OUR most fundamental tool as intelligent beings is language. It is through language that you learn new information and share your knowledge, feelings, and experiences with others. Through language, you can express any thought anyone has ever had and describe any event, real or fictional. The world is controlled through language. Presidents to petty functionaries, generals to GIs, CEOs to clerks—all rely on language to give instructions to others and to gather information.

Language is a necessity for a computer, too. Software is created using special languages that provide instructions for telling the computer what to do. And language defines the data with which the instructions will work. Computer language is similar to human language in many ways. The nouns, verbs, prepositions, and objects found in English, for example, have their counterparts in programming code—or the source code, the actual lines of text that get translated into functioning programs. But software sentences have their own syntax, and the words that make up the languages have their own precise meanings.

Computer language is more exacting and more limited than English. An often-repeated story tells how, in an early attempt to use a computer to translate English into Russian, the phrase "The spirit is willing, but the flesh is weak" was interpreted as "The vodka is ready, but the meat is rotten." The story might be mythical, but it illustrates a reality—that computers and their languages do not do a good job of managing the ambiguities and shades of meaning in human language that any four-year-old understands (although advances in voice recognition have computers understanding what we say, if not what we mean).

If programming languages lack the subtleties of human language, human language cannot match the precision of computer-speak. Try, for example, to describe a simple spiral without using your hands. It's impossible in English. But because math is an integral part of computer languages, those languages cannot only describe a spiral but also can provide the instructions to create an image of that spiral on a display or printer.

Different Programming Languages

Just as there is more than one language for humans, so is there more than one computer language, even for the same type of computer. Generally, the various languages are described as low-level or high-level. The more a computer language resembles ordinary English, the higher its level. Lower-level languages are more difficult to work with, but they usually produce programs that are smaller and faster.

On the lowest level is **machine language**. This is a series of codes, represented by numbers (ones and zeros), used to communicate directly with the internal instructions of the PC's microprocessor. Deciphering machine language code or writing it is as complex a task as one can tackle in computing. Luckily, we don't have to do it. Programs called **interpreters** and **compilers** translate commands written in higher-level languages into machine language. We'll look at both interpreters and compilers later in this chapter.

On a slightly higher level than machine language is **assembly language**, or simply **assembly**, which uses simple command words to supply step-by-step instructions for the processor to carry out. Assembly language directly manipulates the values contained in those memory scratch pads in the microprocessor called **registers**. In machine language, the hexadecimal code 40 increases by one the value contained in the register named AX; assembly language uses the command INC AX to perform the same function. Although assembly language is more intelligible to

humans than machine language codes, assembly is still more difficult to use than higher-level languages. Assembly remains popular among programmers, however, because it creates compact, fast code.

On the high end, languages such as C and Java allow programmers to write in words and terms that more closely parallel English. And the programmer using these languages need not be concerned with such minutiae as registers. The C language is powerful and yet reasonably simple to write and understand. Currently, Java is the rising star among languages because a program written in Java will run on any computer no matter what its operating system. This is a distinct advantage when you're writing programs people will use over the Internet, using anything from PCs to Macs and Sun workstations. Software written in C, in contrast, must be modified to allow a program written for one type of computer to be used on another.

At the highest level are languages such as BASIC (Beginners All-purpose Symbolic Instruction Code), Visual Basic, the DOS batch language, and the macro languages used to automate applications such as Microsoft Office and Corel WordPerfect Office.

Software Construction

A program can be a single file—a record of data or program code saved to a disk drive. But generally, complex software consists of one file that contains a master program—the **kernel**—surrounded by a collection of files that contain subprograms, or **routines**. The kernel **calls** the routines it needs to perform some task, such as display a dialog box or open a file. A routine can also call other routines in the same file, in another file that's part of the program, or in files provided by Windows for common functions. Together, the kernel and subprograms give programs a way to receive, or **input**, information from the keyboard, memory, ports, and files, rules for handling that input data, and a way to send, or **output**, information to the screen, memory, ports, and files.

