Introduction to Trading

Algorithm Development

What Is an Algorithm?

An Algorithm is an effective procedure, a way of getting something done in a finite number of discrete steps.

David Berlinski

Berlinski's definition is exactly right on the money. The word *algorithm* sounds mysterious as well as intellectual but it's really a fancy name for a recipe. It explains precisely the stuff and steps necessary to accomplish a task. Even though you can perceive an algorithm to be a simple recipe, it must, like all things dealing with computers, follow specific criteria:

1. *Input:* There are zero or more quantities that are externally supplied.
2. *Output:* At least one quantity is produced.
3. *Definiteness:* Each instruction must be clear and unambiguous.
4. *Finiteness:* If we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after a finite number of steps.

5. *Effectiveness:* Every instruction must be sufficiently basic that it can in principle be carried out by a person using only pencil and paper. It is not enough that each operation be definite as in (3), but it must also be feasible. [*Fundamentals of Data Structures*: Ellis Horowitz and Sartaj Sahni 1976; Computer Science Press]

These criteria are very precise because they can be universally applied to any type of problem. Don't be turned off thinking this is going to be another computer science text, because even though the criteria of an algorithm seem to be very formal, an algorithm really is straightforward and quite eloquent. It is basically a guide that one must follow to convert a problem into something a computer can solve. Anybody can design an algorithm following these criteria with pencil and paper. The only prerequisite is that you must think like a Vulcan from *Star Trek*. In other words, think in logical terms by breaking ideas down into rudimentary building blocks. This is the first step—translation of idea into an algorithm. It takes practice to do this, and this is in part why programming can be difficult.

Another thing that makes programming difficult is understanding a computer language's syntax. Most people who have been exposed to a programming or scripting language at one time or another in their lives have probably exclaimed something like, "I forgot one stupid semicolon and the entire program crashed! Isn't the computer smart enough to know that? *Arrgh!* I will never be a computer programmer!" The question that is proffered in this temper tantrum is the question of the computer's intelligence. Computers are not smart—they only do what we tell them. It doesn't matter if you spend $500 or $5,000 on the hardware. They do things very quickly and accurately, but their intelligence is a reflection of their programmer's ability to translate idea into algorithmic form and then into proper syntax.

Algorithmic (algo) traders don't necessarily need to be programmers, but they must understand what a computer needs to know to carry out a trading signal, position sizing, and money management. If you can create an algorithm, then you are more than 50 percent there. I say more than 50 percent because most traders will utilize trading software and its associated programming or scripting language. Learning the syntax of a scripting language or a small subset of a programming dialect is much easier than learning an entire programming language like C# or C++. An algo trader only needs to be concerned with the necessary tools to carry out a trading system. The developers of EasyLanguage, AmiBroker, or TradersStudio's main objective was to provide only the necessary tools to put a trading idea into action. They accomplished this by creating a vast library of trading functions, easy access to these functions, and a simplified programming syntax. Now if you want to develop your own testing platform and want to use a full-blown programming language to do so, then you will need to know the language inside-out. If you are interested in doing this, Chapters 5 and 6 will give you a head start. In these chapters, I show how I developed testing platforms in Python and Microsoft VBA from scratch.

However, at this introductory stage, let's take a look at a very simple trading algorithm and the possible exchange between a computer and trader. Pretend a trader wants to develop and test a simple moving-average crossover system and wants to use software specifically designed for system testing. Let's call this first trader AlgoTrader1, and since he has used this particular testing platform he knows it understands a trading vernacular and provides access to the common indicator functions and data. Box 1.1 shows a possible exchange between trader and computer.

Box 1.1 Algo Testing Software

```
AlgoTrader1 – AlgoTester ON
Computer – AlgoTester ready
AlgoTrader1 – load crude oil futures data
Computer – data loaded
AlgoTrader1 – buy whenever close is above moving average
Computer – "moving average" function requires three inputs
AlgoTrader1 - help with moving average function
Computer - function calculates simple, weighted, exponential average
Computer - function syntax moving average (type, price, length)
AlgoTrader1 - buy whenever close is above moving average
  (simple,close,21)
Computer – command completed
AlgoTrader1 –short whenever close is below moving average
  (simple,close,21)
Computer – command completed
AlgoTrader1 – save algorithm as MovAvgCross
Computer – command completed
AlgoTrader1 – run MovAvgCross algorithm
Computer – run completed and results are:
  $12,040 profit, $8,500 draw down, $1,200 avg. win

AlgoTrader1 – load euro currency data
Computer – command completed
AlgoTrader2 – run MovAvgCross algorithm
Computer – run completed and results are:
  -$32,090 profit, $40,000 draw down, $400 avg. win

AlgoTrader1 – edit MovAvgCross algorithm
Computer – command completed
AlgoTrader2 – edit moving average function
Computer - command completed
AlgoTrader2 – change length input to 30
Computer – command completed
AlgoTrader2 – run MovAvgCross algorithm
Computer – run completed and blah blah blah
```

As you can see, the computer had to be somewhat spoon-fed the instructions. The software recognized many keywords such as: **load**, **buy**, **short**, **run**, **edit**, **change**, and **save**. It also recognized the moving average function and was able to provide information on how to properly use it. The trading algorithm is now stored in the computer's library and will be accessible in the future.

This simple exchange between computer and AlgoTrader1 doesn't reveal all the computations or processes going on behind the scene. Loading and understanding the data, applying the algorithm properly, keeping track of the trades, and, finally, calculating all of the performance metrics did not involve any interaction with the trader. All this programming was done ahead of time and was hidden from the trader and this allows an algo trader to be creative without being bogged down in all the minutiae of a testing platform.

Even though the computer can do all these things seamlessly it still needed to be told exactly what to do. This scenario is similar to a parent instructing a child on how to do his first long-division problem. A child attempting long division probably knows how to add, subtract, multiply, and divide. However, even with these "built-in" tools, a child needs a list of exact instructions to complete a problem. An extended vocabulary or a large library of built-in functions saves a lot of time, but it doesn't necessarily make the computer any smarter. This is an example of knowledge versus intelligence—all the knowledge in the world will not necessarily help solve a complex problem. To make a long story short, think like a computer or child when developing and describing a trading algorithm. Box 1.2 shows an algorithmic representation of the long-division process to illustrate how even a simple process can seem complicated when it is broken down into steps.

