1 What Are Programming Languages?

1.1 The wrong question

When one first encounters a new programming language, the first question is usually:

> What can this language “do”?

Implicitly we are comparing the new language with other languages. The answer is very simple: all languages can “do” exactly the same computations! Section 1.8 outlines the justification for this answer. If they can all do the same computations, surely there must be other reasons for the existence of hundreds of programming languages.

Let us start with some definitions:

> A *program* is a sequence of symbols that specifies a computation. A *programming language* is a set of rules that specify which sequences of symbols constitute a program, and what computation the program describes.

You may find it interesting that the definition does not mention the word computer! Programs and languages can be defined as purely formal mathematical objects. However, more people are interested in programs than in other mathematical objects such as groups, precisely because it is possible to use the program—the sequence of symbols—to control the execution of a computer. While we highly recommend the study of the theory of programming, this text will generally limit itself to the study of programs *as they are executed* on a computer.

These definitions are very general and should be interpreted as generally as possible. For example, sophisticated word processors usually have a facility that enables you to “capture” a sequence of key-presses and store them as a *macro* so that you can execute the entire sequence by pressing a single key. This is certainly a program because the sequence of key-presses specifies a computation and the software manual will clearly specify the macro language: how to initiate, terminate and name a macro definition.

To answer the question in the title of the chapter, we first go back to early digital computers, which were very much like the simple calculators used today by your grocer in that the computation that such computers perform is “wired-in” and cannot be changed.

The most significant early advance in computers was the realization (attributed to John von Neumann) that the specification of the computation, the program, can be *stored* in the computer just as easily as the data used in the computation. The *stored-program computer* thus becomes a general-purpose calculating machine and we can change the program just by changing a plugboard of wires, feeding a punched card, inserting a diskette, or connecting to a telephone line.

Since computers are binary machines that recognize only zeros and ones, storing programs in a computer is technically easy but practically inconvenient, since each instruction has to be written as binary digits (*bits*) that can be represented mechanically or electronically. One of the first software tools was the *symbolic assembler*. An assembler takes a program written in *assembly language*, which represents each instruction as a symbol, and translates the symbols into a binary representation suitable for execution on the computer. For example, the instruction:

```
load        R3,54
```

meaning "load register 3 with the data in memory location 54", is much more readable than the equivalent string of bits. Believe it or not, the term *automatic programming* originally referred to assemblers since they automatically selected the right bit sequence for each symbol. Familiar programming languages like C and Pascal are more sophisticated than assemblers because they "automatically" choose addresses and registers, and even "automatically" choose instruction sequences to specify loops and arithmetical expressions.

We are now ready to answer the question in the title of this chapter:

> A programming language is an abstraction mechanism. It enables a programmer to specify a computation abstractly, and to let a program (usually called an assembler, compiler or interpreter) implement the specification in the detailed form needed for execution on a computer.

We can also understand why there are hundreds of programming languages: two different classes of problems may demand different levels of abstraction, and different programmers have different ideas on how abstraction should be done. A C programmer is perfectly content to work at a level of abstraction that requires specification of computations using arrays and indices, while an author of a report prefers to "program" using a language consisting of the functions of a word-processor.

Levels of abstraction can be clearly seen in computer hardware. Originally, discrete components like transistors and resistors were wired directly to one another. Then standard plug-in modules were used, followed by small-scale integrated circuits. Today, entire computers can be constructed from a handful of chips each containing hundreds of thousands of components. No computer engineer would dare to design an "optimal" circuit from individual components if there exists a set of chips that can be adapted do to the same function.

There is a general truth that arises from the concept of abstraction:

> The higher the abstraction, the more detail is lost.

If you write a program in C, you lose the ability you had in assembly language to specify register allocation; if you write in Prolog, you lose the ability you had in C to specify arbitrary linked structures using pointers. There is a natural tension between striving for the concise, clear and reliable expression of a computation in a high-level abstraction, and wanting the flexibility of specifying the computation in detail. An abstraction can never be as exact or optimal as a low-level specification.

In this textbook you will study languages at three levels of abstraction: skipping assembly language, we start with "ordinary" programming languages like Fortran, C, Pascal and the Pascal-like constructs of Ada. Then in Part IV, we discuss languages like Ada and C++ that enable the programmer to construct higher-level abstractions from statements in ordinary languages. Finally, we will describe functional and logic programming languages that work at even higher levels of abstractions.

1.2 Imperative languages

Fortran

Fortran was the first programming language that significantly rose above the level of assembly language. It was developed in the 1950's by an IBM team led by John Backus, and was intended to provide an abstract way of specifying scientific computations. The opposition to Fortran was strong for reasons similar to those advanced against all subsequent proposals for higher-level abstractions, namely, most programmers believed that a compiler could not produce optimal code relative to hand-coded assembly language.