Typically, when a user types information into a program, it is stored as a **variable**. As the term suggests, the information a variable stores varies from one instance to another. Programs on their own are also capable of storing in variables the information based on the results of a calculation or manipulation of data. For example, to assign the value 3 to a variable X, BASIC uses the command X = 3. Assembly language accomplishes the same thing by assigning the value to the AX register with the command MOV AX,3. Some languages require several commands to achieve the same effect another language accomplishes with a single command.

After a program has information in a variable, it can manipulate it with commands that perform mathematical operations on numbers or parse text strings. **Parsing** is the joining, deletion, or extraction of some of the text characters to use them elsewhere in the program. When a variable is text, it is often called a **string**. You can have math strings, but most often, the term *string* refers to an uninterrupted series of alphanumeric and punctuation characters. Through parsing, a program can locate, for example, the spaces in the name "Phineas T. Fogg"; determine which parts of the string make up the first name, the middle initial, and the last name; and assign each segment to a separate variable. A typical math manipulation would be X = 2 + 2, which results in the variable X having the

value 4. If that command is then followed by X = X + 1, the new value of X would be 5. The command X = "New" + " " + "York" assigns the string "New York" to variable X.

Programs can rely on the BIOS (see Chapter 3) to perform many of the input and output functions—such as recognizing keystrokes, displaying keystrokes onscreen, sending data through the parallel and serial ports, reading and writing to RAM, and reading and writing disk files. The programming language still must have commands to trigger the BIOS services. Consider the following series of BASIC commands:

```
OPEN "FOO" FOR OUTPUT AS #1
    WRITE #1, "This is some text."
    CLOSE #1
```

These commands create a file named FOO that contains the text in quotes: This is some text. It then closes the file. The language Pascal does the same with these commands:

```
Assign (TextVariable, "FOO");
    WriteLn (TextVariable, "This is some text.");
    Close (TextVariable);
```

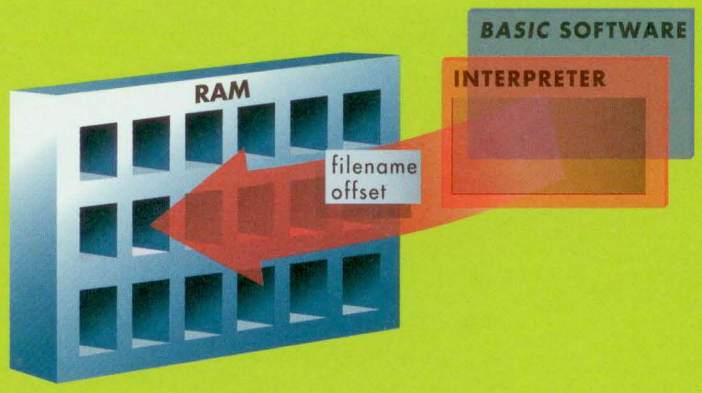
So far, we've described a fairly straightforward scheme: in, process, out. The reality is more complex. A program must be capable of performing different tasks under different circumstances—a feature that accounts for programming languages' power and versatility. And because a program hardly ever proceeds in a straight line from start to finish, there are commands that tell the computer to **branch** to different parts of the program to execute other commands. In BASIC, the command GOTO causes the execution to move to another part of the program. Assembly language does the same with the command JMP (short for jump).

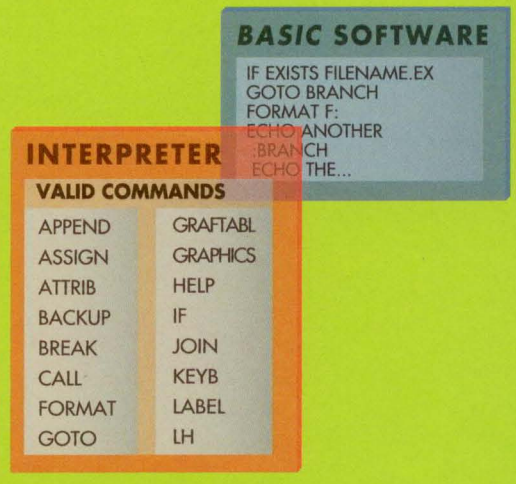
Branching is used in combination with the **Boolean logic** functions of the programming languages. For example, when a program needs to change what it's doing because a particular condition exists, it can use "**if...then...**" The program checks to see whether a certain condition is true, and if it is, the program then performs a certain command. For example, if the variable State is the string "Texas," the program uses the abbreviation TX to address a letter.