Box 1.2 Procedure for Long Division

Suppose you are dividing two large numbers in the problem $n \div m$. In this example, the dividend is n and the divisor is m.

If the divisor is not a whole number, simplify the problem by moving the decimal of the divisor until it is to the right of the last digit. Then, move the decimal of the dividend the same number of places. If you run out of digits in the dividend, add zeroes as placeholders.

When doing long division, the numbers above and below the tableau should be vertically aligned.

Now you are ready to divide. Look at the first digit of the dividend. If the divisor can go into that number at least once, write the total number of times it fits completely above the tableau. If the divisor is too big, move to the next digit of the dividend, so you are looking at a two-digit number. Do this until the divisor will go into the dividend at least once. Write the number of times the divisor can go into the dividend above the tableau. This is the first number of your quotient.

Multiply the divisor by the first number of the quotient and write the product under the dividend, lining the digits up appropriately. Subtract the product from the dividend. Then, bring the next digit from the quotient down to the right of the difference. Determine how many times the divisor can go into that number, and write it above the tableau as the next number of the quotient.

Repeat this process until there are no fully divisible numbers left. The number remaining at the bottom of the subtraction under the tableau is the remainder. To finish the problem, bring the remainder, r, to the top of the tableau and create a fraction, r/m.

A few years ago a client came to me with the following trading system description and hired me to program it. Before reading the description, see if you can see any problems the programmer (me) or a computer might encounter before the directives can be properly carried out.

> Buy when the market closes above the 200-day moving average and then starts to trend downward and the RSI bottoms out below 20 and starts moving up. The sell short side is just the opposite.

Did you see the problems with this description? Try instructing a computer to follow these directions. It doesn't matter if a computer has a vast library of trading functions; it still would not understand these instructions. The good news was, the trader did define two conditions precisely: close greater than 200-day moving average and relative strength index (RSI) below 20. The rest of the instructions were open to interpretation. What does *trending downward* or *bottoming out* mean? Humans can interpret this, but the computer has no idea what you are talking about. I was finally able, after a couple of phone calls, to discern enough information from the client to create some pseudocode. *Pseudocode* is an informal high-level description of the operating principle of a computer program or algorithm. Think of it as the bridge between a native language description and quasi-syntactically correct code that a computer can understand. Translating an idea into pseudocode is like converting a nebulous idea into something with structure. Here is the pseudocode of the client's initial trading idea:

Algorithm Pseudocode

```
if close > 200 day moving average and
close < close[1] and close [1] < close [2] and
close[2] < close[3] and yesterday's 14 day RSI < 20 and
yesterday's 14 day RSI < today's 14 day RSI then BUY
```

If this looks like Latin (and you don't know Latin), don't worry about it. The [1] in the code just represents the number of bars back. So close [2] represents the close price prior to yesterday. If you are not familiar with RSI, you will be after Chapter 2. By the time you are through with this book you will be able to translate this into English, Python, EasyLanguage, AmiBroker, or Excel VBA. Here it is in English.

> If today's close is greater than the 200-day moving average of closing prices and today's close is less than yesterday's close and yesterday's close is less than the prior day's close and the prior day's close is less than the day before that and the 14-day RSI of yesterday is less than 20 and the 14-day RSI of yesterday is less than the 14-day RSI of today, then buy at the market.

Notice how the words *downward* and *bottoming out* were removed and replaced with exact descriptions:

downward: today's close is less than yesterday's close and yesterday's close is less than the prior day's close and the prior day's close is less than the day before. The market has closed down for three straight days.

bottoming out: the 14-day RSI of yesterday is less than 20 and the 14-day RSI of yesterday is less than the 14-day RSI of today. The RSI value ticked below 20 and then ticked up.

Also notice how the new description of the trading system is much longer than the original. This is a normal occurrence of the evolution of idea into an exact trading algorithm.

And now here it is in the Python programming language:

Actual Python Code

```
if myClose[D0] < sAverage(myClose,200,D0,1) and
  myClose[D0] < myClose[D1] and myClose[D2] < myClose[D3] and
  myClose[D1] < myClose[D2] and rsiVal[D1] < 20 and rsiVal[D1]
  < rsiVal[D0]:
    buyEntryName = 'rsiBuy'
    entryPrice = myOpen
```

Don't get bogged down trying to understand exactly what is going on; just notice the similarity between pseudo and actual code. Now this is something the computer can sink its teeth into. Unfortunately, reducing a trading idea down to pseudocode is as difficult as programming it into a testing platform. The transformation from a trader to an algo trader is very difficult and in some cases cannot be accomplished. I have worked with many clients who simply could not reduce what they saw on a chart into concise step-by-step instructions. In other cases the reduction of a trading idea removes enough of the trader's nuances that it turned something that seemed plausible into something that wasn't.

It goes without saying that if you don't have an algorithm, then all the software in the world will not make you a systematic trader. Either you have to design your own or you have to purchase somebody else's. Buying another person's technology is not a bad way to go, but unless the algorithm is fully disclosed you will not learn anything. However, you will be systematic trader. I have spent 27 years evaluating trading systems and have come across good and bad and really bad technology. I can say without a doubt that one single type of algorithm does not stand head and shoulders above all the others. I can also say without a doubt there isn't a correlation between the price of a trading system and its future profitability. The description in Box 1.3 is very similar to a system that sold for over $10,000 in the 1990s.

