CHAPTER
1

Another Pleasant Valley Saturday

Understanding What Computers Really Do

It's All in the Plan

''Quick, Mike, get your sister and brother up, it's past 7. Nicky's got Little League at 9:00 and Dione's got ballet at 10:00. Give Max his heartworm pill! (We're out of them, Ma, remember?) Your father picked a great weekend to go fishing. Here, let me give you 10 bucks and go get more pills at the vet's. My God, that's right, Hank needed gas money and left me broke. There's an ATM over by Kmart, and if I go there I can take that stupid toilet seat back and get the right one.''

''I guess I'd better make a list. . . .''

It's another Pleasant Valley Saturday, and thirty-odd million suburban homemakers sit down with a pencil and pad at the kitchen table to try to make sense of a morning that would kill and pickle any lesser being. In her mind, she thinks of the dependencies and traces the route:

Drop Nicky at Rand Park, go back to Dempster and it's about 10 minutes to Golf Mill Mall. Do I have gas? I'd better check first—if not, stop at Del's Shell or I won't make it to Milwaukee Avenue. Milk the ATM at Golf Mill, then cross the parking lot to Kmart to return the toilet seat that Hank bought last weekend without checking what shape it was. Gotta remember to throw the toilet seat in the back of the van—write that at the top of the list.

By then it'll be half past, maybe later. Ballet is all the way down Greenwood in Park Ridge. No left turn from Milwaukee—but there's the sneak path around behind the mall. I have to remember not to turn right onto Milwaukee like I always do—jot that down. While I'm in Park Ridge I can check to see if Hank's new glasses are in—should call but they won't even be open until 9:30. Oh, and groceries—can do that while Dione dances. On the way back I can cut over to Oakton and get the dog's pills.

In about 90 seconds flat the list is complete:

- Throw toilet seat in van.
- Check gas—if empty, stop at Del's Shell.
- Drop Nicky at Rand Park.
- Stop at Golf Mill teller machine.
- Return toilet seat at Kmart.
- Drop Dione at ballet (remember the sneak path to Greenwood).
- See if Hank's glasses are at Pearle Vision—if they are, make sure they remembered the extra scratch coating.
- Get groceries at Jewel.
- Pick up Dione.
- Stop at vet's for heartworm pills.
- Drop off groceries at home.
- If it's time, pick up Nicky. If not, collapse for a few minutes, then pick up Nicky.
- Collapse!

In what we often call a ''laundry list'' (whether it involves laundry or not) is the perfect metaphor for a computer program. Without realizing it, our intrepid homemaker has written herself a computer program and then set out (acting as the computer) to execute it and be done before noon.

Computer programming is nothing more than this: you, the programmer, write a list of steps and tests. The computer then performs each step and test in sequence. When the list of steps has been executed, the computer stops.

A computer program is a list of steps and tests, nothing more.

Steps and Tests

Think for a moment about what I call a ''test'' in the preceding laundry list. A *test* is the sort of either/or decision we make dozens or hundreds of times on even the most placid of days, sometimes nearly without thinking about it.

Our homemaker performed a test when she jumped into the van to get started on her adventure. She looked at the gas gauge. The gas gauge would tell her one of two things: either she has enough gas or she doesn't. If she has enough gas, then she takes a right and heads for Rand Park. If she doesn't have enough gas, then she takes a left down to the corner and fills the tank at Del's Shell. Then, with a full tank, she continues the program by taking a U-turn and heading for Rand Park.

In the abstract, a test consists of those two parts:

- First, you take a look at something that can go one of two ways.
- Then you do one of two things, depending on what you saw when you took a look.

Toward the end of the program, our homemaker gets home, takes the groceries out of the van, and checks the clock. If it isn't time to get Nicky from Little League, then she has a moment to collapse on the couch in a nearly empty house. If it *is* time to get Nicky, then there's no rest for the ragged: she sprints for the van and heads back to Rand Park.

(Any guesses as to whether she really gets to collapse when the program finishes running?)

