt the printout format, the print rules must be diligently applied. The value held in A must be printed and the line left open for the possibility of printing a zero. If the zero is printed, the line must be closed. The option to close the line after "A" is printed must be valid.

```
10  DEF FN A(X) = INT (X * 100 + .5) / 100
20  PI = 3.1416
30  FOR R = 1 TO 100 STEP 10
40  A = PI * R^2
50  IF A <= 10 THEN HTAB 20
60  IF A > 10 THEN HTAB 19
70  IF A > 100 THEN HTAB 18
80  IF A > 1000 THEN HTAB 17
90  IF A > 10000 THEN HTAB 16
100 PRINT FN A(A);
110 IF ( INT (A * 100 + .5) - INT (A * 10 + .5) * 10) = 0 THEN PRINT "0";
120 PRINT : NEXT R
130 END
RUN
     3.14
   380.13
  1385.45
  3019.08
  5281.03
  8171.30
 11689.89
 15836.81
 20612.04
 26015.59
```

Fig. 22-10. Zero printing.

Table 22-3. Zero Printing Right Justify by Decision Statements

CASE	COLUMN 20
1	3.14
2	380.13
3	1385.45
4	3019.08
5	5281.03
6	8171.30***
7	11689.89
8	15836.81
9	20612.04
10	26015.59
***ZERO PRINTING	

Table 22-4. IF (INT (A*100 + .5) – INT (A*10 + .5) * 10) = 0 THEN PRINT "0"

CASE	INT (A*100 + .5)	INT (A*10 + .5)*10	COLUMN 20
1	314	310	3.14
2	38013	38010	380.13
3	138545	138540	1385.45
4	301908	301910	3019.08
5	528103	528100	5281.03
6	817130	817130	8171.30***
7	1168989	1168990	11689.89
***ZERO PRINTING			

Case No. 1 — A printed, no zero printed, line closed.
Case No. 2 — A printed, zero printed, line closed.

The simplest way to handle both cases is as follows.

1. PRINT FN A(A); — semicolon leaves the line open.
2. THEN PRINT "0"; — semicolon leaves the line open.
3. PRINT — PRINT on a separate line number after the PRINT "0";. This satisfies Case No. 1 — the number is printed, no zero, the PRINT "0"; is false, and a default to the PRINT occurs.

   Case No. 2 — the number is printed, (zero suppressed), THEN PRINT "0";, prints the zero, and a default to the PRINT statement closes the line.

Lines 50 through 90 demonstrate how decision statements and HTAB's can right justify (Table 22-3).

LESSON 23

Approaching the Problem

Programming is the process by which a set of instructions is produced for the computer to make it solve a specific problem.

Before programming, there is preprogramming. Preprogramming is the ability to understand the problem and to be able to successfully work the problem. To be able to program a problem, the programmer must be able to understand and develop each step of the problem. The variables assigned to each formula must be understood. The steps in computing the solution must be understood. The program must produce the correct solution to the problem. If the solution is incorrect, the programmer must determine why the solution is incorrect and rectify the problem.

If you are an accountant, you must be able to solve the problem using a pencil and paper (or mentally) before you can program it. This is a most important fact in programming. You must be very adept at solving the problem because you must explain the complete process to the computer. If you cannot solve the problem using pencil and paper, do not attempt to write a program for the computer to solve it.

Think of programming this way. An understudy machine is taught to do something for you. The understudy is electronic, not human. This electronic understudy does not understand the English language, so it must be instructed in its own language. This electronic understudy, the computer, does not understand what it is doing. It is processing so fast that it does not have time to care what the results are. The speed at which the computer does tedious, repetitious, complex tasks is one of its great advantages. Unless there is a hardware malfunction, the results of running a program are accurate as to input and according to the programming instructions.

To make use of the advantages, the disadvantages must be overcome. The advantages are speed and accuracy. The disadvantages are, procedures used in the solutions must be completely specified, and conversion of the spoken language to the language of the computer must be performed. This aligns the advantages of speed and accuracy versus the disadvantages of procedure and language.

In Lesson 15, the Name and Address program was written to cause the computer to look through a string of characters in order to recognize special markers, called delimiters. In that case, semicolons were used as delimiters. The program also checked for six input errors.

To begin the learning process, write down this string of characters on a piece of paper.

RESIDENT;STREET;CITY STATE ZIP CODE

Now separate the string of characters into different fields when a semicolon is encountered.

RESIDENT;
STREET;
CITY STATE ZIP CODE

What did that accomplish? It accomplished the need to think of the minute steps involved in breaking down data so it can be converted into detailed one step instructions that the computer can process.

The list of characters must have a starting point. The individual may use a finger to point at the first character in the string, or he or she may put a pencil mark on the first character. The first character must be marked as the place to begin, so the field between the first character and the first delimiter can be determined. Each character in the field is checked to see if it is a delimiter. This character checking continues until the first delimiter is discovered. The first field is then separated from the remaining fields by writing down the first character in the field, and each succeeding character in the field is written down until the delimiter is encountered. This process is repeated until the string of characters is separated into three fields.

The programmer must tell the computer each individual step to separate the string of characters. The computer must be told to mark the first character in the string. The computer must be told to look at each character in the string of characters, checking to see if it had encountered a semicolon. If a semicolon is not encountered, the computer must be told what action to take. If a semicolon is encountered, the computer must be told what action to take. The computer must determine when the end of the first field is reached. The first field starts with a special condition, the first character in the string. All other fields start with a semicolon. The last field ends with a special condition, the last character in the string (L = LEN(A$)). All other fields end with a semicolon. These are special starting and ending conditions.

These points must be understood in relation to what is necessary to solve the programming problem. The programmer must thoroughly understand all facets of the problem and be able to solve it before the problem can be detailed in computer language. The problem is now approached from the computer's side of the program.

The line of characters is placed in a string variable. The computer marks the first character in the string as 1, the second character as 2, the third character as 3, etc., until it reaches the end of the string. This is a logical assignment of the character position, as a reference of the programmer, and the computer uses a numeric variable to hold the position value. When a loop is used that will increment the numeric variable, one by one, each individual character will be compared to the delimiter. An IF statement is used to test the individual character with the delimiter, to see if the delimiter has been reached. The LEN function (L = LEN(A$)) is used to store the length of the string variable, and the LEN function stores the number of characters in the string that are to be examined. Three characters have been discussed, (1) the first or beginning character, (2) the semicolon between the fields, and (3) the ending delimiter.

There are three reasons why the semicolon is used as a delimiter.

1. In Applesoft, a comma is used as a special separator in INPUT, READ, GET, and DATA statements. A comma incorrectly placed causes an ?EXTRA IGNORED. A colon is a special separator used to place multiple statements at a single line number. An incorrectly placed colon causes an ?EXTRA IGNORED. These are limitations of the language. All languages have some types of limitations.
2. Few addresses display a semicolon as part of the basic information. A pound sign (#) could have been used except apartment numbers are usually designated by a pound sign. Diligent research of the problem will eliminate programming difficulties. Perhaps there are address formats that use a semicolon, but it is not frequently seen in address formats. One type of research is to write down many varied examples, and chart which examples are most used and which examples are least used.
3. The semicolon is easy for the input operator to produce. It is on the home keys and does not require a shift. A division sign (/) could be used as a delimiter but it is not as easy to produce as a semicolon.

Operator convenience, ease of production, and language compatibility are three points that comprise the major network of logic used to write a program for the computer. To extend a program to the complex task of error checking (similar to the program in Lesson 15), the complete string of information must be checked and tested to insure the correct format before the lines are printed. For each line of output to be correctly printed, the numeric value of the beginning and end of each field and the end of each field must be stored. An efficient way to mark the beginning and end of each field is to store the numeric value in the position with a semicolon delimiter.

Error detecting routines look for three types of errors.

1. Errors that do not "make sense" for purposes of processing data.
2. Errors that cause the program to stop running.
3. Errors that create undesired output.

Errors that do not "make sense" for the purpose of processing data involve the length of the field. Fields of zero length must be checked because the print formula would give an ILLEGAL VALUE error. This can be tested by removing the program lines that check the values of one 1 to DIC, D1C to D2C, and D2C to L.

Errors that cause the program to stop running are those lines with less than two delimiters. While this option is not used in the program under discussion, it could be used as a method to stop the execution of the program.

Errors that create undesired output include those of more than two semicolon delimiters, and the improper use of the LEN function. These errors would indicate that there are more than two delimiters in the string.

A great deal of discussion has centered on the NAME AND ADDRESS program because it has many features that make it a good learning tool.

As an exercise for logic development, we will discuss a situation that happens from time to time. While driving home from work late at night, a thumping sound is heard and the car steering pulls unexpectedly. After the possibility of a flat tire flashes through the driver's mind, the car is stopped for visual inspection. The possible actions are shown in Table 23-1.

Table 23-1. Flat Tire

SITUATION	CATEGORY	ACTION
1. No tire flat	I	1. Continue on the trip
2. One flat tire — the spare is usable	II	2. Exchange the flat and spare
3. One flat tire — the spare is unusable	III	3. Walk for assistance
4. More than one tire flat	IV	4. Walk for assistance

While other options could be cited, the four possible actions from different situations will be discussed. The actions are reached by using common sense and understanding of a given situation.

When the situation is examined, the action will be taken according to NO FLAT TIRES, or THE NUMBER OF FLAT TIRES as shown in Table 23-1. If after observing all tires, and the flat count is zero (0), then action taken is placed in Category I. If the flat count is one (1) and the spare tire is usable, then the action is placed in Category II. If the flat count is one (1) and the spare tire is unusable, then the action taken is placed in Category III. If the flat count is greater than one (1), then the action taken is placed in Category IV.

The overall reaction to the thumping sound and the car pull is included in the general framework.

1. Stop the car.
2. Shut off the engine.
3. Open the door. Exit the car. Close the door.
4. Initialize the flat count to zero.
5. If the left front tire is flat, increment the flat count.
6. If the left rear tire is flat, increment the flat count.
7. If the right rear tire is flat, increment the flat count.
8. If the right front tire is flat, increment the flat count.
9. If the flat count is zero, then continue the trip home.
10. If the flat count is more than one, then call for assistance.
11. Open the trunk.
12. If the spare is unusable then go to step #20.
13. Exchange the flat tire and the spare.
14. Close the trunk.
15. Open the door on the driver's side of the car.
16. Get in the car. Close the door.
17. Start the engine.
18. Continue on the journey home.
19. End the actions.
20. Walk to a phone and call for assistance.

These are decisions and actions involved in the thought process. Most humans do these actions naturally, but they must be completely detailed to the computer.

The flowchart (Fig. 23-1) initializes the flat count to zero (0). An inspection of the tires is conducted, and the flat count is recorded. Since the flat count starts at zero (0) (through initialization) and there are three situations to check, only two decision statements are needed. This is similar to cutting a log into three pieces — only two cuts are needed. In the first flat count decision statement, there are two exit paths. If the flat count is zero (0) and the statement is true, the decision is made to continue the trip home. If the flat count is not zero (0) and the decision is false, the exit path is to the next decision statement. Is the flat count greater than one (1)? This decision statement selects the path to follow if the flat count is one (1), or if the flat count is greater than one (1). If the flat count is greater than one (1) the statement is true, and the exit path flows to the walk for assistance action. If the flat count is one (1), then the statement is false, and the exit path flows to the open the trunk action, and to another decision statement. Is the spare tire usable? The flowchart details the conclusion, either walk for assistance or continue the journey homeward.

Flowcharts can be detailed or general. Fig. 23-1 has both detailed steps

(STOP THE CAR — SHUT OFF ENGINE) and general steps (EXCHANGE FLAT AND THE SPARE). A flowchart can be on many different levels. It can be a long complicated written tool to help the programmer keep the action in the proper sequence. The flowchart can be so simple the programmer doesn't have to write it down. The flowchart can help to clarify a complex point. It can be a step by step set of instructions, complete with line numbers, that will be followed exactly when the program is typed and stored in memory. Since the FLAT TIRE program is not a programmable problem, the flowchart is used to keep the action in program context.

The ability to break the actions into minute, detailed steps is the essence of programming. Breaking the action into small steps helps develop the ability to process data in the same manner as the computer.

To verify that the flowchart works properly, a table of possibilities is constructed (Table 23-2). The table of possibilities follows the flowchart logic to determine that the problem is solved correctly.

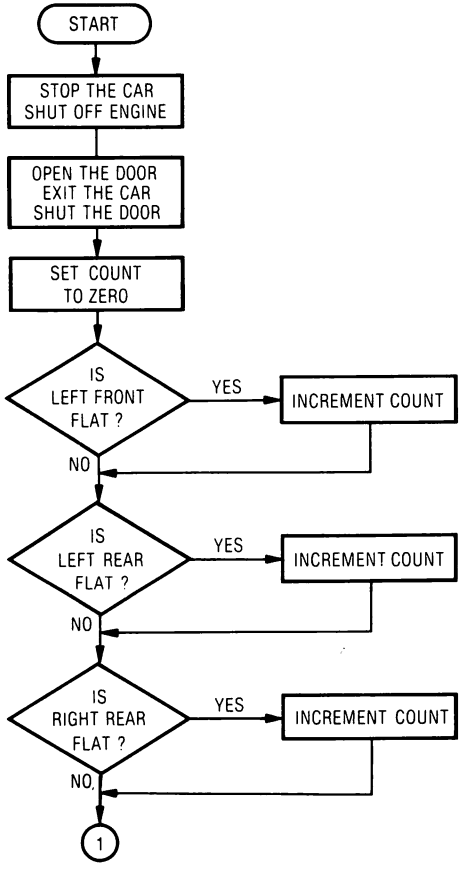


Fig. 23-1. Flat tire flowchart.

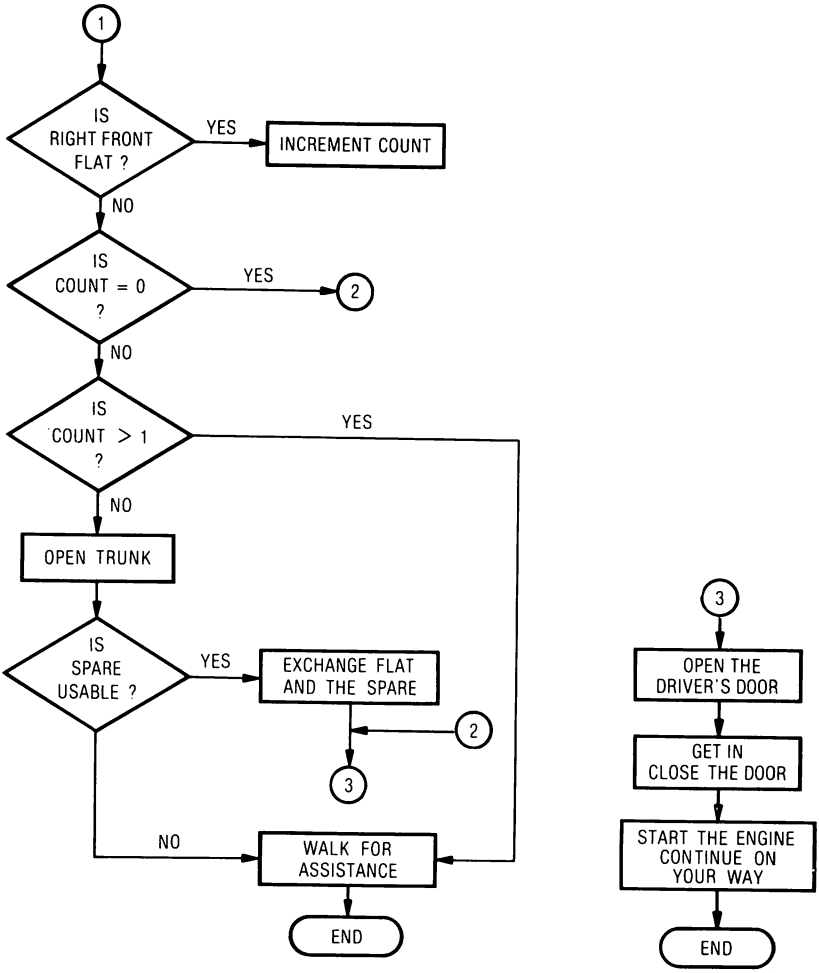


Fig. 23-1 — cont. Flat tire flowchart.

Over half of the thirty-two possibilities are listed in Table 23-2. The table shows the status of four tires, the spare and the course of action to be taken. Table 23-2 was produced by writing down different combinations of tire status and determining the best possible action to take. Common sense was used to confirm the algorithm and produce the table. If the "YES" or "NO" decisions on the flowchart had been switched, the flowchart would give incorrect results even though the logic was correct. For correct results to be produced from the written program, logic, flowcharts, and program coding must be correct.

Table 23-2. Flat Tire Possibilities

LEFT FRONT	LEFT REAR	RIGHT REAR	RIGHT FRONT	SPARE	ACTION TO BE TAKEN
O	O	O	O	O	C
O	O	O	O	F	C
O	O	O	F	O	E & C
O	O	O	F	F	W
O	O	F	O	O	E & C
O	O	F	O	F	W
O	O	F	F	O	W
O	O	F	F	F	W
O	F	O	O	O	E & C
O	F	O	O	O	W
O	F	O	F	O	W
O	F	O	F	F	W
O	F	F	O	O	W
O	F	F	O	F	W
O	F	F	F	O	W
O	F	F	F	F	W
F	O	O	O	O	E & C
F	O	O	O	F	W
F	O	O	F	O	W
F	O	O	F	F	W

TIRE STATUS (FLAT — F, O — OKAY)
CONTINUE — C, WALK FOR ASSISTANCE — W
EXCHANGE FLAT AND SPARE — E

The first two lines of Table 23-2 show the four tires, the spare, and the action to be taken. Both lines show all four tires are okay, but the first line shows the spare okay, while the second line shows the spare is flat. According to logic, if all four tires are okay, the spare does not need to be checked.

There is at least one drawback to producing a table of possibilities to check a logic flowchart. On a complex problem, the table of possibilities may have so many entries, it is unusable. In that case, five or ten comprehensive sample situations are used to try to catch all possible errors.

To sum up, there are three basic steps in programming.

1. The programmer must be able to completely describe the problem to be set into basic programming instructions.
2. The programmer must be able to outline the logical progression from one step to another, especially where decisions need to separate actions into different sections.
3. The programmer must be able to change instructions from English into equivalent computer language by understanding what the computer can process.

LESSON 24

Program Flexibility

The only thing permanent in life is change. This also applies to programs and programming. The banking industry is highly regulated by the government. Although the regulations are rigid, the bank computer programs change constantly. The bank's needs and equipment are constantly changed and updated. Customer's relations with the bank and customer's situations change constantly. The government regulations constantly change so the bank must revise and check their programs to maintain compliance. This beehive of activity affects the bank's programmers who must constantly revise and rewrite programs. The programmer must constantly upgrade his or her education to adapt to the new methods and equipment.

In programming, flexibility is a key word. Is the program flexible? Can the program be easily and quickly changed to accommodate a new situation, a new set of government rules, or a new output format, and produce the correct results? Using a programming team, a long complex program may take a year to write. This program should be flexible enough so minor changes in regulations do not make the program completely obsolete.

If a mailing list has a three line input, can a fourth line be easily inserted into the program and produce a four line output?

In an inventory program, can data be entered both in the alpha and numeric modes?

In an accounts receivable program, can customer information be easily changed without having to rewrite the program and without having to rewrite the whole business package of programs?

The programs used for this example of flexibility are the computation of federal income tax and net income. The user enters the adjusted gross income and the program computes the tax due and the net income. The tax table was taken from the 1979 Tax Rate Schedule for Married Taxpayers Filing Joint Returns and Qualifying Widows and Widowers. For simplicity, only one tax table was used in the program. In reality, the tax rate schedule has four separate tables, (1) for single taxpayers, (2) married filing jointly, (3) married filing separate returns, and (4) heads of households, etc. The

point is that an accountant filing income tax returns for the general public needs all four tables. A flexible program could easily be changed to accept the revised tables, while an inflexible program could not accept the new tables easily.

The inflexible program (Fig. 24-1) is written with IF statements. In this program, it would be difficult to change one table, much less four tables. The flexible program (Fig. 24-2) is written with the tax table in DATA statements. The flexible program would be relatively easy to change, or add to by this simple routine.

```
INPUT "ENTER ADJUSTED GROSS INCOME ";AGI
MENU
1. SINGLE TAX PAYER
2. MARRIED FILING JOINT RETURN
3. MARRIED FILING SEPARATE RETURN
4. HEAD OF THE HOUSEHOLD
INPUT "ENTER STATUS # ";STATUS
ON STATUS GOSUB 2000, 3000, 4000, 5000
```

```
1000 HOME : VTAB 3
1010 INPUT "ENTER ADJUSTED GROSS INCOME ";AGI
1020 IF AGI < 0 THEN END
1030 RESTORE
1040 BF = 0
2000 IF AGI > 3400 THEN 2020
2010 BT = 0:TB = 0:BF = 0: GOTO 7010
2020 IF AGI > 5500 THEN 2040
2030 BT = 0:TB = .14:BF = 3400: GOTO 7010
2040 IF AGI > 7600 THEN 2060
2050 BT = 294:TB = .16:BF = 5500: GOTO 7010
2060 IF AGI > 11900 THEN 2080
2070 BT = 630:TB = .18:BF = 7600: GOTO 7010
2080 IF AGI > 16000 THEN 2100
2090 BT = 1404:TB = .21:BF = 11900: GOTO 7010
2100 IF AGI > 20200 THEN 2120
2110 BT = 2265:TB = .24:BF = 16000: GOTO 7010
2120 IF AGI > 24600 THEN 2140
2130 BT = 3273:TB = .28:BF = 20200: GOTO 7010
2140 IF AGI > 29900 THEN 2160
2150 BT = 4505:TB = .32:BF = 24600: GOTO 7010
2160 IF AGI > 35200 THEN 2180
2170 BT = 6201:TB = .37:BF = 29900: GOTO 7010
2180 IF AGI > 45800 THEN 2200
2190 BT = 8162:TB = .43:BF = 35200: GOTO 7010
2200 IF AGI > 60000 THEN 2220
2210 BT = 12720:TB = .49:BF = 45800: GOTO 7010
2220 IF AGI > 85600 THEN 2240
2230 BT = 19678:TB = .54:BF = 60000: GOTO 7010
```

Fig. 24-1. Inflexible tax program.

```
2240 IF AGI > 109400 THEN 2260
2250 BT = 33502:TB = .59:BF = 85600: GOTO 7010
2260 IF AGI > 162400 THEN 2280
2270 BT = 47544:TB = .64:BF = 109400: GOTO 7010
2280 IF AGI > 215400 THEN 2300
2290 BT = 81464:TB = .68:BF = 162400: GOTO 7010
2300 BT = 117504:TB = .7:BF = 215400
7010 IT = BT + (AGI - BF) * TB: PRINT : PRINT "YOUR INCOME TAX IS ";IT
7020 PRINT : PRINT "YOUR NET IS ";AGI - IT
7030 PRINT : GOTO 1010
RUN
ENTER ADJUSTED GROSS INCOME  20000
YOUR INCOME TAX IS   3225
YOUR NET IS   16775
ENTER ADJUSTED GROSS INCOME   30000
YOUR INCOME TAX IS   6238
YOUR NET IS   23762
ENTER ADJUSTED GROSS INCOME   40000
YOUR INCOME TAX IS   10226
YOUR NET IS   29774
ENTER ADJUSTED GROSS INCOME   220000
YOUR INCOME TAX IS   120724
YOUR NET IS   99276
ENTER ADJUSTED GROSS INCOME   0
YOUR INCOME TAX IS   0
YOUR NET IS   0
ENTER ADJUSTED GROSS INCOME   -1
```

Fig.24-1–cont. Inflexible tax program.

At 2000, 3000, 4000, and 5000 the tables could be placed in DATA statements similar to the flexible tax computation program, Fig. 24-2.

The following variables are used in both programs in Figs. 24-1 and 24-2.

AGI	ADJUSTED GROSS INCOME
UL	UPPER LIMIT OF THE TAX RANGE
BF	BASE FIGURE FROM WHICH THE TAX IS COMPUTED. IF THE BASE FIGURE IS 24,600, THE TAX BASE IS 3272 PLUS 28% OF EVERYTHING OVER 24,600
BT	BASE TAX IS THE SECOND NUMBER IN THE TABLE
TB	TAX BRACKET IS THE THIRD NUMBER IN THE TABLE
IT	INCOME TAX
AGI – IT	NET INCOME IS THE ADJUSTED GROSS INCOME LESS THE INCOME TAX

```
1000 HOME : VTAB 3
1010 INPUT "ENTER ADJUSTED GROSS INCOME    ";AGI
1020 IF AGI < 0 THEN END
1030 RESTORE
1040 BF = 0
1050 READ UL,BT,TB
1060 IF UL = 0 THEN 1080
1070 IF AGI > UL THEN BF = UL: GOTO 1050
1080 IT = BT + (AGI - BF) * TB: PRINT : PRINT "YOUR INCOME TAX IS ";IT
1090 PRINT : PRINT "YOUR NET IS ";AGI - IT
1100  PRINT : GOTO 1010
7100 DATA 3400,0,0
7110 DATA 5500,0,.14
7120 DATA 7600,294,.16
7130 DATA 11900,630,.18
7140 DATA 16000,1404,.21
7150 DATA 20200,2265,.24
7160 DATA 24600,3275,.28
7170 DATA 29900,4505,.32
7180 DATA 35200,6201,.37
7190 DATA 45800,8162,.43
7200 DATA 60000,12720,.49
7210 DATA 85600,19678,.54
7220 DATA 109400,33502,.59
7230 DATA 162400,47544,.64
7240 DATA 215400,81446,.68
7250 DATA 0,117504,.70
RUN
ENTER ADJUSTED GROSS INCOME    20000
YOUR INCOME TAX IS    3225
YOUR NET IS    16775
ENTER ADJUSTED GROSS INCOME    30000
YOUR INCOME TAX IS    6238
YOUR NET IS    23762
ENTER ADJUSTED GROSS INCOME    40000
YOUR INCOME TAX IS    10226
YOUR NET IS    29774
ENTER ADJUSTED GROSS INCOME    220000
YOUR INCOME TAX IS    120724
YOUR NET IS    99276
ENTER ADJUSTED GROSS INCOME    0
YOUR INCOME TAX IS    0
YOUR NET IS    0
ENTER ADJUSTED GROSS INCOME  -1
```

Fig. 24-2. Flexible tax program.

RESTORE has been previously discussed. RESTORE resets the pointer so the data can be reused. RESTORE–READ–DATA allows the data tables to be reused when the program is in constant use. Without the RESTORE, the program would have to be RUN again (started over) if the data were to be reused.

After the adjusted gross income (AGI) is entered, it is checked to see if it is less than zero (0). If the adjusted gross income is less than zero (0), the program ends. The processing starts at the lowest range of the base figure value. If the adjusted gross income is greater than the upper limit value in the first range, the adjusted gross income is tested against the next higher upper limit value. The processing continues until the adjusted gross income is less than the upper limit value in the range. The correct range is found when the adjusted gross income is equal to or greater than the base figure, but less than the upper limit value. The correct range then sets the base tax, tax bracket, and base figure for this range. Both the inflexible and the flexible programs process the ranges in approximately the same manner.

The flexible tax program in Fig. 24-2 gets the base figure from the upper limit value of the previous range. If the adjusted gross income is in the first range, zero (0) to 3400, the base figure is zero (line 1040). In line 1040 BF = 0, the base figure was initialized to zero. When the processing begins in the first range (0–3400), the base figure has been initialized to zero (0).

If the adjusted gross income is greater than $215,400, the upper limit value of the next range is zero (0). The zero (0) indicates there is no upper limit to this range. The upper limit value is zero, so that range applies to any amount greater than $215,400.

Line 1060 (IF UL = 0 THEN 1080) tests the upper limit value for zero. If line 1060 is true, the program branches to line 1080 (IT = BT + (AGI – BF) * TB : PRINT : PRINT "YOUR INCOME TAX IS ";IT) to compute the tax. If the upper limit value is zero (0), the adjusted gross income is greater than $215,400. This sets the base figure as $215,400.

LESSON 25

Circular Lists, Stacks, and Pointers

A circular list is a list from which all insertions are made at one end and all retrievals are made at the other end (Fig. 25-1). This type of list has several names: circular buffer, queue, and FIFO (first in-first out). The pro-

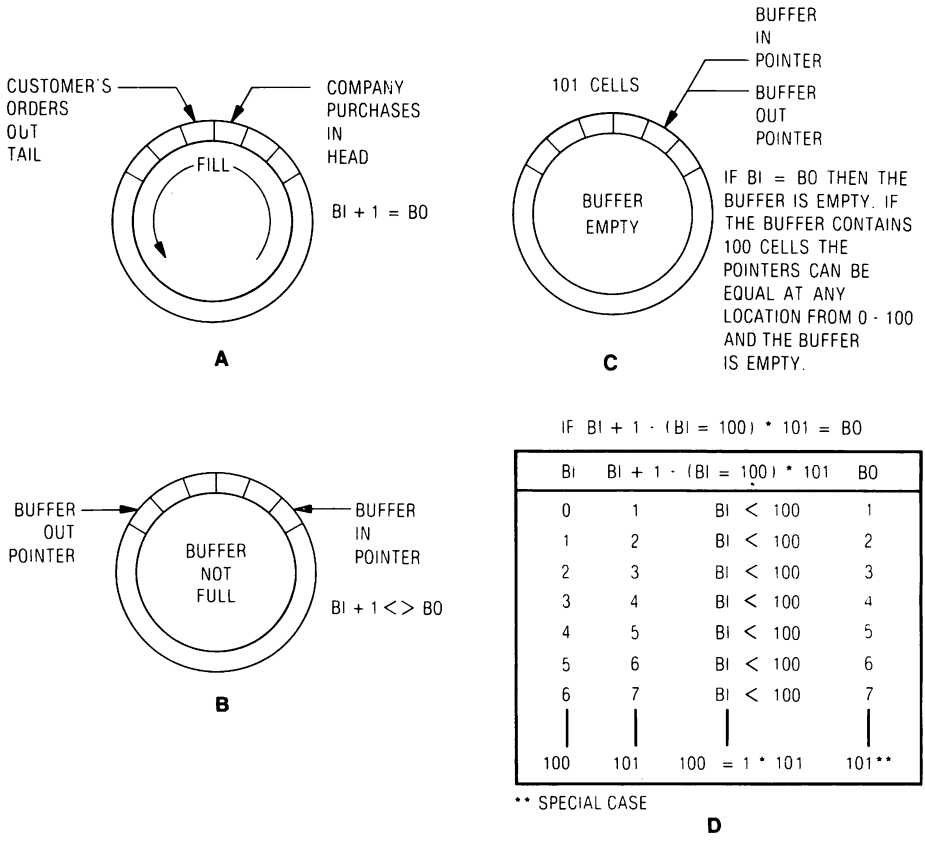


Fig. 25-1. Circular list (FIFO).

gram (Fig. 25-2) written for this lesson uses FIFO in two ways, computer lists and inventory.

A stack is a linear list from which all insertions and all retrievals are made from the top. LIFO (last in-first out) is synonymous with stack (Fig. 25-3). The program (Fig. 25-2) written for this lesson uses LIFO in two ways, computer lists and inventory.

```
5     REM : CIRCULAR LISTS, STACKS, AND POINTERS
10    DIM A$(100),PR(100),AO(100)
20    HOME : VTAB 5: HTAB 12: PRINT ''FIFO/LIFO DEMONSTRATION''
30    VTAB 8: PRINT SPC( 12);''0.END''
40    VTAB 10: PRINT SPC( 12);''1.FIFO''
50    VTAB 12: PRINT SPC( 12);''2.LIFO''
60    VTAB 14: INPUT ''ENTER SELECTION ?'';S
70    IF S = 0 THEN HOME : PRINT ''THAT'S ALL'': END
80    ON S GOSUB 300,600
90    GOTO 20
300   BI = 0:BO = 0:T = 0
310   HOME : VTAB 5: HTAB 12: PRINT ''FIFO ENTRY SYSTEM''
320   VTAB 8: PRINT SPC( 12);''0.ENDING REPORT''
330   VTAB 10: PRINT SPC( 12);''1.ENTER PURCHASE''
340   VTAB 12: PRINT SPC( 12);''2.ENTER ORDER''
350   VTAB 14: INPUT ''ENTER SELECTION ?'';S
360   IF S > 0 THEN 400
370   HOME : VTAB 5: HTAB 12: PRINT ''FIFO ENDING REPORT''
380   VTAB 14: GOSUB 1020: GOSUB 1010: RETURN
400   ON S GOTO 420,500
410   GOTO 310
420   HOME : VTAB 5: HTAB 12: PRINT ''FIFO PURCHASE HANDLER''
430   IF (BI + 1 - (BI = 100) * 101) = BO THEN VTAB 15: PRINT
      ''INVENTORY IS FULL!!!!'': PRINT : PRINT ''NO PURCHASES PERMITTED
      TODAY'': GOSUB 1000: GOTO 310
440   VTAB 8: PRINT ''ENTER DATE(MM/DD/YY),PRICE,AMOUNT''
450   VTAB 10: HTAB 11: INPUT A$(BI),PR(BI),AO(BI)
460   N = T:T = AO(BI) + T: IF T < 1 THEN 490
470   IF T < N THEN AO(BI) = T
480   BI = BI + 1 - (BI + 100) * 101
490   VTAB 12: GOSUB 1020: GOSUB 1000: GOTO 310
500   HOME : IF BO = BI THEN VTAB 8: PRINT ''THERE IS NO INVENTORY IN
      STOCK'': GOSUB 1000: GOTO 310
510   VTAB 4: INPUT ''ENTER NUMBER OF ITEMS ORDERED ?'';NU: IF NU < 1
      THEN 510
515   T = T - NU
520   IF AO(BO) > NU THEN 560
530   PRINT : PRINT AO(BO);'' ITEMS AT $'';PR(BO);'' PURCHASED '';A$(BO):NU
      = NU - AO(BO):BO = BO + 1 - (BO = 100) * 101: IF NU
      = 0 THEN 570
540   IF BI = BO THEN PRINT : PRINT ''WE ARE OUT OF STOCK WITH '';NU;''
      ITEMS'': PRINT : PRINT ''LEFT ON ORDER'': GOSUB 1010: GOTO 310
```

Fig. 25-2. Inventory program.

```
550   GOTO 520
560   PRINT : PRINT NU;" ITEMS AT $";PR(BO);" PURCHASED ";A$(BO):AO(BO)
      = AO(BO) - NU
570   PRINT : GOSUB 1020: GOSUB 1000: GOTO 310
600   BI = 0:T = 0
610   HOME : VTAB 5: HTAB 12: PRINT "LIFO ENTRY SYSTEM"
620   VTAB 8: PRINT SPC( 12);"0.ENDING REPORT"
630   VTAB 10: PRINT SPC( 12);"1.ENTER PURCHASE"
640   VTAB 12: PRINT SPC( 12);"2.ENTER ORDER"
650   VTAB 14: INPUT "ENTER SELECTION ?";S
660   IF S > 0 THEN 700
670   HOME : VTAB 5: HTAB 12: PRINT "LIFO ENDING REPORT"
680   VTAB 14: GOSUB 1020: GOSUB 1010: RETURN
700   ON S GOTO 720,800
710   GOTO 610
720   HOME : VTAB 5: HTAB 12: PRINT "LIFO PURCHASE HANDLER"
730   IF BI = 100 THEN VTAB 15: PRINT "INVENTORY IS FULL!!!!": PRINT :
      PRINT "NO PURCHASES PERMITTED TODAY": GOSUB 1000: GOTO 610
740   VTAB 8: PRINT "ENTER DATE(MM/DD/YY),PRICE,AMOUNT"
750   BI = BI + 1: VTAB 10: HTAB 11: INPUT A$(BI),PR(BI),AO(BI):N = T:T
      = AO(BI) + T: IF T < 1 THEN BI = BI - 1: GOTO 770
760   IF T < N THEN AO(BI) = T
770   VTAB 12: GOSUB 1020: GOSUB 1000: GOTO 610
800   HOME : IF BI = 0 THEN VTAB 8: PRINT "THERE IS NO INVENTORY IN
      STOCK": GOSUB 1000: GOTO 610
810   VTAB 4: INPUT "ENTER NUMBER OF ITEMS ORDERED ?";NU: IF NU < 1
      THEN 810
815   T = T - NU
820   IF AO(BI) > NU THEN 860
830   PRINT : PRINT AO(BI);" ITEMS AT $";PR(BI);" PURCHASED ";A$(BI):NU =
      NU - AO(BI):BI = BI - 1: IF NU = 0 THEN 870
840   IF BI = 0 THEN PRINT : PRINT "WE ARE OUT OF STOCK WITH ";NU;"
      ITEMS": PRINT : PRINT "LEFT ON ORDER": GOSUB 1010: GOTO 610
850   GOTO 820
860   PRINT : PRINT NU;" ITEMS AT $";PR(BI);" PURCHASED ";A$(BI):AO(BI) =
      AO(BI) - NU
870   PRINT : PRINT : GOSUB 1020: GOSUB 1010: GOTO 610
1000  FOR J = 1 TO 1800: NEXT J: RETURN
1010  VTAB 20: PRINT "PRESS RETURN TO CONTINUE!!! ";: GET Q$: RETURN
1020  PRINT "THERE ARE ";T;" ITEMS IN INVENTORY": RETURN
1050  T = 0:PT = BI
1060  FOR J = PT TO 0 STEP - 1
1070  T = T + AO(J): NEXT J: PRINT "THERE ARE ";T;" ITEMS IN INVENTORY":
      RETURN
RUN       FIFO/LIFO DEMONSTRATION
            0.END
            1.FIFO
            2.LIFO
ENTER SELECTION ?1
```

Fig. 25-2–cont. Inventory program.

```
            FIFO ENTRY SYSTEM
             0.ENDING REPORT
             1.ENTER PURCHASE
             2.ENTER ORDER
ENTER SELECTION ?1
            FIFO PURCHASE HANDLER
ENTER DATE(MM/DD/YY),PRICE,AMOUNT
            ?05/17/83,23.00,66.00
THERE ARE 66 ITEMS IN INVENTORY
            FIFO ENTRY SYSTEM
             0.ENDING REPORT
             1.ENTER PURCHASE
             2.ENTER ORDER
ENTER SELECTION ?2
ENTER NUMBER OF ITEMS ORDERED ?4
4 ITEMS AT $23 PURCHASED 05/17/83
THERE ARE 62 ITEMS IN INVENTORY
            FIFO ENTRY SYSTEM
             0.ENDING REPORT
             1.ENTER PURCHASE
             2.ENTER ORDER
ENTER SELECTION ?0
            FIFO ENDING REPORT
THERE ARE 62 ITEMS IN INVENTORY
PRESS RETURN TO CONTINUE!!! FIFO/LIFO DEMONSTRATION
             0.END
             1.FIFO
             2.LIFO
ENTER SELECTION ?2
            LIFO ENTRY SYSTEM
             0.ENDING REPORT
             1.ENTER PURCHASE
             2.ENTER ORDER
ENTER SELECTION ?1
            LIFO PURCHASE HANDLER
ENTER DATE(MM/DD/YY),PRICE,AMOUNT
            ?05/17/83,23.00,66.00
THERE ARE 66 ITEMS IN INVENTORY
            LIFO ENTRY SYSTEM
             0.ENDING REPORT
             1.ENTER PURCHASE
             2.ENTER ORDER
ENTER SELECTION ?2
ENTER NUMBER OF ITEMS ORDERED ?6
6 ITEMS AT $23 PURCHASED 05/17/83
THERE ARE 60 ITEMS IN INVENTORY
PRESS RETURN TO CONTINUE!!! LIFO ENTRY SYSTEM
             0.ENDING REPORT
```

Fig. 25-2–cont. Inventory program.

```
            1.ENTER PURCHASE
            2.ENTER ORDER
ENTER SELECTION ?0
          LIFO ENDING REPORT
THERE ARE 60 ITEMS IN INVENTORY
PRESS RETURN TO CONTINUE!!! FIFO/LIFO DEMONSTRATION
            0.END
            1.FIFO
            2.LIFO
ENTER SELECTION ?0
THAT'S ALL
```

Fig. 25-2–cont. Inventory program.