Box 1.3 Trading Algorithm Similar to One That Sold for $10K in the 1990s Description

Entry Logic:

```
    If the 13-day moving average of closes > the 65-day moving average of closes
and the 13-day moving average is rising and the 65-day moving average is rising
then buy the next day's open
    If the 13-day moving average of closes < the 65-day moving average of closes
and the 13-day moving average is falling and the 65-day moving average is falling
then sell the next day's open
```

Exit Logic:

```
If in a long position then
    set initial stop at the lowest low of the past 13 days
If in a short position then
    set initial stop at the highest high of the past 13 days
Once profit exceeds or matches $700 pull stops to break even
```

If in a long position use the greater of:
- Breakeven stop—if applicable
- Initial stop
- Lowest low of a moving window of the past 23 days

If in a short position use the lesser of:
- Breakeven stop—if applicable
- Initial stop
- Highest high of the moving window of the past 23 days

That is the entire system, and it did in fact sell for more than $10K. This boils down to a simple moving-average crossover system trading in the direction of the shorter- and longer-term trend. The description also includes a complete set of trade management rules: protective, breakeven, and trailing stop. This is a complete trading system description, but as thorough as it seems there are a couple of problems. The first is easy to fix because it involves syntax but the second involves logic. There are two words that describe market direction that cannot be interpreted by a computer. Do you see them? The two words in question are: *rising* and *falling*. Again, descriptive words like these have to be precisely defined. This initial problem is easy to fix—just inform the computer the exact meaning of *rising* and *falling*. Second, it has a weakness from a logical standpoint. The algorithm uses $700 as the profit level before the stop can be moved to break even. Seven hundred dollars in the 1990s is quite a bit different than it is today. The robustness of this logic could be dramatically improved by using a market-derived parameter. One could use a volatility measure like the average range of the past N-days. If the market exhibits a high level of volatility, then the profit objective is larger and the breakeven stop will take longer to get activated. You may ask, "Why is this beneficial?" Market noise is considered the same as volatility, and the more noise, the higher likelihood of wide market swings. If trading in this environment, you want to make sure you adjust your entries and exits so you stay outside the noise bands.

This algorithm was very successful in the 1980s and through a good portion the 1990s. However, its success has been hit-and-miss since. Is this algorithm worth $10K? If you were sitting back in 1993 and looked at the historical equity curve, you might say yes. With a testing platform, we can walk forward the algorithm, and apply it to the most recent data and see how it would have performed and then answer the question. This test was done and the answer came out to be an emphatic *no!*

Had you bought this system and stuck with it through the steep drawdowns that have occurred since the 1990s, you would have eventually made back your investment (although not many people would have stuck with it). And you would have learned the secret behind the system. Once the secret was revealed and your checking account was down the $10K, you might have been a little disappointed knowing that basic tenets of the system had been around for many years and freely disseminated in books and trade journals of that era. The system may not be all that great, but the structure of the algorithm is very clean and accomplishes the tasks necessary for a complete trading system.

The most time-consuming aspect when developing a complete trading idea is coming up with the trade entry. This seems somewhat like backward thinking because it's the exit that determines the success of the entry. Nonetheless, the lion's share of focus has always been on the entry. This system provides a very succinct trade entry algorithm. If you want to develop your own trading algorithm, then you must also provide the computer with logic just as precise and easy to interpret. Getting from the nebula of a trading idea to this point is not easy, but it is absolutely necessary. On past projects, I have provided the template shown in Box 1.4 to clients to help them write their own entry rules in a more easily understood form. You can download this form and a similar exit template at this book's companion website:www.wiley.com/go/ultimatealgotoolbox.

Box 1.4 Simple Template for Entry Rules

Long / Short Entries

Calculations and/or Indicators Involved (specify lookback period). Don't use ambiguously descriptive words like *rising*, *falling*, *flattening*, *topping* or *bottoming out*.

____

____

____

Buy Condition—What must happen for a long signal to be issued? List steps in chronological order. And remember, don't use ambiguously descriptive words like *rising*, *falling*, *flattening*, *topping*, or *bottoming out*.

____

____

____

Sell Condition—What must happen for a short signal to be issued? List steps in chronological order.

____

____

____

Here is one of the templates filled out by a one-time client:

Calculations and/or Indicators Involved (specify lookback period)

Bollinger Band with a 50-day lookback

Buy Condition—What must happen for a long signal to be issued? List steps in chronological order.

1. Close of yesterday is above 50-day upper Bollinger Band
2. Today's close < yesterday's close
3. Buy next morning's open

Sell Condition—What must happen for a short signal to be issued? List steps in chronological order.

1. Close of yesterday is below 50-day lower Bollinger Band
2. Today's close > yesterday's close
3. Sell next morning's open

The simple Bollinger Band system shown in Box 1.4 is seeking to buy after the upper Bollinger Band penetration is followed by a down close. The conditions must occur within one daily bar of each other. In other words, things must happen consecutively: the close of yesterday is > upper Bollinger Band and close of today < yesterday's close. The sell side of things uses just the opposite logic. The template for exiting a trade is very similar to that of entering. Box 1.5 contains a simple template for exit rules and shows what the client from Box 1.4 had completed for his strategy.

Box 1.5 Simple Template for Exit Rules

Exits

Calculations and/or Indicators Involved (specify lookback period). Don't use ambiguously descriptive words like *rising*, *falling*, *flattening*, *topping*, or *bottoming out*.

____

____

____

Long Liquidation Condition—What must happen to get out of a long position? List steps in chronological order.

____

____

____

Short Liquidation Condition—What must happen to get out of a short position? List steps in chronological order.

____

____

____

Calculations and/or Indicators Involved (specify lookback period)
Average true range (ATR) with a 10-day lookback
Moving average with a 50-day lookback

Long Liquidation Condition—What must happen to get out of a long position? List steps in chronological order.

1. Close of yesterday is less than entry price −3 ATR—get out on next open
2. Close of yesterday is less than 50-day moving average—get out on next open
3. Close of yesterday is greater than entry price +5 ATR—get out on next open

Short Liquidation Condition—What must happen to get out of a short position? List steps in chronological order.

1. Close of yesterday is greater than entry price +3 ATR—get out on next open
2. Close of yesterday is greater than 50-day moving average—get out on next open
3. Close of yesterday is less than entry price −5 ATR—get out on next open

There are three conditions that can get you out of a long position. Unlike the entry logic, the timing or sequence of the conditions do not matter. The exit algorithm gets you out on a volatility-based loss, a moving average–based loss/win, or a profit objective. The short liquidation criteria are just the opposite of the long liquidation criteria.

This is a complete, fully self-contained trend-following trading algorithm. It has everything the computer would need to execute the entries and exits. And it could be made fully automated; the computer could analyze the data and autoexecute the trading orders. Any platform such as AmiBroker, TradeStation, Ninja Trader, and even Excel could be used to autotrade this type of system. Any trading idea, if reduced to the exact steps, can be tested, evaluated, and autotraded. Once the programming code has been optimized, verified, validated, and installed on a trading platform, the trader can just let the system run hands off. This is the true beauty of algorithmic system trading: A computer can replicate a trader's ideas and follow them without any type of human emotion.