To get an idea of how programs are written, we'll look at a **flowchart**—a kind of map sometimes used by programmers to lay out the logical connections among different sections of the program code. As an example we'll use a type of game that was popular in the days when text-only adventures like *Zork* were plentiful. In such games, the user types in elementary commands, such as "Go east," "Go north," "Take knife," and "Hit monster." And the game displays a sentence describing the consequences of the player's action. Our example is oversimplified and takes into account only one small portion of such a game. As such, it gives you an idea of how many commands and how much programming logic go into even the simplest code. In our adventure, the player is already on the balcony of a castle turret surrounded by Fire Demons...

How Software Interpreters Work

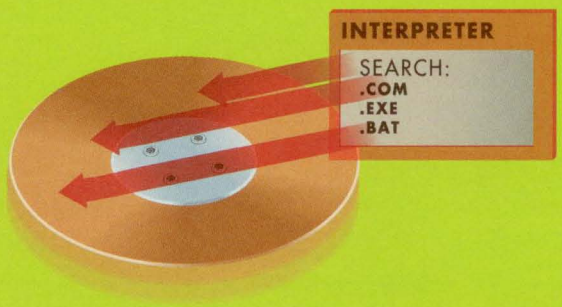
1 Some software programs can't communicate directly with a computer. They are written in **languages** that require an **interpreter**, another program that translates the software's commands into instructions the processor can use. Interpreted programs include DOS batch files, programs written in BASIC, WordPerfect macros, and Java software written for use on the Internet. Each command in the program is written on a separate line. When you launch an interpreted program, an interpreter designed particularly for that language establishes a small area in memory. There, the interpreter puts the name of the file and keeps track of its current place, called the **offset**, which is measured in the number of lines the current command is from the beginning of the file.



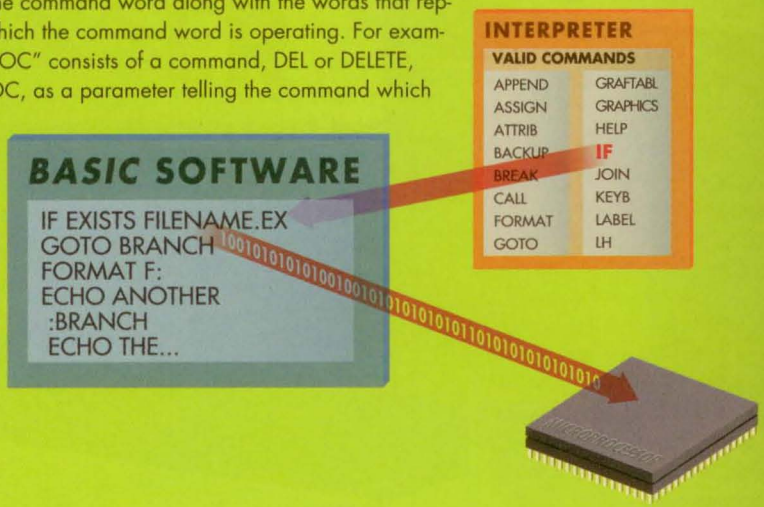


2 As the interpreter reads each line of the file, it compares the first word in the line to a list of valid **commands**. In some instances in a BASIC program, the interpreter will also recognize a **variable** at the start of a line. A variable holds some temporary data, such as a filename or a number.

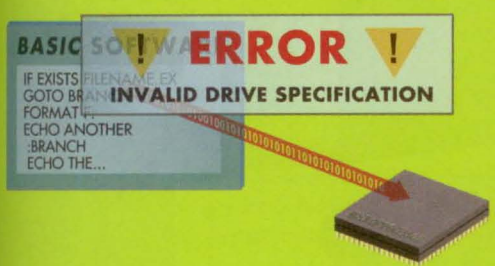
3 In a batch file, if the first word of a line is not found on the approved list, the interpreter will look for a .COM, .EXE, or .BAT file with a name matching the word. If none of these conditions are fulfilled in a batch file, or, if in a BASIC program a matching command word or variable is not found, the interpreter generates an **error message**.



