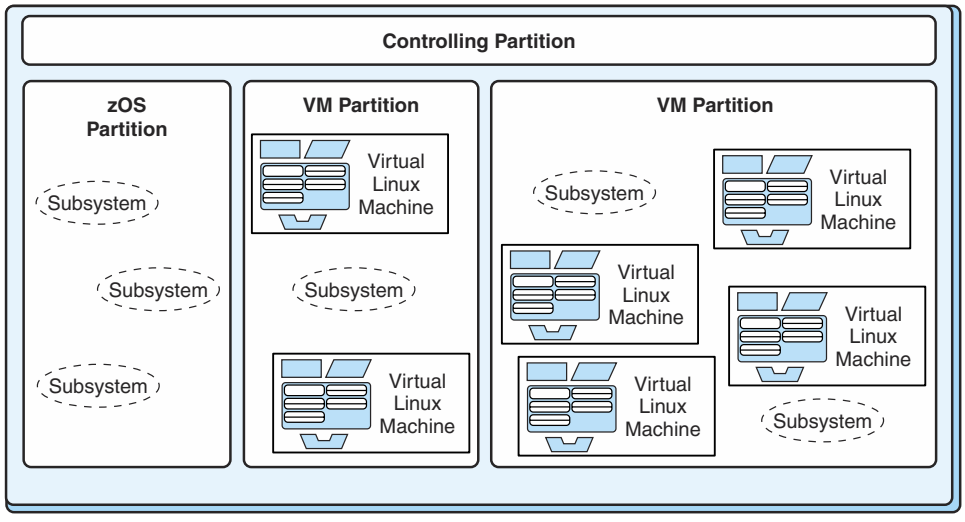

FIGURE 8.5 Linux Machines within Logical Partitions of an IBM zSeries Server

8.4 PROGRAMMING TOOLS

The operating system and its collection of applications provide an interface between the user who is writing the programs and the system that is running them. Other utilities, or programming tools, are necessary to carry out the more mechanical aspects of software creation. We discuss them in the sections below.

8.4.1 Assemblers and Assembly

In our layered system architecture, the level that sits directly on the Operating System layer is the Assembly Language layer. In Chapter 4, we presented a simple, hypothetical machine architecture, which we called MARIE. This architecture is so simple, in fact, that no real machine would ever use it. For one thing, the continual need to fetch operands from memory would make the system very slow. Real systems minimize memory fetches by providing a sufficient number of addressable on-chip registers. Furthermore, the instruction set architecture of any real system would be much richer than MARIE's is. Many microprocessors have over a thousand different instructions in their repertoire.

Although the machine that we presented is quite different from a real machine, the assembly process that we described is not. Virtually every assembler in use today passes twice through the source code. The first pass assembles as much code as it can, while building a symbol table; the second pass completes the binary instructions using address values retrieved from the symbol table built during the first pass.

The final output of most assemblers is a stream of *relocatable* binary instructions. Binary code is relocatable when the addresses of the operands are relative to the location where the operating system has loaded the program in memory, and the operating system is free to load this code wherever it wants. Take, for example, the following MARIE code from Table 4.5:

```
      Load x
      Add y
      Store z
      Halt
x, DEC 35
y, DEC -23
z, HEX 0000
```

The assembled code output could look similar to this:

```
1+004
3+005
2+006
7000
0023
FFE9
0000
```

The "+" sign in our example is not to be taken literally. It signals the program loader (component of the operating system) that the 004 in the first instruction is relative to the starting address of the program. Consider what happens if the loader happens to put the program in memory at address 250h. The image in memory would appear as shown in Table 8.1.

If the loader happened to think that memory at address 400h was a better place for the program, the memory image would look like Table 8.2.

In contrast to relocatable code, *absolute code* is executable binary code that must always be loaded at a particular location in memory. Nonrelocatable code is

Address	Memory Contents
250	1254
251	3255
252	2256
253	7000
254	0023
255	FFE9
256	0000

TABLE 8.1 Memory if Program Is Loaded Starting at Address 250h

Address	Memory Contents
400	1404
401	3405
402	2406
403	7000
404	0023
405	FFE9
406	0000

TABLE 8.2 Memory If Program Is Loaded Starting at 400h

used for specific purposes on some computer systems. Usually these applications involve explicit control of attached devices or manipulation of system software, in which particular software routines can always be found in clearly defined locations.