How to Get My Trading Idea into Pseudocode

The aforementioned template is a great way to get your ideas into written instructions. However, just like with great writing, it will take many revisions before you get to the pseudocode step. Over the years, I have discovered there are two different paradigms when describing or programming a trading system. The first is the easier to program and can be used to describe a system where events occur on a consecutive bar basis. For example: *Buy when today's close is greater than yesterday's and today's high is greater than yesterday's high and today's low is less than yesterday's low*. The other paradigm can be used to describe a trading system that needs things to happen in a sequence but not necessarily on a consecutive bar-by-bar basis. Unfortunately, the latter paradigm requires a heightened level of description and is more difficult to program. Many trading systems can fit inside a cookie-cutter design, but most traders are very creative, and so are their designs. Don't fret, though, because any idea that is reducible to a sequence of steps can be described and programmed.

The Tale of Two Paradigms

I have developed names for these two different models of describing/programming trading algorithms:

1. The variable bar liberal sequence paradigm
2. The consecutive bar exact sequence paradigm

The variable bar paradigm can provide a very robust algorithm and at the same time provide a nearly limitless expression of a trader's creativity. We will begin with this one since it is the less intuitive of the two. What you learn about the variable bar sequence will make shifting to the simpler consecutive bar sequence that much easier. The consecutive bar sequence can usually be programmed by using a single if-then construct. You can see this by referring to my earlier example:

> **If** today's close is greater than yesterday's close and today's high is greater than yesterday's high and today's low is less than yesterday's low, **then** buy at the market.

This paradigm is simple to explain, program, and deploy. There is nothing wrong with this and, remember, many times a simple approach or idea to trading the markets can deliver a very robust algorithm. Both paradigms will be exemplified in different algorithms through the rest of the book. All trading algorithms fall into

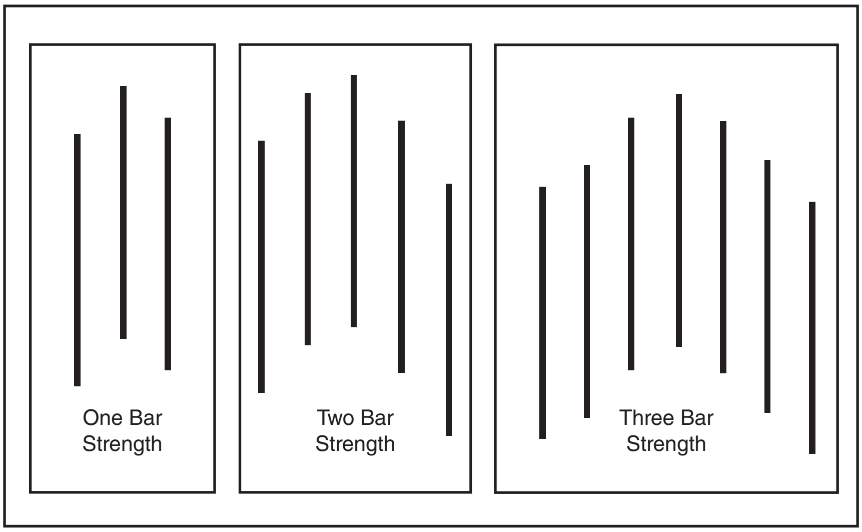

FIGURE 1.1 Examples of strength differences between pivot high points on daily bars.

either paradigm or across both, meaning that some systems require both approaches to be fully described and programmed. Most traders can easily describe and probably program a consecutive bar-type algorithm. However, traders that can't put their ideas into such a simple form and have had little or no programming experience have a more difficult time reducing their ideas into a formal description, not to mention pseudocode.

The Variable Bar Sequence Jumping headfirst into paradigm #1, the variable bar sequence, here is a somewhat creative design that incorporates pivot points and a sequence that can occur over a variable amount of days. For those of you who are not familiar with pivot points, Figure 1.1 shows different examples of pivot high points on daily bars and their differing strengths.

The strength of the pivot bar is based on the number of bars preceding and following the high bar. A two-bar pivot is simply a high bar with two bars before and after that have lower highs. The preceding/following highs do not necessarily have to be in a stairstep fashion. Here is our long entry signal description using step-by-step instructions.

Buy:

Step 1: Wait for a pivot high bar of strength 1 to appear on the chart and mark the high price.

Step 2: Wait for the first low price that is 2 percent lower than the high price marked in Step 1. If one occurs, move to Step 3. If the market moves above the high marked in Step 1, then go back to Step 1 and wait for another pivot high bar.

Step 3: Wait for another pivot high bar that exceeds the price in Step 1. Once a high bar pivot occurs that fits the criteria, mark the high price.

Step 4: Once the subsequent pivot high bar is found, then wait for another low price that is 2 percent below the high marked in Step 3. When a bar's low price fulfills the criteria buy that bar's close. If the market moves above the high marked in Step 3, then go back to Step 3 and wait for another pivot high bar. The new high that just occurred may turn out to be the pivot high bar that you are looking for in Step 3.

Additional notes: If 30 days transpire before proceeding from Step 1 to Step 4, then reset.

Figure 1.2 illustrates the sequence the above description is trying to capture. Ignore the state designations for now. They will become apparent in the following discussion.

It is easy to see the designer/trader of this entry signal is trying to buy after two pivot highs are separated and followed by a pullback of 2 percent. In addition, there are a couple of conditions that must also be met before a buy signal is triggered: (1) the second pivot high price must be higher than the first, and (2) the whole sequence must take less than 30 bars to complete. When a system allows its entry criteria to work across an unknown number of days or price levels, conditions must be used to keep the integrity of the entry signal and limit the duration of the sequence.