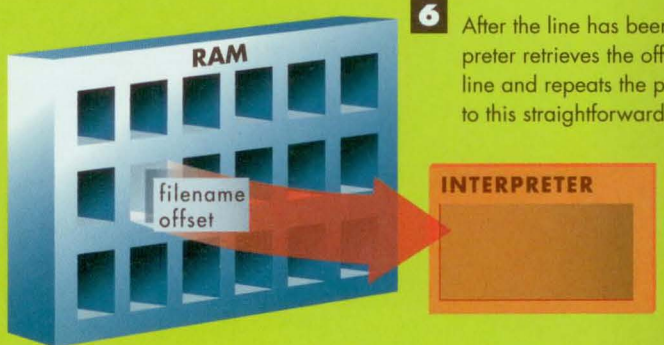
4 If the word is found on the list of valid commands, the interpreter **executes** the entire line, translating the command word along with the words that represent the parameters on which the command word is operating. For example, the line "DEL MYFILE.DOC" consists of a command, DEL or DELETE, and a filename, MYFILE.DOC, as a parameter telling the command which file to delete. The command and parameters are turned into code **tokens**—shorthand abbreviations for instructions that are passed to the microprocessor, which carries out the instructions.



5 If any of the parameters are invalid, or if they attempt to perform a forbidden operation, such as copying a file over itself, the interpreter generates a **syntax error** message.



6 After the line has been processed, the interpreter retrieves the offset location of the next line and repeats the procedure. The exception to this straightforward progression occurs if a command, such as GOTO, **branches** execution to another section of the program.



How a Compiler Creates Software

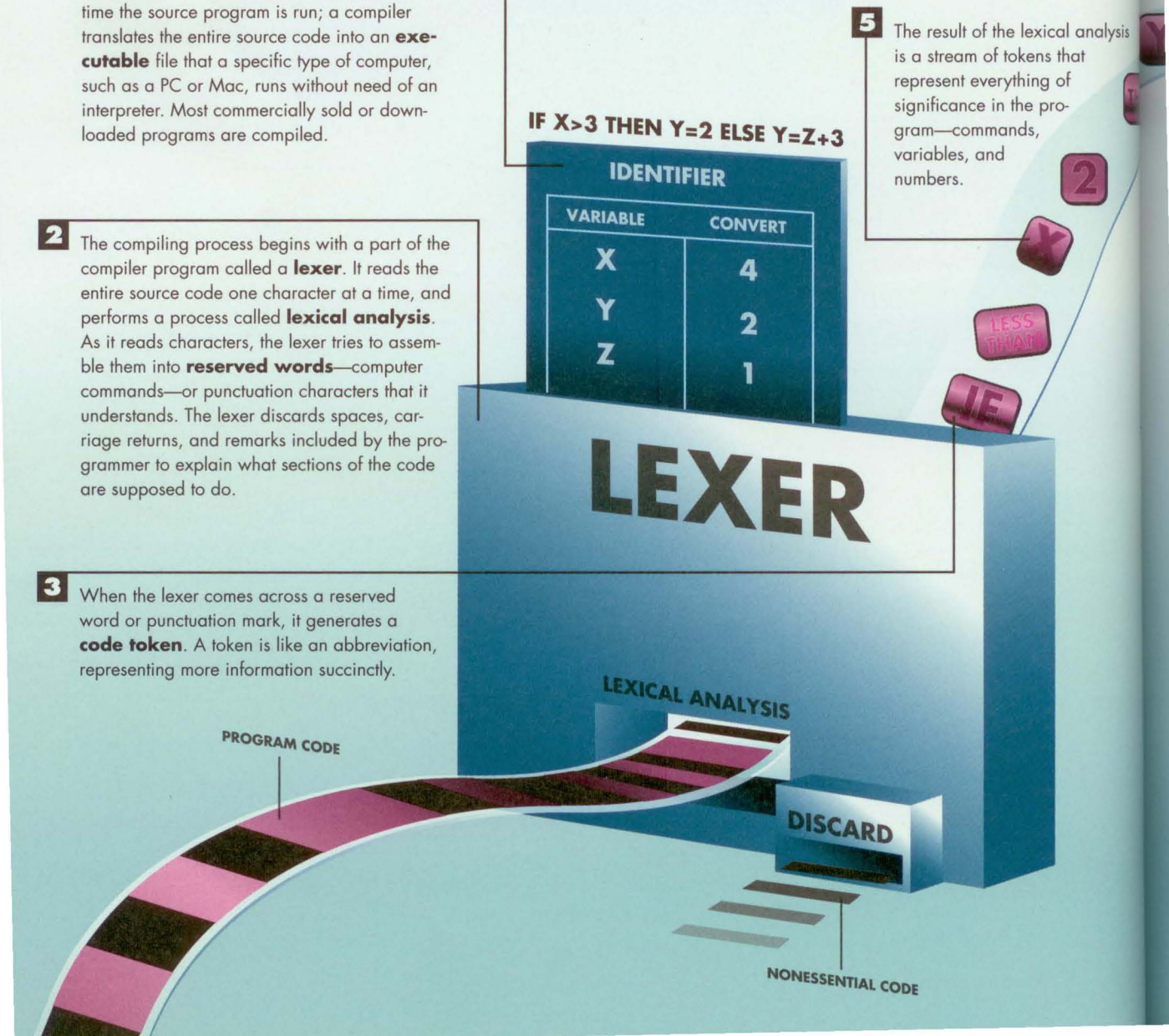
1 The **interpreter** in the previous illustration and a **compiler** are both software programs that translate program source code that humans understand, such as BASIC or C+, into machine language, which computers understand. The difference between a compiler and an interpreter is this: An interpreter translates the source code, line by line, each time the source program is run; a compiler translates the entire source code into an **executable** file that a specific type of computer, such as a PC or Mac, runs without need of an interpreter. Most commercially sold or downloaded programs are compiled.