Of course, binary machine instructions cannot be provided with "+" signs to distinguish between relocatable and nonrelocatable code. The specific manner in which the distinction is made depends on the design of the operating system that will be running the code. One of the simplest ways to distinguish between the two is to use different file types (extensions) for this purpose. The MS-DOS operating system uses a .COM (a COMmand file) extension for nonrelocatable code and .EXE (an EXEcutable file) extension for relocatable code. COM files always load at address 100h. EXE files can load anywhere and they don't even have to occupy contiguous memory space. Relocatable code can also be distinguished from nonrelocatable code by prepending all executable binary code with prefix or preamble information that lets the loader know its options while it is reading the program file from disk.

When relocatable code is loaded into memory, special registers usually provide the base address for the program. All addresses in the program are then considered to be offsets from the base address stored in the register. In Table 8.1, where we showed the loader placing the code at address 0250h, a real system would simply store 0250 in the program base address register and use the program without modification, as in Table 8.3, where the address of each operand

Address	Memory Contents
250	1004
251	3005
252	2006
253	7000
254	0023
255	FFE9
256	0000

TABLE 8.3 Memory If Program Is Loaded at Address 250h Using Base Address Register

becomes an effective address after it has been augmented by the 0250 stored in the base address register.

Regardless of whether we have relocatable or nonrelocatable code, a program's instructions and data must be *bound* to actual physical addresses. The binding of instructions and data to memory addresses can happen at compile time, load time, or execution time. Absolute code is an example of *compile-time binding*, where the instruction and data references are bound to physical addresses when the program is compiled. Compile-time binding works only if the load memory location for a process image is known in advance. However, under compile-time binding, if the starting location of the process image changes, the code must be recompiled. If the load memory location for a process image is not known at compile time, relocatable code is generated, which can be bound either at load time or at run time. *Load-time binding* adds the starting address of the process image to each reference as the binary module is loaded into memory. However, the process image cannot be moved during execution, because the starting address for the process must remain the same. *Run-time binding* (or *execution-time binding*) delays binding until the process is actually running. This allows the process image to be moved from one memory location to another as it executes. Run-time binding requires special hardware support for *address mapping*, or translating from a logical process address to a physical address. A special base register stores the starting address of the program. This address is added to each reference generated by the CPU. If the process image is moved, the base register is updated to reflect the new starting address of the process. Additional virtual memory hardware is necessary to perform this translation quickly.

8.4.2 Link Editors

On most systems, program compiler output must pass through a *link editor* (or *linker*) before it can be executed on the target system. Linking is the process of matching the external symbols of a program with all exported symbols from other files, producing a single binary file with no unresolved external symbols. The principal job of a link editor, as shown in Figure 8.6, is to combine related program files into a unified loadable module. (The example in the figure uses file extensions characteristic of a DOS/Windows environment.) The constituent binary files can be entirely user-written, or they can be combined with standard system routines, depending on the needs of the application. Moreover, the binary linker input can be produced by any compiler. Among other things, this permits various sections of a program to be written in different languages, so part of a program could be written in C++, for ease of coding, and another part might be written in assembly language to speed up execution in a particularly slow section of the code.

As with assemblers, most link editors require two passes to produce a complete load module comprising all of the external input modules. During its first pass, the linker produces a global external symbol table containing the names of each of the external modules and their relative starting addresses with respect to the beginning of the total linked module. During the second pass, all references