Like most first efforts in programming languages, Fortran was seriously flawed, both in details of the language design and more importantly in the lack of support for modern concepts of data and module structuring. As Backus himself said in retrospect:

> As far as we were aware, we simply made up the language as we went along. We did not regard language design as a difficult problem, merely a simple prelude to the real problem: designing a compiler which could produce efficient programs.[1]

Nevertheless, the advantages of the abstraction quickly won over most programmers: quicker and more reliable development, and less machine dependence since register and machine instructions are abstracted away. Because most early computing was on scientific problems, Fortran became the standard language in science and engineering, and is only now being replaced by other languages. Fortran has undergone extensive modernization (1966, 1977, 1990) to adapt it to the requirements of modern software development.

Cobol and PL/I

The Cobol language was developed in the 1950's for business data processing. The language was designed by a committee consisting of representatives of the US Department of Defense, computer manufacturers and commercial organizations such as insurance companies. Cobol was intended to be only a short-range solution until a better design could be created; instead, the language as

[1]R.L. Wexelblat, *History of Programming Languages*, Academic Press, 1981, page 30. Copyright by the ACM, Inc., reprinted by permission.

defined quickly became the most widespread language in its field (as Fortran has in science), and for a similar reason: the language provides a natural means of expressing computations that are typical in its field. Business data processing is characterized by the need to do relatively simple calculations on vast numbers of complex data records, and Cobol's data structuring capabilities far surpass those of algorithmic languages like Fortran or C.

IBM later created the language PL/I as a universal language having all the features of Fortran, Cobol and Algol. PL/I has replaced Fortran and Cobol on many IBM computers, but this very large language was never widely supported outside of IBM, especially on the mini- and micro-computers that are increasingly used in data processing organizations.

Algol and its descendants

Of the early programming languages, Algol has influenced language design more than any other. Originally designed by an international team for general and scientific applications, it never achieved widespread popularity compared to Fortran because of the support that Fortran received from most computer manufacturers. The first version of Algol was published in 1958; the revised version Algol 60 was extensively used in computer science research and implemented on many computers, especially in Europe. A third version of the language, Algol 68, has been influential among language theorists, though it was never widely implemented.

Two important languages that were derived from Algol are Jovial, used by the US Air Force for real-time systems, and Simula, one of the first simulation languages. But perhaps the most famous descendent of Algol is Pascal, developed in the late 1960's by Niklaus Wirth. The motivation for Pascal was to create a language that could be used to demonstrate ideas about type declarations and type checking. In later chapters, we will argue that these ideas are among the most significant concepts ever proposed in language design.

As a practical language, Pascal has one big advantage and one big disadvantage. The original Pascal compiler was itself written in Pascal,[2] and thus could easily be ported to any computer. The language spread quickly, especially to the mini- and micro-computers that were then being constructed. Unfortunately, the Pascal language is too small. The standard language has no facilities whatsoever for dividing a program into modules on separate files, and thus cannot be used for programs larger than several thousand lines. Practical compilers for Pascal support decomposition into modules, but there is no standard method so large programs are not portable.

Wirth immediately recognized that modules were an essential part of any practical language and developed the Modula language. Modula (now in version 3 which supports object-oriented programming) is a popular alternative to non-standard Pascal dialects.

C

C was developed by Dennis Ritchie of Bell Laboratories in the early 1970's as an implementation language for the UNIX operating system. Operating systems were traditionally written in assembly language because high-level languages were considered inefficient. C abstracts away

[2]We won't discuss here how this can be done!

the details of assembly language programming by offering structured control statements and data structures (arrays and records), while at the same time it retains all the flexibility of low-level programming in assembly language (pointers and bit-level operations).

Since UNIX was readily available to universities, and since it is written in a portable language rather than in raw assembly language, it quickly became the system of choice in academic and research institutions. When new computers and applications moved from these institutions to the commercial marketplace, they took UNIX and C with them.

C is designed to be close to assembly language so it is extremely flexible; the problem is that this flexibility makes it extremely easy to write programs with obscure bugs because unsafe constructs are not checked by the compiler as they would be in Pascal. C is a sharp tool when used expertly on small programs, but can cause serious trouble when used on large software systems developed by teams of varying ability. We will point out many of the dangers of constructs in C and show how to avoid major pitfalls.

The C language was standardized in 1989 by the American National Standards Institute (ANSI); essentially the same standard was adopted by the International Standards Organization (ISO) a year later. References to C in this book are to ANSI C[3] and not to earlier versions of the language.

C++

In the 1980's Bjarne Stroustrup, also from Bell Laboratories, used C as the basis of the C++ language, extending C to include support for object-oriented programming similar to that provided by the Simula language. In addition, C++ fixes many mistakes in C and should be used in preference to C, even on small programs where the object-oriented features may not be needed. C++ is the natural language to use when upgrading a system written in C.