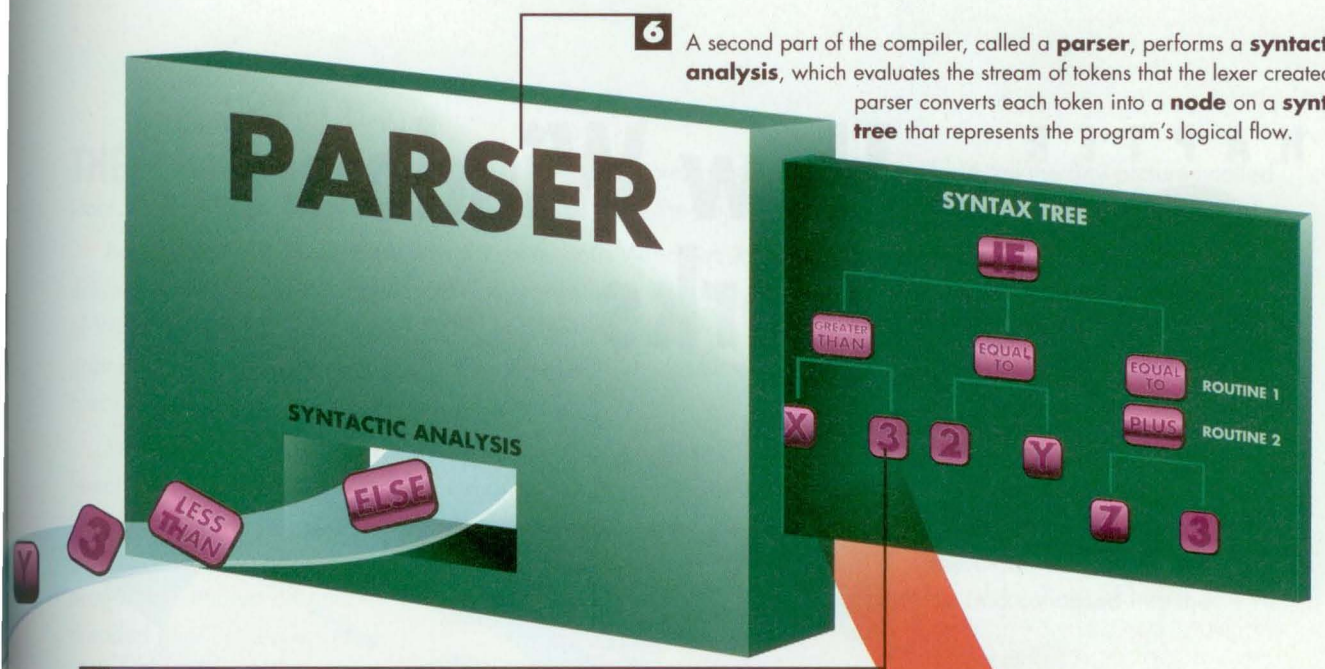
2 The compiling process begins with a part of the compiler program called a **lexer**. It reads the entire source code one character at a time, and performs a process called **lexical analysis**. As it reads characters, the lexer tries to assemble them into **reserved words**—computer commands—or punctuation characters that it understands. The lexer discards spaces, carriage returns, and remarks included by the programmer to explain what sections of the code are supposed to do.