A pointer, as shown in Figs. 25-1 and 25-3, is an address location used to designate the location of data contained in a cell of a linear list. A pointer is considered a pointer only if it points at some data item within the list. An address location is not considered a pointer unless it specifically points to data.

A circular list has two buffer pointers, buffer in (BI) pointer, and a buffer out (BO) pointer. A stack has only one pointer, a buffer in (BI) pointer.

The program in Fig. 25-2, written to demonstrate the circular list, stack, and pointers, accepts only one inventory item. The fields for this one inventory item contain (1) the date the item was purchased by the company, (2) the price of the item, (PR(BI)), and (3) the number of items the company purchased, (AO(BI)).

The circular list and the stack contain 101 cells (DIM A$(100, PR(100), AO(100)), into which purchase information is placed and from which customer orders are taken.

The FIFO (circular list) and the LIFO (stack) give the program flexibility. The program could be used by a company that uses either the FIFO or LIFO inventory method.

Lines 10 through 80 (Fig. 25-2) DIMension the variables and set up the main menu from which to select the topics: 0. END THE PROGRAM, 1. FIFO, or 2. LIFO.

Lines 420 through 490 process the purchasing information for the circular list (FIFO). Line 430 detects the buffer full condition, Fig. 25-1, part A.

The buffer is dimensioned to 101 cells. DIM A$(100) was arbitrarily selected and could have been a smaller number or any number within reasonable limits.

When BO and BI are pointing to adjacent cells, there are no empty cells in the circular list, and consequently, the list is full. In line 430, a special case is used when BI = 100 (Fig. 25-1, part D). The special case is used so the circular list can continue uninterrupted. The BI = 100 portion of the formula is activated when BI = 100.

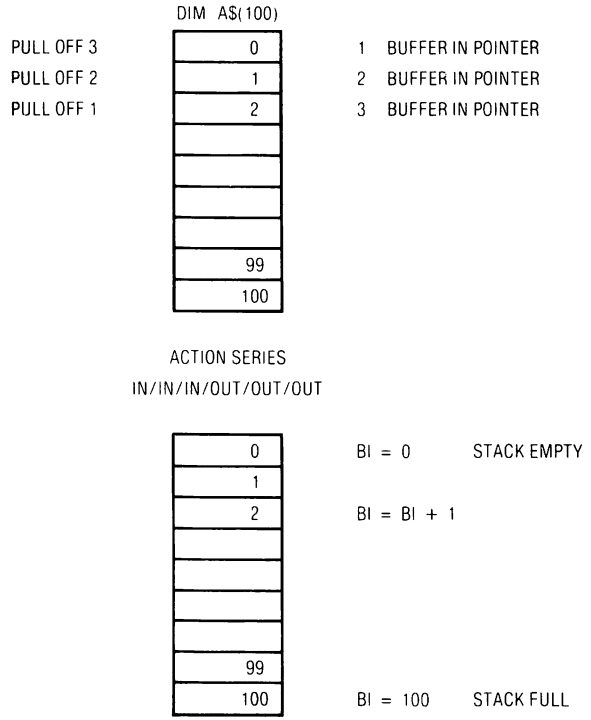


Fig. 25-3. Stack (LIFO) 101 cells.

Line 440 prints out the informational header, ENTER DATE (MM/DD/YY), PRICE, AMOUNT. Line 450 allows the user to enter the date of purchase (A$(BI)), the price of the item (PR(BI)), and the number of items purchased (AO(BI)). These items are placed within a specific cell in the circular list (Fig. 25-4).

In line 460, the total number of items purchased is placed in the variable N. N holds the total number of items for comparison purposes in relation to back orders. When N is greater than T (total items), there are not enough items in stock to fill a customer's order, and the items have to be back ordered. T = AO(BI) + T holds the total number of items purchased (Fig. 25-5).

Line 460 IF T < 1 THEN 490. When T is less than one (1), there are items back ordered and the subroutine at line 1020 prints out a negative value for the number of items in inventory.

Line 470 IF T < N THEN AO(BI) = T. If there is a back order, the next purchase may not eliminate the back order. When the next purchase does not eliminate the back order, or when the purchase equals the back order, the purchase does not go into the buffer. If the purchase is greater *than the*

back order, the excess is stored in N. The excess in N can then be compared to T (1) before the purchase order, and (2) after the purchase order. If the total after the order is less than the order, the purchase must be reduced by the number of items in the order. When the purchase is greater than the back order, the back order is subtracted from the purchase, and the balance of the purchase is stored in a cell in the buffer.

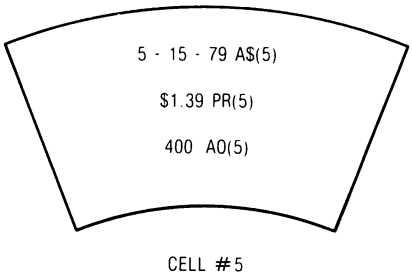


Fig. 25-4. Individual cell and contents.

Line 480 increments the buffer pointer to the next cell in the buffer, and line 490 causes the program to jump to the FIFO menu.

In line 500, when both buffer pointers rest at the same cell in the buffer (Fig. 25-1, part C) the buffer is empty. The buffer has three conditions:

1. BI = BO BUFFER EMPTY — input company purchases only.
2. BI + 1 = BO BUFFER FULL — take out customer's orders only.
3. BI + 1 < > BO company purchases can be placed in the buffer and customer's orders can be taken from the buffer.

Line 510 checks to determine if an order of less than one (1) has been entered. If an order of less than one (1) has been entered, the program returns to line 510 for a legal order value to be entered.

Line 515 subtracts the number of items ordered from the total number of items.

In line 520, if the number of items of a particular purchase in the buffer is greater than the number of items ordered, the program branches to line 560 to process the order.

A$ = date of purchase of items by company.
AO = number of items purchased by the company.
BI = buffer in pointer.
BO = buffer out pointer.
N = holds total number of items for later comparison to total items (T).
NU = number of items ordered by the customer.
PR = price of the items.
PT = temporary pointer.
T = total number of items.

Fig. 25-5. Variables for circular list, stack, and pointers.

If the statement in line 520 is false, the program defaults to line 530 to print out the number of items purchased and updates the buffer, as necessary, to complete the order. The order is processed against inventory buffers until the order is filled. If there are not sufficient items in inventory to fill the order, the inventory buffers are depleted and the balance is back ordered, making T a negative value. AO(BO) is printed each time the inventory is reduced by the order. NU = NU − AO(BO) updates the number of items left on the order that need to be filled. If NU is greater than AO(BO), the first buffer cell is emptied, and this process continues until all buffer cells are emptied, or until the order is filled. If the order is not completely filled, and there is no remaining inventory, the balance of the items is back ordered. The logic of this situation is implemented in lines 520 through 550.

The reduction in inventory is contained in the statement NU = NU − AO(BO). The buffer out cell is computed by the statement BO = BO + 1 − (BO = 100)*101. IF NU = 0 THEN 570 takes care of the condition when the number of items ordered comes out even, with BO entry on the buffer empty.

Line 560 prints out the number of items purchased, the price and the purchase date.

The statement AO(BO) = (BO) − NU in line 560 computes the number of items that remain in a specific cell in the inventory buffer.

When the transactions are completed, the FIFO menu is displayed. Zero (0) selection prints out the ending report of the number of items remaining in inventory (GOSUB 1020). GOSUB 1000 causes PRESS RETURN TO CONTINUE!!! to be printed below the ending report. When RETURN is pressed, the program returns to line 90 — GOTO 20, and the FIFO/LIFO DEMONSTRATION menu is displayed on the screen. Selection zero (0) from the FIFO/LIFO DEMONSTRATION menu causes the program to end.

Selection 2 from the FIFO/LIFO DEMONSTRATION menu causes the program to GOSUB 600 to the LIFO (stack) section of the program.

Line 600 BI = 0 : T = 0 initializes the buffer in pointer and the total to zero. The stack has only one pointer, the buffer in pointer. The LIFO ENTRY SYSTEM menu is printed: 0. ENDING REPORT, 1. ENTER PURCHASE, and 2. ENTER ORDER.

If 1. ENTER PURCHASE is selected, the program branches to line 720, to print out LIFO PURCHASE HANDLER.

LINE 730, IF BI = 100, the program prints out, INVENTORY IS FULL.

1. BI = 100 STACK IS FULL — customer orders can be filled.
2. BI = 0 STACK IS EMPTY — company purchases can be placed in the stack.
3. BI = BI + 1 company purchases may be inserted, and customer's orders may be processed.

Line 750 BI = BI + 1 increments the stack pointer, and purchasing information is entered into the array. A$(BI) is the date of purchase, PR(BI) is the price of the item, and AO(BI) is the number of items purchased.

Line 750 N = T places the total number of items in a variable to be used for comparison later in the program.

Line 760 IF T < N THEN AO(BI) = T. If the total number of items in inventory is less than the number ordered, the purchases go to eliminate the back order. If the purchases are less than, or equal to, the back order, the purchases do not go to inventory. If the purchases are greater than the back order, the excess purchases are stored in N. The excess in N is compared to T before the purchase, and T after the purchase. If T after the purchase is greater than N, the back order is eliminated and the excess purchases go into a cell in the buffer.

Line 800 HOME : IF BI = 0 THEN VTAB 8 : PRINT "THERE IS NO INVENTORY IN STOCK" : GOSUB 1000 : GOTO 610. The BI pointer is set to the top of the stack (Fig. 25-3). Cell number 100 is the bottom of the stack. The top of the stack and the bottom of the stack is a matter of semantics. The important aspect is how the stack is filled and emptied. If the number of items in inventory is greater than the number of items ordered by the customer, the program branches to line 860 to process the order and only this cell in the stack is reduced.

If line 820 is false, the program defaults to line 830 to print out the number of items purchased, subtract the number of items purchased from inventory (NU = NU – AO(BI)), decrement the stack pointer (BI = BI – 1), and go to the next cell to try to complete the order. The order is processed against inventory buffer cells until the order is filled. If there is not sufficient inventory to fill the order, all inventory buffer cells are depleted and the balance is back ordered. T is then a negative value. Line 815 T = T – NU. AO(BI) is printed out each time the inventory is reduced by the order. NU = NU – AO(BI) updates the number of items left on order that need to be filled. If NU is greater than AO(BI), the cell is emptied, and the adjacent cells are emptied, until the order is filled. If the order is not completely filled, and there is no remaining inventory, the remaining items are back ordered. The logic is implemented in lines 828 through 850.

Line 830 IF NU = 0 THEN 870 is true, the program prints an inventory status report of zero (0) items.

When all purchases and orders have been completed, the program returns to line 610 to print out the LIFO ENTRY SYSTEM. Zero (0) selection from this menu prints out an ending inventory report, and press RETURN returns the program to the FIFO/LIFO DEMONSTRATION menu. A zero (0) selection from this menu ends the program.

LESSON 26

Sorting, Searching, and Deleting

The program written for Lesson 26 was originally written for the Apple II computer when cassette tape was the primary method to save and load programs. In the old days, disks were in very limited production and not readily available to the public.

Since cassettes and cassette tapes are an out of date item, the program was rewritten so a file could be saved (and loaded) from either tape or disk.

There was discussion, between the authors, whether to rewrite the program to save and load a file to disk only. The decision was to present the program with both the tape and disk option. The logic behind this decision came from experience in commercial programming. In commercial programming, the rule is to keep the contents of the present program and add options. Fig. 26-1 is the program containing both options, load and save a file to tape and disk. Fig. 26-2 gives the list of statements used to convert the original program (load and save a file to tape) to the present program (load and save a file to tape or disk).

```
1       DIM CA(45),CH(1),DA$(1000),CL(1000) : REM : PHONE LIST
10      SP$ = "          ":SP$ = SP$ + SP$ + SP$
15      DC$ = "713"
16      D$ = CHR$ (4)
20      HOME : VTAB 4: HTAB 12: PRINT "PHONE LISTING"
30      VTAB 10: HTAB 8: PRINT "1.ENTER"
40      VTAB 12: HTAB 8: PRINT "2.MODIFY/DELETE"
50      VTAB 14: HTAB 8: PRINT "3.LIST/SEARCH"
60      VTAB 16: HTAB 8: PRINT "4.SAVE LIST AND END"
70      VTAB 18: HTAB 8: INPUT "ENTER SELECTION ?";MS
80      ON MS GOTO 1000,2000,3000,4400
90      GOTO 20
1000    HOME : VTAB 4: HTAB 12: PRINT "FILE MAINTENANCE"
1010    VTAB 10: HTAB 8: PRINT "1.LOAD OLD FILE"
1020    VTAB 12: HTAB 8: PRINT "2.ENTER NEW ITEMS"
1030    VTAB 14: HTAB 8: PRINT "3.RETURN TO MAIN MENU"
1040    VTAB 16: HTAB 8: INPUT "ENTER SELECTION ?";MS
```

Fig. 26-1. Program and run of the RAM phone list.

```
1050  ON MS GOTO 1070,1200,1400
1060  GOTO 1000
1070  PRINT : PRINT ''IS THE FILE ON (T)APE OR (D)ISK ?'': PRINT : INPUT ''EN
      TER (T) OR (D) ?'';Q$
1075  IF Q$ = ''T'' GOTO 1100
1080  IF Q$ = ''D'' GOTO 1170
1090  GOTO 1000
1100  HOME : VTAB 4: HTAB 4: INPUT ''READY CASSETTE AND PRESS RETURN
      ! '';Q$
1110  RECALL CH
1120  IF CH(0) = 0 THEN PRINT ''THERE IS NO ARRAY ON TAPE'': GOTO 1000
1130  FOR J = 1 TO CH(0)
1140  RECALL CA
1150  DA$(J) = '''': FOR K = 1 TO 44:DA$(J) = DA$(J) + CHR$ (CA(K)):
      NEXT K:CL(J) = CA(45):NEXT J
1160  GOTO 1000
1170  HOME : VTAB 4: HTAB 4: PRINT ''ENTER THE DISK FILE NAME !'': PRINT
1175  PRINT : INPUT ''FILE NAME = '';F$: IF  LEN (F$) = 0 GOTO 1000
1180  PRINT D$;''OPEN '';F$: PRINT D$;''READ '';F$: INPUT CH(0)
1185  FOR J = 1 TO CH(0): INPUT DA$(J),CL(J): NEXT J
1190  PRINT D$;''CLOSE '';F$: PRINT D$;''IN#0'': PRINT D$;''PR#0''
1195  GOTO 1000
1200  HOME :VT = 6: GOSUB 10010
1210  IF CL(CH(0) + 1) = 0 THEN 1000
1260  VT = 12: GOSUB 10080
1310  DA$(CH(0) + 1) = DA$(CH(0) + 1) +
      ''('' + TC$ + '')-'' + LEFT$(PT$,3)
      + ''-'' + RIGHT$ (PT$,4)
1320  PRINT : GOSUB 10000
1330  PRINT : INPUT ''ENTER 'R' TO REENTER ELSE 'RETURN' ?'';Q$ : IF Q$
      < > ''R'' THEN CH(0) = CH(0) + 1
1340  GOTO 1200
1400  GOSUB 1410: GOTO 20
1410  IF CH(0) < 2 THEN RETURN
1420  FOR J = 1 TO CH(0) - 1
1430  M = J: FOR K = J + 1 TO CH(0)
1440  IF LEFT$ (DA$(K),30) < LEFT$ (DA$(M),30) THEN M = K
1450  NEXT K
1460  IF M = J THEN 1480
1470  TC$ = DA$(M):DA$(M) = DA$(J):DA$(J) = TC$:CL(0)
      = CL(M):CL(M) = CL(J):CL(J) = CL(0)
1480  NEXT J
1490  RETURN
2000  HOME : VTAB 4: IF CH(0) = 0 THEN PRINT ''THERE IS NO LIST '';
      CHR$ (7): FOR J = 1 TO 2000: NEXT J: GOTO 20
2010  PRINT ''ENTER NAME TO BE CHANGED'': PRINT : INPUT NA$
2020  IF LEN (NA$) = 0 THEN 20
2030  FOR K = 1 TO CH(0)
2040  IF NA$ < > LEFT$ (DA$(K),CL(K)) THEN 2060
```

Fig.26-1–cont. Program and run of the RAM phone list.

```
2050  GOTO 2100
2060  NEXT K: VTAB 10: HTAB 6: PRINT "THIS NAME NOT ON LIST": PRINT
      CHR$ (7): FOR J = 1 TO 1000 : NEXT J: GOTO 2000
2100  CH(1) = CH(0):CH(0) = K - 1:VTAB 6: PRINT "CURRENT RECORD IS ":
      PRINT
2110  VTAB 8: GOSUB 10000: PRINT : PRINT "ENTER 'C' TO CHANGE, 'D' TO
      DELETE": PRINT : INPUT "ELSE 'RETURN' ?";Q$
2120  IF Q$ < > "C" AND Q$ < > "D" THEN 2240
2125  IF Q$ = "D" THEN DA$(K) = "DELETE" + LEFT$ (SP$,24) + "(000)-
      000-0000": GOTO 2230
2130  VTAB 12: CALL -958: VTAB 12: PRINT "ENTER 'N'-NAME, 'P'-PHONE#,
      'B'-BOTH" : PRINT
2140  T$ = RIGHT$ (DA$(K),14): INPUT "LETTER PLEASE ?";C$: IF C$ < >
      "N" AND C$ < > "P" AND C$ < > "B" THEN 2130
2150  IF C$ = "P" THEN 2170
2160  VT = 14: GOSUB 10010
2170  IF C$ = "N" THEN 2190
2180  VT = 16: GOSUB 10080
2190  IF C$ = "N" THEN DA$(K) = DA$(K) + T$: GOTO 2230
2200  IF C$ = "P" THEN DA$(K) = LEFT$(DA$(K),30)
2220  DA$(K) = DA$(K) + "(" + TC$ + ")-"
       + LEFT$ (PT$,3) + "-" + RIGHT$ (PT$,4)
2230  CH(0) = CH(1): PRINT : INPUT "ANY MORE CORRECTIONS (Y OR N)
      ?";Q$: IF Q$ = "Y" THEN 2000
2240  K = 0: FOR J = 1 TO CH(0)
2250  IF LEFT$ (DA$(J),6) = "DELETE" THEN 2280
2260  K = K + 1: IF K = J THEN 2280
2270  DA$(K) = DA$(J):CL(K) = CL(J)
2280  NEXT J
2290  CH(0) = K
2300  GOSUB 1410: GOTO 20
3000  HOME : VTAB 3: INPUT "ENTER 'S' TO SEARCH OR 'L' TO LIST ?";Q$:
      IF Q$ < > "L" AND Q$ < > "S" THEN 3000
3010  IF Q$ = "S" THEN 3100
3030  FOR J = 1 TO CH(0)
3040  IF J < > INT ((J - 1) / 5) * 5 + 1 THEN 3070
3050  IF J < > 1 THEN PRINT : INPUT "!";Q$
3060  HOME : VTAB 3
3070  PRINT "NAME = "; LEFT$ (DA$(J),30): PRINT SPC( 7);"PHONE # = ";
      RIGHT$ (DA$(J),14): PRINT
3080  NEXT J
3090  PRINT : INPUT "!";Q$: GOTO 20
3100  HOME : VTAB 3: HTAB 12: PRINT "SEARCH SELECTION": PRINT
3110  HTAB 12: PRINT "1.NAME SEARCH": PRINT : HTAB 12: PRINT "2.NUMBER
      SEARCH": PRINT : HTAB 12: PRINT "3.RETURN TO MAIN MENU": PRINT
3120  INPUT "ENTER SEARCH KEY ?"; MS
3130  ON MS GOTO 3150,3250,20
3140  GOTO 3100
3150  HOME : VTAB 4: PRINT "ENTER NAME OR FRAGMENT ?": PRINT :
      INPUT NA$:L = LEN (NA$): IF L = 0 THEN 3100
```

Fig.26-1–cont. Program and run of the RAM phone list.

```
3160  CO = 0: FOR J = 1 TO CH(0): IF L > CL(J) THEN 3220
3170  FOR K = 1 TO CL(J) - L + 1
3180  IF NA$ < > MID$ (DA$(J),K,L) THEN 3210
3190  CH(1) = CH(0):CH(0) = J - 1: PRINT : GOSUB 10000: PRINT :
      INPUT Q$
3200  CH(0) = CH(1):CO = CO + 1: GOTO 3220
3210  NEXT K
3220  NEXT J: IF CO > 0 THEN 3100
3230  PRINT : PRINT "THIS WORD IS NOT ON FILE": FOR J = 1 TO 1000:
      NEXT J: PRINT CHR$ (7): GOTO 3100
3250  HOME : VTAB 6: HTAB 6: INPUT "ENTER AREA CODE,PHONE # ?";
      AC$,PN$
3260  IF LEN (AC$) = 0 THEN AC$ = DC$
3270  TC$ = "(" + AC$ + ")-" + LEFT$(PN$,3)
      + "-" + RIGHT$ (PN$,4)
3280  FOR J = 1 TO CH(0)
3290  IF TC$ < > RIGHT$ (DA$(J),14) THEN 3310
3300  CH(1) = CH(0):CH(0) = J - 1: GOSUB 10000: PRINT :CH(0) = CH(1):
      INPUT Q$: PRINT
3310  NEXT J
3320  GOTO 3100
4000  HOME : VTAB 6: HTAB 10: INPUT "READY CASSETTE TO SAVE FILE !";Q$
4010  STORE CH
4020  FOR J = 1 TO CH(0)
4030  FOR K = 1 TO 44
4040  CA(K) = ASC (  MID$ (DA$(J),K,1))
4050  NEXT K
4060  CA(45) = CL(J)
4070  STORE CA
4080  NEXT J
4090  PRINT CHR$ (7); CHR$ (7)
4100  HTAB 10: PRINT "PHONE SYSTEM IS ENDED"
4110  END
4400  HOME : VTAB 4: HTAB 4: PRINT "SAVE THE FILE ON (T)APE OR (D)ISK ?":
4410  PRINT : INPUT "ENTER (T) OR (D) ?";Q$
4420  IF Q$ = "T" THEN 4000
4430  IF Q$ < > "D" THEN 1000
4440  PRINT : PRINT : PRINT : PRINT "ENTER THE DISK FILE NAME !": PRINT
4450  INPUT "FILE NAME = ";F$
4460  IF  LEN (F$) = 0 THEN  END
4470  PRINT D$;"OPEN ";F$
4480  PRINT D$;"WRITE ";F$
4490  PRINT CH(0)
4500  FOR J = 1 TO CH(0)
4510  PRINT DA$(J);",";CL(J): NEXT J
4520  PRINT D$;"CLOSE ";F$
4530  END
10000 PRINT "NAME = "; LEFT$ (DA$(CH(0) + 1),30): PRINT : PRINT "PHONE
      #= "; RIGHT$ (DA$(CH(0) + 1),14): RETURN
```

Fig.26-1–cont. Program and run of the RAM phone list.

```
10010  VTAB VT: CALL - 958: VTAB VT: PRINT "ENTER NAMES(LESS THAN 31
       CHARACTERS)"
10020  PRINT : INPUT DA$(CH(0) + 1)
10030  CL(CH(0) + 1) = LEN (DA$(CH(0) + 1)): IF CL(CH(0) + 1) = 0 THEN
       RETURN
10040  IF CL(CH(0) + 1) > 30 THEN PRINT : PRINT "NAME IS TOO LONG";
       CHR$ (7): FOR J = 1 TO 1000: NEXT J: GOTO 10010
10050  IF CL(CH(0) + 1) = 30 THEN RETURN
10060  DA$(CH(0) + 1) = DA$(CH(0) + 1) + LEFT$ (SP$,30 - CL(CH(0)
       + 1))
10070  RETURN
10080  VTAB VT: PRINT "ENTER AREA CODE,PHONE NO."
10090  PRINT : PRINT : INPUT AC$, PN$
10100  IF LEN (AC$) = 0 THEN AC$ = DC$
10110  IF LEN (AC$) < > 3 OR LEN (PN$) < > 7 THEN 10080
10120  TC$ = STR$ ( VAL (AC$)):PT$ = STR$ ( VAL (PN$)): IF TC$ < > AC$
       OR PT$ < > PN$ THEN PRINT : PRINT "PLEASE USE NUMERICS"; CHR$
       (7): FOR J = 1 TO 1000: NEXT J: GOTO 10080
10130  RETURN


RUN
                PHONE LISTING
          1.ENTER
          2.MODIFY/DELETE
          3.LIST/SEARCH
          4.SAVE LIST AND END
          ENTER SELECTION ?1
               FILE MAINTENANCE
          1.LOAD OLD FILE
          2.ENTER NEW ITEMS
          3.RETURN TO MAIN MENU
          ENTER SELECTION ?2
ENTER NAME(LESS THAN 31 CHARACTERS)
?JERRY HUGHES
ENTER AREA CODE,PHONE NO.
?409,8325133
NAME = JERRY HUGHES
PHONE #= (409)-832-5133
ENTER 'R' TO REENTER ELSE 'RETURN' ?
ENTER NAME(LESS THAN 31 CHARACTERS)
?MARY ABCDEF
ENTER AREA CODE,PHONE NO.
?,8014982
NAME = MARY ABCDEF
PHONE #= (713)-801-4982
```

Fig.26-1–cont. Program and run of the RAM phone list.

```
ENTER 'R' TO REENTER ELSE 'RETURN' ?
ENTER NAME(LESS THAN 31 CHARACTERS)
?CHARLES NOBLES
ENTER AREA CODE,PHONE NO.
?409,8626148
NAME = CHARLES NOBLES
PHONE #= (409)-862-6148
ENTER 'R' TO REENTER ELSE 'RETURN' ?
ENTER NAME(LESS THAN 31 CHARACTERS)
?JANET RUSSELL
ENTER AREA CODE,PHONE NO.
?409,8606623
NAME = JANET RUSSELL
PHONE #= (409)-860-6623
ENTER 'R' TO REENTER ELSE 'RETURN' ?
ENTER NAME(LESS THAN 31 CHARACTERS)
?ALICE STEWART
ENTER AREA CODE,PHONE NO.
?,8134549
NAME = ALICE STEWART
PHONE #= (713)-813-4549
ENTER 'R' TO REENTER ELSE 'RETURN' ?
ENTER NAME(LESS THAN 31 CHARACTERS)
?JOHN TALBERT
ENTER AREA CODE,PHONE NO.
?409,8326619
NAME = JOHN TALBERT
PHONE #= (409)-832-6619
ENTER 'R' TO REENTER ELSE 'RETURN' ?
ENTER NAME(LESS THAN 31 CHARACTERS)
?
                FILE MAINTENANCE
          1.LOAD OLD FILE
          2.ENTER NEW ITEMS
          3.RETURN TO MAIN MENU
          ENTER SELECTION ?3
              PHONE LISTING
          1.ENTER
          2.MODIFY/DELETE
          3.LIST/SEARCH
          4.SAVE LIST AND END
```

Fig.26-1–cont. Program and run of the RAM phone list.

```
          ENTER SELECTION ?2
ENTER NAME TO BE CHANGED
?ALICE STEWART
CURRENT RECORD IS
NAME = ALICE STEWART
PHONE #= (713)-813-4549
ENTER 'C' TO CHANGE, 'D' TO DELETE
ELSE 'RETURN' ?C
ENTER 'N'-NAME, 'P'-PHONE#, 'B'-BOTH
LETTER PLEASE ?N
ENTER NAME(LESS THAN 31 CHARACTERS)
?ALICE STEWARD
ANY MORE CORRECTIONS (Y OR N) ?Y
ENTER NAME TO BE CHANGED
?JANET RUSSELL
CURRENT RECORD IS
NAME = JANET RUSSELL
PHONE #= (409)-860-6623
ENTER 'C' TO CHANGE, 'D' TO DELETE
ELSE 'RETURN' ?D
ANY MORE CORRECTIONS (Y OR N) ?Y
ENTER NAME TO BE CHANGED
?
              FILE MAINTENANCE
          1.LOAD OLD FILE
          2.ENTER NEW ITEMS
          3.RETURN TO MAIN MENU
          ENTER SELECTION ?3
                PHONE LISTING
          1.ENTER
          2.MODIFY/DELETE
          3.LIST/SEARCH
          4.SAVE LIST AND END
          ENTER SELECTION ?3
ENTER 'S' TO SEARCH OR 'L' TO LIST ?S
          SEARCH SELECTION
          1.NAME SEARCH
          2.NUMBER SEARCH
          3.RETURN TO MAIN MENU
ENTER SEARCH KEY ?1
ENTER NAME OR FRAGMENT ?
?ABC
```

Fig.26-1–cont. Program and run of the RAM phone list.

```
NAME = MARY ABCDEF
PHONE #= (713)-801-4982
?
          SEARCH SELECTION
          1.NAME SEARCH
          2.NUMBER SEARCH
          3.RETURN TO MAIN MENU
ENTER SEARCH KEY ?3
               PHONE LISTING
          1.ENTER
          2.MODIFY/DELETE
          3.LIST/SEARCH
          4.SAVE LIST AND END
          ENTER SELECTION ?3
ENTER 'S' TO SEARCH OR 'L' TO LIST ?L
NAME = ALICE STEWARD
          PHONE # = (713)-813-4549
NAME = CHARLES NOBLES
          PHONE # = (713)-862-6148
NAME = JERRY HUGHES
          PHONE # = (409)-832-5133
NAME = JOHN TALBERT
          PHONE # = (409)-832-6619
NAME = MARY ABCDEF
          PHONE # = (713)-801-4982

               PHONE LISTING
     1.ENTER
     2.MODIFY/DELETE
     3.LIST/SEARCH
     4.SAVE LIST AND END
     ENTER SELECTION ?4
  SAVE THE FILE ON (T)APE OR (D)ISK ?
ENTER (T) OR (D) ?D
ENTER THE DISK FILE NAME !
FILE NAME = RAM PHONE TEST
```

Fig.26-1–cont. Program and run of the RAM phone list.

Following is a list of the variables as they appear in the program.

CA — CA array is used to store and retrieve information from tape. The Applesoft language cannot store string arrays directly. The string arrays are converted to the

```
16   D$ = CHR$ (4)
80   ON MS GOTO 1000,2000,3000,4400
1010 VTAB 10: HTAB 8: PRINT "1.LOAD OLD FILE"
1050 ON MS GOTO 1070,1200,1400
1070 PRINT : PRINT "IS THE FILE ON (T)APE OR (D)ISK ?": PRINT : INPUT
     "ENTER (T) OR (D) ?" ;Q$
1075 IF Q$ = "T" GOTO 1100
1080 IF Q$ = "D" GOTO 1170
1090 GOTO 1000
1170 HOME : VTAB 4: HTAB 4: PRINT "ENTER THE DISK FILE NAME !" : PRINT
1175 PRINT : INPUT "FILE NAME = ";F$: IF    LEN (F$) = 0 GOTO 1000
1180 PRINT D$;"OPEN ";F$: PRINT D$;"READ ";F$: INPUT CH(0)
1185 FOR J = 1 TO CH(0): INPUT D A$(J),CL(J): NEXT J
1190 PRINT D$;"CLOSE ";F$: PRINT D$;"IN#0": PRINT D$;"PR#0"
1195 GOTO 1000
4400 HOME : VTAB 4: HTAB 4: PRINT "SAVE THE FILE ON (T)APE OR (D)ISK ?":
4410 PRINT : INPUT "ENTER (T) OR (D) ?";Q$
4420 IF Q$ = "T" THEN 4000
4430 IF Q$ < > "D" THEN 1000
4440 PRINT : PRINT : PRINT : PRINT "ENTER THE DISK FILE NAME !" : PRINT
4450 INPUT "FILE NAME = ";F$
4460 IF    LEN (F$) = 0 THEN END
4470 PRINT D$;"OPEN ";F$
4480 PRINT D$;"WRITE ";F$
4490 PRINT CH(0)
4500 FOR J = 1 TO CH(0)
4510 PRINT DA$(J);",";CL(J): NEXT J
4520 PRINT D$;"CLOSE ";F$
4530 END
```

Fig.26-2. Lines changed to make program compatible for both disk and tape.

number equivalent in order to store the number. CA(K) = ASC(MID$(DA$(J),K,1)). Lines 4010-4050 of Fig. 26-1. STORE CA saves the file to tape. RECALL CA loads the file from tape. The file is converted into a string by the statement DA$ = DA$ + CHR$(CA(K)) in lines 1110-1150.

CH(0)	CH(0) holds the number of records in the list.
CH(1)	CH(1) is an array used as temporary storage for the record count.
DA$	DA$ is the string array in which the name, area code, and the telephone number are stored.
CL	CL holds the length of DA$ before it is padded to exactly 30 characters.

SP$	SP$ is a string that contains only blank characters and is used to pad DA$ to thirty characters.
DC$	DC$ holds the default value of the area code. This is an easy method to store the local area code. DC$ is used to save typing.
D$	D$ = CHR$(4) is the key to the disk operating system. All commands that relate to DOS must be preceded by PRINT D$.
MS	MS is the variable used for menu selection.
TAPE	A cassette tape is a linear magnetic medium used to store computer programs.
DISK	A disk is a circular magnetic medium to store computer programs.
Q$	Q$ is the string variable that is used to store a Y for yes, or N for no, on which the program makes a decision.
RECALL CH	RECALL CH is the command used to retrieve a real or integer array that has been stored on tape. The array must be DIMensioned in the program. Subscripts are not used when storing or recalling arrays. CA(0), CA(1), CA(2), etc., are stored and recalled as CA. CA(45) contains 45 characters including the padded spaces to make DA$ exactly 30 characters.
RECALL CA	RECALL CA is the command used to store the number of records on tape.
J,K	J and K are used as loop variables.
F$	F$ is the string array used to hold the name of the file on disk.
IN#0	PRINT D$;"IN#0" is the command that changes the input direction to come from the keyboard instead of the disk operating system.
PR#0	PRINT D$;"PR#0" is the command that changes the print direction to go to the CRT instead of to the disk operating system.
GOSUB 10000	GOSUB 10000 causes the "NAME = and PHONE # = " on the screen. A name and phone number from the list is printed after the prompt.
VT	VT is the variable that is used to store the value used in the TAB function.
CL(CH(0) + 1)	CL(CH(0) + 1) holds the length of the name string (DA$) before it is padded to thirty characters.
LEN	LEN is the Applesoft function that returns the number of characters in a string, between 0 and 255.