Since there is variability in the instruction set, the programming of this paradigm is definitely more difficult. However, as mentioned earlier, it is very doable. As a young

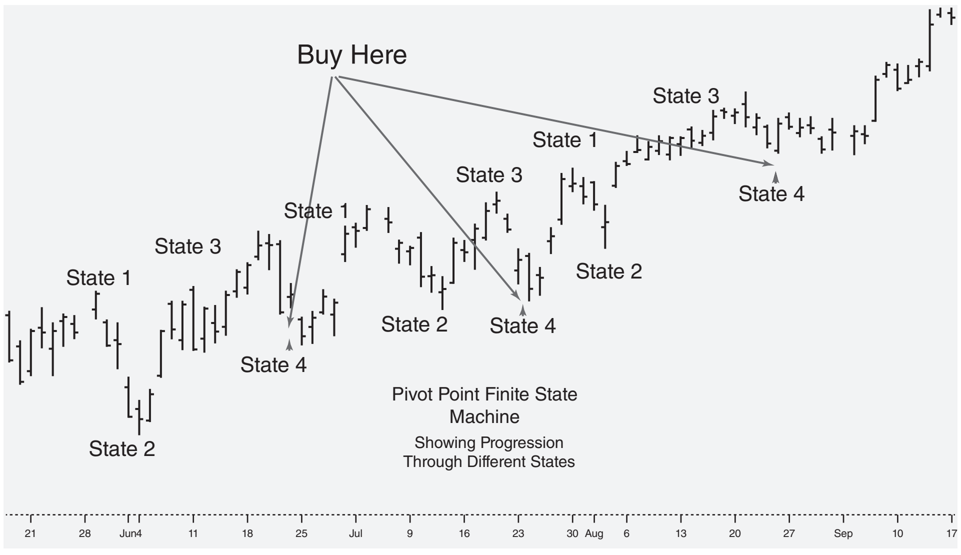

FIGURE 1.2 A depiction of the variable-bar sequence described by the long entry signal.

and inexperienced trading system programmer, I tried to program these variable sequences using a bunch of true/false Boolean flags. Box 1.6 shows an initial attempt using this method to describe the pivot point sequence. The flags are bolded so you can easily see how they are turned on and off. Comments are enclosed in brackets { }.

Box 1.6 Using Boolean Flags to Program a Trading Algorithm

```
HighPivotFound = High of yesterday > High of today and High
  of yesterday > High of two days ago
If HPivot1 = False and HighPivotFound then {first Pivot high found}
   HPivot1 = True
   HPivot1Price = high of yesterday
   HPivot1Cnt = currentBar

If HPivot1 = True and Low < HPivot1Price * 0.98 then
  {2% retracement after first Pivot high}
   LRetrace = True

If HPivot1 = True and LRetrace = False and High of today >
  HighPivot1Price then
   HPivot1Price = High of yesterday {another higher high but
     not a 2% retracement - start over}
   HPivot1Cnt = currentBar

If LRetrace = True and HighPivotFound and High of yesterday
  > HPivot1Price then
   HPivot2 = True {second Pivot high > first Pivot high and
     2% retracement between the two}
   HPivot2Price = High of yesterday

If HPivot2 = True and High of today > HPivot2Price then
   HPivot2Price = High of today {another higher high > second
     Pivot High - keep track of it}

If HPivot2 = True and Low < HPivot2Price * 0.98 then
   Buy this bar on close {Buy Order Placed - entry criteria
     has been met}
   HPivot1 = False {Start Over}
   LRetrace = False
   HPivot2 = False

HPivot1Cnt = HPivot1Cnt + 1
If HPivot1Cnt >= 30 then {reset all Boolean flags - start over}
   HPivot1 = False
   LRetrace = False
   HPivot2 = False
```

As you can see, the description using these flags is somewhat laborious and not that flexible. The flags have to have descriptive names so that the correct flag is being tested and subsequently turned on or off. In addition, all flags must eventually be turned off when either the HPivot1Cnt variable grows equal to or greater than 30 or a LONG position is established. This type of programming will work but isn't very eloquent. Good programmers don't like to use superfluous variable names and many lines of code to complete a task. And really good programmers like for others to easily read their code and comment on how clever the solution was to the problem. Eloquent code is precise and clever. So, after much trial and error as a young programmer, I finally realized that these types of trading algorithms (the paradigm #1 variety) were simply looking for an occurrence of a certain sequence or pattern in the data. When I say pattern, I don't mean a certain price pattern like a candlestick formation. As we have seen before, a trading algorithm is just a sequence of instructions. Remembering back to my compiler design class in college and how we had to write computer code to recognize certain tokens or words (patterns) in a string of characters I felt I could use the same model to program these types of systems, a universal model that could be molded to fit any trading system criteria. This model that programmers use to parse certain words from a large file of words is known as a finite state machine (FSM). The FSM concept is not as daunting as it sounds. If we translate the geek-speak, a FSM is simply a model of a system that shows the different possible states a system can reach and the transitions that move the system from one state to another. The machine starts at a START state and then moves methodically through several states by passing certain logical criteria and then arrives at an ACCEPT state.

Don't let this idea blow your mind because it is quite easy to understand and implement. Let's start off with a very simple FSM to illustrate its simplicity and eloquence. Think of a combination lock that can only be unlocked by inputting the following numbers: 6, 4, 2, 7, 5, and 1. Like most combination locks the numbers need to be inputted in the exact sequence described or the lock will not open. This lock is not very good because it lets you test to see if you have the right number before proceeding to the next input. So you plug in a number, test it, and either try again if it fails or move onto the next number if it is correct. Eventually, with time the correct combination will be inputted and the lock will open. Remember this is just a very simple example of something that can be modeled by a FSM. Without an illustration or diagram, most nonprogrammers could not design even this simple example. A picture is always worth a minimum of a thousand words and the easiest way to create an appropriate FSM is to create a diagram. The diagram in Figure 1.3 describes the FSM that models the combination lock.