More Than Two Ways?

You might object, saying that many or most tests involve more than two alternatives. Ha-ha, sorry, you're dead wrong—*in every case*. Furthermore, you're wrong whether you think you are or not. Read this twice: *Except for totally impulsive or psychotic behavior, every human decision comes down to the choice between two alternatives.*

What you have to do is look a little more closely at what goes through your mind when you make decisions. The next time you buzz down to Yow Chow Now for fast Chinese, observe yourself while you're poring over the menu. The choice might seem, at first, to be of one item out of 26 Cantonese main courses. Not so. The choice, in fact, is between choosing one item and *not* choosing that one item. Your eyes rest on chicken with cashews. Naw, too bland. *That was a test*. You slide down to the next item. Chicken with black mushrooms. Hmm, no, had that last week. *That was another test*. Next item: Kung Pao chicken. Yeah, that's it! *That was a third test*.

The choice was not among chicken with cashews, chicken with black mushrooms, or Kung Pao chicken. Each dish had its moment, poised before the critical eye of your mind, and you turned thumbs up or thumbs down on it, individually. Eventually, one dish won, but it won in that same game of ''to eat or not to eat.''

Let me give you another example. Many of life's most complicated decisions come about due to the fact that 99.99867 percent of us are not nudists. You've been there: you're standing in the clothes closet in your underwear, flipping through your rack of pants. The tests come thick and fast. This one? No. This one? No. This one? No. This one? Yeah. You pick a pair of blue pants, say. (It's a Monday, after all, and blue would seem an appropriate color.) Then you stumble over to your sock drawer and take a look. Whoops, no blue socks. *That was a test*. So you stumble back to the clothes closet, hang your blue pants back on the pants rack, and start over. This one? No. This one? No. This one? Yeah. This time it's brown pants, and you toss them over your arm and head back to the sock drawer to take another look. Nertz, out of brown socks, too. So it's back to the clothes closet . . .

What you might consider a single decision, or perhaps two decisions inextricably tangled (such as picking pants and socks of the same color, given stock on hand), is actually a series of small decisions, always binary in nature: pick 'em or don't pick 'em. Find 'em or don't find 'em. The Monday morning episode in the clothes closet is a good analogy of a programming structure called a *loop*: you keep doing a series of things until you get it right, and then you stop (assuming you're not the kind of geek who wears blue socks with brown pants); but whether you get everything right always comes down to a sequence of simple either/or decisions.

Computers Think Like Us

I can almost hear the objection: ''Sure, it's a computer book, and he's trying to get me to think like a computer.'' Not at all. Computers think like *us*. We designed them; how else could they think? No, what I'm trying to do is get you to take a long, hard look at how *you* think. We run on automatic for so much of our lives that we literally do most of our thinking without really thinking about it.

The very best model for the logic of a computer program is the very same logic we use to plan and manage our daily affairs. No matter what we do, it comes down to a matter of confronting two alternatives and picking one. What we might think of as a single large and complicated decision is nothing more than a messy tangle of many smaller decisions. The skill of looking at a complex decision and seeing all the little decisions in its tummy will serve you well in learning how to program. Observe yourself the next time you have to decide something. Count up the little decisions that make up the big one. You'll be surprised.

And, surprise! You'll be a programmer.

Had This Been the Real Thing . . .

Do not be alarmed. What you have just experienced was a metaphor. It was not the real thing. (The real thing comes later.) I use metaphors a lot in this book. A metaphor is a loose comparison drawn between something familiar (such as a Saturday morning laundry list) and something unfamiliar (such as a computer program). The idea is to anchor the unfamiliar in the terms of the familiar, so that when I begin tossing facts at you, you'll have someplace comfortable to lay them down.

The most important thing for you to do right now is keep an open mind. If you know a little bit about computers or programming, don't pick nits. Yes, there are important differences between a homemaker following a scribbled laundry list and a computer executing a program. I'll mention those differences all in good time.