GOSUB 10010	GOSUB 10010 allows the user to enter a name into the list.
GOSUB 10080	GOSUB 10080 allows the user to enter an area code, and telephone number in the list.
TC$	TC$ is the temporary area code string to check the number entered into the area code. To determine that the area code is numeric, the statement TC$ = STR$(VAL(AC$)) is used.
PT$	PT$ is the temporary phone number string to check that the telephone number entered is numeric. To determine that the phone number is numeric, the statement PT$ = STR$(VAL(PN$)) is used.
R	R is the variable used when the REENTER question is asked.
M	M is a variable used in the delete section of the program. An interchange between the variables J and K (Fig. 26-8) determines if the record is to be deleted.
NA$	NA$ is the string array that holds the name to be changed.
C	C is the variable used when a change is to be made in a name or telephone number.
D	D is the variable used when a record is to be deleted.
N	N is a variable used when the name is to be changed.
P	P is a variable used when the phone number is to be changed.
B	B is a variable used when both the name and phone number are to be changed.
C$	C$ is the string array used to hold the letters N, P, or B when the name, phone, or both are to be changed.
S	S is the variable when the search routine is used.
L	L is the variable used when the file is to be listed.
CO	CO is the variable that holds the count of the number of times a match is found on the list (search).
CHR$(7)	CHR$(7) is the command used to ring the bell in the computer.
AC$	AC$ is the string array that holds the area code.
PN$	PN$ is the string array that holds the telephone number.
STORE CA	STORE CA is the command that stores the number of records on tape.
STORE CH	STORE CH is the command that stores the length of the record on tape.

STR$	STR$ is the Applesoft function that converts a numeric value into a string.
VAL	VAL is an Applesoft function that changes a string value to a numeric value.

The same variables are now presented in alphabetical order.

AC$	AC$ is the string array that holds the area code.
B	B is the variable used when both the name and telephone number are to be changed.
C	C is the variable used when a change is to be made in the name and/or telephone number.
C$	C$ is the string array used to hold the letters N, P, or B when the name, phone, or both are to be changed.
CA	CA is the array used to store and retrieve information from tape. The Applesoft language cannot store string arrays directly. The string arrays are converted to the number equivalent in order to store the number. CA(K) = ASC(MID$(DA$(J),K,1)). Lines 4010-4050 of Fig. 26-1. STORE CA stores the file to tape. RECALL CA loads the file from tape. The file is converted into a string array by the statement DA$ = DA$ + CHR$(CA(K)) in lines 1110-1150.
CH(0)	CH(0) holds the number of records in the list.
CH(1)	CH(1) is an array used as temporary storage for the record count.
CHR$(7)	CHR$(7) is the command used to ring the bell in the computer.
CL	CL holds the length of DA$ before it is padded to exactly thirty characters.
CL(CH(0) + 1)	CL(CH(0) + 1) holds the length of the name string (DA$) before it is padded to thirty characters.
D	D is the variable used when the record is to be deleted.
D$	D$ = CHR$(4) is the key to the disk operating system. All commands that relate to DOS must be preceded by PRINT D$.
DA$	DA$ is the string array in which the name, area code, and the telephone number are stored.
DC$	DC$ is the string array that holds the default value of the area code. DC$ is a method used to save typing.
F$	F$ is the string array used to hold the name of the file stored on disk.
GOSUB 10000	GOSUB 10000 causes the "NAME = and PHONE # = " prompt to be displayed on the screen. A name and

	telephone number from the list is printed after the prompt.
GOSUB 10010	GOSUB 10010 allows the user to enter a name into the list.
GOSUB 10080	GOSUB 10080 allows the user to enter an area code, and telephone number in the list.
IN#0	PRINT D$;"IN#0" is the command that changes the input direction to come from the keyboard instead of the disk operating system.
J,K	J and K are used as loop variables.
LEN	LEN is the Applesoft function that returns the number of characters in a string, between 0 and 255.
M	M is a variable used in the delete section of the program. An interchange between variables J and K (Fig. 26-8) determines if the record is to be deleted.
MS	MS is the variable used for the menu selection.
N	N is the variable used when the name is to be changed.
NA$	NA$ is the string array that holds the name to be changed.
P	P is the variable used when the phone number is to be changed.
PN$	PN$ is the string array that holds the telephone number.
PR#0	PRINT D$;"PR#0" is the command that changes the print direction to go to the CRT instead of to the disk operating system.
PT$	PT$ is the temporary phone number string to check that the telephone number entered is numeric. To determine that the phone number is numeric, the statement PT$ = STR$(VAL(AC$)) is used.
Q$	Q$ is the string variable that is used to store a Y for yes, or N for no, on which the program makes a decision.
R	R is the variable used when the REENTER question is asked.
RECALL CA	RECALL CA is the command used to recall the number of records stored on tape.
RECALL CH	RECALL CH is the command used to retrieve a real or integer array that has been stored on tape. The array must be DIMensioned in the program. Subscripts are not used when storing or recalled as CA. CA(45) contains exactly 45 characters including the padded spaces to make DA$ exactly thirty characters.

S	S is the variable used in the search routine.
SP$	SP$ is a string that contains only blank characters and is used to pad DA$ to thirty characters.
STORE CA	STORE CA is the command that stores the number of records on tape.
STORE CH	STORE CH is the command that stores the length of the record on tape.
STR$	STR$ is an Applesoft function that converts a numeric value into a string.
TAPE	A cassette tape is a linear magnetic medium used to store computer programs.
TC$	TC$ is a temporary area code string used to check the number entered into the area code. To determine that the area code is numeric, the statement TC$ = STR$(VAL(AC$)) is used.
VT	VT is the variable used to store the value used in the TAB function.
VAL	VAL is an Applesoft function that changes a string value to a numeric value.

The program is designed to input up to 1000 names and telephone numbers in a sequential file. The list is sorted alphabetically by name each time it is written to tape or disk. This program is designed to be an introduction to files, file maintenance, sorting, searching, and deleting.

Sorting is the act of placing information in a predetermined sequence. Sorting depends on sequencing items according to a key word. Lists of names are usually keyed or sorted alphabetically on the first letter of the last name. Telephone numbers are usually keyed or sorted on the area code. Mailing lists may be sorted according to the zip code. Lists can be sorted in any manner that meets the needs of the user. Lists are sorted to increase the speed and efficiency of the search and delete functions. From the human point of view, lists are sorted because we expect to see lists in the proper order.

The correct time to sort the list is after file maintenance is complete and before it is saved to tape or disk. File maintenance includes all changes to the list, all updates to the list, and all deletions to the list.

There are several types of sorts used in programming. Ripple, modified ripple, bubble, and Shell-Metzner are some of the better known sorts. Shell-Metzner is the most efficient sort of this group. A detailed discussion of sorts is out of the scope of this book.

In the program in Fig. 26-1, the sort is set up using double nested loops so each item on the list can be compared and ordered (Fig. 26-3). Each comparison is called a pass. The items on the list are compared to each other

during the passes, and items on the list are swapped to place them in the correct order.

A search is an act of examining items on a list to discover whether the key being searched for is on the list. All items related to that key are then dis-

NUMBER OF ITEMS IN THE LIST—5
ORIGINAL ORDER OF THE LIST

DA$(1)	DA$(2)	DA$(3)	DA$(4)	DA$(5)
E	D	C	B	A

DA$(K)	CONTAINS NAME AND TELEPHONE NUMBER
FOR J = 1 TO CH(0) – 1 (FOR J = 1 TO 4)	NUMBER OF PASSES IS ONE LESS THAN THE # OF ITEMS IN THE LIST
FOR K = J + 1 TO CH(0) (FOR K = 2 TO 5) (1st pass only)	IF K STARTED AT #1 THE SAME ITEM WOULD BE COMPARED TO ITSELF

(A) Information for sort.

PASS	J	Ms	J+1	K	DA$(1)	DA$(2)	DA$(3)	DA$(4)	DA$(5)	LINE #	CONDITION	Ma
0	1	1			E	D	C	B	A	1430-1470		
1	1	1	2	2		D				1440	TRUE	2
				3			C			1440	TRUE	3
				4				B		1440	TRUE	4
				5					A	1440	TRUE	5
M=5, J=1					A	D	C	B	E	1460	FALSE	
2	2	2	3	3			C			1440	TRUE	3
				4				B		1440	TRUE	4
				5					E	1440	FALSE	4
M=4, J=2					A	B	C	D	E	1460	FALSE	
3	3	3	4	4	A	B	C	D	E	1440	FALSE	3
				5						1440	FALSE	
NO EXCHANGE M = J												
4	4	4	5	5	A	B	C	D	E	1440	FALSE	4
NO EXCHANGE M = J												

Ms—M AT SORT PASS
Ma—M AFTER CHANGE

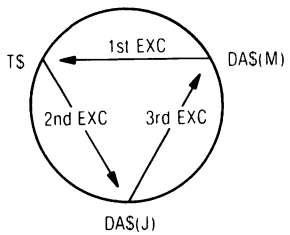
1st PASS	EXCHANGE	M = 5(A), J = 1(E)
2nd PASS	EXCHANGE	M = 4(B), J = 2(D)
3rd PASS	NO EXCHANGE	3rd ITEM IN THE LIST IS IN THE CORRECT ORDER
4th PASS	NO EXCHANGE	4th ITEM IN THE LIST IS IN THE CORRECT ORDER

(B) Computer sort.

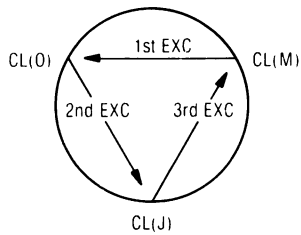
Fig. 26-3. Sorting a list.

played. A search aids the user in discovering all items on the list related to the key. A key can be a name or a fragment of a name or a phone number. How often do you remember the last name of a person but not his or her first name? If the name is on the list, a search will reveal it. A search allows the user to pull one record off the list by using a keyword with which to search. A search can be made any time the user needs information from the records in the list.

```
1470  T$ = DA$(M) : DA$(M) = DA$(J) : DA$(J) = T$CL(0) = CL(M) :
      CL(M) = CL(J) : CL(J) = CL(0)
```



CL = length of DA$ before it is padded. In this example, CL is always 1 character.



(C) Exchange routine.

		T$	DA$(M)	DA$(J)	
ORIGINAL			E	A	
1st PASS	1st EXC	E	E	A	T$ = DA$(5)
	2nd EXC	E	A	A	DA$(5) = DA$(1)
	3rd EXC	A	A	E	DA$(1) = T$
2nd PASS	1st EXC	D	D	B	T$ = DA$(4)
	2nd EXC	D	B	B	DA$(4) = DA$(2)
	3rd EXC	B	B	D	T$ = DA$(2)
3rd PASS	NO EXCHANGE—3rd ITEM IS IN THE CORRECT POSITION				
4th PASS	NO EXCHANGE—4th ITEM IS IN THE CORRECT POSITION				

(D) Table of passes and exchanges.

Fig.26-3–cont. Sorthing a list.

Deletion is the act of removing a record or records from the list (Fig. 26-4). Deleting is used to keep the file as small as possible to use the least memory and to keep the file current. A list containing unneeded names is nonproductive and costly to most users.

Deletions should be made any time names on the list become of no use to the user. If the list is for subscriptions, each name on the list costs money for production costs, mailing costs, and labor. Nonsubscribers' names on the list should be deleted.

```
FILE:
  DA$(1) = "JONES (713)-688-1212"
  DA$(2) = "SMITH (713)-688-1213"                    CH(0) = 4
  DA$(3) = "DELETE (000)-000-0000"
  DA$(4) = "ACTION (713)-688-1214"
```

PASS	K	J	DA$(J)	DA$(K)	LINE 2250	K = J
	0					
1	1	1	JONES	JONES	FALSE	FALSE
2	2	2	SMITH	SMITH	FALSE	FALSE
3	2	3	DELETE		TRUE	JUMPS OVER
4	3	4	ACTION→			

```
FILE:
  DA$(1) = "JONES (713)-688-1212"
  DA$(2) = "SMITH (713)-688-1213"
  DA$(3) = "ACTION (713)-688-1214"
  DA$(4) = "ACTION (713)-688-1214"
2290 CH(0) = K (K = 3)—LAST RECORD DA$(4) IS REMOVED
```

Fig. 26-4. Delete routine.

Line 1 of the program (Fig. 26-1) DIMensions the arrays used in the program. CA(45) is dimensioned to hold the forty-five (45) name and telephone number digits. CH is an array that holds the number of records in the file. DA$ is the array into which the name, area code, prefix, and phone number are placed. The program is designed to accept and retain a file of up to 1000 names and phone numbers. CL is an array to hold the name string (DA$) before it is padded to exactly thirty characters. Since the store and recall commands do not store string arrays, they must be converted into numeric arrays, lines 4010 through 4070.

4010 STORE CH

Stores the number of records to be placed on the tape. In this case, CH(0) = 5 (5 is an arbitrary number).

4020 FOR J = 1 TO CH(0)

Sets the beginning of the loop to store 5 records on tape (CH(0) = 5).

4030 FOR K = 1 TO 44

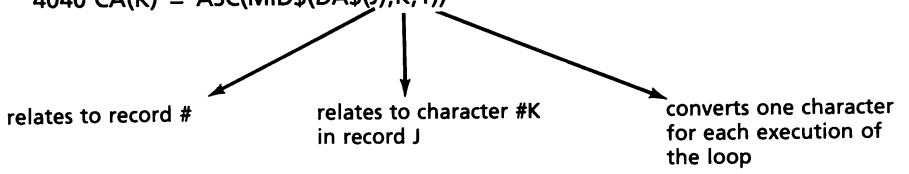
States that there are 44 characters in each record plus CL = 1. CL stores the length of DA$ before it is padded. There are 30 characters in the name string including padded characters (spaces), produced by SP$.

(713) – 688 – 1212

1+3+1+1+3+1+4+30 = 44 characters in DA$. CL = 1. CL is the length of DA$ before it is padded.

30 characters in DA$
14 characters in area code (AC$) and phone number (PN$)
1 character for CL = length of DA$ before padding.
45 = CA(45)

4040 CA(K) = ASC(MID$(DA$(J),K,1))



Line 4040 converts the string array, DA$, into ASCII characters and places it in CA numeric array to be stored on tape.

4050 NEXT K

Completes the conversion of one record.

4060 CA(45) = CL(J)

CL is the length of DA$ before it is padded and stored in CA(45). CL(J) is one number. One added to the 44 characters produced in line 4020 equals 45 numbers; thus, CA(45).

4070 STORE CA

Stores the real array, CA, on tape. The subscript of the array is not indicated when STORE is used. This stores all 45 values from each record.

4080 NEXT J

When one record is converted to a numeric array and stored, the next record is processed. This processing continues until all five records have been stored. Returning to line 1, we see that CL holds the length of DA$ (name) before it is padded. DA$ (1000) can contain 1000 records (name and phone number) and CL (1000) can contain 1000 lengths of names in DA$.

```
10 SP$ = "          " : SP$ = SP$ + SP$ + SP$
```

Line 10 SP$ = " " : SP$ = SP$ + SP$ + SP$, is a string used to place spaces in the name string (DA$) so it is padded to exactly thirty characters, (line 10060). SP$ = " ", sets ten spaces between the quotation marks. SP$ = SP$ + SP$ + SP$, concatenates the SP$ to thirty blank spaces to pad the name string.

```
10060 DA$(CH(0)+1) = DA$(CH(0)+1)+LEFT$(SP$, 30-CL(CH(0)+1))
```

30 spaces

Length of DA$ (name) record being built

10070 RETURN

The seven digits often referred to as the "phone number" consists of seven digits. The area code consists of three digits. The central branch or exchange code consists of three digits. The phone number consists of four digits. In this discussion, the phone number will be defined as the seven digits of the central branch office and the phone number.

Line 15 DC$ = "713," is a line designed to save typing when the area code is in the "713" area. Line 10090 is INPUT AC$, PN$. If a comma (,) is entered before the phone number, then LEN(AC$) = 0. When LEN(AC$) = 0 then the program accepts DC$ = "713" as the area code. Line 15 can be easily changed to accept any area code. The area code in your area would be the choice to enter in line 15, so local phone numbers would be quicker and easier to type.

Line 16 D$ = CHR$(4) is the key to the disk operating system. When data or information is directed to the disk operating system, the deferred command must be preceded by PRINT D$. In line 1180 the file is to be opened, so PRINT D$;"OPEN ";F$: PRINT D$;"READ ";F$, causes the computer to read the file stored on disk.

When the file has been read from disk, the direction of the input and print must be returned to the keyboard and the CRT. To change the direction, PRINT D$;"IN#0", returns the input direction to the keyboard. PRINT D$;"PR#0" causes the PRINT statements to go from the disk operating system to the CRT. Each time the direction is changed, DOS must be aware of the change, or the system and program will not work.

The main menu, PHONE LISTING, is contained in lines 30 through 60.

```
LINE 30 — 1.ENTER              (ON MS GOTO 1000)
LINE 40 — 2.MODIFY/DELETE      (ON MS GOTO 2000)
LINE 50 — 3.LIST/SEARCH        (ON MS GOTO 3000)
```

LINE 60 — 4.SAVE LIST & END (ON MS GOTO 4400)

Line 30 — 1.ENTER is required to be selected as the first step in order to load a file from tape or disk into memory.

When 1.ENTER is selected, the program branches to line 1000. The secondary menu, FILE MAINTENANCE, is contained in lines 1000 through 1030.

LINE 1010 — 1.LOAD OLD FILE (ON MS GOTO 1070)
LINE 1020 — 2.ENTER NEW ITEMS (ON MS GOTO 1200)
LINE 1030 — 3.RETURN TO MAIN MENU (ON MS GOTO 1400)

If option 1 from the secondary menu, FILE MAINTENANCE, is selected, the program branches to line 1070, to ask the user, "IS THE FILE ON (T) TAPE OR (D) DISK?"

If the "T" tape option is selected, the program branches to line 1100 — RECALL CH. The number of records previously SAVEd to tape is stored in the array CH. If CH is greater than zero (line 1120), the records are read into computer memory. In line 1150 DA$(J) is initialized to a null value. The K loop reads the 44 characters in the name and phone number from tape. The forty-fifth character (CA(45)) is a count of the number of characters in the name. The J loop increments until all the records have been placed in memory. The program then jumps to the secondary menu at line 1000.

If option 1, line 1010, from the secondary menu, FILE MAINTENANCE, is selected, the program branches to line 1070 to ask the user, "IS THE FILE ON (T)APE OR (D)ISK?"

If the (D) disk option is selected, the program branches to line 1170.

At line 1170, the user is asked, "ENTER THE DISK FILE NAME!" The user enters the specific name of a file saved to disk.

In line 1180, PRINT D$;"OPEN ";F$: PRINT D$;"READ ";F$: INPUT CH(0), PRINT D$;"OPEN ";F$, is the command that alerts the disk operating system (DOS) that it should open a file on disk having the same name as is stored in F$. After the file is opened, the PRINT D$;"READ ";F$, tells DOS that records are to be read from disk and the INPUT command tells DOS to fill the file's buffer in the computer's memory. The number of records stored on disk are read. The number of records (CH(0)) is used as the ending loop index to read in the number of records stored on disk. INPUT DA$(J), CL(J), reads in the name and telephone number into DA$(J) and the character count into CL(J) (one record) for each loop increment.

When the records stored on disk have been read into memory, and the user is ready to end file usage, the file must be closed before the program ends. PRINT D$;"CLOSE ";F$, closes the file. After a file is read, it should always be closed. This makes a buffer available for use by another file.

The commands to write to disk are used in the combination — WRITE PRINT.

```
4470 PRINT D$;"OPEN ";F$
4480 PRINT D$;"WRITE ";F$
4490 PRINT CH(0) - NUMBER OF RECORDS
4500 FOR J = 1 TO CH(0)
4510 PRINT DA$(J);",";CH(J)
     NEXT J
4520 PRINT D$;"CLOSE ";F$
```

The commands to read from disk are used in the combination — READ INPUT.

```
1180 PRINT D$;"OPEN ";F$
     PRINT D$;"READ ";F$
     INPUT CH(0) - NUMBER OF RECORDS
1185 FOR J = 1 TO CH(0)
     INPUT DA$(J), CL(J)
     NEXT J
1190 PRINT D$;"CLOSE ";F$
```

The commands to return control to the CRT and the screen are as follows.

```
PRINT D$;"IN#0" RETURNS INPUT CONTOL TO THE KEYBOARD
PRINT D$;"PR#0" RETURNS THE PRINT DIRECTION TO THE CRT
```

PRINT D$;"IN#0", returns control to the keyboard and specifies that all input is to come from the keyboard. PRINT D$;"PR#0", specifies that all PRINT statements are to go to the CRT. PR#0 is also the command that turns off the printer, so if this program RUN is to be placed on the printer, the program has to be changed to avoid turning off the printer at each PR#0. To prevent the printer from being turned off, replace PR#0 with PR#1, if the printer is in slot 1.

The program returns to the secondary menu, FILE MAINTENANCE, to line 1000. If selection 2.ENTER NEW ITEMS is selected, the program branches to line 1200.

```
1200 HOME : VT = 6 : GOSUB 10010
```

In line 1200, HOME clears the screen, VT = 6 sets the vertical tab value that is to be used in the subroutine beginning at line 10010.

The subroutine that begins at line 10010 receives the characters in the name. There are four cases to be considered, (1) if the user does not place any characters in DA$ (RETURN is pressed), (2) if the user enters more than 30 characters in DA$, (3) if the user enters exactly 30 characters in DA$, and (4) if the user enters less than 30 characters in DA$.

Case 1, line 10030, if the length of the name string CL(CH(0) + 1) is equal to zero (0), then RETURN was pressed, and no characters were entered into DA$. Case 1 causes the program to RETURN to line 1000, via line 1210.

Case 2, line 10040, if more than thirty characters were entered into DA$, the program prints the message, "NAME IS TOO LONG", rings a bell

(CHR$(7)), pauses for 1000 loop incrementations, and jumps to line 10010, to allow the user to enter another name.

Case 3, if the number of characters entered into DA$ is exactly 30 characters, line 10050, the program RETURNS to line 1210, defaults to 1260, GOSUB 10080.

Case 4, if the number of characters entered into DA$ is less than 30, the program defaults to line 10060. Line 10060 uses the space string (SP$, line 1) to create a DA$ of exactly 30 characters. The program then returns to line 1310 to concatenate the name, area code, prefix, and phone number into DA$ (Fig. 26-1).

```
10010 VTAB VT : CALL -958 : VTAB VT : PRINT "ENTER NAME (LESS THAN 31
      CHARACTERS)"
10020 PRINT : INPUT DA$(CH(0) + 1))
10030 CL(CH(0) + 1) = LEN(DA$(CH(0) + 1)) : IF CL(CH(0) = THEN RETURN
10040 IF CL(CH(0) + 1) > 30 THEN PRINT : PRINT "NAME IS TOO LONG" : CHR$(7)
      : FOR J = 1 TO 1000 : NEXT J : GOTO 10010
10050 IF CL(CH(0) + 1 = 30 THEN RETURN
10060 DA$(CH(0) + 1) = DA$(CH(0) + 1 + LEFT$ (SP$, 30 - CL(CH(0) + 1))
10070 RETURN
```

In line 10010, VTAB VT tabs to line #6 on the screen, and CALL -958 is a machine language call that clears the screen below the cursor.

Line 10020, INPUT DA$ (CH(0) + 1), allows the name to be entered into a specific record number.

```
10030 CL(CH(0)+1) = LEN(DA$(CH(0)+1)) : IF CL(CH(0)+1) = 0 THEN RETURN
```

Line 10030 stores the length of DA$ for the current record being built. This information is stored in CL array. The second statement in line 10030, IF CL(CH(0)+1) = 0 THEN RETURN, checks to see if a record is entered. If no record is entered, the program RETURNS to line 1000.

```
10040 IF CL(CH(0) + 1) > 30 THEN PRINT : PRINT "NAME IS TOO LONG" : CHR$(7):
      for J = 1 TO 1000 : NEXT J : GOTO 10010
```

DA$ can be a total of 30 characters. If the length of the name string is over 30 characters, it is disallowed, because the file is not designed to hold over 30 characters in the name part of DA$.

The CHR$(7) rings the computer bell, and the loop FOR J = 1 TO 1000 pauses for a count of 1000. GOTO 10010 causes the program to jump back to line 10010 to allow the user to enter a name of 30 characters or less.

```
10050 IF CL(CH(0) + 1) = 30 THEN RETURN
```

In line 10050, if the name is exactly 30 characters, it is the proper length to fit the file and the program returns to line 1260.

```
10060 DA$(CH(0) + 1) = DA$(CH(0) + 1 + LEFT$ (SP$,30 - CL(CH(0) + 1) )
1260  VT = 12 : GOSUB 10800
```

```
10080 VTAB VT : PRINT "ENTER AREA CODE, PHONE NO."
10090 PRINT : PRINT : INPUT AC$,PN$
```

Line 10080 prints the user prompt, and line 10090 allows the user to enter the area code, prefix, and the phone number.

```
10100 IF LEN (AC$) = 0 THEN AC$ = DC$
```

The input format is AC$,PN$. If there is no entry into AC$ (a comma is the first character typed), line 10100 is true and then AC$ = DC$, which is the area code "713" (line 15).

```
10100 IF LEN(AC$) < > 3 OR LEN (PN$) < > 7 THEN 10080
```

In line 10110, if the area code is not three characters, or the prefix and phone number is not seven characters, then the program jumps to line 10080 for the correct entry.

```
10120 TC$ = STR$ ( VAL (AC$) ) : PT$ = STR$ ( VAL (PN$) ) : IF TC$ < > AC $ OR PT
      $ < > PN $ THEN PRINT : PRINT "PLEASE USE NUMERICS"; CHR$(7) : FOR J
      = 1 TO 1000 : NEXT J : GOTO 10080
```

Line 10120 converts the string arrays to numeric arrays as a further check that the area code, prefix, and phone number are numerics. TC$ = STR$(VAL(AC$)) converts the area code string to a numeric value and that value is converted to a string. PT$ = STR$(VAL(PN$)) converts the prefix and phone number string to a numeric value and that value is converted to a string. IF TC$ < > AC$ OR PT$ < > PN$ THEN PRINT : PRINT "PLEASE USE NUMERICS", is a check to determine that the area code string and the phone number string have been input correctly. CHR$(7) rings the computer's bell. The pause loop is activated, and the program returns to line 1310 to concatenate the name, area code, and phone number into a single array, DA$.

```
1310 DA$(CH(0) + 1) = DA$(CH(0) + 1) + "(" + TC$ + ")-" + LEFT$(PT$,3) +
     "-" + RIGHT$ (PT$,4)
```

As an example, line 1310 translates to the following

```
JOHN SMITHXXXXXXXXXXXXXXXXXXXX   30 CHARACTERS
(713)-688-1212                   14 CHARACTERS
```

```
1320 PRINT : GOSUB 1000
```

Line 1320 causes the record entered to be printed on the screen by the subroutine at line 10000. A sample printout is shown.

```
NAMEX = XXJOHN SMITH
PHONEX#=X(713) - 688 - 1212
1330 PRINT : INPUT "ENTER 'R' TO REENTER ELSE 'RETURN'?";Q$ : IF Q$ < > "R"
     THEN CH(0) = CH(0) + 1
```

In line 1330, IF Q$ < > "R," then the record count is incremented and a new record may be entered. The program then defaults to line 1340, and jumps to line 1200. Line 1200 clears the screen and jumps to the subroutine at line 10010. If no record is entered (RETURN is pressed), the program jumps to the secondary menu at line 1000.

```
1410 IF CH(0) < 2 THEN RETURN
```

Line 1410 makes the decision that if there is only one record in the file, there is no need to sort the list.

Lines 1420 through 1490 perform the sort routine. In this example (Fig. 26-3), there are five records in the list. The five records "E, D, C, B, A" represent the list of names, area codes, and telephone numbers. Fig. 26-3 shows the details of program lines 1420, 1430, 1440, 1450, 1460, 1470, and 1480. The number of records in the list is CH(0) = 5.

```
1420 FOR J = 1 TO CH(0)
```

Line 1420 determines the maximum number of passes through five records to order the list. This list took only two passes to order.

```
1430 M =J : FOR K = J TO CH(0)
```

In line 1430, the K variable begins at the second record in the list. If both K and J started at the same record, the same record would be compared to itself and this would be a useless comparison.

```
1440 IF LEFT$(DA$(K),30) < LEFT$(DA$(M),30) THEN M = K
```

Line 1440 compares the position of the records in the list. If record K is less than record M, then the value of K is stored in M. This comparison continues for K times.

```
1460 IF M = J THEN 1480
```

If line 1460 is true, the records for a specific pass are in the correct order and no exchange is made. If line 1460 is false, the program defaults to line 1470 to exchange the records in the list. In a sort, all items out of order must be exchanged. The DA$'s are ordered by exchanging records that are out of position.

```
1480 NEXT J
```

Line 1480 processes the next record in the list.

```
1490 RETURN
```

Line 1490 returns the program to the second statement in line 1400, which is GOTO 20. GOTO 20 causes the program to jump to the main menu, PHONE LISTING.

```
40   VTAB 12 : HTAB 8 : PRINT "2.MODIFY/DELETE"
```

Selection 2 of the main menu causes the program to jump to line 2000 to modify or delete records in the list.

```
2000 HOME : VTAB 4 : IF CH(0) = 0 THEN PRINT "THERE IS NO LIST"; CHR$(7) :
     FOR J = 1 TO 2000 : NEXT J : GOTO 20
```

If CH(0) = 0 there are no records in the list.

```
2010 PRINT "ENTER NAME TO BE CHANGED" : PRINT : INPUT NA$
2020 IF LEN(NA$) = 0 THEN 20
```

If the length of the name string is zero, the program branches to the main menu, PHONE LISTING.

```
2030 FOR K = 1 TO CH(0)
```

Line 2030 sets the beginning of the loop to process each record in the list.

```
2040 IF NA$ < > LEFT$ (DA$(K) ) THEN 2060
```

If the name string that was entered does not match any name in the list, then the program jumps to line 2060.

```
2050 GOTO 2100
2060 NEXT K : VTAB 10 : PRINT "THIS NAME IS NOT ON LIST" : PRINT CHR$ (7) :
     FOR J = 1 TO 1000 : NEXT J : GOTO 2000
```

If line 2040 is false, the program defaults to line 2050, and jumps to line 2100.

```
2100 CH(1) = CH(0) : CH(0) = K - 1 : VTAB 6 : PRINT "CURRENT RECORD IS " :
     PRINT
```

CH(1) = CH(0) stores the value of the 5 records in the list in CH(1) for temporary storage. CH(0) = K - 1 stores the value of the record to be changed. Each time the loop executes, it is incremented by one greater than the loop value. K - 1 decrements the loop value to correspond to the number of records on the list.

```
2110 VTAB 8 : GOSUB 10000 : PRINT : PRINT "ENTER 'C' TO CHANGE, 'D' TO
     DELETE" : PRINT : INPUT "ELSE 'RETURN' ?";Q$
```

Line 2110 sets up the record to be modified (GOSUB 10000 prints the current record on the screen), prints the heading to change or delete the record, and requests user input.

```
2120 IF Q$ < > "C" AND Q$ < > "D" THEN 2240
```

If C is not pressed, and D is not pressed, and RETURN is pressed, the program branches to line 2240 to reestablish the list in correct alphabetical order. Line 2300 causes the list to be sorted in alphabetical order and the program jumps to line 20, the main menu, PHONE LISTING.

```
2125 IF Q$ = "D" THEN DA$(K) = "DELETE" + LEFT$ (SP$,24) +
     "(000)-000-0000" : GOTO 2230
```

If D is entered, the program jumps to line 2230 to reset the value of the number of records in the list into CH(0) (CH(0) = CH(1)), and prints the user prompt, "ANY MORE CORRECTIONS (Y OR N)?".

```
2230 CH(0) = CH(1) : PRINT : INPUT "ANY MORE CORRECTIONS (Y OR N) ?";Q$ :
     IF Q$ = "Y" THEN 2000
```

If there are no more corrections to the list, the program defaults to line 2240 to the delete routine (Fig. 26-4).

```
2240 K = 0 : FOR J = 1 TO CH(0)
2250 IF LEFT$ (DA$(J),6 = "DELETE" THEN 2280
2260 K = K + 1 : IF K = J THEN 2280
```

If line 2250 is false, line 2260, K = K + 1, increments the value of K to accommodate the record. If K = J THEN 2280 is false, the record is stored in DA$(J), DA$(K) = DA$(J), and the length of the record CL(K) is also stored, CL(K) = CL(J). The transfer places the record in K.

Records are taken from DA$(J) and placed in DA$(K) unless they are equal to DELETE. If K = J, no action is taken because the record DS$(J) = DA$(K) and the records are not moved. Each time a DELETE record is encountered, J is incremented, but K remains the same. As an example (Fig. 26-4) if the third record on the list is DELETE, then when the fourth record is processed, K is still equal to three (3), but J is equal to four (4). DA$(4) is then moved into DA$(3). If line 2250 is true, K remains the same value for the next loop execution. The next record is written over the deleted record.

```
2300 GOSUB 1410 : GOTO 20
```

Line 2300 causes the program to jump to the subroutine that sorts the list alphabetically, and then branches to the main menu, PHONE LISTING.

Going back to line 2120, IF Q$ < > "C" AND Q$ < > "D" THEN 2240. IF Q$ < > "C" is true, the program defaults to line 2130.

```
2130 VTAB 12 : CALL -958 : BTAB 12 : PRINT "ENTER 'N' -NAME, 'P'-PHONE#,
     'B'-BOTH" : PRINT
```

CALL -958 is a machine call that clears the screen below the cursor at VTAB 12. The name and telephone number headings are printed, and remain on the screen as a prompt during the changes. The program defaults to line 2140 to store the area code and phone number in T$, and requests the user to enter a letter.

```
2140 T$ = RIGHT (DA$(K,14) : INPUT "LETTER PLEASE ?";C$ : IF C$ < > "N" AND
     C$ < > "P" AND C$ < > "B" THEN 2130
```

If the letter B is entered, the program defaults to line 2150, which is "IF C$ = "P" THEN 2170". Since the letter B is entered, line 2150 is false, and the program defaults to line 2160.

```
2160 VT = 14 : GOSUB 10010
```

Line 2160 sets the VTAB variable to 14 and the subroutine at line 10010 asks for the name change to be entered. The subroutine returns to line 2170.

```
2170 IF C$ = "N" THEN 2190
```

The letter B was entered and this makes line 2170 false, and the program defaults to line 2180.

```
2180 VT = 16 : GOSUB 10080
```

The subroutine at 10080 asks the user to enter the new telephone number.

The B for both name and telephone number is not written into the program routine. B is the default value when N or P is not entered. Remember the cliche about cutting a log in two places to get three sticks of wood? This example demonstrates the use of the default value in programming. B was the selection, but B was not written into the program as a decision statement. Another point to be reenforced is, when an IF statement is true, all following statements on that line are executed.

In line 2140, the user enters the letter N for a name change. The program defaults to line 2150.

```
2150 IF C$ = "P" THEN 2170
```

The letter N was entered, so line 2150 is false, and the program defaults to line 2160.

```
2160 VT = 14 : GOSUB 10010
```

The subroutine at line 10010 asks for the name change to be entered. The subroutine at 10010 returns to line 2170.

```
2170 IF C$ = "N" THEN 2190
2190 IF C$ = "N" THEN DA$(K) = DA$(K) + T$ : GOTO 2230
```

Line 2170 is true, so the program branches to line 2190. The name change is concatenated to the area code and telephone number stored in T$. Line 2230 sets CH(0) = CH (1) and asks the user for more corrections. If there are no more corrections, the number of records in the list is stored in CH(0), the list is sorted, and the program jumps to the main menu, PHONE LISTING.

If the user enters the letter P in line 2140, the program defaults to line 2150.

```
2150 IF C$ = "P" THEN 2170
```

The letter P was entered, line 2150 is true, so the program branches to line 2170.

```
2170 IF C$ = "N" THEN 2190
```

The letter P was entered, line 2170 is false, and the program defaults to line 2180.

```
2180 VT = 16 : GOSUB 10080
```

GOSUB 10080 allows the user to enter the new telephone number and returns to line 2190.

```
2190 IF C$ = "N" THEN DA$(K) = DA$(K) + T$ : GOTO 2230
```

The letter P was entered, so line 2190 is false, and the statement GOTO 2230 is not executed. The program defaults to line 2220.

```
2220 IF C$ = "P" THEN DA$(K) = LEFT$(DA$(K),30)
```

Line 2200 is true and sets up DA$(K) so it contains the name in a specific record. The program defaults to line 2220 to concatenate the name and new telephone number into DA$(K).

```
2220 DA$(K) = DA$(K) + "(" + TC$ + ")-" + LEFT$ (PT$,3) + "-" + RIGHT$
     (PT$,4)
```

The program defaults to line 2230 to ask for more corrections. If there are no more corrections, the program defaults to line 2300.

```
2300 GOSUB 1410 : GOTO 20
```

GOSUB 1410 sorts the list and GOTO 20 causes the program to jump to the main menu, PHONE LISTING.