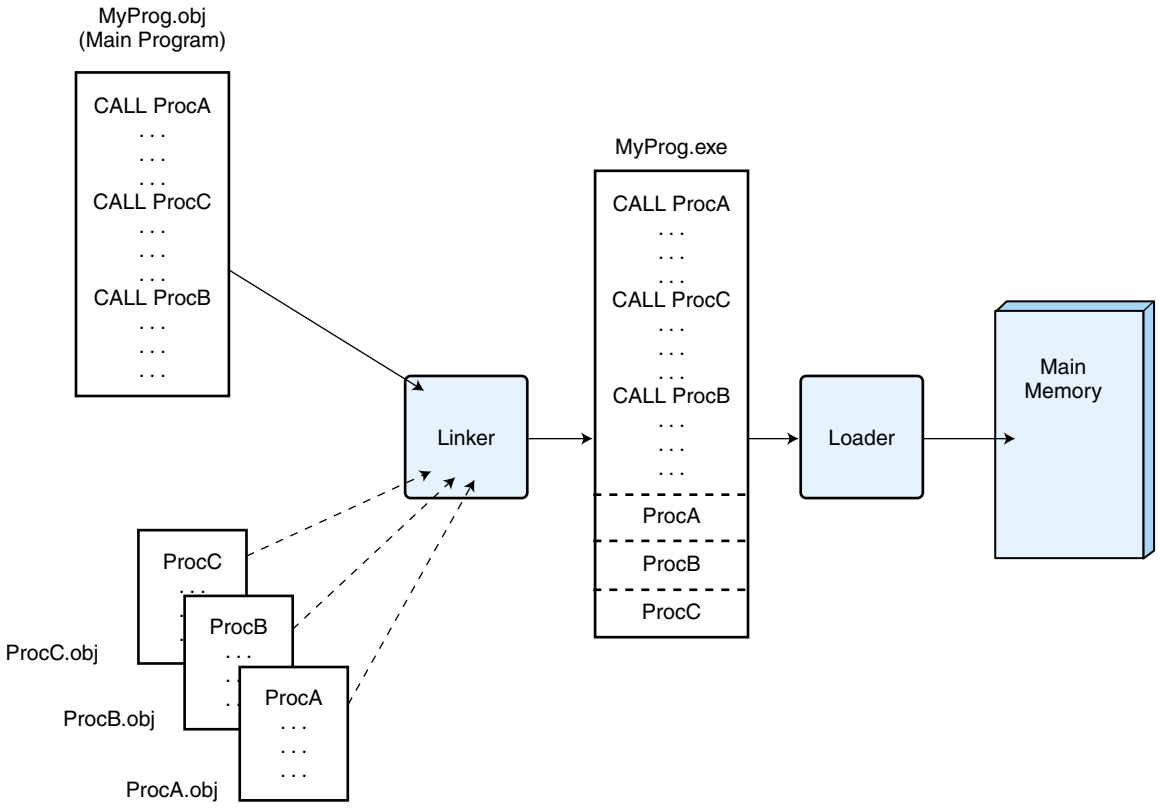

FIGURE 8.6 Linking and Loading Binary Modules

between the (formerly separate and external) modules are replaced with displacements for those modules from the symbol table. During the second pass of the linker, platform-dependent code can also be added to the combined module, producing a unified and loadable binary program file.

8.4.3 Dynamic Link Libraries

Some operating systems, notably Microsoft Windows, do not require link editing of all procedures used by a program before creating an executable module. With proper syntax in the source program, certain external modules can be linked at runtime. These external modules are called *dynamic link libraries* (*DLLs*), because the linking is done only when the program or module is first invoked. The dynamic linking process is shown schematically in Figure 8.7. As each procedure is loaded, its address is placed in a cross-reference table within the main program module.

This approach has many advantages. First, if an external module is used repeatedly by several programs, static linking would require that each of these programs include a copy of the modules' binary code. Clearly, it is a waste of disk