For now, it's still Chapter 1. Take these initial metaphors on their own terms. Later on, they'll help a lot.

Do Not Pass Go

''There's a reason *bored* and *board* are homonyms,'' said my best friend, Art, one evening as we sat (two super-sophisticated twelve-year-olds) playing some game in his basement. (He may have been unhappy because he was losing.) Was it Mille Bornes? Or Stratego? Or Monopoly? Or something else entirely? I confess, I don't remember. I simply recall hopping some little piece of plastic shaped like a pregnant bowling pin up and down a series of colored squares that told me to do dumb things like go back two spaces or put $100 in the pot or nuke Outer Mongolia.

There are strong parallels to be drawn between that peculiar American pastime, the board game, and assembly-language programming. First of all, everything I said before still holds: board games, by and large, consist of a progression of steps and tests. In some games, such as Trivial Pursuit, *every* step on the board is a test: to see if you can answer, or not answer, a question on a card. In other board games, each little square along the path on the board contains some sort of instruction: Lose One Turn; Go Back Two Squares; Take a Card from Community Chest; and, of course, Go to Jail. Things happen in board games, and the path your little pregnant bowling pin takes as it works its way along the edge of the board will change along the way.

Many board games also have little storage locations marked on the board where you keep things: cards and play money and game tokens such as little plastic houses or hotels, or perhaps bombers and nuclear missiles. As the game progresses, you buy, sell, or launch your assets, and the contents of your storage locations change. Computer programs are like that too: there are places where you store things (''things'' here being pure data, rather than physical tokens); and as the computer program executes, the data stored in those places will change.

Computer programs are not games, of course—at least, not in the sense that a board game is a game. Most of the time, a given program is running all by itself. There is only one ''player'' and not two or more. (This is not *always* true, but I don't want to get too far ahead right now. Remember, we're still in metaphor territory.) Still, the metaphor is useful enough that it's worth pursuing.