Entry 3 in the main menu is "3.LIST/SEARCH." This selection causes the program to jump to line 3000.

```
3000 HOME : VTAB 3 : INPUT "ENTER 'S' TO SEARCH OR 'L' TO LIST?";Q$ : IF Q$
     <> "L" AND Q$ <> "S" THEN 3000
3010 IF Q$ = "S" THEN 3100
```

The user enters the letter "L" to list the records. Line 3010 is false, so the program defaults to line 3030.

```
3030 FOR J = 1 TO CH(0)
```

Line 3030 is the beginning of a loop to process the records in the list.

```
3040 IF J <> INT((J - 1) /5) * 5 + 1 THEN 3070
3050 IF J <> 1 THEN PRINT : INPUT "!";Q$
```

Lines 3040 and 3050 use negative logic and are both related (Fig. 26-5). On the first pass of the loop, line 3040 is false and the program defaults to

line 3050. On the first pass J = 1, so line 3050 is false and the program defaults to line 3060, to clear the screen and VTAB 3.

```
3060 HOME : VTAB 3
```

```
3070 PRINT "NAME = "; LEFT$ (DA$(J),30) : PRINT SPC( 7); "PHONE # = ";
     RIGHT$ (DA$(J),14) : PRINT
```

Line 3070 is executed and prints the first record. On loop executions 2, 3, 4, and 5, line 3040 is true and the program branches to print records 2, 3, 4, and 5.

J	J – 1	INT((J – 1)/5)	INT((J – 1)/5)*5 + 1	3040	GOES TO 3050		ACTION OF 3050
1	0	0	1	FALSE	3050	FALSE	GOES TO 3060 PRINTS RECORD #1
2	1	0	1	TRUE	3070		PRINTS RECORD #2
3	2	0	1	TRUE	3070		PRINTS RECORD #3
4	3	0	1	TRUE	3070		PRINTS RECORD #4
5	4	0	1	TRUE	3070		PRINTS RECORD #5
6	5	1	6	FALSE	3050	TRUE	INPUTS "!";Q$ PRESS RETURN TO CONTINUE PRINTS RECORD #6
7	6	1	6	TRUE	3070		PRINTS RECORD #7
8	7	1	6	TRUE	3070		PRINTS RECORD #8
9	8	1	6	TRUE	3070		PRINTS RECORD #9

Fig. 26-5. Relationship of lines 3040 and 3050.

On the sixth loop execution, line 3040 is false (6 = 6). The program defaults to line 3050, "IF J < > 1 THEN PRINT : INPUT "!";Q$". On the sixth execution (6 < > 1), and line 3050 is true. The loop execution stops, "!" is printed, and the computer waits for the user to press RETURN to continue printing the list. Line 3080 NEXT J, completes the loop execution.

```
3090 PRINT : INPUT "!";Q$ : GOTO 20
```

Line 3090 stops the program. When the user presses RETURN, the program jumps to line 20, the PHONE LISTING menu.

Selection 3 on the PHONE LISTING menu is ,"3.LIST/SEARCH," causes the program to jump to line 3000. If S for search is selected at line 3000, the program branches to line 3100 to begin the search.

```
3100 HOME : VTAB 3 : HTAB 12 : PRINT "SEARCH SELECTION" : PRINT
```

Line 3100 prints out three selections.

1.NAME SEARCH
2.NUMBER SEARCH
3.RETURN TO MAIN MENU

Selection 1 causes the program to branch to line 3150 to begin the name search.

```
3150 HOME : VTAB 4 : PRINT "ENTER NAME OR FRAGMENT ?" : PRINT : INPUT
     NA$ : L = LEN (NA$) : IF L = 0 THEN 3100
```

The name search is processed in lines 3160 through 3220. For this example, variables are given specific values to make the learning process easier.

NA$	"ABC" — SEARCH FOR ABC
L	LEN(NA$) = 3
DA$(J)	"EDABC" — THIS NAME IS SEARCHED
CL(J)	LENGTH OF DA$(J) IS 5 CHARACTERS
CO	COUNTING VARIABLE TO COUNT THE NUMBER OF MATCHES FOUND IN THE LIST
CH(0)	CH(0) = 7 — THERE ARE 7 RECORDS IN THE LIST
FOR J = 1	TO CH(0) — FOR J = 1 TO 7
FOR K = 1	TO CL(J) - L + 1 — FOR K = 1 TO 3

Fig. 26-6 should be studied in detail to learn the name search routine. The record count is stored in a temporary location, CH(1). In line 3200, CH(0) = CH(1), the record count is stored in CH(0). This step is necessary to preserve the record count before the program jumps to the subroutine at line 10000.

If at line 3100, the user had entered selection 2, "2.NUMBER SEARCH," the program would jump to line 3250 to search for a specific telephone number.

```
3250 HOME : VTAB 6 : HTAB 6 : INPUT "ENTER AREA CODE, PHONE #
     ?";AC$,PN$
3260 IF LEN (AC$ ) = 0 THEN AC$ = DC$
```

Line 3260 allows the user to enter a comma for the area code, if the user wants the default area code of 713. The area code and the telephone number are always the last 14 characters of DA$. The telephone number for a specific record is RIGHT$(DA$(J),14).

```
3270 TC$ = "(" + AC$ + ")-" + LEFT$ (PN$,4)
```

Line 3270 concatenates the area code and the telephone number in the proper format and stores it in the temporary string variable, TC$.

```
3280 FOR J = 1 TO CH(0)
```

Line 3280 sets up the loop that will list the area codes and telephone numbers on the list.

SPECIFIC VARIABLES USED FOR THE NAME SEARCH

NA$	LEN(NA$)	DA$(J)	LEN(DA$(J))	# OF RECORDS	CL(J) – L + 1
ABC	3	EDABC	5	7	3
				FOR J = 1 TO 7	FOR K = 1 TO 3

PROGRAM LINES 3160 — 3220 FOR NAME SEARCH

```
3160  CO = 0 : COUNTS THE NUMBER OF MATCHES IN THE LIST.
      FOR J = 1 TO CH(0) – FOR J = 1 TO 7
3170  FOR K = 1 TO CL(J) – L + 1 – FOR K = 1 TO 3
            IF NA$ < > MID$(DA$(J),     K, L) THEN       3210
               ABC              EDABC
      1st  PASS  ABC (TRUE)  < > EDA    1, 3 K=1, L=3  3210
      2nd  PASS  ABC (TRUE)  < > DAB    2, 3 K=2, L=3  3210
      3rd  PASS  ABC (FALSE) < > ABC    3, 3 K=3, L=3  3190
3190  CH(1) = CH(0) 7 = 7 — THE RECORD COUNT MUST BE STORED BEFORE
      THE GOSUB 10000 OR RECORD COUNT WILL BE WIPED OUT
      CH(0) = J – 1 — SEE Fig. 25-2 OR Fig. 25-3
      GOSUB 10000 — PRINTS OUT THE RECORD
      INPUT Q$ — STOPS THE PROGRAM — PRESS RETURN TO CONTINUE
3200  CH(0) = CH(1) 7 = 7 — CH(1) RESTORES THE RECORD COUNT TO CH(0)
      CO = CO + 1 — INCREMENTS THE COUNT TO DETERMINE IF A MATCH
      OCCURS MORE THAN ONCE IN THE LIST
      GOTO 3220 — PREVENTS THE SAME ITEM ON THE LIST FROM BEING
      MATCHED TWICE
3210  NEXT K
3220  NEXT J — SEARCHES THE NEXT RECORD
```

Fig. 26-6. Name search.

```
3290 IF TC$ < > RIGHT$ (DA$(J),14) THEN 3310
```

All telephone numbers are composed of 14 numeric characters located in the last 14 places in DA$ as RIGHT$(DA$(J),14), so the comparison is relatively simple. It is much simpler than the name search, though not nearly so flexible.

```
3300 CH(1) = CH(0) : CH(0) = J – 1 : GOSUB 10000 : PRINT :
     CH(0) = CH(1) : INPUT Q$ : PRINT
```

The record count is again stored in a temporary location, CH(1), so it will not be lost during the execution of the subroutine at line 10000. The subroutine prints out both the name, area code, and telephone number produced from the number search. On returning from the subroutine, the record count is again stored in CH(0) (CH(0) = CH(1)).

```
3310 NEXT J
```

Each record on the list is searched for the telephone number in question and line 3320, GOTO 3100, causes the program to jump to the "SEARCH SELECTION" menu. Selection 3, "3.RETURN TO MAIN MENU," causes a jump to line 20, the PHONE LISTING menu. Selection 4, "4.SAVE LIST AND END," from the PHONE LISTING menu causes the program to end.

LESSON 27

Formulas

Lesson 27 contains three programs dealing with formulas for (1) decimal to hexadecimal conversion (Fig. 27-1), (2) hexadecimal to decimal conversion (Fig. 27-6), and (3) systematic and efficient output (Fig. 27-9).

"Formula" has many different meanings, but a good definition is "the rule for doing something." A formula is a recipe or a prescription. Formulas have been used and demonstrated in almost every lesson of this book.

The computer, which understands only binary (1 and 0), must use a formula to convert any input to binary. Program computations are carried out in binary, and this binary must be converted to the required type of output. The formulas to make these internal conversions reside in the Applesoft interpreter. These conversion formulas are not readily visible. The formulas to be discussed are the ones you write.

The most efficient way to use a formula in programming is to input the

```
5    REM : DECIMAL TO HEXADECIMAL
10   HOME : VTAB 6
20   INPUT "ENTER DECIMAL INTEGER ?";DEC
30   IF DEC < 1 THEN END
40   DEC =  INT (DEC)
50   HEX = 0:HX$ = "" ""
60   FOR J = 0 TO 15: IF DEC < 16 ↑ J THEN 80
70   NEXT J: PRINT "THE NUMBER IS TOO LARGE" : PRINT : GOTO 20
80   FOR K = J - 1 TO 0 STEP - 1
90   HEX =  INT (DEC / 16 ↑ K)
100  HX$ = HX$ + CHR$ (HEX + 48 + (HEX>9) * 7)
110  DEC = DEC - HEX * 16 ↑ K
120  NEXT K
130  PRINT : HTAB 9: PRINT "HEX DISPLAY IS ";HX$
140  PRINT : GOTO 20

]RUN
ENTER DECIMAL INTEGER ?863
HEX DISPLAY IS 35F
```

Fig. 27-1. Decimal to hexadecimal conversion program.

data in a variable, or variables. The variables in the formula receive the data values, use these values to make computations, and output the information in a variable. In this way, the data entered and the information output can be easily changed and the program remains relatively constant.

In the first program (Fig. 27-1) a decimal value is converted to a hexadecimal value. In the second program (Fig. 27-6) a hexadecimal value is converted to a decimal value. The conversion formulas point out two important aspects of the computer, (1) humans speak decimal, and computers speak hexadecimal, before it is converted to binary, and (2) decimal contains numeric characters, while hexadecimal contains alpha and numeric characters. In the conversion process, decimal is entered as numerics and converted to string arrays. In converting hexadecimal to decimal, the hexadecimal is entered as a string array and converted to numerics.

The decimal system uses a base of 10. The hexadecimal system uses a base of 16. The decimal figure 312 means:

$$\begin{aligned} 3 * 10^2 &= 300 \\ + 1 * 10^1 &= 10 \\ + 2 * 10^0 &= 2 \\ \hline & 312 \end{aligned}$$

The hexadecimal system uses the decimal numbers from 0 through 9, as the first ten digits, and uses the letters A = 10, B = 11, C = 12, D = 13, E = 14, and F = 15, as the last five letters (Fig. 27-2). The hexadecimal number 35F means:

$$\begin{aligned} 3 * 16^2 &= 768 \\ + 5 * 16^1 &= 80 \\ + F * 16^0 &= 15 \\ \hline & 863 \end{aligned}$$

Fig. 27-3 shows the manual system for converting decimal to hexadecimal. Fig. 27-4 details the decimal to hexadecimal conversion of the program statements in relation to the manual conversion of decimal to hexadecimal. Figs. 27-2, and 27-5 should be studied closely before going to the conversion program. Fig. 27-2 presents the relationship between the first fifteen decimal numbers and the first fifteen hexadecimal numbers. Fig. 27-5 shows the ASCII numerics and the character strings they represent. Conversion of string arrays to numeric arrays and conversion of numeric arrays to string arrays was discussed in Lesson 7. The sequence of ASCII characters for ten numerics and twenty-six alpha characters goes from 48 through 90. The character strings represented by the ASCII characters run from zero (0) to zebra (Z). Between the numeric and alpha characters is an intervening group of special characters (58 – 64) that inter-

rupt the chain of continuity. This interruption is very important to understand. It is programmed in line 100 of Fig. 27-1. The hexadecimal number is converted into a string array to accommodate both alpha and numeric characters.

```
100 HX$ = HX$ + CHR$ (HEX + 48 + (HX > 9) * 7)
```

The ASCII number 48 represents the string character zero (0). If the ASCII characters were set so 48 through 57 represented zero (0) through nine (9), and ASCII 58 through 84 represented A through Z, there would be no need for the logical expression, ((HEX > 9) * 7), in line 100. Since ASCII 59 through 64 interrupt the continuity of the numeric and alpha characters, + 48 is necessary to establish base zero for the character string. The + 48 corrects for the offset of the ASCII values.

HEXADECIMAL	DECIMAL
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
A	10
B	11
C	12
D	13
E	14
F	15

Fig. 27-2. Comparing the first sixteen hexadecimal and decimal digits.

NUMBER IN DECIMAL	DIVIDE BY	QUOTIENT	REMAINDER IN DECIMAL	REMAINDER IN HEXADECIMAL
863	863/16	53	15	F
	53/16	3	5	5
	3/16	0	3	3

3 5 F

Fig. 27-3. Manual system for decimal to hexadecimal conversion.

Line 100 converts in this manner: (HEX>9) * 7) is a logical expression used as a bridge between ASCII 48 through 57 (0–9), and ASCII 65 through 90 (A through Z). When HEX is less than 9, the expression is FALSE times *zero (0)*. Zero times 7 = 0. If HEX is greater than 9, the expression is TRUE or one (1). One times 7 = 7. The hexadecimal alpha values A through F are used in the conversion, so only the ASCII values from 65 through 70 are

needed for the conversion from decimal (10 – 15) to hexadecimal (A – F). The following example shows how line 100 works.

```
HEX = 6
HX$ = HX$ + CHR$ (6 + 48 + 0) = CHR$(54)
CHR$(54) = "6"
HEX = 14
HX$ = HX$ + CHR$ (14 + 48 + (1*7) ) = CHR$(69)
CHR$ (69) = "E"
```

In line 50, we see HX$ = " ". This means HX$ is initialized to a null value.

Fig. 27-4 shows three loop executions by each program statement involved in converting decimal 863 to hexadecimal 35F. This was done to

20	INPUT "ENTER DECIMAL INTEGER ?";DEC	863
30	IF DEC < 1 THEN END	
40	DEC = INT(DEC)	If a real number is input, this statement converts it to an integer.
50	HEX = 0 : HX$ = " "	Initializes variables. HX$ is a null string with no characters.
60	FOR J = 0 TO 15 : IF DEC < 16 > J THEN 80	Numbers larger than $1.15*10^{18}$ will not be accepted.
70	NEXT J : PRINT " THE NUMBER IS TOO LARGE"	Sets up the position of DEC – 863 = 16^3
80	FOR K = J–1 TO 0 STEP –1	The largest power is divided 1st. When the J loop checks the size of DEC, J is 1 more than the greatest power when J jumps out of the loop. J–1 is the offset for the correct power. K sets up positional values.
90	HEX = INT(DEC/16 > K)	Divides DEC by the positional value to give the HEX value. 1. INT(DEC/16^2) = 863/16^2 = HEX 3 R95 2. INT(DEC/16^1) = 95/16^1 = HEX 5 R15 3. INT(DEC/16^0) = 15/16^0 = HEX F (15)
100	HX$ = HX$ + CHR$(HEX + 48 + (HEX > 9) *7)	1. 15 + 48 + (1*7) = 70 ASC(70) = F 2. 5 + 48 + (0*7) = 53 ASC(53) = 5 3. 3 + 48 + (0*7) = 51 ASC(51) = 3
110	DEC = DEC – HEX*16 > K	1. DEC = 863 – HEX(3)*16^2 = 768 2. DEC = 95 – HEX(5)*16^1 = 80 3. DEC = 15 – HEX(15)*16^0 = 15
120	NEXT K	
130	PRINT "HEX DISPLAY IS ";HX$	35F

Fig. 27-4. Decimal to hex conversion program statements as related to manual conversion to hex.

show how the individual loop executions compared with the manual conversion.

The formulas in Fig. 27-6 convert hexadecimal to decimal. Since hexadecimal must contain both alpha and numeric characters, the value is entered as a string array. The string array is converted into a numeric variable to print out the decimal number. The hexadecimal number 35F, is entered and converted to the decimal value, 863. The manual conversion is shown in Fig. 27-7. The program statements and manual conversion are detailed in Fig. 27-8. The three loop executions are included with each program statement to view the change in values during the computation.

ASC	CHR$	
48	0	CONVERSION OF A NUMERIC ARRAY TO
49	1	A STRING ARRAY
50	2	ASC(65) = A
51	3	
52	4	CONVERSION OF A NUMERIC ARRAY TO
53	5	A STRING ARRAY
54	6	CHR$(A) = 65
55	7	
56	8	
57	9	
58	:	
59	;	
60	<	
61	=	
62	>	
63	?	
64	@	
65	A	
66	B	
67	C	
68	D	
69	E	
70	F	
90	Z	

Fig. 27-5. ASCII characters.

The hexadecimal value is entered as a string array in line 10.

```
10 INPUT "ENTER HEX VALUE ?";Q$
```

Line 40 converts the hexadecimal value entered into Q$ into a numeric variable by the ASC function. In conjunction with the loop beginning at line 30, line 40 processes the characters, one at a time.

```
J = 1 : HEX = 3 (CHR$(51))
J = 2 : HEX = 5 (CHR$(53))
J = 3 : HEX = F (CHR$(70))
```

```
1   REM : HEXADECIMAL TO DECIMAL
5   HOME : VTAB 6
10  INPUT "ENTER HEX VALUE T";Q$
20  IF LEN (Q$) = 0 THEN END
30  DEC = 0: FOR J = 1 TO LEN (Q$)
40  HX = ASC ( MID$ (Q$,J,1)): IF (HX > 47 AND HX < 58) OR (HX > 64
      AND HX < 71) THEN DEC = DEC * 16 + HX - 48 - (HX > 58) * 7
50    NEXT J
60    PRINT : PRINT "DEC = ";DEC: PRINT
70    GOTO 10

]RUN
ENTER HEX VALUE ?35F
DEC = 863
```

Fig. 27-6. Hexadecimal to decimal conversion program.

If the hexadecimal character is a numeric value, it must be between the ASCII values of 48 through 57. If the hexadecimal character is an alpha character, it must be between 65 through 90. The statement in line 40, IF (HX > 47 AND HX < 58) OR (HX > 64 AND HX < 71) THEN DECX = DEC * 16 + HX - 48 - (HX > 58) * 7, is similar to a summing statement (for example, DEC = DEC + HX). This statement converts from a hexadecimal value entered to the decimal number (Fig. 27-8) in that it takes the sum of the computed DEC and adds it to DEC on each loop execution.

Line 50 is the foot of the loop, and line 60 displays the numeric number that has been converted from the hexadecimal value.

```
60 PRINT "DEC = ";DEC
```

Line 70 gives the user a chance to enter another hexadecimal value.

The formulas in the OGIVE Program (Fig. 27-9) produce systematic and efficient output and are applied to a statistical problem. The statistical problem randomly inputs student grades. The grades are output in seventeen different ranges. The grades are used to produce an ogive of cumulative distribution.

The definition of cumulative distribution is "heaped up," or "growing in amount."

HEXADECIMAL NUMBER	POSITIONAL VALUE	MULTIPLY BY
35F	F * 16^0	F * 1 = 15
	5 * 16^1	5 * 16 = 80
	3 * 16^2	3 * 256 = 768
		863

Fig. 27-7. Manual system for hexadecimal to decimal conversion.

5 HOME : VTAB 5	
10 INPUT "ENTER HEX VALUE ?";Q$	35F — HEX is input in a string array. It may contain alpha and numeric characters.
20 IF LEN(Q$) = 0 THEN END	
30 DEC = 0 : FOR J = 1 TO LEN(Q$)	Initialized DEC to zero. Sets up loop to check each character.
40 HX = ASC(MID$(Q$,J,1)) :	Converts the string array to a numeric array one character at a time. Picks off the 1st, 2nd, and 3rd character.
IF (HX >47 AND HX <58) OR (HX >64 AND HX < 71) THEN	ASC numerics 48 through 57 represent the numbers zero to 9. ASC numbers 64 through 70 represent the letters A - F (Fig. 27-5).
DEC = DEC*16+HX−48−(HX>58)*7 1. DEC=0*16+51−48−(51>58)=3 DEC=3 (0 * 7) 2. DEC=3*16+48−53−(53>58)=53 DEC=53 (0 * 7) 3. DEC=53*16=848+70−48−(70>58) DEC = 863 (1 * 7)	Positional value 35F 16^0 * 15 = 15 16^1 * 5 = 80 16^2 * 3 = 768 863
50 NEXT J	
60 PRINT : PRINT "DEC = ";DEC	DEC = 863
70 GOTO 10	

Fig. 27-8. Comparing program statements and manual conversion.

```
5     REM : OGIVE PROGRAM
10    HOME
20    DIM CG(17)
30    GOSUB 800
40    T = 0: FOR J = 1 TO 17
50    IF J < > INT ((J - 1) / 5) * 5 + 1 THEN 80
60    IF J < > 1 THEN PRINT "OGIVE COUNT=";T;" OF 80 =";T / 80;"%":
      INPUT "!";Q$
70    PRINT "RANGE#     BASE     TOP     COUNT"
80    R = J: GOSUB 900:UL = RL:R = J - 1: GOSUB 900:LL
      = RL + 1: IF LL = 1 THEN LL = 0
90    PRINT SPC(  3);J;: HTAB 13: PRINT LL;: HTAB 20: PRINT UL;: HTAB 30:
      PRINT CG(J)
100   T = T + CG(J): NEXT J
110   PRINT "OGIVE COUNT=";T;" OF 80 =";T / 80;"%"
120   END
800   FOR J = 1 TO 80:SG = INT ( RND(1.0) * 101): IF SG = 0 THEN
      SG = 1
```

Fig. 27-9. OGIVE program.

```
810  IF SG > 32 THEN 830
820  CG((SG - 1) / 16 + 1) = CG((SG - 1) / 16 + 1) + 1: GOTO 880
830  IF SG > 64 THEN 850
840  CG(3 + (SG - 33) / 8) = CG(3 + (SG - 33) / 8) + 1:: GOTO 880
850  IF SG > 92 THEN 870
860  CG(7 + (SG - 65) / 4) = CG(7 + (SG - 65) / 4) + 1:: GOTO 880
870  CG(14 + (SG - 93) / 2) = CG(14 + (SG - 93) / 2) + 1
880  NEXT J: RETURN
900  RL = R * 16 - (R > 2) * (R - 2) * 8 - (R > 6)
       * (R - 6) * 4 - (R > 13) * (R - 13)
       * 2
910  RETURN


RUN
 RANGE #      BASE      TOP      COUNT
     1          0        16        10
     2         17        32        14
     3         33        40         7
     4         41        48         7
     5         49        56        11
OGIVE COUNT=49 OF 80 =.6125%
!
 RANGE #      BASE      TOP      COUNT
     6         57        64         7
     7         65        68         3
     8         69        72         2
     9         73        76         4
    10         77        80         2
OGIVE COUNT=67 OF 80 =.8375%
!
 RANGE #      BASE      TOP      COUNT
    11         81        84         1
    12         85        88         3
    13         89        92         1
    14         93        94         3
    15         95        96         2
OGIVE COUNT=77 OF 80 =.9625%
!
 RANGE #      BASE      TOP      COUNT
    16         97        98         1
    17         99       100         2
OGIVE COUNT=80 OF 80 =1%
```

Fig.27-9–cont. OGIVE program.

Ogive is a distribution curve or graph in which the frequencies are cumulative. A teacher takes the 80 student grades, places them in 17 ranges of varying widths, adds the number of grades in each range, and plots them on a graph (Fig. 27-10).

The program in Fig. 27-9 uses a RND function to produce the 80 grades. The grades are placed in ranges by a formula and the ranges are computed by a formula. A formula is used to break the printout into a heading and five ranges and then the printout continues. The ogive is produced by hand to demonstrate the use of the output (Fig. 27-10). One point to note — since the grades are produced by the RND function, *NO TWO PROGRAM RUNS OR GRAPHS WILL BE THE SAME.* The ogive does not pattern itself in the manner of the standard distribution curve.

Line 20 DIM CG(17), dimensions the numeric array for the seventeen grade ranges.

Line 30 branches to the subroutine at line 800, which will produce the 80 student grades. Lines 810 through 880 place them in the proper range.

In line 800, FOR J = 1 TO 80, is the beginning of a loop to produce the 80 student grades. SG = INT(RND(1.0) * 101) produces student grades from 0 to 100. The purpose of IF SG = 0 THEN SG = 1 is a special case. If the RND function produces a student grade of zero (0), the grade has to be modified to fit the program pattern. The grade ranges have no place for a grade of zero (0), so zero grade will be replaced with a grade of one (1).

Lines 810, 830, and 850 set up to divide the ranges into four divisions, (1) two ranges containing 16 scores each, (2) four ranges containing 8 scores each, (3) six ranges containing 4 scores each, and (4) four ranges containing 2 scores each (Figs. 27-11 and 27-12).

```
810 IF SG > 32 THEN 830      (handles grades from 1 to 32)
830 IF SG > 64 THEN 850      (handles grades from 33 to 64)
                             (handles grades 65 to 92 if statement is false)
850 IF SG > 92 THEN 870      (handles grades 93 to 100 if statement is true)
```

If line 810, IF SG > 32 THEN 830, is false, the program defaults to line 820 to tabulate the grades in the first two grade ranges from zero (0) to 32.

```
820 CG( (SG - 1) / 16 + 1) = CG( (SG - 1) / 16 - 1) / 16 + 1) + 1 : GOTO 880
```

The summing statement in line 820 sums the number of grades in each of the first two grade ranges (Fig. 27-12).

There are statements similar to the statement at line 820 at lines 840, 860, and 870. These statements compute and total the number of student grades in each range (Fig. 27-12).

```
830 IF SG > 64 THEN 850
```

If line 830 is false, line 840 increments the number of grades in the proper grade range from 33 to 64 (Fig. 27-12).

```
850 IF SG > 92 THEN 870
```

If line 850 is false, line 860 increments the number of grades in the proper

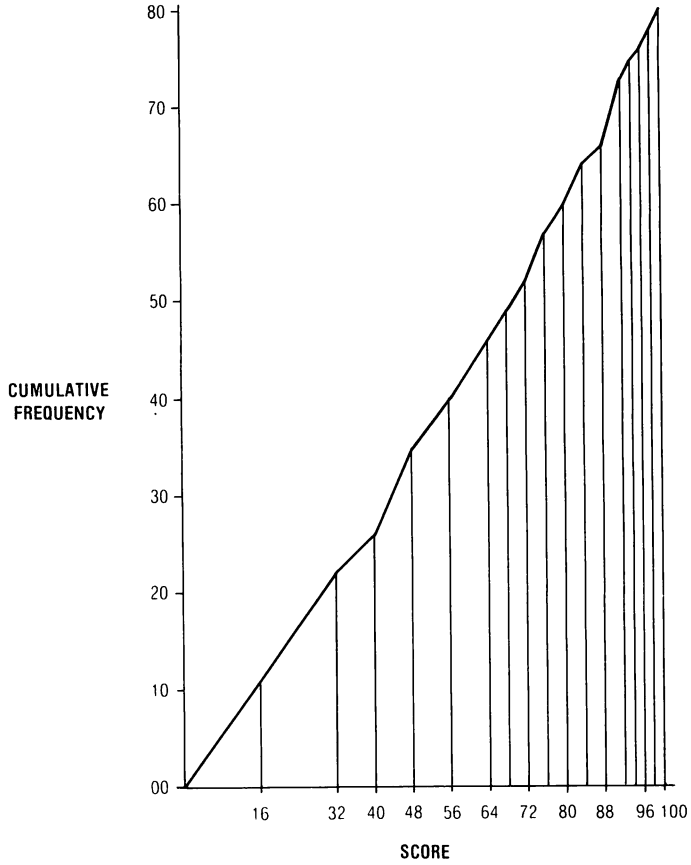


Fig. 27-10. OGIVE of the distribution of 80 scores.

range from 65 to 92. If line 850 is true, line 870 increments the number of grades in the proper range from 93 to 100 (Fig. 27-12).

```
880 NEXT J : RETURN
```

NEXT J completes the loop to input 80 student grades and to place them in the proper range, and RETURN causes the subroutine to return to line 40.

```
40 T = 0 : FOR J = 1 TO 17
```

The totaling variable is initialized to zero, and the 17 grade ranges are placed in a loop structure for processing.

```
50 IF J < > INT((J - 1)/5)*5 + 1 THEN 80
60 IF J < > 1 THEN PRINT "OGIVE COUNT = ";T;" OF 80 = ";T/80;"%" : INPUT
   Q$
```

Lines 50 and 60 control the printout. The printout changes its routine every five (5) lines. The details of the RUN can be viewed in Fig. 27-9. Statements similar to the statements in lines 50 and 60 are routinely used formulas to control printouts to the screen.

```
70 PRINT "RANGE# BASE TOP COUNT"
```

Line 70 prints the headings for each column in the printout.

```
80 R = J : GOSUB 900 : UL = RL : R = J - 1 : GOSUB 900 : LL = RL + 1 : IF LL = 1
THEN LL = 0
```

R = J is used as a temporary storage for the range loop J. If J is not stored, it will be lost during the jump to, or return from the subroutine.

J	LL(BASE)	UL(TOP)	SPREAD	NO. OF RANGES
1	0	16	16	2
2	17	32	16	
3	33	40	8	4
4	41	48	8	
5	49	56	8	
6	57	64	8	
7	65	68	4	7
8	69	72	4	
9	73	76	4	
10	77	80	4	
11	81	84	4	
12	85	88	4	
13	89	92	4	
14	93	94	2	4
15	95	96	2	
16	97	98	2	
17	99	100	2	

Fig. 27-11. Ranges and score points.

The branch to the subroutine at line 900 computes the range level (RL) by using a logical comparison of all ranges (Fig. 27-13).

```
900 RL = R * 16 - (R>2) * (R - 2) * 8 - (R>6) * (R -6) * 4 - (R>13) * (R - 13) * 2
```

R * 16 - (R > 2), the range number times 16 (# of grade points in the first two ranges), separates the first two ranges. The logical expression (- (R > 2)), separates the first two grade ranges based on a spread of 16 points. If R is less than two (2), the statement is true, or one (1). When the logical expression is true, it is activated for the first two grade loop values (R = J) If R is greater than two (2), the logical expression is false, or zero (0). When the expression is false (0), zero (0) times 16 = 0, and the logical expression

```
810  IF SG > 32 THEN 830
820  CG((SG - 1)/16 + 1) = CG((SG - 1)/16 + 1) + 1 : GOTO 880
                 GRADE |  (SG - ?)  | CG((SG - ?)/ ? + ?
                   1   |     0      |        1
                  17   |    16      |        2
                  32   |    31      |        2
830  IF SG > 64 THEN 850
840  CG(3 + (SG - 33)/8) = CG(3 + (SG - 33)/8) + 1 : GOTO 880
                  33   |     0      |        3
                  41   |     8      |        4
                  49   |    16      |        5
                  57   |    24      |        6
                  64   |    31      |        6
850  IF SG > 92 THEN 870
860  CG(7 + (SG - 65)/4) = CG(7 + (SG - 65)/4 + 1 : GOTO 880
                  65   |     0      |        7
                  69   |     4      |        8
                  73   |     8      |        9
                  77   |    12      |       10
                  81   |    16      |       11
                  85   |    20      |       12
                  89   |    24      |       13
                  92   |    27      |       13
870  CG(14 + (SG - 93)/2) = CG(14 + (SG - 93)/2 + 1
                  93   |     0      |       14
                  95   |     2      |       15
                  97   |     4      |       16
                  99   |     6      |       17
                 100   |     7      |       17
```

Fig. 27-12. Grades and ranges.

is not included in any other computation (Fig. 27-14). Line 900 produces the range levels from one (1) through 17. This is the power of using formulas in programming.

Lines 80 and 900 work in conjunction to produce the upper level (TOP), UL = RL, of the range, and the lower level (BASE), LL = RL + 1, of the range.

When the subroutine at line 900 has completed processing each range level, it returns to line 80, UL = RL, to store the range level computed, into the upper level variable.

R = J − 1 decrements the value of the loop variable stored in R, so the upper level and the lower level remain at the same value. The program jumps to the subroutine at line 900 to process the range level again. When the subroutine at line 900 returns to line 80, the range limit is incremented by one (1), (LL = RL + 1), to produce the lower limit (BASE) of the grade range. Line 80 and the formula at line 900 have now produced the upper and lower limits of the grade range from 1 to 100. The grade range runs from

```
40 FOR J = 1 TO 17
80 R = J : GOSUB 900 : UL = RL : R = J - 1 : GOSUB 900 :
   LL = RL + 1 : IF LL = 1 THEN LL = 0
```

RL = R * 16 - (R > 2)	LL = RL + 1	IF LL = 1 THEN LL = 0 J - 1
		0
1 * 16 = 16	0	1
2 * 16 = 32	17	2
RL = (R - 2) * 8 - (R > 6)		
3 - 2 = 1 * 8 = 8	33	3
4 - 2 = 2 * 8 = 16	41	4
5 - 2 = 3 * 8 = 24	49	5
6 - 2 = 4 * 8 = 32	57	6
RL = (R - 6) * 4 - (R > 13)		
7 - 6 = 1 * 4 = 4	65	7
8 - 6 = 2 * 4 = 8	69	8
9 - 6 = 3 * 4 = 12	73	9
10 - 6 = 4 * 4 = 16	77	10
11 - 6 = 5 * 4 = 20	81	11
12 - 6 = 6 * 4 = 24	85	12
13 - 6 = 7 * 4 = 28	89	13
RL = (R - 13) * 2		
14 - 13 = 1 * 2 = 2	93	14
15 - 13 = 2 * 2 = 4	95	15
16 - 13 = 3 * 2 = 6	97	16
17 - 13 = 4 * 2 = 8	99	17

Fig. 27-13. Compute lower limit range.

one (1) to 100, so a special case must be accommodated to produce a grade range from zero (0) to 100.

The last statement in line 80 handles this special case. IF LL = 1 THEN LL = 0 converts the lower level of the first grade range from one (1) to zero (0).

```
90 PRINT SPC(3);J; : HTAB 13 : PRINT LL;: HTAB 20 : PRINT UL;: HTAB 30 : PRINT
   CG(J)
```

Line 90 prints the output information under the proper headings.

```
100 T = T + CG(J) : NEXT J
```

T = T + CG(J) totals the number of grades in each grade range, and NEXT J completes the loop structure, so all 17 grade ranges are produced.

```
110 PRINT "OGIVE COUNT = ";T;" OF 80 = ";T/80;"%"
```

Line 110 prints out the final ogive count as a check to determine if all 80 student grades have been input and processed. The ogive count was printed

```
40  FOR J = 1 TO 17
80  R = J : GOSUB 900 : UL = RL : R = J - 1
```

RL =	R * 16 - (R > 2)	UL = RL	R = J - 1
	1 * 16 = 16	16	1
	2 * 16 = 32	32	2
RL = (R - 2)	* 8 - (R > 6)		
	3 - 2 = 1 * 8 = 8	40	3
	4 - 2 = 2 * 8 = 16	48	4
	5 - 2 = 3 * 8 = 24	56	5
	6 - 2 = 4 * 8 = 32	64	6
RL = (R - 6)	* 4 - (R > 13)		
	7 - 6 = 1 * 4 = 4	68	7
	8 - 6 = 2 * 4 = 8	72	8
	9 - 6 = 3 * 4 = 12	76	9
	10 - 6 = 4 * 4 = 16	80	10
	11 - 6 = 5 * 4 = 20	84	11
	12 - 6 = 6 * 4 = 24	88	12
	13 - 6 = 7 * 4 = 28	92	13
RL = (R - 13)	* 2		
	14 - 13 = 1 * 2 = 2	94	14
	15 - 13 = 2 * 2 = 4	96	15
	16 - 13 = 3 * 2 = 6	98	16
	17 - 13 = 4 * 2 = 8	100	17

Fig. 27-14. Compute upper limit range.

for each of the five ranges by line 60. The percentage is printed out as 1%. If "T/80" had been multiplied by 100 ((T/80)*100)) the percentage would have been 100%. Line 120 ENDs the program.

Formulas produce fast, efficient, orderly output. The program written without formulas would take approximately six times the number of lines to solve the same problem. When possible, use formulas to save memory space, increase speed, and efficiency, and to systematize output.

LESSON 28

Cash Flow

Double subscripted arrays were introduced in Lesson 14. In that lesson, a business program was presented that accepted inputs of gross income and expenses and produced outputs of net income and column totals.

Double subscripted arrays may be thought of as an arrangement of numbers in rows and columns. Values in a double subscripted array table may be accessed by specifying the row and column of the array. For example, CF(2,3) accesses the value stored in row 2, column 3, of the CF array. The size of the table is determined by the DIMension statement. Each element of the table can be manipulated as if it were a simple variable. Each element of the table can be operated on by the arithmetic operators.