The diagram is made up of circles, which represent the different **STATES** and connectors that show how the machine moves from one state to another. This FSM has a total of seven states that include a **START** and **ACCEPT** state. Pretend you are

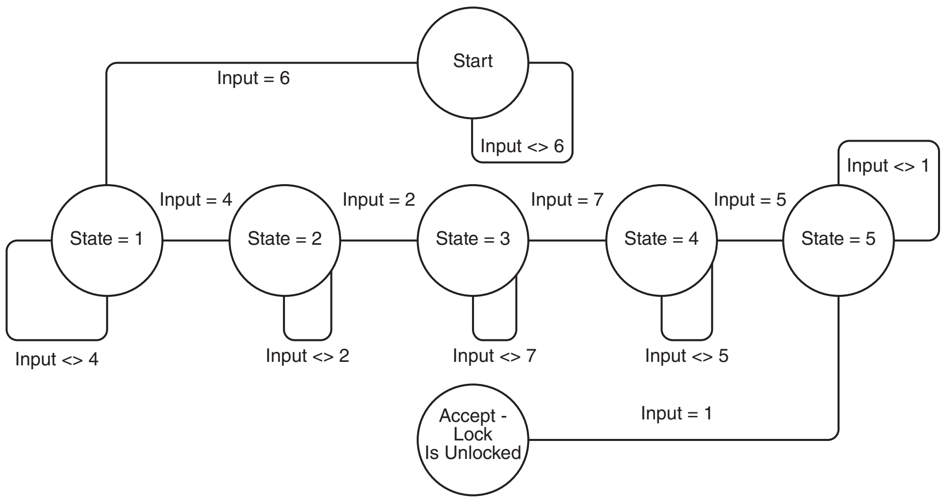

FIGURE 1.3 An FSM that models the workings of a combination lock.

sitting in front of a small screen and a numeric touchpad similar to one that is found on a cell phone, and it is prompting you to input a number from one to nine. At this point, the machine is set to the START state. If you get lucky right off the bat, you input the number six and the screen prompts you to input the second number. If you aren't lucky, it will ask you to re-input the first number in the combination. So following along with the diagram, you can see how the number six moves the machine from the START state to STATE 1. If the number six is not inputted, it moves right back to the START state. The machine moves along the paths visiting each STATE as long as the proper number is inputted. Notice you are not penalized if you don't input the proper number; the machine just sits in that particular state until the proper number is inputted.

In our pivot point example, there will also be a START and ACCEPT state. Refer back to the step-by-step instructions of the pivot point entry technique and visualize the steps as different states. The START (STEP 1) state will be assigned the value 0 and the ACCEPT (STEP 4) state will have the value 4. In between, the START and ACCEPT states will be three other states that the FSM can assume. These are intermediate states that must be achieved in sequential order. The START state tries to get the ball rolling and looks at every bar to see if a pivot high (of strength 1) has occurred. If one does, then the machine moves onto the next state, and then on to the next, and then on to the next, and then finally to the ACCEPT state. As soon as the FSM assumes the ACCEPT state, a buy order is placed at the market. The illustration in Figure 1.4 is the FSM diagram of our pivot-point long-entry algorithm.

This FSM diagram looks to be much more complicated than the combination lock, but it really isn't. There are fewer states but many more connectors. In the

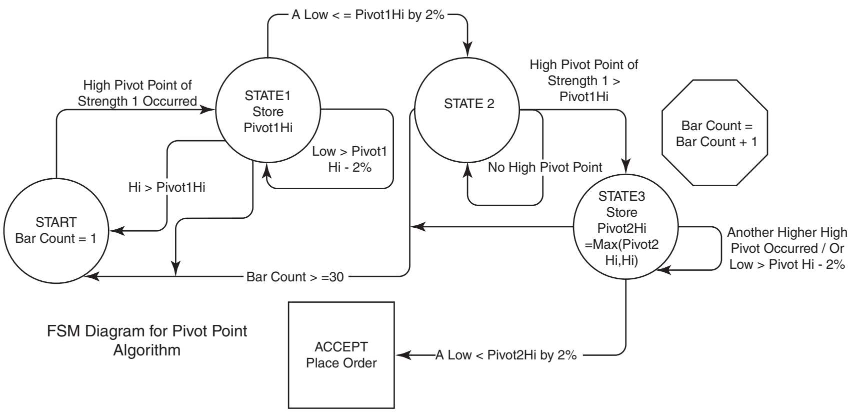
A Low < = Pivot1Hi by 2%
High Pivot Point of Strength 1 Occurred
STATE1 Store Pivot1Hi
STATE 2
High Pivot Point of Strength 1 > Pivot1Hi
Bar Count = Bar Count + 1
Low > Pivot1 Hi - 2%
No High Pivot Point
Hi > Pivot1Hi
STATE3 Store Pivot2Hi =Max(Pivot2 Hi,Hi)
START Bar Count = 1
Another Higher High Pivot Occurred / Or Low > Pivot Hi - 2%
Bar Count > =30
FSM Diagram for Pivot Point Algorithm
ACCEPT Place Order
A Low < Pivot2Hi by 2%

FIGURE 1.4 An FSM that models the pivot-point long-entry algorithm described in this chapter.

combination lock FSM there was only one connector connecting the different states together and the machine never went backward. Once a state was attained, the machine stayed there until the criteria were met to advance to the next state. This pivot-point FSM can move forward and backward.

Stepping through this diagram will help make it less scary. Keep in mind that all of these machines gobble up one bar at a time and then make a decision. Starting at the START state the machine looks for a pivot high of strength one. Once the machine finds this pivot point it then moves to STATE1 and starts looking for a low price that is 2 percent or less than the pivot high found in the START state. There are four paths out of STATE1: (1) a low is found that fits the criteria, (2) a low price fitting the criteria is not found, stay in STATE1, (3) 30 bars have occurred since the pivot high is found, and (4) a high exceeds the pivot high price. Only one path leads to STATE2. All other paths either return to the START state or loop back to the current state. Once the machine attains STATE2 it starts analyzing the data by gobbling each bar and searches for a higher pivot high point. Unlike STATE1 there are only three paths coming out of this state: (1) a higher pivot high is found, (2) a higher pivot high is not found, and (3) 30 bars have occurred since the pivot high was found in the START state. There is only one path to STATE3 and that is a higher pivot high. All other paths either loop or return the machine to the START state. Assuming a higher pivot high is found the machine moves to STATE3. Once in STATE3 the machine will only return to the START state if 30 bars have come and gone before the machine can attain the ACCEPT state. If new highs are found, the machine continues to stay in STATE3 and keeps track of the new highs, all the while looking for that particular low that is 2 percent or lower than the most recent pivot high. Eventually things fall into place and the machine attains the ACCEPT state and a buy order is placed. You can now refer back to Figure 1.2 and see when the machine attains the various states.