The Game of Big Bux

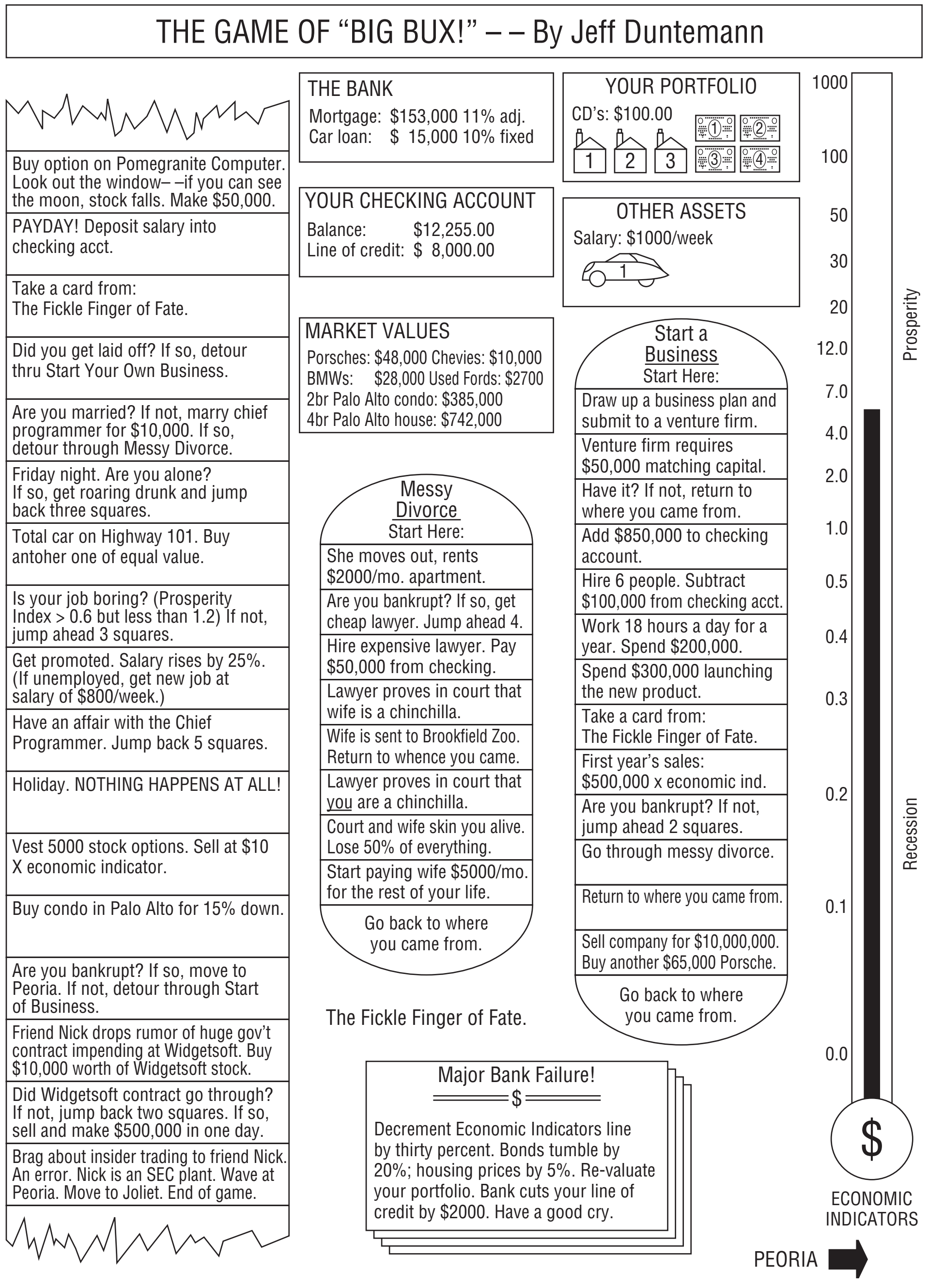
I've invented my own board game to continue down the road with this particular metaphor. In the sense that art mirrors life, the Game of Big Bux mirrors life in Silicon Valley, where money seems to be spontaneously created (generally in somebody else's pocket) and the three big Money Black Holes are fast cars, California real estate, and messy divorces. There is luck, there is work, and assets often change hands *very* quickly.

A portion of the Big Bux game board is shown in Figure 1-1. The line of rectangles on the left side of the page continues all the way around the board. In the middle of the board are cubbyholes to store your play money and game pieces; stacks of cards to be read occasionally; and short detours with names such as Messy Divorce and Start a Business, which are brief sequences of the same sort of action squares as those forming the path around the edge of the board. These are ''side paths'' that players take when instructed, either by a square on the board or a card pulled during the game. If you land on a square that tells you to Start a Business, you go through that detour. If you jump over the square, you don't take the detour, and just keep on trucking around the board.

Unlike many board games, you don't throw dice to determine how many steps around the board you take. Big Bux requires that you move *one* step forward on each turn, *unless* the square you land on instructs you to move forward or backward or go somewhere else, such as through a detour. This makes for a considerably less random game. In fact, Big Bux is a pretty linear experience, meaning that for the most part you go around the board until you're told that the game is over. At that point, you may be bankrupt; if not, you can total up your assets to see how well you've done.

There is some math involved. You start out with a condo, a cheap car, and $250,000 in cash. You can buy CDs at a given interest rate, payable each time you make it once around the board. You can invest in stocks and other securities whose value is determined by a changeable index in economic indicators, which fluctuates based on cards chosen from the stack called the Fickle Finger of Fate. You can sell cars on a secondary market, buy and sell houses, condos, and land; and wheel and deal with the other players. Each time you make it once around the board, you have to recalculate your net worth. All of this involves some addition, subtraction, multiplication, and division, but there's no math more complex than compound interest. Most of Big Bux involves nothing more than taking a step and following the instructions at each step.

Is this starting to sound familiar?