The cash flow program (Fig. 28-1) does an analysis of an investment in income producing property to assist a potential buyer to determine if the purchase will be profitable. Information concerning the value of the property, loan amount, length of loan, interest rate, net operating income, cash equity, and personal income, are entered (Fig. 28-2). The data entered is processed by the program (Fig. 28-3). The program outputs financial information and ratios to aid in the decision whether to purchase the property (Fig. 28-4). The cash flow program was written to be used as a valid investment tool. It was written according to accounting and real estate guidelines.

```
10    REM : CASH FLOW PROGRAM TO
20    REM : DETERMINE INVESTMENT
30    REM : YIELDS ON INCOME PROP-
40    REM : ERTY:::COPYWRITED 1980
50    REM : BRIAN D. BLACKWOOD AND
60    REM : GEORGE H. BLACKWOOD
70    REM : 7020 BURLINGTON
80    REM : BEAUMONT, TEXAS 77706
90    REM : 713-866-6141
140   DEF FN R(Z) = ( INT (Z * 1000 + .5) / 1000)
145   DEF FN A(X) = INT ((X) + .5)
```

Fig. 28-1. Cash flow program and run.

```
150  DIM H1$(15),NUM(2),H4(15,1)
160  HOME : VTAB 3
200  H1$ = "NET OP INC     LOAN VAL INT RATE     LOAN LEN"
210  PRINT H1$
220  VTAB 4: INPUT " ";NOI: VTAB 4: HTAB 12: INPUT " ";PV: VTAB 4:
     HTAB 23: INPUT " ";I: VTAB 4: HTAB 33: INPUT " ";LL
240  H2$ = "ASSET COST     ASSET LIFE SALVAGE VALUE": VTAB 5: PRINT H2$
245  VTAB 6: HTAB 1: INPUT " ";CA
250  VTAB 6: HTAB 15: INPUT " ";LA: VTAB 6: HTAB 26: INPUT " ";SV
255  DIM CF(LL + 1,14)
260  H3$ = "RATE OF DEP     YRLY INCOME CASH EQUITY"
270  VTAB 7: PRINT H3$
280  VTAB 8: HTAB 4: INPUT " ";RD : VTAB 8: HTAB 14: INPUT " ";YI:
     VTAB 8: HTAB 26: INPUT " ";CE
290  IF I > = 1 THEN I = I / 100 : GOTO 290
292  IF RD < 1 THEN RD = 100
310  GOSUB 8000
320  DP = 1 / LA:DEP = (RD / 100) * DP:BV = CA - SV:TB = BV
360  CF(1,8) = 0:CF(1,9) = 0:NE = YI:CF(0,14) = 0:J = 1: GOSUB 7000
365  CF(0,14) = IRS
370  GOSUB 9000
380  AN = PV / DF: FOR J = 1 TO LL
385  CF(J,2) = NOI
390  I1 = PV * I:CF(J,3) = I1
395  CF(LL + 1,3) = CF(LL + 1,3) + CF(J,3)
410  PR = AN - I1:CF(J,4) = PR
415  CF(LL + 1,4) = CF(LL + 1,4) + CF(J,4)
420  BR = PV - PR:PV = BR
425  REM : COMPUTE CASH FLOW
430  CF(J,5) = CF(J,2) - (CF(J,3) + CF(J,4))
435  CF(LL + 1,5) = CF(LL + 1,5) + CF(J,5)
462  D1 = TB * DEP
464  CF(J,6) = D1
466  CF(LL + 1,6) = CF(LL + 1,6) + CF(J,6)
470  TB = TB - D1
500  CF(J,7) = CF(J,3) + CF(J,6)
505  CF(LL + 1,7) = CF(LL + 1,7) + CF(J,7)
520  CF(J,8) = CF(J,2) - CF(J,7)
523  IF CF(J,8) < 0 THEN CF(J,8) = 0
525  CF(LL + 1,8) = CF(LL + 1,8) + CF(J,8)
540  CF(J,9) = CF(J,2) - CF(J,7)
545  CF(J,9)= ( SGN (CF(J,9)) - 1) * CF(J,9) / 2
547  CF(LL + 1,9) = CF(LL + 1,9) + CF(J,9)
560  CF(J,10) = CF(J,8) * CF(J - 1,14)
565  CF(LL + 1,10) = CF(LL + 1,10) + CF(J,10)
580  CF(J,11) = CF(J,9) * CF(J - 1,14)
585  CF(LL + 1,11) = CF(LL + 1,11) + CF(J,11)
590  CF(J,0) = YI + CF(J,10) - CF(J,11)
595  NE = YI + CF(J,10) - CF(J,11) : GOSUB 7000:CF(J,14) = IRS
600  CF(J,12) = CF(J,5) + CF(J,11) - CF(J,10)
```

Fig.28-1–cont. Cash flow program and run.

```
610   CF(LL + 1,12) = CF(LL + 1,12) + CF(J,12)
660   CF(J,13) = CF(J,12) + CF(J,4)
670   CF(LL + 1,13) = CF(LL + 1,13) + CF(J,13): NEXT J:CF(LL + 1,2) =
      CF(1,2) * LL
675   CF(J,14) = IRS
680   VTAB 9: INPUT "YRS OWNED=";YO
690   IF YO = 0 THEN 900
695   IF YO < 1 OR YO > LL THEN 680
697   VTAB 9: CALL -958
700   FOR K = 3 TO 13
710   CF(0,K) = 0
720   FOR J = 1 TO YO
730   CF(0,K) = CF(0,K) + CF(J,K)
740   NEXT J,K
750   CF(0,2) = YO * CF(1,2)
760   IRS = CF(YO,14)
770   NE = CF(YO,0)
780   VTAB 9: PRINT "YRS OWNED=";YO; TAB( 16);"END INC=";
      FN A(CF(YO,0));"(";IRS;")"
810   VTAB 11: HTAB 16: PRINT YO;" YR TOT ";YO:"YR AV";
      TAB( 35);"YIELD"
820   VTAB 12: PRINT "CASH FLOW"; TAB(18); FN A(CF(0,5)); TAB ( 28);
      FN A(CF(0,5) / YO); TAB( 35); FN R(CF(0,5) / (YO * CE)
830   VTAB 14: PRINT "TAX SAV'S"; TAB( 11); FN A(CF(0,11))
840   VTAB 16: PRINT "TAX PAY"; TAB( 11); FN A(CF(0,10)); TAB( 19);
      FN A(CF(0,11) - CF(0,10))
850   VTAB 18: PRINT "CASH BENEFITS"; TAB( 18); FN A(CF(0,12)); TAB( 28);
      FN A(CF(0,12) / YO); TAB( 35); FN R(CF(0,12) / (YO * CE))
860   VTAB 19: PRINT "ADD:PRINCIPAL"; TAB( 18); FN A(CF(0,4))
870   VTAB 21: PRINT "TOTAL CASH AND"
880   VTAB 22: PRINT "AMORTIZATION"; TAB( 18); FN A(CF(0,13)); TAB( 28);
      FN A(CF(0,13) / YO); TAB( 35); FN R(CF(0,13) / (YO * CE))
890   GOTO 680
900   H1$(0) = " 0.INCOME":H4(0,0) = 6:H4(0,1) = 0:H1$(1) = " 1.TAX
      PAYABLE":H4(1,0) = 3:H4(1,1) = 7
910   H1$(2) = " 2.NET OP INCOME":H4(2,0) = 6:H4(2,1) = 6:H1$(3) =
      " 3.INTEREST":H4(3,0) = 8:H4(3,1) = 0
920   H1$(4) = " 4.PRINCIPAL":H4(4,0) = 9:H4(4,1) = 0:H1$(5) = " 5.CASH
      FLOW":H4(5,0) = 4:H4(5,1) = 4
930   H1$(6) = "6.TOT DEPRECIATION ":H4(6,0) = 9:H4(6,1) = 7:H1$(7) =
      " 7.INC TAX DEDUCTS":H4(7,0) = 7:H4(7,1) = 7
940   H1$(8) = "8.TAXABLE INC.":H4(8,0) = 7:H4(8,1) = 4:H1$(9) = "
      9.TAXABLE LOSS":H4(9,0) = 7:H4(9,1) = 4
950   H1$(10) = "10.TAX PAYABLE":H4(10,0) = 3:H4(10,1) = 7:H1$(11) =
      "11.TAX SAVINGS":H4(11,0) = 3:H4(11,1) = 7
960   H1$(12) = "12.CASH AVAILABLE":H4(12,0) = 4:H4(12,1) = 9:H1$(13) =
      "13.TOTAL BENEFITS":H4(13,0) = 5:H4(13,1) = 8
965   H1$(14) = "14.TAX BRACKET":H4(14,0) = 3:H4(14,1) = 7:H1$(15) =
      "15.ANNUAL PAYMENT":H4(15,0) = 6:H4(15,1) = 7
970   CALL -936: HTAB 14: PRINT "TABLE LISTING":PRINT : PRINT "ENTER 3
```

Fig.28-1–cont. Cash flow program and run.

```
      COLUMN VALUES FOR TABLE PRINT''
 980  FOR J = 0 TO 14 STEP 2: PRINT H1$(J); TAB(  19);H1$(J + 1): NEXT J:
      PRINT
 990  FOR N = 0 TO 2: PRINT ''COLUMN '';N + 1;'' = '';: INPUT NUM(N):
      IF NUM(N) < 0 OR NUM(N) > 15 THEN 1010
1000  NEXT N
1010  N = N - 1: IF N < 0 THEN 3000
1020  FOR L = 1 TO LL
1030  IF L < > INT ((L - 1) / 15) * 15 + 1 THEN 1080
1040  IF L < > 1 THEN INPUT ''!'';A$
1050  CALL -936: PRINT ''YEARS'';: FOR J = 0 TO N: HTAB (J + 1) * 10:M
      = NUM(J): PRINT MID$(H1$(M),4,H4(M,0));: NEXT J: PRINT
1060  FOR J = 0 TO N: HTAB (J + 1) * 10:M = NUM(J): IF H4(M,1) > 0
      THEN PRINT RIGHT$ (H1$(M),H4(M,1));
1070  NEXT J: PRINT : PRINT
1080  PRINT L;: FOR J = 0 TO N
1090  HTAB (J + 1) * 10:M = NUM(J): IF M = 14 THEN 1110
1095  IF M < > 15 THEN 1100
1096  PRINT FN A(AN);: GOTO 1120
1100  PRINT FN A(CF(L,M));: GOTO 1120
1110  PRINT FN R(CF(L,M));
1120  NEXT J: PRINT : NEXT L
1130  INPUT ''!'';A$: GOTO 970
3000  END
7000  RESTORE
7010  BS = 0
7020  READ UL,BF,IRS
7030  IF NE > = UL AND UL < > 0 THEN BS = UL: GOTO 7020
7040  CF(J,14) = IRS
7050  CF(J,1) = BF + CF(J - 1,14) * (CF(J,0) - BS)
7100  DATA 3400,0,0
7110  DATA 5500,0,.14
7120  DATA 7600,294,.16
7130  DATA 11900,630,.18
7140  DATA 16000,1404,.21
7150  DATA 20200,2265,.24
7160  DATA 24600,3273,.28
7170  DATA 29900,4505,.32
7180  DATA 35200,6201,.37
7190  DATA 45800,8162,.43
7200  DATA 60000,12720,.49
7210  DATA 85600,19678,.54
7220  DATA 109400,33502,.59
7230  DATA 162400,47544,.64
7240  DATA 215400,81464,.68
7250  DATA 0,117504,.70
7900  RETURN
8000  FOR J = 3 TO 13
8010  CF(LL + 1,J) = 0
8020  NEXT J
```

Fig.28-1–cont. Cash flow program and run.

```
8030  RETURN
9000  FOR J = 1 TO LL
9010  DF = DF + 1 / (1 + I) ↑ J
9020  NEXT J
9030  RETURN

RUN
NET OP INC   LOAN VAL   INT RATE   LOAN LEN
  80000
               600000
                           12
                                       25
ASSET COST          ASSET LIFE      SALVAGE VALUE
  80000
                        40
                                     50000
RATE OF DEP         YRLY INCOME     CASH EQUITY
  200
                      40000
                                     200000

YRS OWNED=1
YRS OWNED=1     END INC=28495(.29)
                    1 YR TOT        1 YR AV        YIELD
CASH FLOW           3500            3500           .018
TAX SAV'S 11505
TAX PAY 0          11505
CASH BENEFITS      15005            15005          .075
ADD:PRINCIPAL       4500
TOTAL CASH AND
AMORTIZATION       19505            19505          .098
YRS OWNED=2
YRS OWNED=2     END INC=32145(.33)
                    2 YR TOT        2 YR AV        YIELD
CASH FLOW           7000            3500           .018
TAX SAV'S 19360
TAX PAY 0          19360
CASH BENEFITS      26360            13180          .066
ADD:PRINCIPAL       9540
TOTAL CASH AND
AMORTIZATION       35900            17950          .09
YRS OWNED=10
YRS OWNED=10    END INC=37014(.39)
                   10 YR TOT        10 YR AV       YIELD
CASH FLOW          35000            3500           .018
TAX SAV'S 63463
TAX PAY 0          63463
CASH BENEFITS      98463            9846           .049
```

Fig.28-1–cont. Cash flow program and run.

```
ADD:PRINCIPAL          78969
TOTAL CASH AND
AMORTIZATION          177432          17743          .089
YRS OWNED=20
YRS OWNED=20    END INC=52367(.44)
                    20 YR TOT       20 YR AV        YIELD
CASH FLOW             70000           3500          .018
TAX SAV'S  66253
TAX PAY  44386  21867
CASH BENEFITS         91867           4593          .023
ADD:PRINCIPAL        324235
TOTAL CASH AND
AMORTIZATION          416102          20805          .104
YRS OWNED=0
                                TABLE LISTING
ENTER 3 COLUMN VALUES FOR TABLE PRINT
 0. INCOME              1. TAX W/O INVEST
 2. NET OP INCOME       3. INTEREST
 4. PRINCIPAL           5.CASH FLOW
 6. TOT DEPRECIATION    7. INC TAX DEDUCTS
 8. TAXABLE INC.        9. TAXABLE LOSS
10. TAX WITH INVEST    11. TAX SAVINGS
12. CASH AVAILABLE     13. TOTAL BENEFITS
14. TAX BRACKET        15. ANNUAL PAYMENT
COLUMN 1 = ?0
COLUMN 2 = ?1
COLUMN 3 = ?2
                                TAX W/O          NET OP
 YEARS          INCOME          INVEST          INCOME
   1             28495           5556            80000
   2             32145           6225            80000
   3             31849           6217            80000
   4             32631           6475            80000
   5             33412           6733            80000
   6             34197           6992            80000
   7             34989           7253            80000
   8             35796           7520            80000
   9             36008           7638            80000
  10             37014           8031            80000
  11             38059           8438            80000
  12             39151           8864            80000
  13             40300           9312            80000
  14             41515           9786            80000
  15             42810          10291            80000
   !

                                TAX W/O          NET OP
 YEARS          INCOME          INVEST          INCOME
  16            44195           10831           80000
```

Fig.28-1–cont. Cash flow program and run.

```
 17          45687          11413          80000
 18          47300          12042          80000
 19          50212          13398          80000
 20          52367          14346          80000
 21          54725          15384          80000
 22          57312          16522          80000
 23          60160          17775          80000
 24          65950          20621          80000
 25          69818          22516          80000
  !
```

```
                              TABLE LISTING
ENTER 3 COLUMN VALUES FOR TABLE PRINT
 0. INCOME                    1. TAX W/O INVEST
 2. NET OP INCOME             3. INTEREST
 4. PRINCIPAL                 5. CASH FLOW
 6. TOT DEPRECIATION          7. INC TAX DEDUCTS
 8. TAXABLE INC.              9. TAXABLE LOSS
10. TAX WITH INVEST          11. TAX SAVINGS
12. CASH AVAILABLE           13. TOTAL BENEFITS
14. TAX BRACKET              15. ANNUAL PAYMENT
COLUMN 1 = ?3
COLUMN 2 = ?4
COLUMN 3 = ?5

                                                  CASH
 YEARS          INTEREST          PRINCIPAL      FLOW
   1             72000              4500          3500
   2             71460              5040          3500
   3             70855              5645          3500
   4             70178              6322          3500
   5             69419              7081          3500
   6             68569              7931          3500
   7             67618              8882          3500
   8             66552              9948          3500
   9             65358             11142          3500
  10             64021             12479          3500
  11             62524             13976          3500
  12             60847             15653          3500
  13             58968             17532          3500
  14             56864             19636          3500
  15             54508             21992          3500
   !

                                                  CASH
 YEARS          INTEREST          PRINCIPAL      FLOW
  16             51869             24631          3500
  17             48913             27587          3500
  18             45603             30897          3500
  19             41895             34605          3500
```

Fig.28-1–cont. Cash flow program and run.

20	37743	38757	3500
21	33092	43408	3500
22	27883	48617	3500
23	22049	54451	3500
24	15515	60985	3500
25	8196	68304	3500

!

TABLE LISTING

ENTER 3 COLUMN VALUES FOR TABLE PRINT

0. INCOME	1. TAX W/O INVEST
2. NET OP INCOME	3. INTEREST
4. PRINCIPAL	5. CASH FLOW
6. TOT DEPRECIATION	7. INC TAX DEDUCTS
8. TAXABLE INC.	9. TAXABLE LOSS
10. TAX WITH INVEST	11. TAX SAVINGS
12. CASH AVAILABLE	13. TOTAL BENEFITS
14. TAX BRACKET	15. ANNUAL PAYMENT

COLUMN 1 = ?1
COLUMN 2 = ?10
COLUMN 3 = ?13

YEARS	TAX W/O INVEST	TAX WITH INVEST	TOTAL BENEFITS
1	5556	0	19505
2	6225	0	16395
3	6217	0	17295
4	6475	0	17191
5	6733	0	17169
6	6992	0	17234
7	7253	0	17393
8	7520	0	17652
9	7638	0	18634
10	8031	0	18964
11	8438	0	19417
12	8864	0	20002
13	9312	300	20732
14	9786	1515	21620
15	10291	2810	22682

!

YEARS	TAX W/O INVEST	TAX WITH INVEST	TOTAL BENEFITS
16	10831	4195	23936
17	11413	5687	25400
18	12042	7300	27097
19	13398	10212	27893
20	14346	12367	29890
21	15384	14725	32184
22	16522	17312	34805

Fig.28-1–cont. Cash flow program and run.

```
 23          17775          20160          37791
 24          20621          25950          38535
 25          22516          29818          41985
  !

                        TABLE LISTING
ENTER 3 COLUMN VALUES FOR TABLE PRINT
 0. INCOME              1. TAX W/O INVEST
 2. NET OP INCOME       3. INTEREST
 4. PRINCIPAL           5. CASH FLOW
 6. TOT DEPRECIATION    7. INC TAX DEDUCTS
 8. TAXABLE INC.        9. TAXABLE LOSS
10. TAX WITH INVEST    11. TAX SAVINGS
12. CASH AVAILABLE     13. TOTAL BENEFITS
14. TAX BRACKET        15. ANNUAL PAYMENT
COLUMN 1    = ?
?REENTER
?-1
```

Fig.28-1–cont. Cash flow program and run.

The cash flow program demonstrates the power of the double subscripted array by producing twenty-five rows (25 years is the length of the loan, in this example) and fourteen columns. Fig. 28-3 shows six rows and fourteen columns of the cash flow problem. The columns are J,1 through J,14. Columns within the array are operated on to produce other columns. The cash flow column (CF(J,5)) is produced by subtracting the amortized principal (CF(J,4)) and the yearly interest (CF(J,3)) from the net operating income (CF(J,2)).

PURCHASE PRICE		
LAND	50,000	
BUILDING	750,000	
TOTAL PURCHASE PRICE	800,000	
SALVAGE VALUE	50,000	
CASH EQUITY	200,000	
NET OPERATING INCOME	80,000	per year
OUTSIDE INCOME	40,000	per year
MORTGAGE		
LIFE OF THE LOAN	25	years
INTEREST RATE	12	%
ANNUAL PAYMENTS (INTEREST & PRINCIPAL)	76,500	
DEPRECIATION		
LIFE OF THE BUILDING	40	years
DEPRECIATION METHOD	200	% double declining balance

Fig. 28-2. Cash flow and tax benefits of investment property ownership.

YEAR	NET OPERATING INCOME J,2	INTEREST J,3	AMORTI-ZATION PRINCIPAL J,4	CASH FLOW J,5	TOTAL DEPRECIAT. J,6	TOTAL DEDUC-TIONS J,7
1	8,000	72,000	4,500	3,500	35,000	107,000
2	8,000	71,460	5,040	3,500	33,250	104,710
3	8,000	70,855	5,645	3,500	31,588	102,443
4	8,000	70,178	6,322	3,500	30,008	100,186
5	8,000	69,419	7,081	3,500	28,508	97,927
TOTALS	40,000	353,912	28,588	17,500	158,354	512,266

TAXABLE INCOME J,8	TAX LOSS J,9	TAX PAYABLE J,10	TAX SAVINGS J,11	CASH AVAILABLE AFTER MORT-GAGE & TAX BENEFIT J,12	TOTAL CASH & AMORTI-ZATION BENEFITS J,13	TAX BRACKET J,14
0	27,000	0	11,610	15,110	19,610	.32
0	24,710	0	7,907	11,407	16,447	.37
0	22,443	0	8,304	11,804	17,449	.37
0	20,186	0	7,469	10,969	17,291	.37
0	17,927	0	6,633	10,133	17,214	.37
0	122,266	0	41,923	59,423	88,011	

Fig. 28-3. Cash flow input information.

```
CF(J,5) = CF(J,2) - (CF(J,3) + CF(J,4))
```

Net operating income is entered into the variable NOI, and an assignment statement places the net operating income into CF(J,2).

```
CF(J,2) = NOI
```

SUMMARY OF CASH & BENEFITS	10 YR. TOTAL	10 YR. AVG.	YIELD
CASH FLOW (BEFORE INCOME TAX EFFECT)	35,000	3,500	.018
TAX SAVINGS . . . . . +63,023			
TAX PAYABLE . . . . . – 0			
SUB TOTAL	63,023		
TOTAL CASH BENEFITS AFTER TAXES	98,023	9,802	.049
ADD PRINCIPAL PAID ON MORTGAGE	78,969		
TOTAL CASH & AMORTIZATION BENEFITS AFTER TAXES	176,992	17,699	.088

Fig. 28-4. Summary of cash and tax benefits.

The variables used as they appear in the program follow.

H1$	Header.
H2$	Header.
H3$	Header.
NOI	Net operating income.
PV	Amount of the loan.
I	Rate of interest.
LL	Life of the loan.
CA	Cost of the asset.
LA	Life of the asset.
SV	Salvage value of the asset.
CF	Cash flow array variable.
RD	Rate of depreciation.
YI	Your personal income outside of what the property will generate.
CE	Cash equity — the down payment on the property.
DF	The discount factor to determine the annual payment of the loan.
DP	1 /LA is the depreciation factor.
DEP	Depreciation per year — (RD/100)*DP.
RD	Rate of depreciation.
BV	Book value of the asset.
TB	Total book value of the asset.
NE	Net income.
RESTORE	Allows values in the data table to be reused without having to reRUN the program.
BS	A base from which the income tax is computed.
UL	The upper limit of the tax range. The first value in the data statement.
BF	The base figure is the amount of tax that must be paid in a specific tax bracket. The second value in the data statement.
IRS	The tax bracket of the investor. The third value in the data statement.
J – K	Loop variables.
AN	Annual payment on the loan.
CF(J,2)	Used to store the value of the net operating income.
CF(LL + 1,2)	Used to store the total value of the net operating income.
I1	Yearly interest.
CF(J,3)	Used to store the value of the yearly interest.
CF(LL + 1,3)	Total interest paid during the life of the loan.

PR	Principal value paid on the loan.
CF(J,4)	Principal value paid on the loan is stored in this element.
CF(LL + 1,4)	Total principal paid on the loan.
BR	Balance remaining on the loan.
CF	Variable for the cash flow array.
CF(J,5)	Element that holds the cash flow values.
CF(LL + 1,5)	Used to hold the total value of the cash flow.
D1	Total depreciation for one year.
CF(J,6)	Element that holds the depreciation.
CF(LL + 1,6)	Total depreciation for the period analyzed.
TB	Total book value.
CF(J,7)	Total deductions — interest and depreciation.
CF(LL + 1,7)	Total deductions for the period analyzed.
CF(J,8)	Taxable income.
CF(LL + 1,8)	Total taxable income.
CF(J,9)	Tax loss.
CF(LL + 1,9)	Total taxable loss.
CF(J,10)	Tax payable.
CF(LL + 1,10)	Total tax payable for the period analyzed.
CF(J,11)	Tax savings.
CF(LL + 1,11)	Tax savings for the period analyzed.
CF(J,0)	Yearly tax payable — CF(J,0) — CF(J,11).
CF(J,12)	Cash available after mortgage payments and payment of income tax.
CF(LL + 1,12)	Total of the cash available after the mortgage payments and income tax effect for the period analyzed.
CF(J,13)	Total cash and amortization benefits after taxes.
CF(LL + 1,13)	Total of cash and amortization benefits after taxes for the period analyzed.
CF(J,14)	Holds the tax bracket of the investor for the specific year.
YO	Years owned.
H1$(0)	See Fig. 28-7.
H1$(15)	See Fig. 28-7.
H4(0,0)	See Fig. 28-7.
H4(15,1)	See Fig. 28-7.
NUM(J)	Number of columns to be printed.
N	Number of the column.
M = NUM(J)	Loop variable J, placed in array NUM, stored in the variable M.

Following are shown the same variables, but in alphabetical order.

AN	Annual payment on the loan.
BF	The base figure is the amount of tax that must be paid in a specific tax bracket. The base figure plus the bracket percentage over the base figure.
BR	Balance remaining on the loan.
BS	A base from which the income tax is computed.
BV	Book value of the asset.
CA	Cost of the asset.
CE	Cash equity — the down payment on the property.
CF	Cash flow array variable.
CF(J,0)	Yearly tax payable — CF(J,0) — CF(J,11).
CF(J,2)	Used to store the value of the net operating income.
CF(LL + 1,2)	Used to store the total value of the net operating income.
CF(J,3)	Used to store the value of the yearly interest.
CF(LL + 1,3)	Total interest paid during the life of the loan.
CF(J,4)	Principal value paid on the loan is stored in this element of the table.
CF(LL + 1,4)	Total principal paid on the loan. At the end of the program this value should be equal to the original loan value.
CF(J,5)	This element of the table holds the cash flow values.
CF(LL + 1,5)	Used to hold the total value of the cash flow.
CF(J,6)	Element that holds the depreciation.
CF(LL + 1,6)	Holds the total depreciation for the period analyzed.
CF(J,7)	Holds the value of the total deductions — interest and depreciation.
CF(LL + 1,7)	Holds the total value of the deductions for the period analyzed.
CF(J,8)	Stores the value of the taxable income.
CF(LL + 1,8)	Holds the total value of the taxable income for the period analyzed.
CF(J,9)	Holds the value of the tax loss.
CF(LL + 1,9)	Holds the total value of the tax loss for the period analyzed.
CF(J,10)	Holds the value of the tax payable.
CF(LL + 1,10)	Holds the total value for the tax payable for the period analyzed.
CF(J,11)	Holds the value of the tax savings.
CF(LL + 1,11)	Holds the total value of the tax savings for the period analyzed.

CF(J,12)	Cash available after mortgage payments and income tax.
CF(LL + 1,12)	Holds the total value of the cash available for the period analyzed.
CF(J,13)	Total cash and amortization benefits after taxes.
CF(LL + 1,13)	Total cash and amortization benefits for the period analyzed.
CF(J,14)	Holds the tax bracket of the investor for a specific year.
DEP	Depreciation per year — (RD/100)*DP.
DF	The discount factor is used to determine the annual payment on the loan.
DP	The depreciation factor — 1/LA.
D1	Total depreciation for one year.
H1$	Header.
H1$(0)	See Fig. 28-7.
H1$(15)	See Fig. 28-7.
H2$	Header.
H3$	Header.
H4(0,0)	See Fig. 28-7.
H4(15,1)	See Fig. 28-7.
I	Rate of interest.
I1	Yearly interest.
IRS	The tax bracket of the investor. The third value in the data statement.
J — K	Loop variables.
LA	Life of the asset.
LL	Life of the loan.
M = NUM(J)	Loop variable J, placed in the array NUM, and stored in the variable M.
N	Number of the column to be printed.
NE	Net income.
NOI	Net operating income.
NUM(J)	Number of columns to be printed.
PR	Principal value remaining on the loan.
PV	Principal value of the loan.
RD	Rate of depreciation.
RESTORE	Allows the values in the data statements to be reused without having to reRUN the program.
SV	Salvage value of the asset.
TB	Total book value of the asset.
UL	The upper limit of the tax range. The first value in the data statement.

YI	Your personal income outside what the property will generate.
YO	Years owned.

The double declining balance depreciation constant is computed in line 320, and applied to the remaining depreciation (line 462) as each year is incremented during the life of the loan loop.

There are three subroutines in the program. Line 310 calls the subroutine at line 8000. Line 370 calls the subroutine at line 7000. Line 370 calls the subroutine at line 9000.

The subroutine at line 7000 sets the RESTORE command so the values in the DATA table can be reused on each life of the loan loop execution. The income base (BS = 0) is set to zero in order to be able to use the tax table. The tax table selected is the 1982 IRS tax table for married couples filing jointly. Line 7020 reads the first value in the tax table as the upper limit (UL) of the tax range, the second value as the base figure for tax payment in that bracket (BF), and the third value as the tax bracket (IRS).

Line 7030 tests to determine where net income (NE) is equal to or greater than the upper limit of the tax table and when the upper limit is not equal to zero (0). If this is true, then the entry in the table needed to compute the tax has not been located, and the program goes back to read the next entry.

Line 7040 assigns the income tax bracket rate (IRS) to CF(J,14) so the tax bracket can be printed out in table form.

Line 7050 computes the tax that would be paid to the IRS if the investment is not made.

Lines 7100 through 7220 compute the tax consequences for each year of the life of the loan, so the potential buyer can determine the profitability of ownership. The program computes the tax consequences on personal income in two cases, (1) with ownership of the property, and (2) without ownership of the property. See the run of the program in Fig. 28-1.

The subroutine that begins at line 8000 zeros out the total line of the table. Only columns three through thirteen are to hold computations (FOR J = 3 TO 13). The totals are to be placed in the row below the last row in the table (LL + 1). The life of the loan variable (LL) is incremented by one (LL + 1), so the totals line is always related to the life of the loan.

The subroutine that begins at line 9000 computes the discount factor (DF). The annual payment is equal to the loan amount divided by the discount factor (AN = PV / DF).

The discount factor is computed using the life of the loan (FOR J = 1 TO LL). The inverse of one (1) plus the interest rate (1/(1 + I)) is raised to the power of the loop variable (1 / (1 + I) ↑ J) (1 + I raised to the J power). This value is computed yearly and is summed on each execution of the loop and the value is placed in the discount factor (DF) variable.

Lines 380 through 670 are the life of the loan loop that computes the rows and columns in the cash flow array.

After the table is generated, which takes about sixty (60) seconds on a twenty-five year loan, the program defaults to line 780. Line 780 requests the user to enter the years owned. This entry allows the user to view the cash benefits, tax benefits, and ratios for periods of one year. This table can be displayed for each year of the life of the loan. This output helps the user determine which period of holding time results in the greatest profit.

If zero (0) is entered after YRS OWNED =, the program branches to the menu at line 900 to print out the individual columns of the table.

The menu selections and headings for the tables are set up in lines 900 through 965. This ingenious method uses single subscripted string arrays to hold the column header, and double subscripted numeric arrays to hold the number of spaces in the first and second lines of the headers (Figs. 28-5, 28-6, and 28-7). H1$ (0 through 15) holds the menu selection headings of each column. The string arrays and the numeric arrays are related. The first subscript in the numeric array relates to the string array, and the second subscript relates to the line of the header. Some of the two line headers are

H1$	1st LINE	2nd LINE	LENGTH 1st LINE	LENGTH OF 2nd LINE
H1$(0) = " 0.INCOME"	INCOME	NONE	H4(0,0) = 6	H4(0,1) = 0
H1$(1) = " 1.TAX PAYABLE"	TAX	PAYABLE	H4(1,0) = 3	H4(1,1) = 7
H1$(2) = " 2.NET OP INCOME"	NET OP	INCOME	H4(2,0) = 6	H4(2,1) = 6
H1$(3) = " 3.INTEREST"	INTEREST	NONE	H4(3,0) = 8	H4(3,1) = 0
H1$(4) = " 4.PRINCIPAL"	PRINCIPAL	NONE	H4(4,0) = 9	H4(4,1) = 0
H1$(5) = " 5.CASH FLOW	CASH	FLOW	H4(5,0) = 4	H4(5,1) = 4
H1$(6) = " 6.TOT DEPRECIATION"	TOT DEPRECIATION		H4(6,0) = 9	H4(6,1) = 7
H1$(7) = " 7.INC TAX DEDUCTS"	INC TAX	DEDUCTS	H4(7,0) = 7	H4(7,1) = 7
H1$(8) = " 8.TAXABLE INC."	TAXABLE	INC.	H4(8,0) = 7	H4(8,1) = 4
H1$(9) = " 9.TAXABLE LOSS"	TAXABLE	LOSS	H4(9,0) = 7	H4(9,1) = 4
H1$(10) = "10.TAX PAYABLE"	TAX	PAYABLE	H4(10,0) = 3	H4(10,0) = 7
H1$(11) = "11.TAX SAVINGS"	TAX	SAVINGS	H4(11,0) = 3	H4(11,1) = 7
H1$(12) = "12.CASH AVAILABLE"	CASH	AVAILABLE	H4(12,0) = 4	H4(12,1) = 9
H1$(13) = "13.TOTAL BENEFITS"	TOTAL	BENEFITS	H4(13,0) = 5	H4(13,1) = 8
H1$(14) = "14.TAX BRACKET"	TAX	BRACKET	H4(14,0) = 3	H4(14,1) = 7
H1$(15) = "15.ANNUAL PAYMENT"	ANNUAL	PAYMENT	H4(15,0) = 6	H4(15,1) = 7

THE 1st ALPHA CHARACTER OF H1$(?) IS ALWAYS THE 4th CHARACTER OF THE HEADER

Fig. 28-5. Header construction.

incorrectly separated, as in line 930 (Fig. 28-5) but they produce the correct results.

The logic of the algorithm stores the values to indicate a one line header (H4((0,0) = 6 : H4(0,1) = 0), or a two line header (H4(5,0) = 4 : H(5,1) = 4), and indicates how many characters and spaces are contained in the first line and how many characters and spaces are contained in the second line, Figs. 28-6 and 28-7.

```
H1$(0) = "0.INCOME"----------HEADER STRING ARRAY
H4(0,0) = 6----------SIX CHARACTERS IN THE FIRST LINE
H4(0,1) = 0----------ZERO CHARACTERS IN THE SECOND LINE
```

After all fifteen values are set for the menu, line 970 clears the screen (CALL -936) and prints TABLE LISTING and ENTER 3 VALUES FOR TABLE PRINT.

Line 980 prints out the menu selection. Line 990 sets up a loop to output to the screen three columns (0–2). The user enters the number of the column in a single subscripted array and the array is checked to determine if the value is between one (1) and fifteen (15). If the number is not between one and fifteen, the program jumps out of the loop to line 1010.

In line 1010, N = N − 1 : IF N < 0 THEN 3000, the value of N is decremented to produce the correct value of N. When the program jumps out of the loop, the loop value is one more than the correct value. To produce the correct value of N, one must be subtracted. Determining the correct value of the loop variable was discussed in Lesson 8.

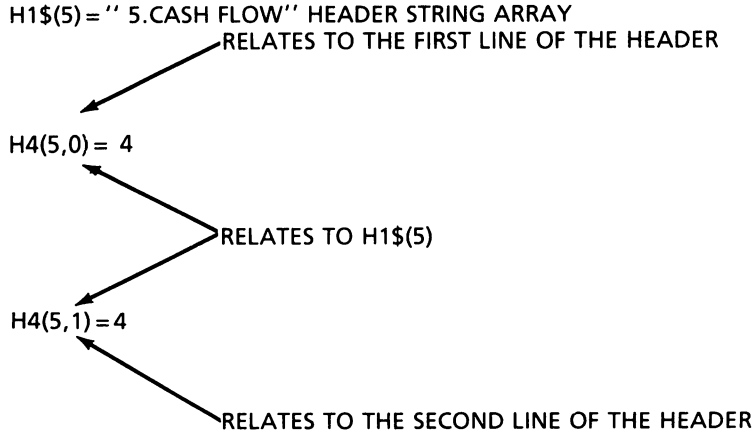


Fig. 28-6. Header detail.