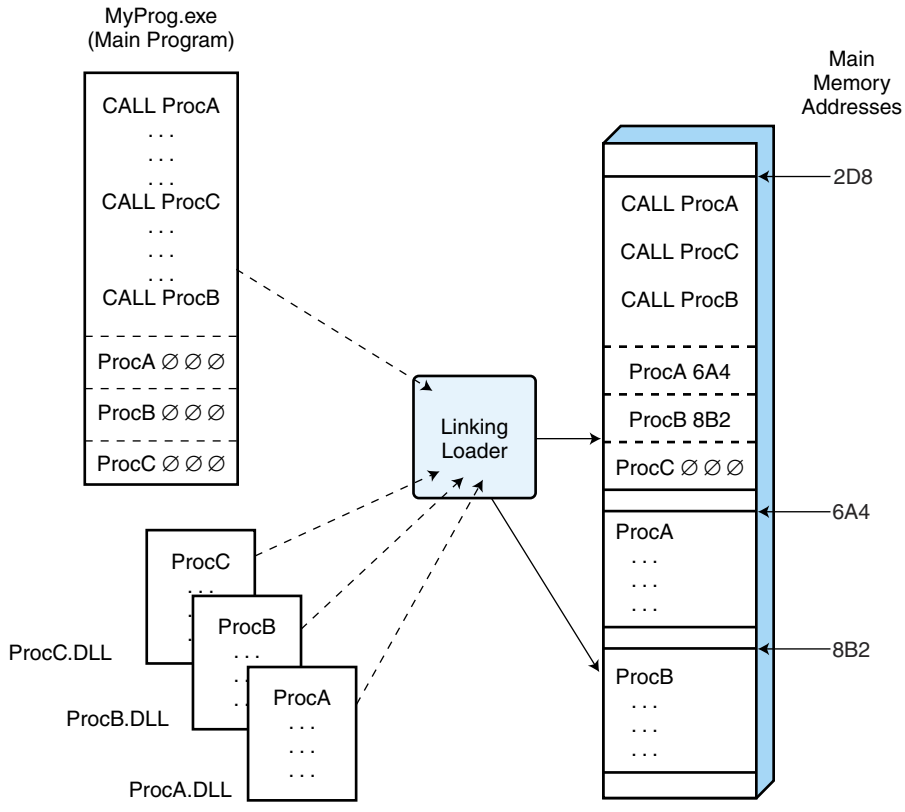

FIGURE 8.7 Dynamic Linking with Load Time Address Resolution

space to have multiple copies of the same code hanging around, so we save space by linking at runtime. The second advantage of dynamic linking is that if the code in one of the external modules changes, then each module that has been linked with it does not need to be relinked to preserve the integrity of the program. Moreover, keeping track of which modules employ which particular external modules can be difficult—perhaps impossible—for large systems. Thirdly, dynamic linking provides the means whereby third parties can create common libraries, the presence of which can be assumed by anyone writing programs for a particular system. In other words, if you are writing a program for a particular brand of operating system, you can take for granted that certain specific libraries will be available on every computer running that operating system. You need not concern yourself with the operating system's version number, or patch level, or anything else that is prone to frequent changes. As long as the library is never deleted, it can be used for dynamic linking.

Dynamic linking can take place either when a program is loaded or when an unlinked procedure is first called by a program while it is running. Dynamic linking at load time causes program startup delays. Instead of simply reading the program's

binary code from the disk and running it, the operating system not only loads the main program, but also loads the binaries for all modules that the program uses. The loader provides the load addresses of each module to the main program prior to the execution of the first program statement. The time lag between the moment the user invokes the program and when program execution actually commences may be unacceptable for some applications. On the other hand, run-time linking does not incur the startup penalties of load-time linking, because a module is linked only if it is called. This saves a considerable amount of work when relatively few of a program's modules are actually invoked. However, some users object to perceived erratic response times when a running program frequently halts to load library routines.

A less obvious problem with dynamic linking is that the programmer writing the module has no direct control over the contents of the dynamic link library routine. Hence, if the authors of the link library code decide to change its functionality, they can do so without the knowledge or consent of the people who use the library. In addition, as anyone who has written commercial programs can tell you, the slightest changes in these library routines can cause rippling effects throughout an entire system. These effects can be disruptive and very hard to track down to their source. Fortunately, such surprises are rare, so dynamic linking continues to be an approach favored for the distribution of commercial binary code across entire classes of operating systems.

8.4.4 Compilers

Assembly language programming can do many things that higher-level languages can't do. First and foremost, assembly language gives the programmer direct access to the underlying machine architecture. Programs used to control and/or communicate with peripheral devices are typically written in assembly due to the special instructions available in assembly that are customarily not available in higher-level languages. A programmer doesn't have to rely on operating system services to control a communications port, for example. Using assembly language, you can get the machine to do anything, even those things for which operating system services are not provided. In particular, programmers often use assembly language to take advantage of specialized hardware, because compilers for higher-level languages aren't designed to deal with uncommon or infrequently used devices. Also, well-written assembly code is blazingly fast. Each primitive instruction can be honed so that it produces the most timely and effective action upon the system.

These advantages, however, are not sufficiently compelling reasons to use assembly language for general application development. The fact remains that programming in assembly language is difficult and error-prone. It is even more difficult to maintain than it is to write, especially if the maintenance programmer is not the original author of the program. Most importantly, assembly languages are not portable to different machine architectures. For these reasons, most general-purpose system software contains very few, if any, assembly instructions. Assembly code is used only when it is absolutely necessary to do so.