THE GAME OF “BIG BUX!” – – By Jeff Duntemann

Buy option on Pomegranite Computer. Look out the window– –if you can see the moon, stock falls. Make $50,000.

PAYDAY! Deposit salary into checking acct.

Take a card from: The Fickle Finger of Fate.

Did you get laid off? If so, detour thru Start Your Own Business.

Are you married? If not, marry chief programmer for $10,000. If so, detour through Messy Divorce.

Friday night. Are you alone? If so, get roaring drunk and jump back three squares.

Total car on Highway 101. Buy antoher one of equal value.

Is your job boring? (Prosperity Index > 0.6 but less than 1.2) If not, jump ahead 3 squares.

Get promoted. Salary rises by 25%. (If unemployed, get new job at salary of $800/week.)

Have an affair with the Chief Programmer. Jump back 5 squares.

Holiday. NOTHING HAPPENS AT ALL!

Vest 5000 stock options. Sell at $10 X economic indicator.

Buy condo in Palo Alto for 15% down.

Are you bankrupt? If so, move to Peoria. If not, detour through Start of Business.

Friend Nick drops rumor of huge gov’t contract impending at Widgetsoft. Buy $10,000 worth of Widgetsoft stock.

Did Widgetsoft contract go through? If not, jump back two squares. If so, sell and make $500,000 in one day.

Brag about insider trading to friend Nick. An error. Nick is an SEC plant. Wave at Peoria. Move to Joliet. End of game.

THE BANK
Mortgage: $153,000 11% adj.
Car loan: $ 15,000 10% fixed

YOUR CHECKING ACCOUNT
Balance: $12,255.00
Line of credit: $ 8,000.00

MARKET VALUES
Porsches: $48,000 Chevies: $10,000
BMWs: $28,000 Used Fords: $2700
2br Palo Alto condo: $385,000
4br Palo Alto house: $742,000

YOUR PORTFOLIO
CD’s: $100.00
1 2 3
1 2 3 4

OTHER ASSETS
Salary: $1000/week
1

Messy Divorce
Start Here:
She moves out, rents $2000/mo. apartment.
Are you bankrupt? If so, get cheap lawyer. Jump ahead 4.
Hire expensive lawyer. Pay $50,000 from checking.
Lawyer proves in court that wife is a chinchilla.
Wife is sent to Brookfield Zoo. Return to whence you came.
Lawyer proves in court that you are a chinchilla.
Court and wife skin you alive. Lose 50% of everything.
Start paying wife $5000/mo. for the rest of your life.
Go back to where you came from.

The Fickle Finger of Fate.

Start a Business
Start Here:
Draw up a business plan and submit to a venture firm.
Venture firm requires $50,000 matching capital.
Have it? If not, return to where you came from.
Add $850,000 to checking account.
Hire 6 people. Subtract $100,000 from checking acct.
Work 18 hours a day for a year. Spend $200,000.
Spend $300,000 launching the new product.
Take a card from: The Fickle Finger of Fate.
First year’s sales: $500,000 x economic ind.
Are you bankrupt? If not, jump ahead 2 squares.
Go through messy divorce.
Return to where you came from.
Sell company for $10,000,000. Buy another $65,000 Porsche.
Go back to where you came from.

Major Bank Failure!
$
Decrement Economic Indicators line by thirty percent. Bonds tumble by 20%; housing prices by 5%. Re-valuate your portfolio. Bank cuts your line of credit by $2000. Have a good cry.

1000
100
50
30
20
12.0
7.0
4.0
2.0
1.0
0.5
0.4
0.3
0.2
0.1
0.0
Prosperity
Recession
$
ECONOMIC INDICATORS

PEORIA

Figure 1-1: The Big Bux game board

Playing Big Bux

At one corner of the Big Bux board is the legend Move In, as that's how people start life in California—no one is actually *born* there. That's the entry point at which you begin the game. Once moved in, you begin working your way around the board, square by square, following the instructions in the squares.