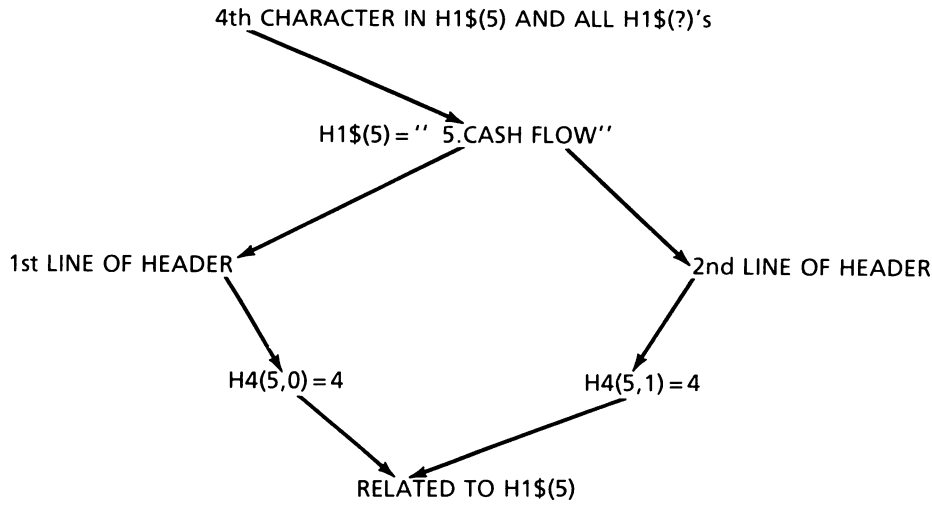


Fig. 28-7. Header detail.

IF N < 0 THEN 3000. When the user is through working with the program, an entry of − 1 (less than zero) causes a branch to line 3000 to end the program.

Line 1020 is the beginning of the loop to compute the values on the table. LL is the variable to hold the life of the loan. The only function of line 1040 is to stop the program. On the first loop pass, L = 1. The program defaults through lines 1030, 1040, and prints the headings in lines 1050 and 1060. On subsequent loop passes, the program branches from 1030 to 1080. When line 1030 is false, the program defaults to line 1040. If L is not one (1) (L = 16) the program inputs "!" and stops. This allows the user to view and study the fifteen (15) rows of the column printed on the screen. When the user is ready for the program to continue, RETURN is pressed.

Line 1050 (Fig. 28-8) clears the screen and prints the first line of the column header on the screen. In line 1050, N is the variable that holds the number of columns, and HTAB(J + 1)*10 sets the column in the correct position on the screen. The column value is stored in the variable M.

In line 1060 (Fig. 28-9) the second line of the header is printed. N is the variable that holds the number of columns. The value of the column is placed in the variable M.

When the column headings have been printed, the program defaults to line 1080 to print L, which represents the year (FOR L = 1 TO LL − LINE 1020). The second statement in line 1080, for J = 0 TO N, is the beginning of a loop that controls the number of columns to be printed. Line 1090 tabs to the proper location on the screen to print the columns. M = NUM(J) stores the number of the column to be printed in the variable M.

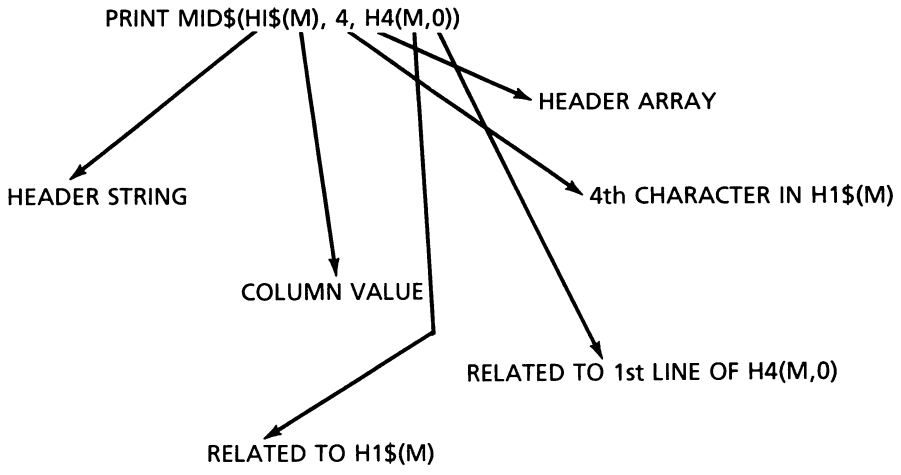


Fig. 28-8. Header detail.

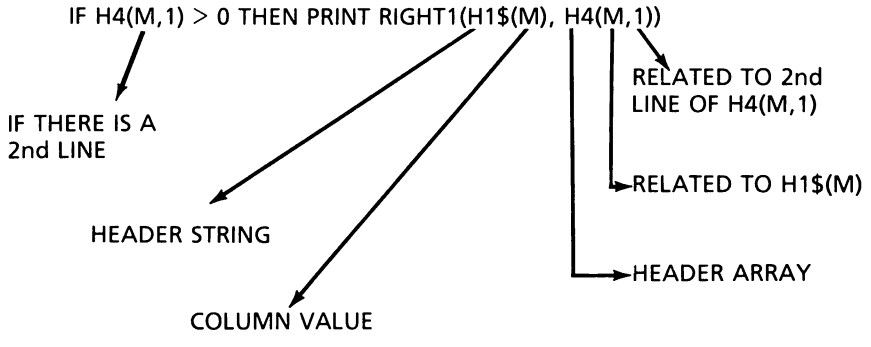


Fig. 28-9. Header detail.

annual payment. The annual payment on the loan is computed by dividing the amount of the loan (AN) by the discount factor (DF) (line 380).

The SGN function (Fig. 28-10) is used in line 545. The SGN function returns only three values +1, 0, and −1. These values relate to the greater

If line 1090 is true, the program branches to line 1110, to print out the rounded (line 140) cash flow array values for row L, and column M. If M = 14 is false, the program defaults to line 1095. If line 1095 is true, the program branches to line 1100 to print out the integer function (line 145) of the cash flow array, row L, column M. If M < > 15 is false, the program defaults to line 1096. PRINT FN A(AN) is the integer (line 145) cash flow array of the

```
SGN FUNCTION MAKES ALL NEGATIVE NUMBERS—POSITIVE
SGN FUNCTION MAKES ALL ZERO NUMBERS—ZERO
SGN FUNCTION MAKES ALL POSITIVE NUMBERS—ZERO

TAXABLE LOSS     NET OPERATING INCOME      DEDUCTIONS
CF(J,9)      = CF(J,2)          - CF(J,7)

CF(J,9) = (SGN(CF(J,9)) - 1)*CF(J,9)/2

IF CF(J,9) < 0 THEN SGN IS -1
  IF CF(J,9) = -2000
    (SGN(CF(J,9)) - 1)*CF(J,9)/2
      (SGN(-2000) -1)* -2000/2
        (-1 -1)* -2000/2
          -2* -2000/2
            +4000/2
              = 2000

  IF CF(J,9) = 0 THEN SGN = 0
    (0 - 1)* 0/2 = 0

  IF CF(J,9) = + 2000 THEN SGN = +1
    (SGN(CF(J,9)) -1)* CF(J,9)/2
        (+1 -1)* 2000/2
            0 * 2000/2
              = 0
```

Fig. 28-10. Sign (SGN) function.

than zero, zero, and less than zero. In this case, if the tax loss is a positive number, SGN(CF(J,9)) = +1, (SGN(CF(J,9)) − 1) = 0, and zero times CF(J,9)/2 = zero (0). When the tax loss is a negative number, (SGN(CF(J,9) − 1) = −2. When −2 is multiplied by a negative value, CF(J,9)/2, the factor (−2 / +2) divides out leaving a negative value. A negative times a negative is a positive value. The purpose of the SGN function is to make all negative numbers positive, leave zero as a zero, and make all positive numbers zero. This causes the tax savings to be calculated. The SGN function is used so a tax loss, which is a negative number, can be converted into a positive number. The positive number is then multiplied by the previous year's tax rate, (CF(J − 1),14)), and the results are placed in the tax savings column, CF(J,11). CF(J − 1,14) gives the tax rate for the previous year which is used to compute this year's taxes. CF(J,14) gives the tax rate for the previous year.

The cash flow program outputs financial information for a single year or a number of years. This information assists the prospective buyer to determine how the purchase will affect his or her cash flow and future net worth. This information will aid the potential buyer whether to buy, or not to buy, the property.

LESSON 29

Numerical Programs

QUADRATIC FORMULAS

A polynomial is an algebraic expression consisting of two or more rational or integral terms. The polynomial consists of more than one monomial in the form of AX^n. A is a real number coefficient, X is a variable, and n is the power (integer) to which it is raised. A polynomial in X assumes the following form.

$$A_nX^n + A_{(n-1)}X^{(n-1)} + \ldots A_1X + A_0$$

Fig. 29-1 deals with a polynomial of 2nd degree, and $X^2 + 5X + 6$ is a specific polynomial. If a polynomial is the product of two polynomials, it can be factored. Factoring plays a major role in the solution of a polynomial $((X + 3)(X + 2)) - (X = -3, X = -2)$. However, when a 2nd degree polynomial cannot be easily factored, it must be solved by using the quadratic formula.

```
100 REM -PROGRAM TO COMPUTE THE ROOTS OF A QUADRATIC EQUATION OF
    THE FORM
110 REM -: Y = A*XΛ2 + B*X + C
120 REM -BY ENTERING THE VALUES A,B, AND C
130 REM -THE PROGRAM ALSO INDICATES IF THE ROOTS ARE REAL OR
    "IMAGINARY"
140 REM -IF ALL THREE VALUES ARE ZERO THEN THEN PROGRAM TERMINATES
150 PRINT "ENTER 3 VALUES (A,B,C) ";
160 INPUT A,B,C
170 IF A = 0 AND B = 0 AND C = 0 GOTO 250
180 D1 = B Λ 2
190 D2 = 4 * A * C
200 D = ( SQR ( ABS (D1 - D2))) / (2 * A)
210 B = - B / (2 * A)
220 IF (D1 - D2) > 0 THEN GOSUB 260: GOTO 150
230 IF (D1 - D2) = 0 THEN GOSUB 300: GOTO 150
240 GOSUB 330: GOTO 150
250 END
```

Fig. 29-1. Quadratic formula program.

```
260 PRINT : PRINT "TWO SOLUTIONS - BOTH REAL": PRINT
270 PRINT "SOLUTION 1 = ";B + D
280 PRINT "SOLUTION 2 = ";B - D: PRINT
290 RETURN
300 PRINT : PRINT "ONE REAL SOLUTION": PRINT
310 PRINT "THE SINGLE SOLUTION = ";B: PRINT
320 RETURN
330 PRINT : PRINT "TWO SOLUTIONS - BOTH IMAGINARY": PRINT
340 PRINT "SOLUTION 1 = ";B;" + J";D
350 PRINT "SOLUTION 2 = ";B;" - J";D: PRINT
360 RETURN
RUN
ENTER 3 VALUES (A,B,C) ?1,5,6

TWO SOLUTIONS — BOTH REAL

SOLUTION 1 = -2
SOLUTION 2 = -3

ENTER 3 VALUES (A,B,C) ?1,2,1

ONE REAL SOLUTION

THE SINGLE SOLUTION = -1

ENTER 3 VALUES (A,B,C) ?1,-1,12

TWO SOLUTIONS - BOTH IMAGINARY

SOLUTION 1 = .5 + J3.4278273
SOLUTION 2 = .5 - J3.4278273

ENTER 3 VALUES (A,B,C) ?0,0,0
```

Fig. 29-1–cont. Quadratic formula program.

The quadratic formula is an equation in the form $AX^2 + BX + C = 0$. In this expression, A, B, and C are real numbers, and A must not be zero (0).

If A is not equal to zero (0), then the roots of the equation $AX^2 + BX + C = 0$ are determined by the following equation.

$$X = \frac{-B \pm SQR(B^2 - 4AC)}{2A}$$

The number $B^2 - 4AC$ which appears under the radical sign (SQR) is called the discriminant of the quadratic equation. The discriminant is used to determine the nature of the roots of the quadratic equation. The nature of the roots is determined if the discriminant is greater than zero (0), equal to zero (0), or less than zero (0).

If $B^2 - 4AC > 0$ (line 220 is TRUE) the equation has two real and unequal solutions, or roots.

If $B^2 - 4AC = 0$ (line 230 is TRUE) the equation has one real solution, or root.

If $B^2 - 4AC < 0$ (line 230 is FALSE) the equation has two complex solutions.

The program (Fig. 29-1) lines 180 through 210, sets up to compute the quadratic formula in four parts. These are (1) D1 = B^2, (2) D2 = 4AC, (3) D = SQR(D1 − D2)/(2A), and (4) B = −B/(2*A).

The equation $X^2 + 5X + 6$ is used to demonstrate a discriminant value greater than zero.

When the discriminant is one (1), that is, greater than zero (1 > 0), the program branches to the subroutine that begins at line 260. The discriminant and base were computed in the main body of the program, and the two solutions are printed out.

$$X1 = \frac{-5 + 1}{2} = \frac{-4}{2} = -2$$

$$X2 = \frac{-5 - 1}{2} = \frac{-6}{2} = -3$$

When a polynomial takes the form $X^2 + 2X + 1$, the discriminant, $(+2^2) - 4*1*1$, is equal to zero (0), line 230. When the discriminant is equal to zero, the program branches to the subroutine at line 300. When the discriminant is equal to zero, there is only one root to the equation.

$$X1 = \frac{-2 + 0}{2} = \frac{-2}{2} = -1$$

$$X2 = \frac{-2 + 0}{2} = \frac{-2}{2} = -1$$

In the third case, line 230 is FALSE; $X^2 - X + 12 = 0$ causes the discriminant to be less than zero, $((+1^2) - 4*1*12)$ equals the SQR(−47). The situation of a negative square root causes the discriminant to be handled by a special case, line 200, D = (SQR (ABS(D1 − D2)). In case 1, the discriminant is greater than zero, and case 2, when the discriminant is equal to zero, the ABS command is unnecessary. In efficient programming, a single method should be searched for in order to calculate a value in the same way for all cases. Thus, the D = (SQR(ABS(D1 − D2)) calculates the discriminant in all three cases.

When the discriminant is less than zero, there are two roots, both being complex. Complex solutions are output in a different form than in cases 1 and 2.

The complex solution has a real part and an imaginary (bad word description) part. To take the square root of a negative number, a minus one (−1) is factored from the negative to produce a positive number. The negative one (−1) is referred to as the "J" operator and is printed out to designate the imaginary part of the complex number.

1 + j2
6 − j4.7

A complex number takes the form A + jB, (not directly related to AX^2 + BX + C = 0). The real value A is called the real part of the complex number. The jB is called the imaginary part of the complex number.

When the discriminant is less than zero ($X^2 - X + 12 = 0$), the program branches to line 330 to print out the two complex solutions.

SOLUTION 1 = .5 + j3.4278273
SOLUTION 2 = .5 − j3.4278273

MATRIX ADDITION

Fig. 29-2 is a program to add matrices. A matrix is a rectangular array of scalars. A scalar is a real number capable of being represented as a point on a scale. Each matrix contains a fixed number of rows and a fixed number of columns. The row value is listed first, and the column value is then listed. A 2 × 3 matrix has the following configuration. It has two rows and three columns.

1 2 3
4 5 6

A 3 × 2 matrix has the following configuration. It has three rows and two columns.

1 4
2 5
3 6

A square matrix has the same number of rows and columns. A 3 × 3 matrix has the following configuration.

1 4 7
2 5 8
3 6 9

The size of a matrix is designated by the letter M for the number of rows, and N for the number of columns. A rectangular matrix could be either an M × N, or an N × M, while a square matrix would be either an M × M, or an N × N.

The lower case letter "i" represents the internal row number of a matrix, and the lower case "j" represents the internal column number of the matrix.

$A = a_{ij}$

$A(3 \times 3) = a_{11}\ a_{12}\ a_{13}$

$a_{21}\ a_{22}\ a_{23}$

$a_{31}\ a_{32}\ a_{33}$

When matrices are added, the same row-column value in each matrix is added.

$A + B = (a_{ij}) + (b_{ij}) = (a_{ij} + b_{ij})$

Thus, when adding, the matrices must be the same dimensions. A 2 × 3 matrix can be added to a 2 × 3 matrix, and a 4 × 4 matrix can be added to a 4 × 4 matrix. A 2 × 3 matrix cannot be successfully added to a 4 × 4 matrix, because the row-column combinations are not compatible. To add matrices A and B, they must be first read into memory. The addition takes place in doubly nested loops, lines 290 through 360. The outer loop, I, line 290, represents the number of rows of the matrix. The loop index goes from row one to the number of rows in the matrix.

```
290 FOR I = 1 TO M
```

The inner J loop, line 300, represents the number of columns in the matrix. The loop index goes from column one (1), to the number of columns (N).

```
300 FOR J = 1 TO N
```

The A and B matrices are summed into the C matrix.

```
320 C(I,J) = A(I,J) + B(I,J)
```

The A matrix and the B matrix are read from data statements and are printed out as a debugging procedure. For correct output, it is essential that the input be correct. When the matrices are printed out, the programmer can view each step. If the A or B matrix output data is incorrect, it can be checked and corrected before it is used to multiply into the C matrix.

```
100 REM MATRIX ADDITION
110 REM -THIS PROGRAM WILL READ TWO MATRICES FROM DATA STATE-
    MENTS
120 REM -AND ADD THEM TOGETHER
130 REM -MATRIX A AND B ARE SUMMED INTO MATRIX C
140 REM -THE NUMBER OF ROWS AND COLUMNS ARE STORED AS DATA TO
    MAKE THE PROGRAM MORE FLEXIBLE
150 REM - I IS THE ROW VARIABLE
160 REM - J IS THE COLUMN VARIABLE
170 REM - M IS THE # OF ROWS IN THE MATRIX
180 REM - N IS THE # OF COLUMNS IN THE MATRIX
190 READ M,N
200 DIM A(M,N),B(M,N),C(M,N)
210 FOR I = 1 TO M
220 FOR J = 1 TO N
230 READ A(I,J)
240 PRINT A(I,J);" ";
250 NEXT J
260 PRINT
270 NEXT I
280 PRINT : PRINT
```

Fig. 29-2. Matrix addition program.

```
290 FOR I = 1 TO M
300 FOR J = 1 TO N
310 READ B(I,J)
320 C(I,J) = A(I,J) + B(I,J)
330 PRINT B(I,J);" ";
340 NEXT J
350 PRINT
360 NEXT I
370  PRINT : PRINT
380 PRINT SPC( 12);"MATRIX ADDITION": PRINT
390 PRINT SPC( 5);" COL1   COL2   COL3   COL4   COL5"
400 FOR I = 1 TO M
410 PRINT "ROW";I; SPC( 2);
420 FOR J = 1 TO N
430 HTAB (J - 1) * 7 + 8: PRINT C(I,J);
440 NEXT J
450 PRINT : PRINT
460 NEXT I
470 DATA 3
480 DATA 5
490 DATA 1,4,2,3,5
500 DATA 4,2,3,2,1
510 DATA 0,0,3,2,5
520 DATA 2,3,1,0,1
530 DATA 2,1,5,0,0
540 DATA 3,2,1,0,2
550 END
RUN
1  4  2  3  5
4  2  3  2  1
0  0  3  2  5

2  3  1  0  1
2  1  5  0  0
3  2  1  0  2
```

MATRIX ADDITION

	COL1	COL2	COL3	COL4	COL5
ROW 1	3	7	3	3	6
ROW 2	6	3	8	2	1
ROW 3	3	2	4	2	7

Fig.29-2–cont. Matrix addition program.

SCALAR-MATRIX MULTIPLICATION

Fig. 29-3 is a program to multiply a matrix by a scalar. When a matrix is multiplied by a scalar, each row-column value is multiplied by the scalar.

```
290 C(I,J) = S * A(I,J)
```

```
100 REM SCALAR * MATRIX MULTIPLICATION
110 PRINT : PRINT
120 REM - I IS THE ROW VARIABLE
130 REM - J IS THE COLUMN VARIABLE
140 REM - M IS THE # OF ROWS IN THE MATRIX
150 REM - N IS THE # OS COLUMNS IN THE MATRIX
160 REM - S IS THE SCALAR VALUE
170 READ M,N,S
180 DIM A(M,N),C(M,N)
190 FOR I = 1 TO M
200 FOR J = 1 TO N
210 READ A(I,J)
220 PRINT A(I,J);'' '';
230 NEXT J
240 PRINT
250 NEXT I
260 PRINT
270 FOR I = 1 TO M
280  FOR J = 1 TO N
290 C(I,J) = S * A(I,J)
300 NEXT J
310 NEXT I
320 PRINT SPC( 8);''SCALAR ' '';S;'' ' TIMES MATRIX'': PRINT
330 PRINT SPC( 5);'' COL1   COL2   COL3   COL4   COL5''
340 FOR I = 1 TO M
350 PRINT ''ROW'';I; SPC( 2);
360 FOR J = 1 TO N
370 HTAB (J - 1) * 7 + 8: PRINT C(I,J);
380 NEXT J
390 PRINT : PRINT
400 NEXT I
410 DATA 3
420 DATA 5
430 DATA 2
440 DATA 1,4,2,3,5
450 DATA 4,2,3,2,1
460 DATA 0,0,3,2,5
470 END
RUN

1  4  2  3  5
4  2  3  2  1
0  0  3  2  5

                SCALAR '2' TIMES MATRIX

          COL1    COL2    COL3    COL4    COL5
ROW 1      2       8       4       6       10
ROW 2      8       4       6       4       2
ROW 3      0       0       6       4       10
```

Fig. 29-3. Scalar matrix multiplication program.

MATRIX MULTIPLICATION

The next step in the progression of matrix operations is matrix multiplication (Fig. 29-4). Without discussing the logic, multiplying matrices does not follow the routine of adding matrices. In this example, the X matrix is a 5 × 3 matrix, and the Y matrix is a 3 × 3 matrix.

X = 5 × 3 (column of "X" matrix)
3 × 3 = Y (row of "Y" matrix)

The resultant Z matrix is a 5 × 3 matrix.

```
100 REM - MATRIX MULTIPLICATION
110 REM - I IS THE ROW VARIABLE
120 REM - J IS THE COLUMN VARIABLE
130 REM - K IS THE COLUMN VARIABLE OF 'A1' AND THE ROW VARIABLE OF
    'B'
140 REM - N IS THE NUMBER OF COLUMNS IN THE MATRIX
150 REM - 'A' MATRIX 2 (X) 3
160 REM - 'A1' MATRIX 3 (X) 2
170 REM - 'B' MATRIX 2 (X) 2
180 REM - 'C' MATRIX 3 (X) 2
190 READ M,N
200 PRINT : PRINT ''THE 'A' MATRIX IS:'': PRINT
210 DIM A(M,N),A1(N,M),B(M,M),C(N,M)
220 FOR I = 1 TO M
230 FOR J = 1 TO N
240 A(I,J) = 0
250 READ A(I,J)
260  PRINT A(I,J);'' '';
270 NEXT J
280 PRINT
290 NEXT I
300 PRINT : PRINT ''THE 'A' MATRIX IS TRANSPOSED'': PRINT
310 FOR I = 1 TO N
320 FOR J = 1 TO M
330 A1(I,J) = A(J,I)
340 PRINT A1(I,J);'' '';
350 NEXT J
360 PRINT
370 NEXT I
380 PRINT : PRINT ''THE 'B' MATRIX IS:'': PRINT
390 FOR I = 1 TO M
400 FOR J = 1 TO M
410 B(I,J) = 0
420 READ B(I,J)
430 PRINT B(I,J);'' '';
440 NEXT J
450 PRINT
460 NEXT I
470 FOR I = 1 TO N
```

Fig. 29-4. Matrix multiplication program.

```
480 FOR J = 1 TO M
490 C(I,J) = 0
500 FOR K = 1 TO M
510 C(I,J) = C(I,J) + A1(I,K) * B(K,J)
520 NEXT J
530 NEXT I
540 PRINT : PRINT "THE PRODUCT MATRIX IS:": PRINT
550 FOR I = 1 TO N
560 FOR J = 1 TO M
570  PRINT C(I,J);" "
580 NEXT J
590 PRINT
600 NEXT I
610 DATA 2 : REM — M
620 DATA 3 : REM — N
630 DATA 2,3,4
640 DATA 1,2,2
650 DATA 2,1
660 DATA 7,3
670 END
RUN

THE 'A' MATRIX IS:

2  3  4
1  2  2

THE 'A' MATRIX IS TRANSPOSED

2  1
3  2
4  2

THE 'B' MATRIX IS:

2  1
7  3

THE PRODUCT MATRIX IS:

4  2
6  3
8  4
```

Fig.29-4–cont. Matrix multiplication program.

The rule for multiplying matrices is that the column value of the X matrix must equal the row value of the Y matrix. The resultant matrix takes the dimensions of the X matrix row, and the Y matrix column. If the C matrix (line 320, Fig. 29-2) has been previously zeroed out, the multiplying statements are as follows.

```
290 FOR I = 1 TO 5
300 FOR J = 1 TO 3
310 FOR K = 1 TO 3
320 C(I,J) = C(I,J) + A(I,K) * B(K,J)
```

```
330 NEXT K
340 NEXT J
350 NEXT I
```

The program in Fig. 29-4 is written to read in the A as a 2 × 3 matrix. Lines 220 through 270 read in and print out a 2 × 3 A matrix. The A (2 × 3) matrix cannot be multiplied by the B (2 × 2) matrix because the A column value is 3, and the B row value is 2. Therefore, the A matrix is transposed to an "A1" (3 × 2) matrix. Lines 310 through 370 transpose the A matrix into the A1 (3 × 2) matrix.

The outer loop variable I line 310, has an index from one (1) to three (3). The inner loop variable J line 320, has an index from one (1) to two (2).

```
320 A1(I,J) = A(J,I)
```

The loop variables are reversed in the A matrix so the transposition can take place.

The B (2 × 2) matrix is read into memory and printed out on lines 390 through 460.

Lines 470 through 530 zero out the elements of the C matrix, multiply the A matrix by the B matrix, and place the results in the C matrix.

GAUSSIAN ELIMINATION

Now that we've learned to add and multiply matrices, how are we going to use this information? In many aspects of the working world, situations present a system of equations that contain several unknown variables. One method of solving a system of equations is by elimination of variables.

$$
\begin{array}{rr}
(1) & 2X + 9Y = 10 \\
(2) & 4X - 2Y = 0 \\
\text{(1) Multiply by } -2 & -4X - 18Y = -20 \\
 & 4X - 2Y = 0 \\
\hline
 & -20Y = -20 \\
 & Y = 1 \\
 & 4X - 2 = 0 \\
 & 4X = 2 \\
 & X = .5
\end{array}
$$

For two equations and two unknowns, the solution is rather simple. Multiply the first equation by −2. Then add the two equations, to produce the result $-20Y = -20$. Dividing by 20, we get $Y = 1$. Substitute the $Y = 1$ into equation number two to determine that $X = .5$.

With three or more equations, the solution is not so simple. One method to solve a system of equations is to use the Gaussian elimination method. Before attempting the program, let's solve a system of equations using the

Gaussian elimination method. This is the same problem solved by the program in Fig. 29-5.

(1) 3X − 2Y = 1 Divide #1 by (3)
(2) 5X + Y = 6 Divide #2 by (5)

(1) 1X − .667Y = .333
(2) 1X + .2Y = 1.2 Multiply by (− 1)

(1) 1X − .667Y = .333
(2) − 1X − .2Y = − 1.2

0X − .867Y = 0.867 Divide #2 by (− .867)
Y = 1
5X + 1 = 6
5X = 5
X = 1

The Gaussian elimination method produces zeros in the lower left triangle until only one variable is unknown. The value of the variable is then back substituted to determine the values of the other unknowns.

```
100 REM
110 REM
120 REM    GAUSSIAN ELIMINATION
130 REM
140 REM
150 REM -THIS PROGRAM WILL SOLVE A SERIES OF EQUATIONS USING THE
    MATRIX TECHNIQUE
160 REM -OF GAUSSIAN ELIMINATION
170 REM -THE PROGRAM ALLOWS THE USER TO ENTER THE COEFFICIENTS FOR
    EACH VARIABLE
180 REM -IN THE EQUATION AND THE CONSTANT THAT IS ON THE RIGHT SIDE
    OF THE EQUALS SIGN
190 REM -THE PROGRAM THEN PRINTS OUT THE INTERMEDIATE STEPS OF
    TRANSFORMING THE EQUATIONS
200 REM -WHEN THE PROCESS IS COMPLETE, THE VALUES ARE DISPLAYED
210 DEF FN A(X) = INT (X * 1000 + .5) / 1000
220 DIM A(25,25)
230 PRINT : PRINT
240 INPUT "ENTER NUMBER OF EQUATIONS: ";N
250 PRINT
260 FOR I = 1 TO N
270 FOR J = 1 TO N + 1
280 PRINT "A(";I;",";J;") = ";
290 INPUT " ";A(I,J)
300 NEXT J
310 NEXT I
320 PRINT
```

Fig. 29-5. Gaussian elimination program.

```
330 FOR I = 1 TO N
340 FOR J = I TO N
350 D = A(J,I)
360 IF D = 0 THEN D = 1E - 25
370 FOR K = I TO N + 1
380 A(J,K) = A(J,K) / D
390 PRINT "A(";J;",";K;") = "; FN A(A(J,K));" D = "; FN A(D)
400 NEXT K
410 NEXT J
420 PRINT
430 FOR J = I + 1 TO N
440 FOR K = I TO N + 1
450 A(J,K) = A(J,K) - A(I,K)
460 PRINT "A(";I;",";J;") = "; FN A(A(I,J));" A(";J;",";K;") = "; FN A(A(J,K))
470 NEXT K
480 NEXT J
490 NEXT I
500 PRINT
510 PRINT : PRINT "X(1) IS 1ST COLUMN VARIABLE:"
520 PRINT : PRINT "X(2) IS 2ND COLUMN VARIABLE:": PRINT
530 PRINT "X(3) IS 3RD COLUMN VARIABLE: ETC.": PRINT
540 FOR K = N TO 1 STEP - 1
550 X(K) = A(K,N + 1)
560 FOR W = N TO K STEP - 1
570 X(K) = X(K) - A(K,W + 1) * X(W + 1)
580 NEXT W
590 PRINT "X(";K;") = ";X(K)
600 NEXT K
610 PRINT : PRINT
620 END
RUN

ENTER NUMBER OF EQUATIONS: 2

A(1,1) = 3
A(1,2) = -2
A(1,3) = 1
A(2,1) = 5
A(2,2) = 1
A(2,3) = 6

A(1,1) = 1  D = 3
A(1,2) = -.667  D = 3
A(1,3) = .333  D = 3
A(2,1) = 1  D = 5
A(2,2) = .2  D = 5
A(2,3) = 1.2  D = 5

A(1,2) = -.667  A(2,1) = 0
A(1,2) = -.667  A(2,2) = .867
A(1,2) = -.667  A(2,3) = .867
A(2,2) = 1  D = .867
```

Fig.29-5–cont. Gaussian elimination program.

```
A(2,3) = 1  D = .867
A(2,3) = 1  A(3,2) = -1
A(2,3) = 1  A(3,3) = -1
X(1) IS 1ST COLUMN VARIABLE:
X(2) IS 2ND COLUMN VARIABLE:
X(3) IS 3RD COLUMN VARIABLE: ETC
X(2) = 1
X(1) = 1
```

Fig.29-5–cont. Gaussian elimination program.

Now let's see how to tell the computer how to solve a system of equations using th Gaussian elimination system (Fig. 29-5).

Line 220 DIM (25,25) dimensions memory to store up to twenty-five equations. Line 240 allows the user to enter the number of equations to be solved. Lines 260 through 310 allow the user to enter the values into the elements of the matrix. The input prompt (see RUN) shows which element is to receive the coefficient of the variable.

Lines 330 through 360 determine the divisor for the first equation (3X/3 = 1), and the second equation (5X/5 = 1), and divides each real value by the divisor. This sets the equations so that the coefficient of the first column variable is one, and prepares to subtract one equation from the other.

Line 360 IF D = 0 THEN D = 1E–25 prevents the division by zero error. This value is very close to zero. If D equals zero (0), the program stops running. If D equals 1E – 25 the program runs, but the answer is not absolutely correct.

Lines 430-480 subtract equation number two from equation number one. The Y is then – .867. The loop that begins at line 330 increments for the second time to divide the Y by – .867 so the resulting value of Y is equal to one.

Lines 540-610 solve the system of equations through back substitution. Line 540 is the beginning of a loop whose beginning index starts at N (number two in this example) and decrements to one. Line 550 assigns the value stored in A(K,N + 1) (A(2,3)), in this example into X(K) (X(2)). This back substitution is completed in lines 560-580. The results are printed out in reverse order. The definitions in lines 510, 520, and 530 are valid if there are no more than three equations.

NEWTON-RAPHSON

Fig. 29-6 is a program and RUN of the Newton-Raphson method to determine the roots of a function. The Newton-Raphson method is probably the most widely used method to determine the root(s) of a function because of its rapid convergence and its ease in programming. To use the Newton-

```
100 REM NEWTON RAPHSON ITERATION METHOD
110 REM -USING THE EQUATION Y=X^3 AS THE BASIS FOR THE EXAMPLE. THE
    NEXT GUESS IS FOUND
120 REM -BY CALCULATING X - ( (X^3) / (3*X^2) )
130 REM -AS SOON AS THE DIFFERENCE IN THE GUESSES IS LESS THAN .00005
    THEN ITERATION PROCESS QUITS
140 DEF FN A(X) = INT (X * 10000 + .5) / 10000
150 PRINT "N","X","Y"
160 Y = 0
170 N = 1
180 X = 1
190 X1 = X - ((X^3) / (3 * X^2))
200 PRINT N, FN A(X), FN A(X^3)
210 N = N + 1
220 IF ( ABS (X1 - X)) < = .00005 GOTO 250
230 X = X1
240 GOTO 190
250 PRINT : PRINT "THE FUNCTION CROSSES THE 'X' AXIS AT:": PRINT
260 PRINT "    X = ";X
270 END
RUN
```

N	X	Y
1	1	1
2	.6667	.2963
3	.4444	.0878
4	.2963	.026
5	.1975	7.7E-03
6	.1317	1.2E-03
7	.0878	7E-04
8	.0585	2E-04
9	.039	1E-04
10	.026	0
11	.0173	0
12	.0116	0
13	7.7E-03	0
14	5.1E-03	0
15	3.4E-03	0
16	2.3E-03	0
17	1.5E-03	0
18	1E-03	0
19	7E-04	0
20	5E-04	0
21	3E-04	0
22	2E-04	0
23	1E-04	0

```
THE FUNCTION CROSSES THE 'X' AXIS AT:
   X = 1.33657182E-04
```

Fig. 29-6. Newton-Rhapson method for determining roots of a function.

Raphson method two criteria must be fulfilled, (1) the function must be continuous over the defined area, and (2) the function must have a derivative. A relatively simple function that fits the criteria is $Y = X^3$. It is known that $Y = X^3$ crosses the X-Y axis at $X = 0$ and $Y = 0$, so the results of the program are easy to check.

A series of points that fits the function $Y = X^3$ is as follows.

X	Y
-5	-125
- .	
-4	-64
-3	-27
-2	-8
-1	-1
0	0
1	1
2	8
3	27
4	64
5	125

The derivative of X^3 is $3X^2$.

The function is divided by the first derivative. This value is subtracted from the value of X_n. This value is assigned to the next value of X.

$$X_{(N+1)} = X_n - \frac{f(X)}{f'(X)}$$

$$X2 = X1 - \frac{f(X)}{f'(X)}$$

$$Y = X - ((X^3) / (3X^2))$$

To determine the root of a function, (1) guess where the function will cross the X axis, and (2) use the first approximation to compute the second, third, etc., approximation.

The first approximation, $X = 1$ is placed in the program in Fig. 29-6, in line 180. Line 200 prints out the number of the increment N, the value of X to four places, and the value of Y to four places. Line 210 counts the number of increments before the final X value is printed.

Line 220, IF (ABS(X1 - X)) < = .00005 GOTO 250 determines how precise the value of X will be.

The program increments until the value of ABS(X1 - X) is less than, or equal to, .00005. After the value of X1 is calculated, it is assigned to the variable X, line 230. This places the calculated X1 value into X so the program can perform the next approximation.

The Newton-Raphson method does not always determine the root of a function. It will fail if the function does not cross the X axis. It will also fail if the function lies entirely on the X axis or oscillates rapidly across the X axis from the positive Y to the negative Y area.