The diagram in Figure 1.4 looks clean and neat, but that's not how it started out. When I diagram a trading algorithm, I use pen/pencil and paper. A pencil is great if you don't want to waste a lot of paper. Also, you don't have to have a complete diagram to move onto the pseudocode step. Trust me when I say a diagram can help immensely. Something that seems impossible to program can be conquered by drawing a picture and providing as much detail on the picture as possible. Box 1.7 contains the pseudocode for the pivot point entry algorithm using the FSM diagram.

Box 1.7 Pivot Point Entry Algorithm for Figure 1.4

```
If initialize then
State = Start

HiPivotFound = High of yesterday > High of today and
     High of yesterday > High of prior day

If State = Start and HiPivotFound then
     State = 1
     BarCount = 1
     Pivot1Hi = High price of HiPivotFound

If State <> Start then BarCount = BarCount + 1

If State = 1 then
     If Low of today < Pivot1Hi * 0.98 then
            State = 2
     If High of today > Pivot1Hi then
            State = Start

If State = 2 then
     If HiPivotFound then
             Pivot2Hi = High Price of HiPivotFound
             If Pivot2Hi > Pivot1Hi then
                      State = 3

If State = 3 then
     If Low of today < Pivot2Hi * 0.98 then
             State = Accept
     If High of today > Pivot2Hi then
             Pivot2Hi = High of today

If State = Accept then
     Buy this market on close
     State = Start {start over}

If BarCount = 30 then
     State = Start
     BarCount = 0
```

You might understand this code or you may not. If you have never programmed or scripted, then you might have a problem with this. If you fully understand what is going on here, just bear with us a few moments. Even if you understood the diagram, this code may not be fully self-explanatory. The reason you may not understand this is because you don't know the *syntax* or the *semantics* of this particular language. Syntax was discussed earlier in this chapter, and remember, it's just the structure of the language. I tried my best to utilize a very generic pseudocode language for all the examples throughout the book and hopefully this will help with its understanding. In the pseudocode example, the language uses *if*s and *then*s to make yes-or-no decisions. If something is true, then do something, or if something is false, then do something. These decisions divert the flow of the computer's execution. The code that is indented below the if-then structure is controlled by the decision. This structure is the syntax. Now the logic that is used to make the decision and then what is carried out after the decision is the semantics. In our example, we will eventually tell the computer that if State = 1, then do something. Syntax is the grammatical structure of a language and semantics is the meaning of the vocabulary of symbols arranged within that structure. You many notice the similarity in the pseudocode and the FSM diagram. The logic that transitions the pivot point FSM from one **state** to another is mirrored almost exactly in the code. This tells you the time you spent planning out your program diagram is time well spent. So let's start with the pseudocode and make sure the FSM diagram is fully implemented.

The START state is simply looking for a pivot point high of strength 1. When one occurs the machine then shifts gears and enters STATE1. Once we have entered the STATE1 phase, the machine stays there until one of three criteria is met: (1) the elusive low that is 2 percent or lower than the Pivot1Hi, (2) a high price that exceeds the Pivot1Hi, or (3) 30 bars transpire prior to attaining the ACCEPT state. If you look at the STATE1 block of code, you will only see the logic for the first two criteria. You may ask where is the logic that kicks the machine back to the START state once 30 bars have been completed prior to the ACCEPT state. If you look further down in the code, you will see the block of code that keeps track of the BarCount. If at any time the BarCount exceeds 30 and the machine is not in the START state, the machine is automatically reset to the START state. If a low price is observed that is lower than 2 percent of the Pivot1Hi price, then the machine transitions to STATE2. However, if a high price is observed that exceeds Pivot1Hi, then the machine reverses back to the START state and it once again starts looking for a new PivotHi. Assuming the machine does make it to STATE2, it then starts looking for another PivotHi price that is greater than Pivot1Hi. A transition from STATE2 can only occur when one of two criteria are met: (1) BarCount exceeds 30, then it's back to the START state, or (2) a higher HiPivot than Pivot1Hi is observed. If the latter criterion is fulfilled, then it is onto STATE3. STATE3 looks for a low price that is 2 percent less than the Pivot2Hi price so it can transition to the ACCEPT state. At this point the BarCount is its only enemy. It doesn't care if a high price exceeds the Pivot2Hi. If it does, then it simply replaces the Pivot2Hi with this price and continues searching for a price that is 2 percent lower than Pivot2Hi. Eventually, the machine resets itself or it finally transitions to the ACCEPT state and buys market on close (MOC).

As you can see, the description of the FSM diagram is all almost completely repeated in the description of the pseudocode. If the two descriptions agree, then it is on to the actual programming of the algorithm. Now that this more complicated paradigm has been explained, I think it is time to give it a better name. How about the *FSM paradigm*? Sounds better than *the variable bar liberal sequence paradigm*, doesn't it?

The Consecutive Bar Sequence I saved *the consecutive bar exact sequence paradigm* for the end because the FSM paradigm is much more difficult to understand and much less intuitive. Armed with our new knowledge of diagrams, it is now time to move to this simpler paradigm because it will be used a lot more in your programming/testing of trading algorithms. As I stated before, most trading algorithms will be fully describable in a step-by-step fashion. Here the logic will not be needed to be modeled by a machine; a recipe approach will do just fine. Guess what type of diagram is used to model these recipe types of instructions. If you guessed a flowchart (FC), then pat yourself on the back. A good computer programming 101 instructor introduces her students to the flowchart concept/diagram way before they even sit down in front of a keyboard. Figure 1.5 shows a very simple FC.

Can you see what is being modeled there? It's quite simple; the FC diagram starts at the top and makes one decision and, based on that decision, it carries out its objective. It starts out with you waking up and observing the time and making a decision to take the bus or the subway. This is a way oversimplified example, but it covers everything an FC is designed to do: start, make decisions, carry out the appropriate instructions based on the decisions, and then finish. Figure 1.6 shows an ever so slightly more complicated FC that deals with what this book is all about, a trading algorithm.