Some of the squares simply tell you to do something, such as ''Buy a Condo in Palo Alto for 15% down.'' Many of the squares involve a test of some kind. For example, one square reads: ''Is your job boring? (Prosperity Index 0.3 but less than 4.0.) If not, jump ahead three squares.'' The test is actually to see if the Prosperity Index has a value between 0.3 and 4.0. Any value outside those bounds (that is, runaway prosperity or Four Horsemen–class recession) is defined as Interesting Times, and causes a jump ahead by three squares.

You always move one step forward at each turn, unless the square you land on directs you to do something else, such as jump forward three squares or jump back five squares, or take a detour.

The notion of taking a detour is an interesting one. Two detours are shown in the portion of the board I've provided. (The full game has others.) Taking a detour means leaving your main path around the edge of the game board and stepping through a series of squares somewhere else on the board. When you finish with the detour, you return to your original path right where you left it. The detours involve some specific process—for example, starting a business or getting divorced.

You can work through a detour, step by step, until you hit the bottom. At that point you simply pick up your journey around the board right where you left it. You may also find that one of the squares in the detour instructs you to go back to where you came from. Depending on the logic of the game (and your luck and finances), you may completely run through a detour or get thrown out of the detour somewhere in the middle. In either case, you return to the point from which you originally entered the detour.

Also note that you can take a detour from within a detour. If you detour through Start a Business and your business goes bankrupt, you leave Start a Business temporarily and detour through Messy Divorce. Once you leave Messy Divorce, you return to where you left Start a Business. Ultimately, you also leave Start a Business and return to wherever you were on the main path when you took the detour. The same detour (for example, Start a Business) can be taken from any of several different places along the game board.

Unlike most board games, the Game of Big Bux doesn't necessarily end. You can go round and round the board basically forever. There are three ways to end the game:

- *Retire*: To do this, you must have assets at a certain level and make the decision to retire.

- *Go bankrupt*: Once you have no assets, there's no point in continuing the game. Move to Peoria in disgrace.
- *Go to jail*: This is a consequence of an error of judgment, and is not a normal exit from the game board.

Computer programs are also like that. You can choose to end a program when you've accomplished what you planned, even though you could continue if you wanted. If the document or the spreadsheet is finished, save it and exit. Conversely, if the photo you're editing keeps looking worse and worse each time you select Sharpen, you stop the program without having accomplished anything. If you make a serious mistake, then the program may throw you out with an error message and corrupt your data in the bargain, leaving you with less than nothing to show for the experience.

Once more, *this is a metaphor*. Don't take the game board too literally. (Alas, Silicon Valley life was *way* too much like this in the go-go 1990s. It's calmer now, I've heard.)

Assembly Language Programming As a Board Game

Now that you're thinking in terms of board games, take a look at Figure 1-2. What I've drawn is actually a fair approximation of assembly language as it was used on some of our simpler computers about 25 or 30 years ago. The column marked ''Program Instructions'' is the main path around the edge of the board, of which only a portion can be shown here. This is the assembly language computer program, the actual series of steps and tests that, when executed, cause the computer to do something useful. Setting up this series of program instructions is what programming in assembly language actually *is*.

Everything else is odds and ends in the middle of the board that serve the game in progress. Most of these are storage locations that contain your data. You're probably noticing (perhaps with sagging spirits) that there are a *lot* of numbers involved. (They're weird numbers, too—what, for example, does ''004B'' mean? I deal with that issue in Chapter 2.) I'm sorry, but that's simply the way the game is played. Assembly language, at its innermost level, is nothing *but* numbers, and if you hate numbers the way most people hate anchovies, you're going to have a rough time of it. (I like anchovies, which is part of my legend. Learn to like numbers. They're not as salty.) Higher-level programming languages such as Pascal or Python disguise the numbers by treating them symbolically—but assembly language, well, it's you and the numbers.