Today, virtually all system and application programs use higher-level languages almost exclusively. Of course, "higher-level" is a relative term, subject to misunderstanding. One accepted taxonomy for programming languages starts by calling binary machine code the "first-generation" computer language (*1GL*). Programmers of this 1GL formerly entered program instructions directly into the machine using toggle switches on the system console! More "privileged" users punched binary instructions onto paper tape or cards. Programming productivity vaulted upward when the first assemblers were written in the early 1950s. These "second-generation" languages (*2GLs*) eliminated the errors introduced when instructions were translated to machine code by hand. The next productivity leap came with the introduction of compiled symbolic languages, or "third-generation" languages (*3GLs*), in the late 1950s. FORTRAN (*FORM*ula *TRAN*slation) was the first of these, released by John Backus and his IBM team in 1957. In the years since, a veritable alphabet soup of 3GLs has poured onto the programming community. Their names are sometimes snappy acronyms, such as COBOL, SNOBOL, and COOL. Sometimes they are named after people, as with Pascal and Ada. Not infrequently, 3GLs are called whatever their designers feel like calling them, as in the cases of C, C++, and Java.

Each "generation" of programming languages gets closer to how people think about problems and more distant from how machinery solves them. Some fourth- and fifth-generation languages are so easy to use that programming tasks formerly requiring a trained professional programmer can easily be done by end users, the key idea being that the user simply tells the computer what to do, not how to do it. The compiler figures out the rest. In making things simpler for the user, these latter-generation languages place substantial overhead on computer systems. Ultimately, all instructions must be pushed down through the language hierarchy, because the digital hardware that actually does the work can execute only binary instructions.

In Chapter 4, we pointed out that there is a one-to-one correspondence between assembly language statements and the binary code that the machine actually runs. In compiled languages, this is a one-to-many relationship. For example, allowing for variable storage definitions, the high-level language statement, `x = 3*y,` would require at least 12 program statements in MARIE's assembly language. The ratio of source code instructions to binary machine instructions becomes smaller in proportion to the sophistication of the source language. The "higher" the language, the more machine instructions each program line typically generates. This relationship is shown in the programming language hierarchy of Figure 8.8.

The science of compiler writing has continued to improve since the first compilers were written in the late 1950s. Through its achievements in compiler construction, the science of software engineering proved its ability to convert seemingly intractable problems into routine programming tasks. The intractability of the problem lies in bridging the semantic gap between statements that make sense to people and statements that make sense to machines.

Most compilers effect this transformation using a six-phase process, as shown in Figure 8.9. The first step in code compilation, called *lexical analysis*, aims to extract meaningful language primitives, or *tokens*, from a stream of textual source code. These tokens consist of reserved words particular to a language (e.g., `if`,

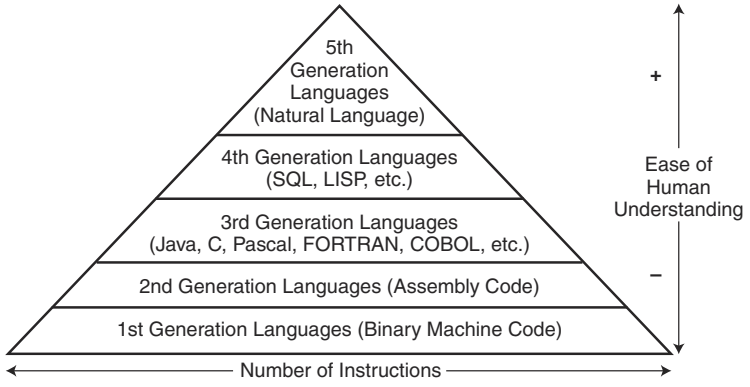

FIGURE 8.8 A Programming Language Hierarchy

`else`), Boolean and mathematical operators, literals (e.g., 12.27), and programmer-defined variables. While the lexical analyzer is creating the token stream, it is also building the framework for a symbol table. At this point, the symbol table most likely contains user-defined tokens (variables and procedure names), along with annotations as to their location and data type. Lexical errors occur when characters or constructs foreign to the language are discovered in the source code.