3 When the lexer comes across a reserved word or punctuation mark, it generates a **code token**. A token is like an abbreviation, representing more information succinctly.

4 When the lexer finds a string of characters that don't form a reserved word, the lexer assumes that those characters stand for a **variable**. It assigns the variable a place in an **identifier table** that tracks the name and contents of every variable in the program. Then the lexer generates a **variable token** that points to the variable's position in the identifier table. When the lexer finds a string of numeric characters, the lexer converts the string into an integer and produces an **integer token** to stand for it.

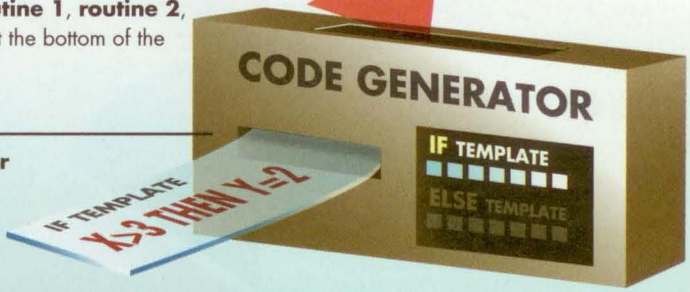
5 The result of the lexical analysis is a stream of tokens that represent everything of significance in the program—commands, variables, and numbers.





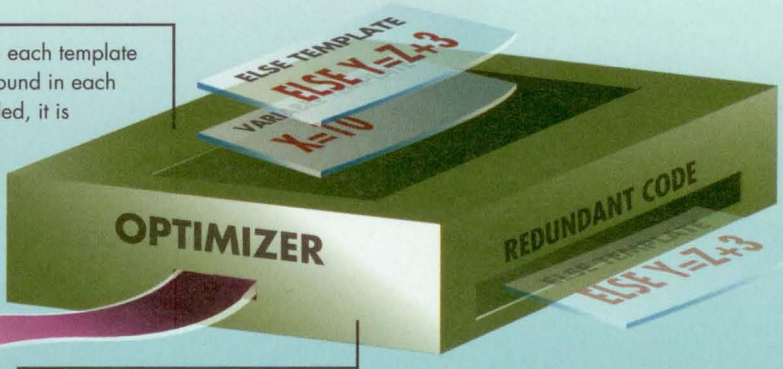
6 A second part of the compiler, called a **parser**, performs a **syntactic analysis**, which evaluates the stream of tokens that the lexer created. The parser converts each token into a **node** on a **syntax tree** that represents the program's logical flow.

7 Each node on the tree represents a program operation that generates data or an instruction that is passed to the node above it. The node, in turn, performs another operation and passes that result to the node above it. When the parser is finished, the compiler has converted the entire program into a tree that represents the program's structure. The topmost node is called the **program**, and the nodes that pass results to it are **routine 1**, **routine 2**, and so forth, all the way down to very specific nodes at the bottom of the tree.



8 A third part of the compiler called the **code generator** works its way through the syntax tree, producing segments of machine code of each node. To each node on the tree, the generator matches a template of machine code to the operation assigned to that node.

9 The generator fills the blanks in each template with the values and variables found in each node. After each template is filled, it is added to the string of binary numbers that constitute the machine language and values of the program.



10 In a final state, an **optimizer** inspects the code produced by the code generator, looking for redundancies. The optimizer eliminates any operation that produces results identical to those of the preceding operation, making the program turned out by the compiler smaller and faster.