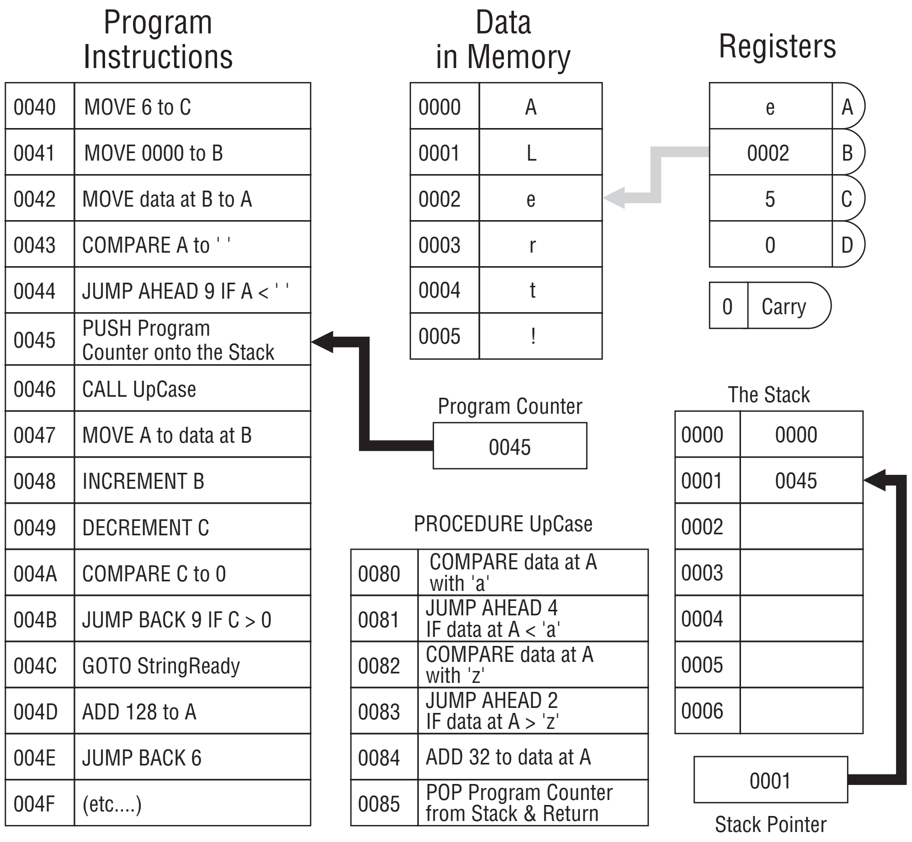


Figure 1-2: The Game of Assembly Language

I should caution you that the Game of Assembly Language represents no real computer processor like the Pentium. Also, I've made the names of instructions more clearly understandable than the names of the instructions in Intel assembly language. In the real world, instruction names are typically things like `STOSB`, `DAA`, `INC`, `SBB`, and other crypticisms that cannot be understood without considerable explanation. We're easing into this stuff sidewise, and in this chapter I have to sugarcoat certain things a little to draw the metaphors clearly.

Code and Data

Like most board games (including the Game of Big Bux), the assembly language board game consists of two broad categories of elements: game steps and places to store things. The ''game steps'' are the steps and tests I've been speaking of all along. The places to store things are just that: cubbyholes into which you can place numbers, with the confidence that those numbers will remain where you put them until you take them out or change them somehow.

In programming terms, the game steps are called *code*, and the numbers in their cubbyholes (as distinct from the cubbyholes themselves) are called *data*. The cubbyholes themselves are usually called *storage*. (The difference between the places you store information and the information you store in them is crucial. Don't confuse them.)

The Game of Big Bux works the same way. Look back to Figure 1-1 and note that in the Start a Business detour, there is an instruction reading ''Add $850,000 to checking account.'' The checking account is one of several different kinds of storage in the Game of Big Bux, and money values are a type of data. It's no different conceptually from an instruction in the Game of Assembly Language reading `ADD 5 to Register A`. An `ADD` instruction in the code alters a data value stored in a cubbyhole named `Register A`.