BISECTION METHOD TO DETERMINE THE ROOTS OF A FUNCTION

The bisection method to determine the roots of a function is used when the derivative of a function cannot be easily determined (Fig. 29-7).

```
10  DEF FN FT(X) = X^3 - 12 * X^2 + 47 * X - 60
100 REM -FINDING ROOTS BY THE BISECTION METHOD
110 REM -THIS PROGRAM USE THE BISECTION METHOD TO SCAN FOR ROOTS
    TO AN EQUATION
120 REM -LISTED IN LINE 10. THE USER SPECIFIES A RANGE TO SEARCH (FROM
    X1 TO X2) AND AND INCREMENT TO USE (X3)
130 REM -IF THE PROGRAM FINDS THAT THE FUNCTION HAS CROSSED THE X
    AXIS SOMEWHERE BETWEEN X1 AND X1+X3 THEN
140 REM -THEN THE BISECTION TECHNIQUE IS USED TO PINPOINT THE VALUE
    OF THE ROOT
150 REM -IF THE FUNCTION IS ABOVE THE X AXIS AT BOTH X1 AND X1+X3
    BUT HAS GONE BELOW THE X AXIS BETWEEN THE TWO, THE PROGRAM
    WON'T FIND THE ROOT
160 REM -IN THIS CASE THE INCREMENT VALUE (X3) SHOULD BE MADE
    SMALLER TO FIND THE PLACE ROOT
170 DEF FN RN(R) = INT (R * X5 + .5) / X5
180 HOME : LIST 10
190 E = 1E - 6
200 X1 = 0: REM -START RANGE
210 X2 = 10: REM -STOP RANGE
220 X3 = 1: REM -INCREMENT
230 X4 = .001
240 X5 = 10000
250 PRINT "BEGIN 'X' AT: ";X1
260 PRINT " END 'X' AT: ";X2
270 PRINT "INCREMENT BY: ";X3
280 PRINT : PRINT
290 Y1 = FN FT(X1)
300 IF ABS (Y1) < E THEN XR = X1: GOTO 500
310 XL = X1
320 YL = Y1
330 IF X1 > X2 THEN 560
340 X1 = X1 + X3
350 Y1 = FN FT(X1)
352 PRINT : PRINT : PRINT "LEFT CALCULATED BOUNDARY = ";XL: PRINT
    "FUNCTION VALUE AT LCB = ";YL
355 PRINT : PRINT "RIGHT CALCULATED BOUNDARY = ";X1: PRINT
    "FUNCTION VALUE AT RCB =";Y1
```

Fig. 29-7. Bisection method for determining roots of a function.

```
360 IF ABS (Y1) < E THEN XR = X1: GOTO 500
370 IF SGN (YL) * SGN (Y1) > 0 THEN 310
380 G1 = XL:V1 = YL
390 G2 = X1:V2 = Y1
400 XG = G1 + (G2 - G1) / 2: IF ABS (FN FT(XG)) < E THEN 480
410 IF ABS (G2 - G1) < E THEN 480
420 YG = FN FT(XG)
423 PRINT : PRINT "LEFT BISECTED BOUNDARY = ";G1: PRINT "FUNCTION
    VALUE AT LBB = ";V1
424 PRINT : PRINT "RIGHT BISECTED BOUNDARY =";G2: PRINT "FUNCTION
    VALUE AT RBB = ";V2
426 PRINT : PRINT "NEXT GUESS =";XG: PRINT "FUNCTION VALUE AT NG =
    ";YG
430 IF SGN (V1) * SGN (YG) > 0 THEN 460
440 G2 = XG:V2 = YG
450 GOTO 400
460 G1 = XG:V1 = YG
470 GOTO 400
480 XR = XG
490 REM *FOUND A ROOT
500 PRINT : PRINT : PRINT "A ROOT EXISTS AT "; FN RN(XR)
505 PRINT : PRINT : PRINT
510 XL = XR + X4 * X3
520 YL = FN FT(XL)
530 X1 = XL + X3
540 Y1 = FN FT(X1)
550 GOTO 352
560 END
RUN

10  DEF FN FT(X)=XΛ3 - 12 * XΛ2 + 47 * X - 60
BEGIN 'X' AT: 0
END 'X' AT: 10
INCREMENT BY: 1
LEFT CALCULATED BOUNDARY = 0
FUNCTION VALUE AT LCB = -60
RIGHT CALCULATED BOUNDARY = 1
FUNCTION VALUE AT RCB = -24
LEFT CALCULATED BOUNDARY = 1
FUNCTION VALUE AT LCB = -24
RIGHT CALCULATED BOUNDARY = 2
FUNCTION VALUE AT RCB = -6
LEFT CALCULATED BOUNDARY = 2
FUNCTION VALUE AT LCB = -6
RIGHT CALCULATED BOUNDARY = 3
FUNCTION VALUE AT RCB = 0
A ROOT EXISTS AT 3
LEFT CALCULATED BOUNDARY = 3.001
```

Fig.29-7–cont. Bisection method for determining roots of a function.

```
FUNCTION VALUE AT LCB = 1.99697912E-03
RIGHT CALCULATED BOUNDARY = 4.001
FUNCTION VALUE AT RCB = -1.00015104E-03
LEFT BISECTED BOUNDARY = 3.001
FUNCTION VALUE AT LBB = 1.99697912E-03
RIGHT BISECTED BOUNDARY = 4.001
FUNCTION VALUE AT RBB = -1.00015104E-03
NEXT GUESS = 3.501
FUNCTION VALUE AT NG = .374748483
LEFT BISECTED BOUNDARY = 3.501
FUNCTION VALUE AT LBB = 1.99697912E-03
RIGHT BISECTED BOUNDARY = 4.001
FUNCTION VALUE AT RBB = -1.00015104E-03
NEXT GUESS = 3.751
FUNCTION VALUE AT NG = .233561739
LEFT BISECTED BOUNDARY = 3.751
FUNCTION VALUE AT LBB = 1.99697912E-03
RIGHT BISECTED BOUNDARY = 4.001
FUNCTION VALUE AT RBB = -1.00015104E-03
NEXT GUESS = 3.876
FUNCTION VALUE AT NG = .122093305
LEFT BISECTED BOUNDARY = 3.876
FUNCTION VALUE AT LBB = 1.99697912E-03
RIGHT BISECTED BOUNDARY = 4.001
FUNCTION VALUE AT RBB = -1.00015104E-03
NEXT GUESS = 3.9385
FUNCTION VALUE AT NG = .0612673014
LEFT BISECTED BOUNDARY = 3.9385
FUNCTION VALUE AT LBB = 1.99697912E-03
RIGHT BISECTED BOUNDARY = 4.001
FUNCTION VALUE AT RBB = -1.00015104E-03
  NEXT GUESS = 3.96975
FUNCTION VALUE AT NG = .0302222818
LEFT BISECTED BOUNDARY = 3.96975
FUNCTION VALUE AT LBB = 1.99697912E-03
RIGHT BISECTED BOUNDARY = 4.001
FUNCTION VALUE AT RBB = -1.00015104E-03
NEXT GUESS = 3.985375
FUNCTION VALUE AT NG = .0146218389
LEFT BISECTED BOUNDARY = 3.985375
FUNCTION VALUE AT LBB = 1.99697912E-03
RIGHT BISECTED BOUNDARY = 4.001
FUNCTION VALUE AT RBB = -1.00015104E-03
NEXT GUESS = 3.9931875
FUNCTION VALUE AT NG = 6.81217015E-03
```

Fig.29-7–cont. Bisection method for determining roots of a function.

```
LEFT BISECTED BOUNDARY = 3.9931875
FUNCTION VALUE AT LBB = 1.99697912E-03
RIGHT BISECTED BOUNDARY = 4.001
FUNCTION VALUE AT RBB = -1.00015104E-03
NEXT GUESS = 3.99709375
FUNCTION VALUE AT NG = 1.90621817E-03
LEFT BISECTED BOUNDARY = 3.99709375
FUNCTION VALUE AT LBB = 1.99697912E-03
RIGHT BISECTED BOUNDARY = 4.001
FUNCTION VALUE AT RBB = -1.00015104E-03
NEXT GUESS = 3.99904688
FUNCTION VALUE AT NG = 9.53122973E-04
LEFT BISECTED BOUNDARY = 3.99904688
FUNCTION VALUE AT LBB = 1.99697912E-03
RIGHT BISECTED BOUNDARY = 4.001
FUNCTION VALUE AT RBB = -1.00015104E-03
NEXT GUESS = 4.00002344
FUNCTION VALUE AT NG = -2.35885382E-05
LEFT BISECTED BOUNDARY = 3.99904688
FUNCTION VALUE AT LBB = 1.99697912E-03
RIGHT BISECTED BOUNDARY = 4.00002344
FUNCTION VALUE AT RBB = -1.00015104E-03
NEXT GUESS = 3.99953516
FUNCTION VALUE AT NG = 4.64841723E-04
LEFT BISECTED BOUNDARY = 3.99953516
FUNCTION VALUE AT LBB = 1.99697912E-03
RIGHT BISECTED BOUNDARY = 4.00002344
FUNCTION VALUE AT RBB = -1.00015104E-03
NEXT GUESS = 3.9997793
FUNCTION VALUE AT NG = 2.20701098E-04
LEFT BISECTED BOUNDARY = 3.9997793
FUNCTION VALUE AT LBB = 1.99697912E-03
RIGHT BISECTED BOUNDARY = 4.00002344
FUNCTION VALUE AT RBB = -1.00015104E-03
NEXT GUESS = 3.99990137
FUNCTION VALUE AT NG = 9.86009836E-05
LEFT BISECTED BOUNDARY = 3.99990137
FUNCTION VALUE AT LBB = 1.99697912E-03
RIGHT BISECTED BOUNDARY = 4.00002344
FUNCTION VALUE AT RBB = -1.00015104E-03
NEXT GUESS = 3.9999624
FUNCTION VALUE AT NG = 3.75956297E-05
LEFT BISECTED BOUNDARY = 3.9999624
FUNCTION VALUE AT LBB = 1.99697912E-03
RIGHT BISECTED BOUNDARY = 4.00002344
```

Fig.29-7–cont. Bisection method for determining roots of a function.

```
FUNCTION VALUE AT RBB = -1.00015104E-03
NEXT GUESS = 3.99999292
FUNCTION VALUE AT NG = 7.07805157E-06
LEFT BISECTED BOUNDARY = 3.99999292
FUNCTION VALUE AT LBB = 1.99697912E-03
RIGHT BISECTED BOUNDARY = 4.00002344
FUNCTION VALUE AT RBB = -1.00015104E-03
NEXT GUESS = 4.00000818
FUNCTION VALUE AT NG = -8.32974911E-06
LEFT BISECTED BOUNDARY = 3.99999292
FUNCTION VALUE AT LBB = 1.99697912E-03
RIGHT BISECTED BOUNDARY = 4.00000818
FUNCTION VALUE AT RBB = -1.00015104E-03
NEXT GUESS = 4.00000055
FUNCTION VALUE AT NG = 1.25318766E-05
LEFT BISECTED BOUNDARY = 4.00000055
FUNCTION VALUE AT LBB = 1.99697912E-03
RIGHT BISECTED BOUNDARY = 4.00000818
FUNCTION VALUE AT RBB = -1.00015104E-03
NEXT GUESS = 4.00000436
FUNCTION VALUE AT NG = -4.51505184E-06
LEFT BISECTED BOUNDARY = 4.00000055
FUNCTION VALUE AT LBB 5 1.99697912E-03
RIGHT BISECTED BOUNDARY = 4.00000436
FUNCTION VALUE AT RBB = -1.00015104E-03
NEXT GUESS = 4.00000246
FUNCTION VALUE AT NG= -2.66730785E-06
LEFT BISECTED BOUNDARY = 4.00000055
FUNCTION VALUE AT LBB = 1.99697912E-03
RIGHT BISECTED BOUNDARY = 4.00000246
FUNCTION VALUE AT RBB = -1.00015104E-03
NEXT GUESS = 4.0000015
FUNCTION VALUE AT NG = -1.65402889E-06
A ROOT EXISTS AT 4
LEFT CALCULATED BOUNDARY = 4.00100103
FUNCTION VALUE AT LCB = -1.00108981E-03
RIGHT CALCULATED BOUNDARY = 5.00100103
FUNCTION VALUE AT RCB = 2.00501084E-03
LEFT BISECTED BOUNDARY = 4.00100103
FUNCTION VALUE AT LBB = -1.00108981E-03
RIGHT BISECTED BOUNDARY = 5.00100103
FUNCTION VALUE AT RBB = 2.00501084E-03
NEXT GUESS = 4.50100103
FUNCTION VALUE AT NG = -.375248879
LEFT BISECTED BOUNDARY = 4.50100103
```

Fig.29-7–cont. Bisection method for determining roots of a function.

```
FUNCTION VALUE AT LBB = -1.00108981E-03
RIGHT BISECTED BOUNDARY = 5.00100103
FUNCTION VALUE AT RBB = 2.00501084E-03
NEXT GUESS = 4.75100103
FUNCTION VALUE AT NG = -.32743457
LEFT BISECTED BOUNDARY = 4.75100103
FUNCTION VALUE AT LBB = -1.00108981E-03
RIGHT BISECTED BOUNDARY = 5.00100103
FUNCTION VALUE AT RBB = 2.00501084E-03
NEXT GUESS = 4.87600103
FUNCTION VALUE AT NG = -.203777343
LEFT BISECTED BOUNDARY = 4.87600103
FUNCTION VALUE AT LBB = -1.00108981E-03
RIGHT BISECTED BOUNDARY = 5.00100103
FUNCTION VALUE AT RBB = 2.00501084E-03
NEXT GUESS = 4.93850103
FUNCTION VALUE AT NG = -.111884207
LEFT BISECTED BOUNDARY = 4.93850103
FUNCTION VALUE AT LBB = -1.00108981E-03
RIGHT BISECTED BOUNDARY = 5.00100103
FUNCTION VALUE AT RBB = 2.00501084E-03
NEXT GUESS = 4.96975103
FUNCTION VALUE AT NG = -.0577806532
LEFT BISECTED BOUNDARY = 4.96975103
FUNCTION VALUE AT LBB = -1.00108981E-03
RIGHT BISECTED BOUNDARY = 5.00100103
FUNCTION VALUE AT RBB = 2.00501084E-03
NEXT GUESS = 4.98537603
FUNCTION VALUE AT NG = -.028609544
LEFT BISECTED BOUNDARY = 4.98537603
FUNCTION VALUE AT LBB = -1.00108981E-03
RIGHT BISECTED BOUNDARY = 5.00100103
FUNCTION VALUE AT RBB = 2.00501084E-03
NEXT GUESS = 4.99318853
FUNCTION VALUE AT NG = -.0134841502
LEFT BISECTED BOUNDARY = 4.99318853
FUNCTION VALUE AT LBB = -1.00108981E-03
RIGHT BISECTED BOUNDARY = 5.00100103
FUNCTION VALUE AT RBB = 2.00501084E-03
NEXT GUESS = 4.99709478
FUNCTION VALUE AT NG = -5.78519702E-03
LEFT BISECTED BOUNDARY = 4.99709478
FUNCTION VALUE AT LBB = -1.00108981E-03
RIGHT BISECTED BOUNDARY = 5.00100103
FUNCTION VALUE AT RBB = 2.00501084E-03
```

Fig.29-7–cont. Bisection method for determining roots of a function.

```
NEXT GUESS = 4.9990479
FUNCTION VALUE AT NG = -1.90153718E-03
LEFT BISECTED BOUNDARY = 4.9990479
FUNCTION VALUE AT LBB = -1.00108981E-03
RIGHT BISECTED BOUNDARY = 5.00100103
FUNCTION VALUE AT RBB = 2.00501084E-03
NEXT GUESS = 5.00002447
FUNCTION VALUE AT NG = 4.88460064E-05
LEFT BISECTED BOUNDARY = 4.9990479
FUNCTION VALUE AT LBB = -1.00108981E-03
RIGHT BISECTED BOUNDARY = 5.00002447
FUNCTION VALUE AT RBB = 2.00501084E-03
NEXT GUESS = 4.99953619
FUNCTION VALUE AT NG = -9.27060843E-04
LEFT BISECTED BOUNDARY = 4.99953619
FUNCTION VALUE AT LBB = -1.00108981E-03
RIGHT BISECTED BOUNDARY = 5.00002447
FUNCTION VALUE AT RBB = 2.00501084E-03
NEXT GUESS = 4.99978033
FUNCTION VALUE AT NG = -4.3925643E-04
LEFT BISECTED BOUNDARY = 4.99978033
FUNCTION VALUE AT LBB = -1.00108981E-03
RIGHT BISECTED BOUNDARY = 5.00002447
FUNCTION VALUE AT RBB = 2.00501084E-03
NEXT GUESS = 4.9999024
FUNCTION VALUE AT NG = -1.95235014E-04
LEFT BISECTED BOUNDARY = 4.9999024
FUNCTION VALUE AT LBB = -1.00108981E-03
RIGHT BISECTED BOUNDARY = 5.00002447
FUNCTION VALUE AT RBB = 2.00501084E-03
NEXT GUESS = 4.99996343
FUNCTION VALUE AT NG = -7.32243061E-05
LEFT BISECTED BOUNDARY = 4.99996343
FUNCTION VALUE AT LBB = -1.00108981E-03
RIGHT BISECTED BOUNDARY = 5.00002447
FUNCTION VALUE AT RBB = 2.00501084E-03
NEXT GUESS = 4.99999395
FUNCTION VALUE AT NG = -1.21891499E-05
LEFT BISECTED BOUNDARY = 4.99999395
FUNCTION VALUE AT LBB = -1.00108981E-03
RIGHT BISECTED BOUNDARY = 5.00002447
FUNCTION VALUE AT RBB = 2.00501084E-03
NEXT GUESS = 5.00000921
FUNCTION VALUE AT NG = 1.83284283E-05
LEFT BISECTED BOUNDARY = 4.99999395
```

Fig.29-7–cont. Bisection method for determining roots of a function.

```
FUNCTION VALUE AT LBB = -1.00108981E-03
RIGHT BISECTED BOUNDARY = 5.00000921
FUNCTION VALUE AT RBB =2.00501084E-03
NEXT GUESS = 5.00000158
FUNCTION VALUE AT NG = 3.06963921E-06
LEFT BISECTED BOUNDARY = 4.99999395
FUNCTION VALUE AT LBB = -1.00108981E-03
RIGHT BISECTED BOUNDARY = 5.00000158
FUNCTION VALUE AT RBB = 1.00501084E-03
NEXT GUESS = 4.99999776
FUNCTION VALUE AT NG = -4.55975533E-06
A ROOT EXISTS AT 5
LEFT CALCULATED BOUNDARY = 5.00099967
FUNCTION VALUE AT LCB = 2.00216472E-03
RIGHT CALCULATED BOUNDARY = 6.00099967
FUNCTION VALUE AT RCB = 6.01100224
LEFT CALCULATED BOUNDARY = 6.00099967
FUNCTION VALUE AT LCB = 6.01100224
RIGHT CALCULATED BOUNDARY = 7.00099967
FUNCTION VALUE AT RCB = 24.0260003
LEFT CALCULATED BOUNDARY = 7.00099967
FUNCTION VALUE AT LCB = 24.0260003
RIGHT CALCULATED BOUNDARY = 8.00099968
FUNCTION VALUE AT RCB = 60.0469966
LEFT CALCULATED BOUNDARY = 8.00099968
FUNCTION VALUE AT LCB = 60.0469966
RIGHT CALCULATED BOUNDARY = 9.00099967
FUNCTION VALUE AT RCB = 120.073991
LEFT CALCULATED BOUNDARY = 9.00099967
FUNCTION VALUE AT LCB = 120.073991
RIGHT CALCULATED BOUNDARY = 10.0009997
FUNCTION VALUE AT RCB = 210.106983
```

Fig.29-7–cont. Bisection method for determining roots of a function.

The logic of the bisection method is to set a specific range to determine if the function crosses the X axis. The function (F1) is assigned a specific X value which is the beginning range on the X axis. The beginning X value is added to the X at the end of the range, and the values are averaged. The average X value is used to calculate the second function value (F3).

When F1 is multiplied by F3, the value is greater than zero, (above the X axis), equal to zero (on the X axis), or less than zero (below the X axis).

The value of F1 * F3 is used to adjust the beginning or ending range value to determine where the function crosses the X axis. The program in Fig. 29-7 was developed to determine all the roots of a function over a wide range.

To test the program, the beginning range X1, ending range X2, and increment X3, lines 200-220, were assigned values. Lines 200-240 could be replaced with INPUT statements for user convenience. The function (line 10) was placed in a DEF FN statement. The user may enter almost any function in line 10 to determine the roots.

The program will not accurately determine the roots of a function that is rapidly oscillating, for example, a trigonometric function that oscillates across the X axis very rapidly.

In line 290, Y1 = FN FT (X1), the function value (Y) is calculated at the beginning of the X range. This produces a value in relation to the Y axis.

Line 300 tests to determine if the X1 location produces a root of the function. If ABS(Y1) < E (1E-6) the function is located on the X axis, and the program branches to line 500.

The routine that begins at line 500 prints out the value of the root, and resets the values over the X range so the next root can be discovered.

If line 300 is FALSE, the program defaults to line 310. If the Y value was not zero, the beginning range X and Y values are assigned to the left side of the subrange, into XL and YL.

Line 330 is a check to determine if the entire "X" range has been scanned. If it has, the program branches to line 560, which is the END of the program.

In line 340, the increment (X3 = 1) is added to the beginning of the subrange and the value is assigned to X1. The value of the function is again calculated in line 350.

Lines 352, 355, 423, 424, and 426 were an afterthought to allow the beginning programmer to view how the program searched the X axis to determine when a root had been found. A nonteaching program would delete lines 352, 355, 423, 424, and 426.

Line 360 again checks to determine if a root has been discovered. Line 370 uses the SGN function (Table 29-1) to determine the location of the function in relation to the Y axis.

If the Y value has the same SGN at the left subrange boundary as the SGN at the right boundary, the function has not crossed the X axis. The program branches to line 310 to reset the left boundary values.

If line 370 is FALSE, the program defaults to set the right subrange values. V1, and V2 are values of the function at the left and right boundaries of the subrange.

Line 400 calculates the next guess as half the distance between guess #1 and guess #2. This is the logic of the bisection method, it cuts the range in half on each guess.

If line 410 is TRUE, the difference between the two guesses is very close to the root of the function.

Line 420 determines the value of the function at the calculated guess.

Line 430 is similar to line 370. It checks to determine the function in rela-

tion to the X axis. Fig. 29-8 shows the variable relationships *before* if line 430 is TRUE. Fig. 29-9 shows the variable relationships *after* if line 430 is FALSE. Fig. 29-10 shows the variable relationships on the second guess. These three figures illustrate the logic of cutting a subrange in half and again determining on which side of the break the function crosses the X axis.

Table 29-1. Table of Possibilities for Y Value of the Function FT to Determine if the Function Has Crossed the X Axis

		SGN(YL)		
		−1	0	+1
SGN(Y1)	−1	BOTH VALUES BELOW THE "X" AXIS	XL IS A ROOT	YL ABOVE "X" AXIS Y1 BELOW "X"
	0	X1 IS A ROOT LINE 360 GOT IT	X1 IS A ROOT LINE 360 GOT IT	X1 IS A ROOT LINE 360 GOT IT
	+1	YL BELOW "X" AXIS Y1 ABOVE "X" AXIS	XL IS A ROOT	BOTH VALUES ABOVE THE "X" AXIS

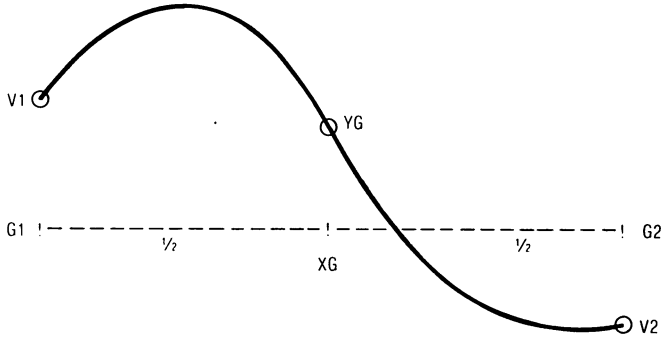


Fig. 29-8. Values *before* if line 430 is true.

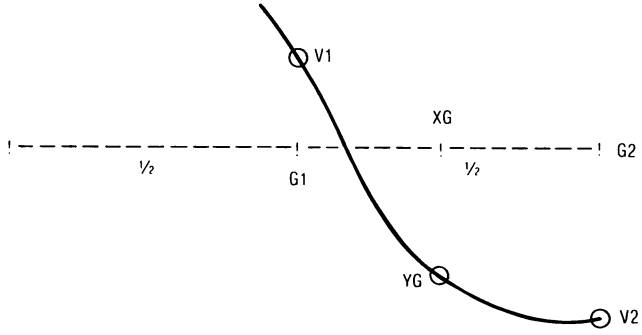


Fig. 29-9. Values *after* if line 430 is false.

Lines 380 and 390 save the values of the left boundary in G1 and the function at the left boundary in V1, and save the value at the right boundary in G2 and its function value in V2. These values will be used to start the bisection logic.

In Line 400, XG is calculated to be half the distance between G1 and G2. The second statement in line 400 checks to determine if the value of the function at that point is less than the error allowed for the root value.

Line 410 checks to determine if the difference between the left and right subrange boundaries is small enough to make further guessing unnecessary.

In line 420, the value of the function at XG is determined and stored in YG.

Line 430 will, in a manner similar to line 370, determine whether the function at XG is on the same side of the X axis as V1. The statement will be TRUE if both points are either above or below the X axis. The program will then branch to line 460 to reset the left subrange boundary to the value of XG.

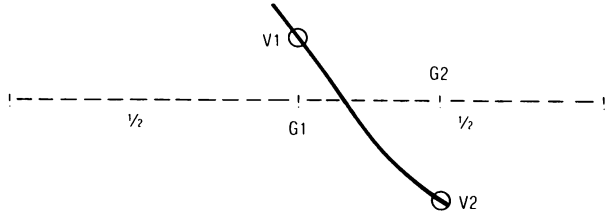


Fig. 29-10. Values on the second guess.

If line 430 is FALSE, then the program defaults to line 440 to set the right subrange boundary to the calculated guess (XG) which cuts the subrange in half. The value of the function at the new right boundary is then transferred into V2. The program then branches back to line 400 to repeat the process.

Line 460 eliminates the left half of the subrange by replacing the value of G1 by XG, and stores the value of the function at XG into V1.

When a root is discovered, the program branches to line 500. The routine that begins at line 500, prints the rounded value of the root to five places (X5 = 10000). Line 510 reestablishes the left boundary of the subrange very close to the location where the root was found. (XL = XR + (X4 = .001) * (X3 = 1).

Line 520 calculates the function value at the new left boundary of the subrange.

Line 530 calculates the new right boundary of the subrange, and line 540 calculates the function value at the right boundary of the subrange.

Line 550 causes the program to jump back to determine if there are any roots in the range.

TRAPEZOIDAL METHOD

The trapezoidal method can be used to approximate the area under a function over a specific range. When a function is integrated it gives the area of the function. The trapezoidal rule is used to determine area when a function is difficult to integrate. The program in Fig. 29-11 uses X^2 as the function to integrate: This is a simple function to integrate, but it is an easy example for a learning experience.

$$X \wedge 2\ DX = \frac{X \wedge 3}{3} = \left(\frac{8}{3} - \frac{1}{3}\right) = \frac{7}{3} = 2.333$$

The trapezoidal method of approximation uses the following formula.

$$T = (B - A)/(2*N) * (Y(0) + 2Y(1) + 2Y(2) + \ldots 2Y(n-1) + Y(n))$$

The trapezoidal method divides the area under a curve (Fig. 29-12), calculates the area of each trapezoid, and adds all these values to produce a final value.

The trapezoidal program (Fig. 29-11) allows the user to enter the beginning ranges of the function in relation to the X axis, and the number of divisions (lines 140-160).

The subroutine that begins at line 230 calculates the size of the step used to divide the area (DX = (B − A) / N). Line 240 assigns the beginning range on the X axis (A), to the variable DS. The multiplying factor is calculated in line 250 and the area (MS) is initialized to zero (0).

```
100 REM -AREA UNDER A FUNCTION USING THE TRAPEZOIDAL METHOD
110 REM -THIS PROGRAM WILL CALCULATE THE AREA UNDER A CURVE FROM
    POINT "A" TO POINT "B" USING THE DEFINED FUNCTION FA AT LINE 130
120 REM -THE USER IS ASKED TO ENTER THE A AND B POINTS AND THE NUM-
    BER OF DIVISIONS TO BE USED IN THE CALCULATION
130 DEF FN FA(B)=B^2
140 PRINT : INPUT "BEGIN 'X' AT: ";A
150 PRINT : INPUT " END 'X' AT: ";B
160 PRINT : INPUT "# OF DIVISIONS: ";N
170 PRINT : PRINT : GOSUB 230: PRINT
180 PRINT "THE AREA UNDER THE FUNCTION FROM X = ";A: PRINT
190 PRINT "TO X = ";B: PRINT
200 PRINT SPC( 10);"EQUALS ' ";T;" ' "
210 PRINT : PRINT
220 END
230 DX = (B - A) / N
240 DS = A
```

Fig.29-11. Trapezoidal program to determine area under a function.

```
250 MF = (B - A) /(2 * N)
260 MS = 0
270 FOR J = 0 TO N
280 TA = ABS ( FN FA(DS))
290 IF J = 0 OR J = N THEN 310
300 TA = TA * 2
310 MS = MS + TA
320 DS = DS + DX
330 NEXT J
340 T = MF * MS
350 RETURN
RUN
BEGIN 'X' AT: 1
  END 'X' AT: 2
# OF DIVISIONS: 4
THE AREA UNDER THE FUNCTION FROM X = 1
TO X = 2
    EQUALS '2.34375'
```

Fig.29-11–cont. Trapezoidal program to determine area under a function.

The loop in line 270 increments from zero, the beginning on the X axis, to the number of divisions selected by the user.

There are three cases to be handled. Case 1, the beginning of the range is squared and added to the total value. Case 2, the end of the range is squared and added to the total value. Case 3, all intermediate values are multiplied by two (2).

Line 280 adds the total area of each loop increment, and line 310 sums up the total as each area is calculated.

Line 320, DS = DS + DX, adds the step increment (DX = (B − A)/ N) on each loop execution.

When the last loop execution is complete, the total area (MS) is multiplied by the multiplying factor (MF) to produce the total area under the function.

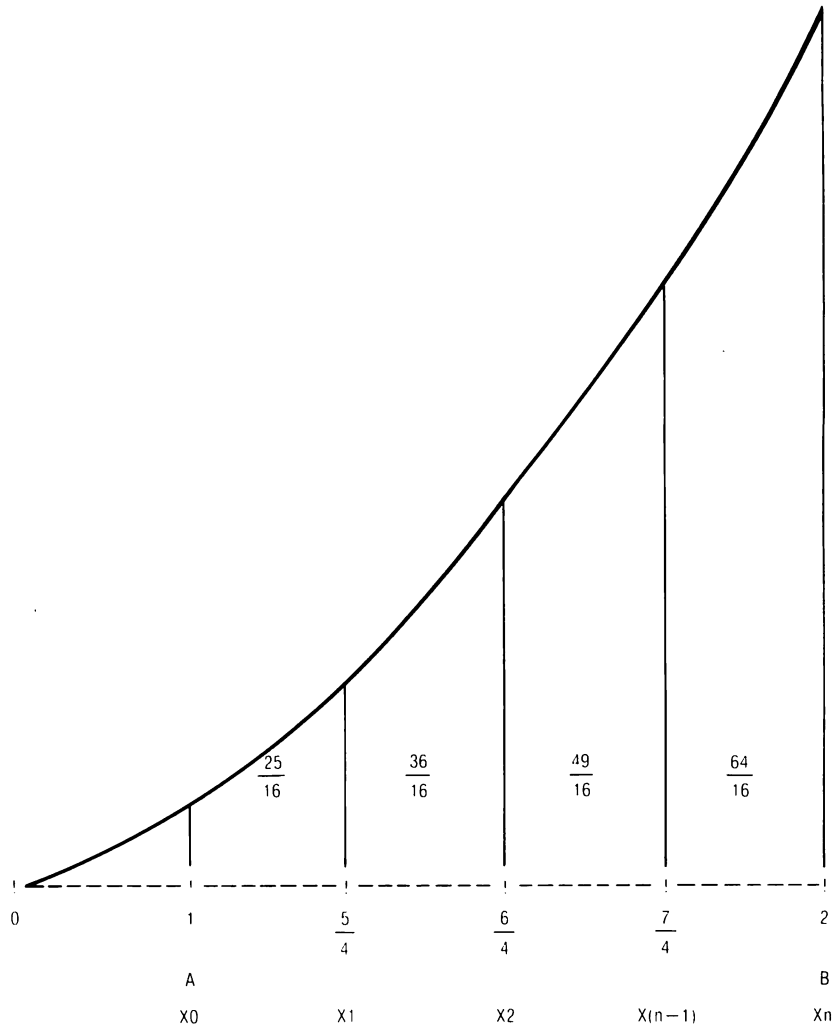


Fig. 29-12. Trapezoidal method for area under a curve.

SECTION II

80 Column Mode

LESSON 30

80 Column Mode

The 80 column mode in the Apple IIe is designed primarily for the Apple Writer II word processing system. The special keys on the Apple IIe are designed especially for the Apple Writer II (with the exception of a memory test function). Those special keys are TAB, DELETE, SOLID APPLE, OPEN APPLE, CAPS LOCK (upper and lower case), and the four special character keys. These keys were discussed in Lesson 3.

Programs can be written and run in the 80 column mode. The programs must be written in UPPER CASE (CAPS LOCK KEY DEPRESSED). Lower case can be used in the DATA, PRINT, and REM statements. Both upper and lower case characters can be produced on the printer, and the output will be produced in the 80 column mode. The software that controls the 80 column mode produces some unexpected results when the printer is turned "on," and "off." (In this example, the printer was a TRS-80, DWP-410 impact printer, MICROTEC, INC., RV-611C parallel printer interface card in slot #1.) To activate the 80 column card type this command.

```
PR#3
```

Once the 80 column card is activated, it can be changed between the 40 column mode and the 80 column mode in one of two ways.

CONTROL Q — 40 column mode with the 80 column card active.
CONTROL R — 80 column mode with the 80 column card active.
ESCAPE 40 — 40 column mode with the 80 column card active.
ESCAPE 80 — 80 column mode with the 80 column card active.

When the 80 column card is inactive, the cursor is a blinking checkerboard pattern. When the 80 column card is active, the cursor is solid and nonblinking.

The following sequence of commands causes the 80 column card to be activated, turns the printer "on," RUNs the program, and turns the printer "off."

PR#3 — 80 column card active.

PR#1 — Turns the printer "on." It also causes any letters typed to the screen to be superimposed over the cursor.

RUN — The "R" is overwritten by the "U," which is overwritten by the "N." This causes the program to RUN and the output goes to the printer and the screen.

PR#0 — Turns the printer "off," but causes the program listing to the screen to be printed in the 80 column mode. However, there is a space between each character. This happens to the screen, but if the program is RUN again to the printer (before the printer is turned "off"), there is no change to the printer output.

```
210 PRINT "47"
2 1 0  P R I N T  " 4 7 "
```

The following list of commands turns the 80 column card "on," turns the printer "on," RUNs the program, and turns the printer "off" with no abnormal spacing to the CRT.

PR#3 — 80 column card active.

PR#1 — Turns the printer "on" and causes any character typed on the screen to be superimposed over the cursor.

RUN— "R" is overwritten by "U," which is overwritten by "N."

PR#3 — Turns the printer "off," leaves the 80 column card active, and does not cause unusual spacing on the screen.

As discussed previously, programs written in the 80 column mode must use upper case letters. DATA, PRINT, and REM statements can use lower case letters.

Fig. 30-1 is a program and RUN to demonstrate the use of VTAB, HTAB, and TAB functions. The VTAB function operates in the range from one (1) through twenty-three (23). VTAB 0 causes an ?ILLEGAL QUANTITY ERROR. VTAB 24 causes the top line of the screen to be pushed up (out of sight) as the screen scrolls.

HTAB tabs from one (1) through forty (40), even in the 80 column mode. HTAB 0 produces undesirable results, but no error message. Fig. 30-2 demonstrates the effects of using an HTAB greater than forty (40). The program causes forty asterisks to be printed on one line. When the forty limit is reached, the HTAB function closes out the line, and prints the last ten asterisks on separate lines. It is suggested that an HTAB greater than thirty-nine (39) not be used in the 80 column mode. Use TAB or SPC to format at greater than thirty-nine columns.

The TAB function ranges from a value of TAB(1) through TAB(79). TAB 0 causes the output to be printed incorrectly, but does not generate an error message. TAB 80 causes the print to wrap around the screen and be printed in column 1, row 2.

```
100 REM —this is a program to demonstrate the tab and space functions
110 HOME
120 VTAB 1: HTAB 1: PRINT "A"; TAB(79);"B"
130 VTAB 11: PRINT SPC( 40);"C"
140 VTAB 23: PRINT "D"; TAB( 79);"E";: VTAB 12
150 END
RUN
A                                                                    B
                                        C
D                                                                    E
```

NOTE — THE RUN OF Fig. 30-1 OUTPUTS TO THE PRINTER IN THE MODE DEMONSTRATED ABOVE.
THE OUTPUT TO THE SCREEN OF THE APPLE IIe COMPUTER IS SHOWN BELOW.

```
A                                                                    B


                                        C


D                                                                    E
```

Fig. 30-1. 80 column card active. TAB and SPC functions.

```
100 FOR J = 1 TO 50
110 HTAB (J): PRINT "*";
120 NEXT J
130 END
RUN
****************************************
*
  *
    *
      *
        *
          *
            *
              *
                *
                  *
```

Fig. 30-2. 80 column card active. HTAB function demonstration.

Fig. 30-3 demonstrates that eighty columns can be printed by using the integers zero (0) through 79, or one (1) through 80, to print out 80 columns across the screen or the printer.

Fig. 30-4 demonstrates how the program is written using upper case letters (note — 40 column printout) but the PRINT statements can output both upper case and lower case letters. PRINT SPC(60) moves the cursor out sixty spaces and prints Brian; SPC(5) moves the cursor out five more spaces and prints George. The TAB function tabs out (50 + J) and prints the value of J for each increment of the loop.

```
100 REM — FOR J = 0 TO 79 PRODUCES THE SAME RESULTS AS 'FOR J = 1 TO
    80
110 FOR J = 1 TO 80
120 PRINT ''*'';
130 NEXT J
140 END
RUN
********************************************************************************
```

Fig. 30-3. 80 column card active. 80 column output.

```
100 VTAB 10
110 PRINT SPC( 60);''Brian''; SPC( 5):''George''
120 FOR J = 1 TO 5
130 PRINT TAB( 50 + J);'' J = '';J
140 NEXT J
150 END
RUN
                                                    Brian       George
                                                     J = 1
                                                       J = 2
                                                         J = 3
                                                           J = 4
                                                             J = 5
```

Fig. 30-4. 80 column card active. SPC and TAB function demonstration.