This diagram is a flowchart of the entry technique of a very popular mean reversion trading system. This system trades in the direction of the long-term trend after a price pullback. The trend is reflected by the relationship of the closing price and its associated 200-day moving average. If the close is greater than the average, then the trend is up. The pullback is reflected by the 14-period RSI indicator entering into oversold territory—in this case, a reading of 20 or less. The diagram illustrates two decisions or tests and, if both tests are passed, then the system enters a long position

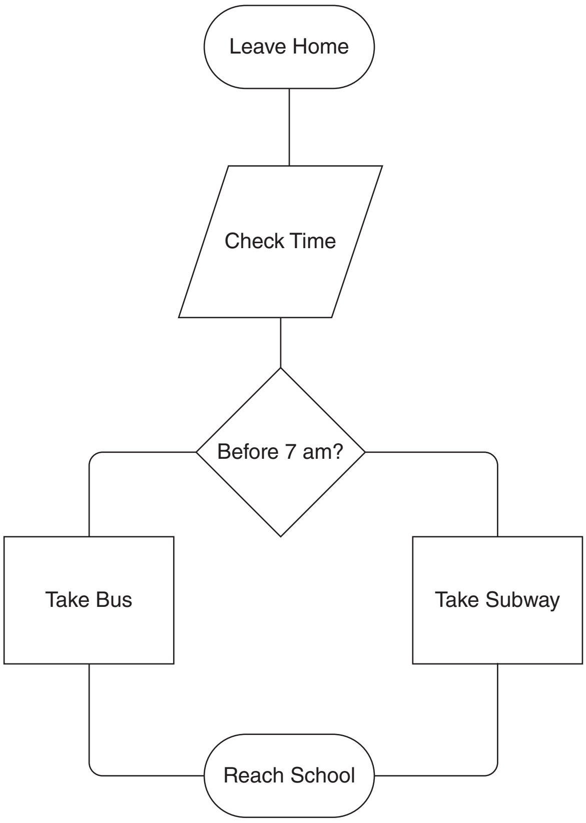

FIGURE 1.5 A very simple FC.

on an MOC order. The flow of the algorithm is linear, meaning it flows from top to bottom. The flow is diverted based on simple decisions, but it always ends at the bottom of the flowchart—either buying MOC or starting on a new bar of data. Here is the pseudocode of this mean reverting system; look quickly or you will miss it:

```
'Simple FC type trading algorithm
'An example of a mean reversion system

If Close of today > 200 day average of Close then
       If RSI(14) < 20 then
              Buy MOC
```

That's the entire entry technique, in a nutshell. Was it necessary to create a flowchart diagram? In this case, no. But as you will find out in later chapters, most trading algorithms are not this easily definable.

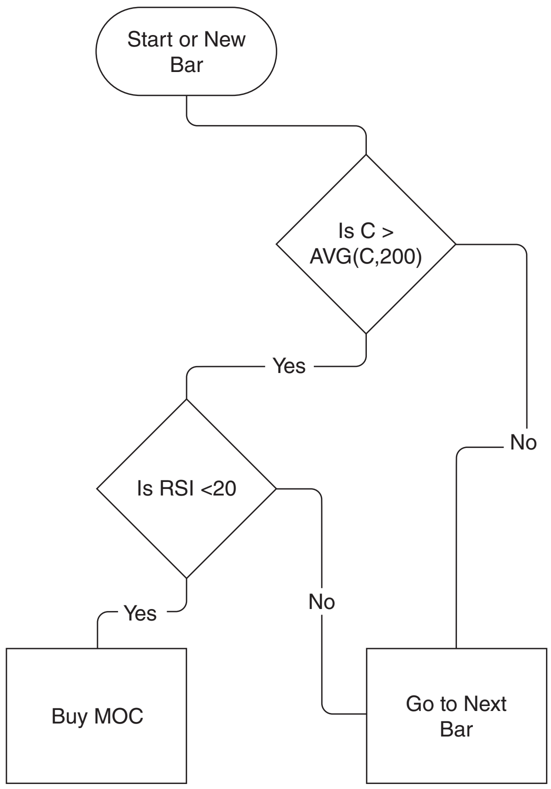

FIGURE 1.6 An FC of a trading algorithm.

Summary

In this chapter, the necessary tools for creating a trading algorithm were introduced. Describing your ideas on paper (real or virtual) is the very first step in the algorithm development process. A template was shown that can be used to help facilitate this. Just getting a trading idea on paper is not sufficient to move onto the pseudocode phase. All ideas must be translated into mathematical expressions and Boolean logic. These expressions and logic must be in a form a computer can understand. Ambiguous directions must be eliminated and replaced with pure logic. Once the idea is written down and further reduced into an extremely logical list of directives and calculations or formulae, the trader must then decide what type of paradigm to use to get the logic into something that can be eventually programmed or scripted. The FSM, as well as the FC methods, were both introduced and used to convert trading schemes into complete pseudocodes. These two paradigms have always been present, and many algorithmic traders have used them without realizing the different programming methods. With knowledge of the two different methods, hopefully an algorithmic trader can make a choice on which is best to use and get from idea to actual code in a lot less time and a lot less frustration.

Here again is a summary of the two methods:

- **Flowchart:** The "flow" of a flowchart is a process. The flowchart shows the steps and actions to achieve a certain goal. Use this method if you can define your trading logic in a step-by-step or bar-by-bar basis. A large portion of trading systems will fit into this programming paradigm.
- **Finite state machine:** The "flow" in a state diagram is always from state to state. Most FSMs have a START and ACCEPT state, similar to the beginning and end of a flowchart. Machine diagrams describe a closed system composed of multiple discrete states. The "state" in this case determines how a system behaves to stimulus or events. Use this method if criteria have to be met in a sequential manner, but the amount of time between the beginning and criteria completion is variable. Pattern-based systems will usually fall into this paradigm.

Once a potential programming method is chosen, it is time to "draw" the corresponding diagram. A diagram doesn't have to be pretty, but it must try to cover all the bases or what-if scenarios a trading system may encounter to carry out its objectives. A thoroughly thought-out diagram providing as much information as possible will definitely enable a trading algorithm to be quickly translated into pseudocode and eventually programmed into a testing platform. Over time, as your experience grows with programming trading systems, these diagrams will begin to appear in your mind's eye. However, this takes time and a lot of experience.

If you made it through this chapter, then, at least, you are now in possession of a trading algorithm that is quite similar to one that actually sold for thousands of dollars in the 1990s.