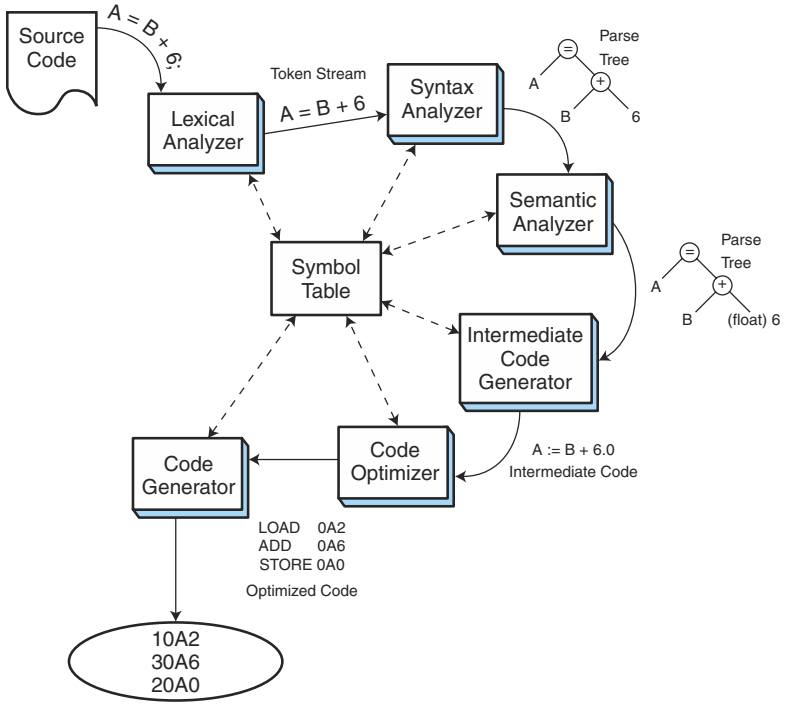

FIGURE 8.9 The Six Phases of Program Compilation

The programmer-defined variable `1DaysPay`, for example, would produce a lexical error in most languages because variable names typically cannot begin with a digit. If no lexical errors are found, the compiler proceeds to analyze the syntax of the token stream.

Syntax analysis, or *parsing*, of the token stream involves creation of a data structure called a *parse tree* or *syntax tree*. The inorder traversal of a parse tree usually gives the expression just parsed. Consider, for example, the following program statement:

```
monthPrincipal = payment - (outstandingBalance * interestRate)
```

One correct syntax tree for this statement is shown in Figure 8.10.

The parser checks the symbol table for the presence of programmer-defined variables that populate the tree. If the parser encounters a variable for which no description exists in the symbol table, it issues an error message. The parser also detects illegal constructions such as $A = B + C = D$. What the parser does not do, however, is check that the = or + operators are valid for the variables A, B, C, and D. The *semantic analyzer* does this in the next phase. It uses the parse tree as input and checks it for appropriate data types using information from the symbol table. The semantic analyzer also makes appropriate data type promotions, such as changing an integer to a floating-point value or variable, if such promotions are supported by the language rules.

After the compiler has completed its analysis functions, it begins its synthesis phase using the syntax tree from the semantic analysis phase. The first step in code synthesis is to create a pseudo-assembly code from the syntax tree. This code is often referred to as *three-address code*, because it supports statements such as `A = B + C`, which most assembly languages do not support. This intermediate code enables compilers to be portable to many different kinds of computers.

Once all of the tokenizing, tree-building, and semantic analyses are done, it becomes a relatively easy task to write a 3-address code translator that produces output for a number of different instruction sets. Most systems' ISAs use 2-address code, so the addressing mode differences have to be resolved during the translation process. (Recall that the MARIE instruction set is a 1-address archi-

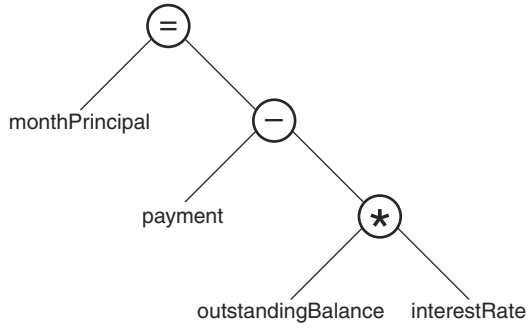

FIGURE 8.10 A Syntax Tree

tecture.) The final compiler phase, however, often does more than just translate intermediate code to assembly instructions. Good compilers make some attempt at code optimization, which can take into account different memory and register organizations as well as supply the most powerful instructions needed to carry out the task. Code optimization also involves removing unnecessary temporary variables, collapsing repeated expressions into single expressions, and flagging dead (unreachable) code.