Fig. 30-5 is a program that demonstrates how a COMMA affects spacing. A RUN of the program causes a different output on the screen than it does on the printer. On the screen, COMMA (,) spacing causes the first print to be placed in columns 1, 2, and 3, fifteen (15) spaces are skipped, and all other values, except the characters placed in columns 1, 2, and 3, are overwritten in the same three columns (16, 17, and 18).

Fig. 30-6 is a program written to demonstrate SEMICOLON (;) spacing. As in the 40 column mode, the output is packed.

Fig. 30-7 demonstrates spaces between quotation marks for spacing output. The SPC(16) and TAB(16) in the REM statements produce similar results.

PRINT L(I,J,K);" "; causes the triple subscripted array program to output a total of fifteen (15) characters and sixty-four (64) spaces. The fifteen characters and sixty-four spaces add to a total of seventy-nine (79) columns to fill the 80 column screen.

Fig. 30-8 is a program to set the scrolling window (screen) to its maximum limits. POKE is a command to place a value into a memory location and may be used in the immediate or deferred mode. POKE is followed by two expressions.

POKE 32,0

```
100 N = 5
110 DIM L(5,5,5)
120 FOR I = 1 TO N
130 DEF FN FA(B)=BΛ2
140 FOR K = 1 TO N
150 L(I,J,K) = I * 100 + J * 10 + K
160 NEXT K,J,I
170 FOR I = 1 TO N
180 FOR J = 1 TO N
190 FOR K = 1 TO N
200 PRINT L(I,J,K),
210 NEXT K
220 PRINT
230 NEXT J
240 PRINT
250 NEXT I
260 END
RUN
 111          112          113
 114          115
 121          122          123
 124          125
 131          132          133
 134          135
 141          142          143
 144          145
 151          152          153
 154          155
 211          212          213
 214          215
 221          222          223
 224          225
 231          232          233
 234          235
 241          242          243
 244          245
 251          252          253
 254          255
 311          312          313
 314          315
 321          322          323
 324          325
 331          332          333
 334          335
 341          342          343
 344          345
 351          352          353
 354          355
 411          412          413
```

Fig. 30-5. 80 column card active. Triple subscripted array comma spacing.

```
414             415
421             422             423
424             425
431             432             433
434             435
441             442             443
444             445
451             452             453
454             455
511             512             513
514             515
521             522             523
524             525
531             532             533
534             535
541             542             543
544             545
551             552             553
554             555
```

Fig. 30-5–cont. 80 column card active. Triple subscripted array comma spacing.

The first expression is the specific memory location from −65535 to 65536. The second expression is an integer from zero (0) to 255.

There are four memory locations that deal with cursor window position. The POKEs are ABSOLUTE values.

POKE 32,0 — Sets left window cursor position.
POKE 33,80 — Sets window width position. Locations 32 and 33 sum to give the right edge of the window.
POKE 34,0 — Sets top window cursor position.
POKE 35,24 — Sets bottom window cursor position.

Using the four POKE commands in line 10 (Fig. 30-8) sets the cursor positions to their maximum limits on the 80 column screen.

```
100 N = 5
110 DIM L(5,5,5)
120 FOR I = 1 TO N
130 FOR J = 1 TO N
140 FOR K = 1 TO N
150 L(I,J,K) = I * 100 + J * 10 + K
160 NEXT K,J,I
170 FOR I = 1 TO N
180 FOR J = 1 TO N
190 FOR K = 1 TO N
200 PRINT L(I,J,K);
210 NEXT K
220 PRINT
```

Fig. 30-6. 80 column card active. Triple subscripted array semicolon spacing.

```
230 NEXT J
240 PRINT
250 NEXT I
260 END
RUN
111112113114115
121122123124125
131132133134135
141142143144145
151152153154155
211212213214215
221222223224225
231232233234235
241242243244245
251252253254255
311312313314315
321322323324325
331332333334335
341342343344345
351352353354355
411412413414415
421422423424425
431432433434435
441442443444445
451452453454455
511512513514515
521522523524525
531532533534535
541542543544545
551552553554555
```

Fig.30-6–cont. 80 column card active. Triple subscripted array semicolon spacing.

```
100 N = 5
110 DIM L(5,5,5)
120 FOR I = 1 TO N
130 FOR J = 1 TO N
140 FOR K = 1 TO N
150 L(I,J,K) = I * 100 + J * 10 + K
160 NEXT K,J,I
170 FOR I = 1 TO N
180 FOR J = 1 TO N
190 FOR K = 1 TO N
200 REM —PRINT L(I,J,K);SPC(16); PRODUCES 16 SPACES AS DOES THE LINE
    BELOW
210 REM —PRINT L(I,J,K);TAB(16); PRODUCES 16 SPACES AS DOES THE LINE
    BELOW
220 PRINT L(I,J,K);" ";: REM —16 SPACES BETWEEN QUOTES
```

Fig. 30-7. 80 column card active. Triple subscripted array quotation mark *spacing.*

```
230 NEXT K
240 PRINT
250 NEXT J
260 PRINT
270 NEXT I
280 END
RUN
111          112          113          114          115
121          122          123          124          125
131          132          133          134          135
141          142          143          144          145
151          152          153          154          155

211          212          213          214          215
221          222          223          224          225
231          232          233          234          235
241          242          243          244          245
251          252          253          254          255

311          312          313          314          315
321          322          323          324          325
331          332          333          334          335
341          342          343          344          345
351          352          353          354          355

411          412          413          414          415
421          422          423          424          425
431          432          433          434          435
441          442          443          444          445
451          452          453          454          455

511          512          513          514          515
521          522          523          524          525
531          532          533          534          535
541          542          543          544          545
551          552          553          554          555
```

Fig.30-7–cont. 80 column card active. Triple subscripted array question mark spacing.

```
10 POKE 32,0: POKE 33,80: POKE 34,0: POKE 35,24: HOME
20 PRINT SPC( 60); "Brian"; SPC( 5);"George"
30 END
RUN
                                                    Brian      George
```

Fig. 30-8. 80 column card active. Demonstration of POKE commands to control window size.

Many Apple IIe memory locations contain specific values that are necessary for proper operation of the computer. An illegal value POKEd into one of these memory locations may alter the system, program, or the Applesoft language. Apple reference manuals give a complete list of the values stored in each memory location.

The *command* to determine the value stored in a memory location is PRINT PEEK (J)

```
POKE 32,0
PRINT PEEK(32)     (RETURNS A VALUE OF ZERO)
0
```

SUMMARY — 80 COLUMN CARD ACTIVE

1. PR#3	Activates the 80 column card.
2. ESCAPE 40	40 column mode.
3. ESCAPE 80	80 column mode.
4. CONTROL Q	40 column mode.
5. CONTROL R	80 column mode.
6. ESCAPE CONTROL Q	Deactivates the 80 column card.

80 COLUMN CARD ACTIVATED

7. PR#1	Turns "on" the printer if the printer interface card is in slot #1.
8. PR#3	Turns "off" the printer but leaves the 80 column card activated.
9. PR#0	Turns off the printer but distorts the output to the CRT.
10. COMMA (,) SPACING	Prints in the first column, skips 15 columns and overprints all columns of data (other than the first columns).
11. SEMICOLON (;)	Semicolon packs output.
12. VTAB	Legal values from one (1) through twenty-three (23).
13. HTAB	Legal values one (1) through forty (40). Does not extend past 40 columns in the 80 column mode. It is suggested that HTAB > 39 NOT BE USED.
14. PRINT TAB	Legal values one (1) to eighty (80).
15. PRINT SPC	Legal values one (1) to eighty (80).

LESSON 31

Formatting in the Eighty Column Mode

Most printers have at least an eighty (80) column capability, and many have a 163 column capability at twelve characters per inch (12 pitch).

The program in Fig. 31-1 is one solution to the problem of formatting output in Applesoft. Using variables which indicate the type of value to be output, the size of the field, the type of justification, etc., and whether the output is to go to the printer or CRT, this program will format a line of output and help create the displays.

VARIABLE DICTIONARY

The variable dictionary is written directly in the program from lines 10 through 130.

Numeric input (NI), is the variable into which all numeric values are stored for output.

The field size (FS) is determined by how many columns the numeric input (NI) is going to use.

The column position (CP) is the position in which the field for outputting the value is to start. If the print string (P$) has been built to a column position of 40, and a column of 30 is specified, it will not cause the printer to back up. The program was written to add spaces to extend the print line further. The print line should be built one field at a time from left to right.

The decimal rounding (DR) variable is used to round a real number to a specific number of decimal places. When the decimal rounding variable is equal to zero (DR = 0), it means the numeric value is to be rounded to the one's place. There is no maximum or minimum limit placed on DR. If the number in NI is to the power of E+10 or E−10 (scientific notation), the rounding function will be bypassed.

FP$ is the string variable into which literals are placed for outputting. *This is* for headings or lines of explanation.

P$ is the string variable that outputs to the printer.

LP is the variable used to determine the length of FP$.

LL is the line length. In line 1000, LL is set to 75. The line length can be the same value as the line length of the printer.

SP$ is a string variable that holds 256 spaces. This is used for packing spaces into the printline and justifying the individual fields.

FJ is field justification. FJ = 0 will left justify the field. FJ = 1 will right justify the field. FJ = 2 will center justify the field.

Field type (FT) is the variable which determines if the field to be output is string or numeric type. FT = 0 indicates numeric, while FT = 1 indicates a string.

The variables J and L are used as loop variables, counters and position holders.

NS is the variable which indicates whether the eighty column card is active or not. This is used to return control to DOS(NS = 0) or the 80 column card(NS = 3) as is the case.

OS is used to indicate where the display line is being output. OS = 3 means the display is in 80 column mode. Any other value of OS means that the display is to a printer.

```
1    REM *FORMATTER SUBROUTINE
10   NI = 0: REM *NUMERIC INPUT
20   FS = 0: REM *FIELD SIZE
30   CP = 0: REM *COLUMN POSITION
40   DR = 0: REM *DECIMAL PLACES TO ROUND TO
50   FP$ = " ": REM *STRING FIELD VARIABLE
60   P$ = " ": REM *PRINT OUT STRING
70   LP = 0: REM *LENGTH OF FP$
80   LL = 0: REM *LINE LENGTH
90   SP$ = "          ":SP$ = SP$ + SP$:SP$ = SP$ + SP$:SP$ = SP$ + SP$:SP$
     = SP$ + SP$:SP$ = SP$ + LEFT$ (SP$,127)
110  FJ = 0: REM *FIELD JUSTIFICATION(0=LEFT;1=RIGHT;2=CENTER)
120  FT = 0: REM *FIELD TYPE(0=NUMBERIC;1=STRING)
130  OS = 1: REM *OUTPUT SLOT(3=80 COLUMN CARD;1=PRINTER)
150  GOTO 1000
200  IF FS > 0 THEN 230
210  PRINT P$:P$ = " ": IF FS = 0 THEN RETURN
220  FOR J = 1 TO ABS (FS): PRINT : NEXT J: RETURN
230  IF LEN (P$) < CP THEN P$ = P$ + LEFT$ (SP$,CP - LEN (P$))
240  IF FT THEN 350
250  IF ABS (NI) > = 1E10 OR ABS (NI) < = 1E - 10 THEN FP$ = STR$ (NI):
     GOTO 350
260  NI = INT (NI * 10^DR + .5) / 10^DR
270  FP$ = STR$ (NI)
280  FOR J = 1 TO LEN (FP$)
290  IF MID$ (FP$,J,1) = "." THEN 320
300  NEXT J:FP$ = FP$ + "."
310  IF DR < 1 THEN 350
```

Fig. 31-1. 80 column card active. 80 column formatter program.

```
320  IF LEN (FP$) = J + DR THEN 350
330  J = DR + J - LEN (FP$)
340  FOR L = 1 TO J:FP$ = FP$ + "0": NEXT L
350  LP = LEN (FP$)
360  IF LP > = FS THEN 420
370  ON FJ GOTO 390,400
380  FP$ = FP$ + LEFT$ (SP$,FS - LP): GOTO 420
390  FP$ = LEFT$ (SP$,FS - LP) + FP$: GOTO 420
395  REM
400  L = INT ((FS - LP) / 2):J = FS - LP - L: IF L > 0 THEN FP$ = LEFT$ (SP$,L) +
     FP$
410  IF J > 0 THEN FP$ = FP$ + LEFT$ (SP$,J)
420  P$ = P$ + FP$: IF LEN (P$) > LL THEN PRINT LEFT$ (P$,LL):P$ = RIGHT$
     (P$, LEN (P$) - LL)
430  RETURN
435  REM
1000 LL = 75:SN = 0: IF PEEK (33) > 40 THEN SN = 3
1010 D$ = CHR$ (4): IF OS = 3 THEN 1020
1015 PRINT D$;"PR#";OS: PRINT CHR$ (9);LL;"N"
1020 OR J = 1 TO LL - 1: PRINT "*";: NEXT J: PRINT
1030 CP = 0:FT = 1:FP$ = "TEST HEADER"
1040 FS = 75:FJ = 2
1050 GOSUB 200
1060 FS = - 2: GOSUB 200
1070 FT = 1:FS = 15:FJ = 1
1080 FOR M = 1 TO 4: READ FP$:CP = 15 * (M - 1)
1090 GOSUB 200: NEXT M:FS = - 1: GOSUB 200
1100 FT = 0:FJ = 1:DR = 3
1110 AVG = 0:CNT = 0
1120 FOR M = 1 TO 2
1130 FS = 15
1140 FOR N = 1 TO 4
1150 READ D
1160 AVG = AVG + D:CNT = CNT + 1
1170 NI = D:CP = 15 * (N - 1)
1180 GOSUB 200
1190 NEXT N
1200 FS = 0: GOSUB 200
1210  NEXT M
1220 GOSUB 200
1230 FT = 1:FS = 15:FP$ = "AVERAGE=":FJ = 1:CP = 35: GOSUB 200
1240 FT = 0:FJ = 0:DR = 0:CP = 50:NI = AVG / CNT: GOSUB 200
1250 FS = 0: GOSUB 200
1260 IF OS < > 3 THEN PRINT D$;"PR #";SN
1270 END
2500 DATA 1ST POINT,2ND POINT,3RD POINT,4TH POINT
3500 DATA 150.3427,860.2176,1021.3347,15.32
3510 DATA 106.72,75.712,89.098,569.0119
```

Fig. 31-1–cont. 80 column card active. 80 column formatter program.

PROGRAM TO CONTROL OUTPUT TO THE PRINTER

When the program (Fig. 31-1) is RUN it defaults through lines 1 to 130 to initialize the variables. Line 150 causes the program to GOTO 1000.

MAIN PROGRAM LINES 1000-1050

At Line 1000, LL = 75, the length of the line is initialized to 75 columns. The length of the line on the printer will be 75 columns long. SN is then set to zero to indicate that the 80 column card is inactive. If the length of the line is greater than forty (LL > 40 THEN SN = 3), then SN is set to three to have the routine return control to the 80 column card.

```
1010 D$ = CHR$ (4) : IF SN = 3 THEN 1020
```

Line 1010 initializes D$ the key to DOS, turns the printer on, and checks to see if output is being generated to the CRT in 80 column mode(OS = 3). If this is TRUE, then the program goes to line 1020.

```
1015 PRINT D$;"PR#";OS: PRINT CHR$ (9);LL;"N"
```

Line 1015 is performed when the printer is to be the output device. This tells DOS to set the output direction to slot number OS and turns off the output to the CRT.

```
1020 FOR J = 1 TO LL-1: PRINT "*";: NEXT J: PRINT
```

Line 1020 prints out a test heading of asterisks to show the range of the printline.

```
1030 CP = 0:FT = 1:FP$ = "TEST HEADER"
```

Line 1030 sets the column position to zero, sets the field type to one (1), which is a string value, and stores the literal, "TEST HEADER" in FP$. The field is to be center justified (FJ = 2).

Line 1040 initializes the field size to seventy five (75), and center justifies the field. The program jumps to the subroutine that begins at line 200. The column position should run from zero to one less than the line length (74 in this program), and the column position, plus field size should be less than, or equal to, the line length (CP + FS < = LL).

FUNCTION OF FORMATTING SUBROUTINE AT 200

The subroutine at line 200 examines parameters that are set in the main program and either prints out the print string (P$) or builds a value in a field onto the tail end of the print string. These parameters include what type of field justification, field size, field type, and decimal rounding if the field is numeric.

```
200 IF FS > 0 THEN 230
```

If the field size is greater than zero, it means something is to be added to P$, either a numeric value, or a string value.

If the field size is less than or equal to zero, P$ is printed, and P$ is then initialized to a null value. IF FS = 0 the program returns to line 1060.

```
(210) IF FS = 0 THEN RETURN
```

If FS is less than zero, the third statement in line 210 is FALSE, and the program defaults to line 220.

```
220 FOR J = 1 TO ABS (FS) : PRINT: NEXT J: RETURN
```

Line 220 causes a number of blank lines to be printed. The number of blank lines is determined by the absolute value of the field size. For example, if FS = − 3, P$ is printed and three (blank) lines are skipped. If FS = 0, PRINT P$, and don't leave any blank lines. If FS is greater than zero, add a numeric or literal value to P$, and don't print anything.

```
230 IF LEN (P$) < CP THEN P$ = P$ + LEFT$ (SP$,CP-LEN (P$) )
```

To get to line 230, FS > 0, which means a numeric value, or literal, is to be added to P$. If the length of P$ is greater than the column position on the printer, then add as many spaces as necessary to get to the correct column position. The column position (CP) changes with every field. If additional spaces do not need to be inserted before a field is printed, that is, the boundaries of adjacent fields touch, then column position can remain unchanged.

```
240 IF FT THEN 350
```

Line 240 separates the field type into numeric and string. FT = 0 is a numeric input, and FT = 1 is a string value.

If FT = 0, a numeric value is to be added to P$, and the program defaults to line 250.

```
250 IF ABS (NI) > = 1E+10 OR ABS (NI) < = 1E − 10 THEN FP$ = STR$ (NI) : GOTO
    350
```

If the absolute value of the numeric input value is greater than or equal to 1E + 10 or less than or equal to 1E − 10, then the value is output in scientific notation. If the value is to be output in scientific notation, the program converts NI to a string representation and jumps to line 350.

```
260 NI = INT (NI * 10 ∧ DR + .5 / 10 ∧ DR
```

Line 260 causes the numeric input value to be rounded.

```
270 FP$ = STR$ (NI)
```

The *numeric* input is converted to a string value to be able to examine it for the decimal point and the correct number of decimal places.

Lines 280 through 300 are a loop used to search the string value to discover if a decimal point needs to be added. If there is no decimal point, the second statement in line 300 (FP$ = FP$ + ".") adds a decimal point.

```
310 IF DR < THEN 350
```

If the decimal rounding is less than one, then the value is rounded to the one's place or above (10's, 100's etc.) and no trailing zeros need to be added.

```
320 IF LEN (FP$) = J + DR THEN 350
```

If line 320 is TRUE, the numeric input has the correct number of decimal places, and no trailing zeros are added. If line 320 is false, then trailing zeros must be added.

```
330 J = DR + J - LEN (FP$)
```

Line 330 calculates the number of trailing zeros to be added. For example, FP$ = "173.4," and decimal rounding is three (DR = 3), then two trailing zeros must be added. J in this case is 4 (decimal point is the 4th character in FP$), and LEN (FP$) is five. (The value of J was determined in lines 280-300).

```
J = 3(DR) + 4 (J) - 5(LEN(FP$) ) = 2
```

The loop at line 340 adds the trailing zeros to FP$.

```
350 LP = LEN (FP$)
```

Line 350 is the point where the numeric value and the string value join to follow one logical path.

```
360 IF LP > = FS THEN 420
```

If the length of the literal (FP$) is greater than or equal to the field size, no field justification is necessary, and the field justification routine is bypassed. Line 420 concatenates the output value onto P$ and returns.

If line 360 is FALSE, the program defaults to the field justification routine at line 370. If the field is to be left justified (FJ = 0), the program defaults to line 380. If the field is to be right justified (FJ = 1), the program branches to line 390. If the field is to be center justified (FJ = 2), the program branches to line 400. Lines 380, 390, and 400 add blanks to the field for the correct justification code, and concatenate that value onto P$.

```
380 FP$ = FP$ + LEFT$ (SP$,FS-LP) : GOTO 420
```

If the field is to be left justified, spaces are added to the end of FP$ to fill the field size. The second statement in line 380 branches to 420 to concatenate FP$ onto the end of P$.

```
390 FP$ = LEFT$ (SP$,FS-LP) + FP$: GOTO 420
```

Line 390 right justifies the output value by adding spaces to the beginning of FP$, then branches to 420

```
400 L = INT ( (FS-LP)/2):J = FS - LP - L: IF L > 0 THEN FP$ = LEFT$ (SP$,L) + FP$
```

If the field is to be center justified, the unused portion of the field size (FS-LP) is divided in half. The correct number of spaces to be added to the end of FP$ is then calculated. For example, if FS is 15 and LP is 8, then L = INT (15-8)/2 = INT (7/2) = 3. Therefore, three spaces will be added to the beginning of FP$ (L = 3). J will be the number of spaces left after three spaces are added to the eight characters in FP$. J = 15 - 8 - 3 = 4. L on the left side = 3, and J on the right side = 4. If L is greater than zero, then add that many spaces to the beginning of FP$. If J is greater than zero (line 410), then add the spaces to the end of FP$. The length of FP$ is equal to the field size (FS), for all three justification cases.

```
420 P$ = P$ + FP$: IF LEN (P$) > LL THEN PRINT LEFT$ (P$,LL): P$=RIGHT$
    (P$,LEN (P$)-LL)
```

Line 420 concatenates the literal (FP$) onto the print line stored in P$. If the length of P$ goes over the number of characters in the line length, the correct number of characters will be printed, and P$ will be replaced with unprinted characters, and the subroutine returns. The first time the subroutine at line 200 is called, it returns to line 1060 in the main program.

MAIN PROGRAM LINES 1060-3510

```
1060 FS = -2: GOSUB 200
```

When the field size is set to a negative value, it indicates lines are to be skipped (line 220). The number of lines to be skipped causes the subroutine at 200 to output into P$ and sent this line to the printer. In this case, two blank lines are skipped.

```
1070 FT = 1:FS = 15:FS = 1
1100 FT = 0:FJ = 1: DR = 3
1230 FT = 1:FS = 15:FP$ = "AVERAGE=":FJ=1:CP = 35: GOSUB 200
1240 FT = 0:FJ = 0: DR = 0:CP = 50:NI = AVG / CNT: GOSUB 200
```

Lines 1070,1100,1230, and 1240 are the lines where the information is input to determine how the output is formatted. Field type, field size, field justification, decimal rounding, and column position are set in the lines.

```
1080 FOR M = 1 TO 4: READ FP$:CP = 15 * (M - 1 )
```

Four heading literals are read from data statements, and the third statement in line 1080 calculates the heading position at 0, 15, 30, and 45 to place the literals.

```
1090 GOSUB 200: NEXT M:FS = - 1: GOSUB 200
```

Line 1090 puts the literals read in from line 1080 on P$, and closes the loop. FS = - 1 causes P$ to output one blank line by calling the subroutine at line 200.

```
1100 FT = 0:FJ = 1:DR = 3
```

Line 1100 controls the formatting of the field type. FT = 0 is a numeric field, while FT = 1 is a string field. The field justification is of three types, FJ = 0 is left justified, FJ = 1 is right justified, and FJ = 2 is center justified. Decimal rounding is set to three places. See discussion of subroutine at 200, lines 250 through 340.

```
1110 AVG =0:CNT =0
```

Line 1110 initializes the value of average and the counter to zero. The values input are to be counted, summed, and averaged.

Lines 1120 through 1210 contain doubly nested M and N loops to read the values in the data statements, sum the values, count the number of values, and assign the values in the data statements to the numeric input (NI). These values are placed into P$ and output to the printer by calling the subroutine at line 200.

```
1120 M LOOP — NUMBER OF LINES INPUT
1130 FIELD SIZE SET TO 15 CHARACTERS
1140 READ IN FOUR NUMBERS FROM DATA STATEMENTS
1150 READ STATEMENT
1160 SUMS AND COUNTS THE NUMBERS READ IN
1170  ASSIGNS THE NUMBER READ IN TO THE NUMERIC INPUT (NI),
     AND COMPUTES COLUMN POSITION
1180 ADDS NUMERIC INPUT TO P$
1190 NEXT N
1200  SETS FIELD SIZE TO ZERO, AND PRINTS A LINE OF FOUR NUM-
     BERS JUST CONSTRUCTED
1210 NEXT M
1220 GOSUB 200
```

Line 1220 prints out a blank line since FS = 0, consequently P$ is a null value (line 210).

```
1230 FT = 1:FS = 15:FP$ = "AVERAGE=":FJ = 1:CP =35: GOSUB 200
```

Line 1230 sets the field type to a string value (FT = 1), sets the field size to 15 (FS = 15). It assigns the literal to FP$ (FP$ = "AVERAGE="), sets the field to be right justified (FJ = 1), sets the column position to 35, and calls the subroutine at line 200 to put FP$ onto P$.

```
1240 FT = 0:FJ = 0:DR = 0:CP = 50:NI = AVG / CNT: GOSUB 200
```

The field type is set to numeric (FT = 0), the field is left justified (FJ = 0), the decimal round is set to the one's place (DR = 0), the column position is

set to fifty. Line 1240 also assigns the average to the numeric input, and calls the subroutine at 200 to put the numeric input value onto P$.

Lines 1230 and 1240 combine to print "AVERAGE = 361.", by right justifying the literal whose field ends in position 49 and left justifying the numeric value whose field starts in column 50. By setting the borders of these two fields adjacent to one another and using the proper justification, the two fields are made to appear as one.

```
1250 FS = 0: GOSUB 200
```

Line 1250 sets the field size to zero, and calls the subroutine at line 200 to print out P$ (line 420).

```
1260 IF OS < > 3 THEN PRINT D$;"PR#";SN
```

Line 1260 turns the printer off if the 80 column card is not active and the program ends.

```
********************************************************************************
                                TEST HEADER
  1ST POINT       2ND POINT       3RD POINT       4TH POINT
    150.343         860.218        1021.335         15.320
    106.720          75.712          89.098        569.012
                                  AVERAGE=361.
```

Fig. 31-2. 80 column card active. Run of the 80 column formatter program.

```
********************************************************************************
                                TEST HEADER
  1ST POINT       2ND POINT       3RD POINT       4TH POINT
    150.343         860.218        1021.335         15.320
    106.720          75.712          89.098        569.012
                                  AVERAGE=361.0
```

Fig. 31-3. 80 column card active. Run of the 80 column formatter program after modification of lines 1070, 1100, 1240.

```
********************************************************************************
                                TEST HEADER
  1ST POINT       2ND POINT       3RD POINT       4TH POINT
    150.343         860.218        1021.335         15.320
    106.720          75.712          89.098        569.012
                                  AVERAGE=361.
```

Fig. 31-4. 80 column card active. Run of the 80 column formatter program after modification of line 1070.

Fig. 31-2 was created by the RUN of the program in Fig. 31-1.

Fig. 31-3 was created by a modification of lines 1070, 1100, and 1240 in the program in Fig. 31-1.

```
1070 FT =1:FS =15: FJ =2
1100 FT =0:FJ =1: DR =2
1240 FT =0:FJ =0: DR = 0:CP =50:NI =AVG / CNT: GOSUB 200
```

Fig. 31-4 was created by a modification of line 1070 in the program in Fig. 31-1.

```
1070 FT =1:FS =12:FJ = 0
```

Index

SAMS APPLE® BOOKS

Many thanks for your interest in this Sams Book about Apple II® microcomputing. Here are a few more Apple-oriented Sams products we think you'll like:

POLISHING YOUR APPLE®, Vol. 1

Clearly written, highly practical, concise assembly of all procedures needed for writing, disk-filing, and printing programs with an Apple II. Positively ends your searches through endless manuals to find the routine you need! By Herbert M. Honig. 80 pages, 5½ x 8½, comb. ISBN 0-672-22026-1. © 1982.
Ask for No. 22026 ..... **$4.95**

POLISHING YOUR APPLE®, Vol. 2

A second Apple II timesaver that guides intermediate-level programmers in setting up professional-looking menus, using effective error trapping, and making programs that run without the need for detailed explanations. Includes many sample routines. By Herbert M. Honig. 112 pages, 5½ x 8½, soft. ISBN 0-672-22160-8. © 1983.
Ask for No. 22160 ..... **$4.95**

APPLESOFT LANGUAGE (2nd Edition)

Quickly introduces you to Applesoft syntax and programming, including advanced techniques, graphics, color commands, sorts, searches, and more! New material covers disk operations, numbers, and number programming. Many usable routines and programs included. By Brian D. Blackwood and George H. Blackwood. 274 pages, 6 x 9, comb. ISBN 0-672-22073-3. © 1983.
Ask for No. 22073 ..... **$13.95**

APPLE® II APPLICATIONS

Presents a series of board-level interfacing applications you can modify if necessary to help you use an Apple II as a development system, a data-acquisition or control device, or for making measurements. Includes programs. By Marvin L. De Jong. 256 pages, 5½ x 8½, soft. ISBN 0-672-22035-0. © 1983.
Ask for No. 22035 ..... **$13.95**

DISKS, FILES, AND PRINTERS FOR THE APPLE® II

Provides you with basic-to-advanced details for using disks, files, and printers with an Apple II, including outstanding explanations for programming with sequential-access, random-access, and executive files. By Brian D. Blackwood and George H. Blackwood. 216 pages, 6 x 9, comb. ISBN 0-672-22163-2. © 1983.
Ask for No. 22163 ..... **$15.95**

THE APPLE® II CIRCUIT DESCRIPTION

Provides you with a detailed circuit description for all revisions of the Apple II and Apple II+ motherboard, including the keyboard and power supply. Highly valuable data that includes timing diagrams for major signals, and more. By Winston D. Gayler. 176 pages plus foldouts, 8½ x 11, comb. ISBN 0-672-21959-X. © 1983.
Ask for No. 21959 ..... **$22.95**

INTERMEDIATE LEVEL APPLE® II HANDBOOK

Provides you with a nicely paced transition from Integer BASIC into machine- and assembly-language programming with the Apple II. Covers text display, video POKEs, graphics, using machine language with BASIC, memory addresses, debugging, and more. By David L. Heiserman. 328 pages, 6 x 9, comb. ISBN 0-672-21889-5. © 1983.
Ask for No. 21889 ..... **$16.95**

APPLE® FORTRAN

Only fully detailed Apple FORTRAN manual on the market! Ideal for Apple programmers of all skill levels who want to try FORTRAN in a business or scientific program. Many ready-to-run programs provided. By Brian D. Blackwood and George H. Blackwood. 240 pages, 6 x 9, comb. ISBN 0-672-21911-5. © 1982.
Ask for No. 21911 ..... **$14.95**

APPLE® II ASSEMBLY LANGUAGE

Shows you how to use the 3-character, 56-word vocabulary of Apple's 6502 to create powerful, fast-acting programs! For beginners or those with little or no assembly language programming experience. By Marvin L. De Jong. 336 pages, 5½ x 8½, soft. ISBN 0-672-21894-1. © 1982.
Ask for No. 21894 ..... **$15.95**

ENHANCING YOUR APPLE® II — Vol. 1

Shows you how to mix text, LORES, and HIRES anywhere on the screen, how to open up whole new worlds of 3-D graphics and special effects with a one-wire modification, and more. Tested goodies from a trusted Sams author! By Don Lancaster. 232 pages, 8½ x 11, soft. ISBN 0-672-21846-1. © 1982.
Ask for No. 21846 ..... **$17.95**

CIRCUIT DESIGN PROGRAMS FOR THE APPLE® II

Programs quickly display "what happens if" and "what's needed when" as they apply to periodic waveform, rms and average values, design of matching pads, attenuators, and heat sinks, solution of simultaneous equations, and more. By Howard M. Berlin. 136 pages, 8½ x 11, comb. ISBN 0-672-21863-1. © 1982.
Ask for No. 21863 ..... **$15.95**

APPLE® INTERFACING

Brings you real, tested interfacing circuits that work, plus the necessary BASIC software to connect your Apple to the outside world. Lets you control other devices and communicate with other computers, modems, serial printers, and more! By Jonathan A. Titus, David G. Larsen, and Christopher A. Titus. 208 pages, 5½ x 8½, soft. ISBN 0-672-21862-3. © 1981.
Ask for No. 21862 ..... **$11.95**

INTIMATE INSTRUCTIONS IN INTEGER BASIC

Explains flowcharting, loops, functions, graphics, variables, and more as they relate to Integer BASIC. Used with *Applesoft Language* (No. 22073), it gives you everything you need to program BASIC with your Apple II or Apple II Plus. By Brian D. Blackwood and George H. Blackwood. 160 pages, 5½ x 8½, soft. ISBN 0-672-21812-7. © 1981.
Ask for No. 21812 ..... **$8.95**

MOSTLY BASIC: APPLICATIONS FOR YOUR APPLE® II, BOOK 1

Twenty-eight debugged, fun-and-serious BASIC programs you can use immediately on your Apple II. Includes a telephone dialer, digital stopwatch, utilities, games, and more. By Howard Berenbon. 160 pages, 8½ x 11, comb. ISBN 0-672-21789-9. © 1980.
Ask for No. 21789 ..... **$13.95**

MOSTLY BASIC: APPLICATIONS FOR YOUR APPLE® II, BOOK 2

A second gold mine of fascinating BASIC programs for your Apple II, featuring 3 dungeons, 11 household programs, 6 on money or investment, 2 to test your ESP level, and more — 32 in all! By Howard Berenbon. 224 pages, 8½ x 11, comb. ISBN 0-672-21864-X. © 1981.
Ask for No. 21864 ..... **$12.95**

SAMS SOFTWARE FOR THE APPLE®
FINANCIAL PLANNING WITH VISICALC® AND THE APPLE® II

Automatically sets up your VisiCalc spreadsheet to perform 16 different calculations commonly needed in business and financial planning, and lets you compare as many as four possibilities. Works with 80-column board if you have one. You'll need VisiCalc, 64K RAM, and one disk drive. ISBN 0-672-29059-6.
Ask for No. 29059 ..... **$79.95**

FINANCIAL PLANNING WITH MULTIPLAN™ AND THE APPLE® II

Same as *Financial Planning with VisiCalc*, except works with Multiplan spreadsheet, 64K RAM, and one disk drive. ISBN 0-672-29058-8.
Ask for No. 29058 ..... **$79.95**

MONEY TOOL

Helps you manage income, expenses, and tax information for home or small business. Can reconcile checking, provide simple reports, and more. By Herb Honig. Takes 48K RAM, Applesoft in ROM, and one disk drive. ISBN 0-672-26113-8.
Ask for No. 26113 ..... **$59.95**

FINANCIAL FACTS

Instantly computes the majority of data you'll commonly need in personal and small-business financial management, and prints out the major factors. By Ed Hanson. Takes 48K RAM, Applesoft in ROM, and one disk drive. ISBN 0-672-26099-9.
Ask for No. 26099 ..... **$59.95**

INSTANT RECALL

Friendly, unconventional, and instantaneous data handler. Each free-form, alphanumeric screenful you enter is an 840-character page you can edit, file, or print out as it appears. By Charles R. Landers. Takes 48K RAM, Applesoft in ROM, and one disk drive. ISBN 0-672-26097-2.
Ask for No. 26097 ..... **$59.95**

PEN-PAL

Sophisticated, powerful, affordable word processor. Provides block movement, line deletion, character and text insertion, and more. Takes 48K RAM, Applesoft in ROM, and one disk drive. ISBN 0-672-26115-4.
Ask for No. 26115 ..... **$59.95**

HELLO CENTRAL !

Versatile, menu-controlled terminal program you can use with any compatible modem to communicate with networks and other computers. Has built-in text editor, auto dialing, much more. By Bruce Kallick. Takes 48K RAM, Applesoft in ROM, one disk drive, and modem. ISBN 0-672-26081-6.
Ask for No. 26081 ..... **$99.95**

You can usually find these Sams products at better computer stores, bookstores, and electronic distributors nationwide.

If you can't find what you need, call Sams at 800-428-3696 toll-free or 317-298-5566, and charge it to your MasterCard or Visa account. Prices subject to change without notice.

For a free catalog of all Sams Books available, write P.O. Box 7092, Indianapolis, IN 46206.

SAMS

Applesoft® For The IIe®

- Written for the Apple IIe computer using Applesoft language.
- Instructions are presented in a lesson-type format using terms easily understood.
- Rules for programming are given in a logical and progressive method.
- Programming techniques expand from a simple level to advanced.
- Details how to write and run programs in an 80 column mode.
- Presents an 80 column formatter program.

Howard W. Sams & Co., Inc.
4300 West 62nd Street, Indianapolis, Indiana 46268 U.S.A.

$19.95/22259

ISBN: 0-672-22259-0