Code and data are two very different kinds of critters, but they interact in ways that make the game interesting. The code includes steps that place data into storage (`MOVE` instructions) and steps that alter data that is already in storage (`INCREMENT` and `DECREMENT` instructions, and `ADD` instructions). Most of the time you'll think of code as being the master of data, in that the code writes data values into storage. Data does influence code as well, however. Among the tests that the code makes are tests that examine data in storage, the `COMPARE` instructions. If a given data value exists in storage, the code may do one thing; if that value does not exist in storage, the code will do something else, as in the Big Bux JUMP BACK and JUMP AHEAD instructions.

The short block of instructions marked `PROCEDURE` is a detour off the main stream of instructions. At any point in the program you can duck out into the procedure, perform its steps and tests, and then return to the very place from which you left. This allows a sequence of steps and tests that is generally useful and used frequently to exist in only one place, rather than as a separate copy everywhere it is needed.

Addresses

Another critical concept lies in the funny numbers at the left side of the program step locations and data locations. Each number is unique, in that a location tagged with that number appears only *once* inside the computer. This location is called an *address*. Data is stored and retrieved by specifying the data's address in the machine. Procedures are called by specifying the address at which they begin.

The little box (which is also a storage location) marked `PROGRAM COUNTER` keeps the address of the next instruction to be performed. The number inside the program counter is increased by one (we say, ''incremented'' each time an instruction is performed *unless the instructions tell the program counter to do something else*. For example: notice the `JUMP BACK 9` instruction at address 004B. When this instruction is performed, the program counter will ''back up'' by nine locations. This is analogous to the ''go back three spaces'' concept in most board games.

Metaphor Check!

That's about as much explanation of the Game of Assembly Language as I'm going to offer for now. This is still Chapter 1, and we're still in metaphor territory. People who have had some exposure to computers will recognize and understand some of what Figure 1-2 is doing. (There's a real, traceable program going on in there—I dare you to figure out what it does—and how!) People with no exposure to computer innards at all shouldn't feel left behind for being utterly lost. I created the Game of Assembly Language solely to put across the following points:

- *The individual steps are very simple:* One single instruction rarely does more than move a single byte from one storage cubbyhole to another, perform very elementary arithmetic such as addition or subtraction, or compare the value contained in one storage cubbyhole to a value contained in another. This is good news, because it enables you to concentrate on the simple task accomplished by a single instruction without being overwhelmed by complexity. The bad news, however, is the following:
- *It takes a lot of steps to do anything useful:* You can often write a useful program in such languages as Pascal or BASIC in five or six lines. You can actually create useful programs in visual programming systems such as Visual Basic and Delphi *without writing any code at all*. (The code is still there ... but it is ''canned'' and all you're really doing is choosing which chunks of canned code in a collection of many such chunks will run.) A useful assembly language program cannot be implemented in fewer than about 50 lines, and anything challenging takes hundreds or thousands—or tens of thousands—of lines. The skill of assembly language programming lies in structuring these hundreds or thousands of instructions so that the program can still be read and understood.
- *The key to assembly language is understanding memory addresses:* In such languages as Pascal and BASIC, the compiler takes care of where something is located—you simply have to give that something a symbolic name, and call it by that name whenever you want to look at it or change it. In assembly language, you must always be cognizant of where things are in your computer's memory. Therefore, in working through this book, pay special attention to the concept of memory addressing, which is nothing more than the art of specifying where something is. The Game of Assembly Language is peppered with addresses and instructions that work with addresses (such as `MOVE data at B`

`to C`, which means move the data stored at the address specified by register B to register C). Addressing is by far the trickiest part of assembly language, but master it and you've got the whole thing in your hip pocket.

Everything I've said so far has been orientation. I've tried to give you a taste of the big picture of assembly language and how its fundamental principles relate to the life you've been living all along. Life is a sequence of steps and tests, and so are board games—and so is assembly language. Keep those metaphors in mind as we proceed to get real by confronting the nature of computer numbers.