After all of the instructions have been generated and optimizations have been made where possible, the compiler creates binary object code, suitable for linking and execution on the target system.

8.4.5 Interpreters

Like compiled languages, interpreted languages also have a one-to-many relationship between the source code statements and executable machine instructions. However, unlike compilers, which read the entire source code file before producing a binary stream, interpreters process one source statement at a time.

With so much work being done "on the fly," interpreters are typically much slower than compilers. At least five of the six steps required of compilers must also be carried out in interpreters, and these steps are carried out in "real time." This approach affords no opportunity for code optimization. Furthermore, error detection in interpreters is usually limited to language syntax and variable type checking. For example, very few interpreters detect possible illegal arithmetic operations before they happen or warn the programmer before exceeding the bounds of an array.

Some early interpreters, notably some BASIC interpreters, provided syntax checking within custom-designed editors. For instance, if a user were to type "`esle`" instead of "`else`" the editor would immediately issue a remark to that effect. Other interpreters allow use of general-purpose text editors, delaying syntax checking until execution time. The latter approach is particularly risky when used for business-critical application programs. If the application program happens to execute a branch of code that has not been checked for proper syntax, the program crashes, leaving the hapless user staring at an odd-looking system prompt, with his files perhaps only partially updated.

Despite the sluggish execution speed and delayed error checking, there are good reasons for using an interpreted language. Foremost among these is that interpreted languages allow source-level debugging, making them ideal for beginning programmers and end users. This is why, in 1964, two Dartmouth professors, John G. Kemeny and Thomas E. Kurtz, invented BASIC, the *Beginners All-purpose Symbolic Instruction Code*. At that time, students' first programming experiences involved punching FORTRAN instructions on 80-column cards. The cards were then run through a mainframe compiler, which often had a turnaround time measured in hours. Sometimes days would elapse before a clean compilation and execution could be achieved. In its dramatic departure from compiling statements in batch mode, BASIC allowed students to type program statements during an interactive terminal session. The BASIC interpreter, which was continually

running on the mainframe, gave students immediate feedback. They could quickly correct syntax and logic errors, thus creating a more positive and effective learning experience.

For these same reasons, BASIC was the language of choice on the earliest personal computer systems. Many first-time computer buyers were not experienced programmers, so they needed a language that would make it easy for them to learn programming on their own. BASIC was ideal for this purpose. Moreover, on a single-user, personal system, very few people cared that interpreted BASIC was much slower than a compiled language.

8.5 JAVA: ALL OF THE ABOVE

In the early 1990s, Dr. James Gosling and his team at Sun Microsystems set out to create a programming language that would run on any computing platform. The mantra was to create a "write once, run anywhere" computer language. In 1995, Sun released the first version of the Java programming language. Owing to its portability and open specifications, Java has become enormously popular. Java code is runnable on virtually all computer platforms, from the smallest handheld devices to the largest mainframes. The timing of Java's arrival couldn't have been better: It is a cross-platform language that was deployable at the inception of wide-scale Internet-based commerce, the perfect model of cross-platform computing. Although Java and some of its features were briefly introduced in Chapter 5, we now go into more detail.

If you have ever studied the Java programming language, you know that the output from the Java compiler is a binary *class* file. This class file is executable by a *Java Virtual Machine* (*JVM*), which resembles a real machine in many respects. It has private memory areas addressable only by processes running within the machine. It also has its own *bona fide* instruction set architecture. This ISA is stack-based to keep the machine simple and portable to practically any computing platform.

Of course, a Java Virtual Machine isn't a real machine. It is a layer of software that sits between the operating system and the application program: a binary class file. Class files include variables as well as the *methods* (procedures) for manipulating those variables.

Figure 8.11 illustrates how the JVM is a computing machine in miniature with its own memory and method area. Notice that the memory heap, method code, and "native method interface" areas are shared among all processes running within the machine.

The memory *heap* is main memory space that is allocated and deallocated as data structures are created and destroyed through thread execution. Java's deallocation of heap memory is (indelicately) referred to as *garbage collection*, which the JVM (instead of the operating system) does automatically. The Java *native method area* provides workspace for binary objects external to Java, such as compiled C++ or assembly language modules. The JVM *method area* contains the binary code required to run each application thread living in the JVM. This is where the class variable data structures and program statements required by the