Note that C++ is an evolving language and your reference manual or compiler may not be fully up-to-date. The discussion in this book follows *The Annotated C++ Reference Manual* by Ellis and Stroustrup (as reprinted in 1994) which is the basis for the standard now being considered.

Ada

In 1977 the United States Department of Defense decided to standardize on one programming language, mainly to save on training and on the cost of maintaining program development environments for each military system. After evaluating existing languages, they chose to ask for the development of a new language to be based on a good existing language such as Pascal. Eventually one proposal was chosen for a language which was named Ada, and a standard was adopted in 1983. Ada is unique in several aspects:

- Most languages (Fortran, C, Pascal) were proposed and developed by a single team, and only standardized after they were in widespread use. For compatibility, all the unintentional

[3]Technically, the ANSI standard was withdrawn with the appearance of the ISO standard, but colloquially the language is still known as ANSI C.

mistakes of the original teams were included in the standard. Ada was subjected to intense review and criticism before standardization.

- Most languages were initially implemented on a single computer and were heavily influenced by the quirks of that computer. Ada was designed to support writing portable programs.

- Ada extends the scope of programming languages by supporting error handling and concurrent programming which are traditionally left to (non-standard) operating system functions.

Despite technical excellence and advantages of early standardization, Ada has not achieved widespread popularity outside of military and other large-scale projects (such as commercial aviation and rail transportation). Ada has received a reputation as a difficult language. This is because the language supports many aspects of programming (concurrency, exception handling, portable numerics) that other languages (like C and Pascal) leave to the operating system, so there is simply more to learn. Also, good and inexpensive development environments for education were not initially available. Now with free compilers and good introductory textbooks available, Ada is increasingly used in the academic curriculum, even as a "first" language.

Ada 95

Exactly twelve years after the finalization of the first standard for the Ada language in 1983, a new standard for the Ada language has been published. The new version, called Ada 95, corrects some mistakes in the original version. But the main extension is support for true object-oriented programming including inheritance which was left out of Ada 83 because it was thought to be inefficient. In addition, the Ada 95 standard contains annexes that describe standard (but optional) extensions for real-time systems, distributed systems, information systems, numerics and secure systems.

In this text, the name "Ada" will be used unless the discussion is specific to one version: "Ada 83" or "Ada 95". Note that in the literature, Ada 95 was referred to as Ada 9X since the exact date of standardization was not known during the development.

1.3 Data-oriented languages

In the early days of programming several very influential languages were designed and implemented that had one characteristic in common: the languages each had a preferred data structure and an extensive set of operations for the preferred structure. These languages made it possible to create sophisticated programs that were otherwise difficult to write in languages such as Fortran that simply manipulated computer words. In the following subsections we will survey some of these languages.[4]

[4]You may wish to defer reading this section and return to it after studying Parts I and II.

Lisp

Lisp's basic data structure is the linked list. Originally designed for research in the theory of computation, much work on artificial intelligence was carried out in Lisp. The language was so important that computers were designed and built to be optimized for the execution of Lisp programs. One problem with the language was the proliferation of different dialects as the language was implemented on different machines. The Common Lisp language was later developed to enable programs to be ported from one computer to another. Currently, a popular dialect of Lisp is CLOS which supports object-oriented programming.

The three elementary operations of Lisp are: `car(L)` and `cdr(L)`[5] which extract the head and tail of a list `L`, respectively, and `cons(E, L)` which creates a new list from an element `E` and an existing list `L`. Using these operations, functions can be defined to process lists containing non-numeric data; such functions would be extremely difficult to program in Fortran.

We will not discuss Lisp further because many of its basic ideas have been carried over in modern functional programming languages such as ML which we will discuss in Chapter 16.

APL

The APL language evolved from a mathematical formalism used to describe calculations. The basic data structures are vectors and matrices, and operations work directly on such structures without loops. Thus programs are very concise compared with similar programs in ordinary languages. A difficulty with APL is that the language carried over a large set of mathematical symbols from the original formalism. This requires a special terminal and makes it difficult to experiment with APL without investing in costly hardware; modern graphical user interfaces which use software fonts have solved this problem that slowed acceptance of APL.

Given a vector:

$$\mathsf{V} = 1\ 5\ 10\ 15\ 20\ 25$$

APL operators can work directly on V without writing loops on the vector indices:

$$\begin{array}{lll} +/\mathsf{V} & = 76 & \text{Reduction of addition (add elements)} \\ \phi\ \mathsf{V} & = 25\ 20\ 15\ 10\ 5\ 1 & \text{Reverse the vector} \\ & & \\ 2\ 3\rho\ \mathsf{V} = & \begin{array}{rrr} 1 & 5 & 10 \\ 15 & 20 & 25 \end{array} & \text{Redimension V as } 2 \times 3 \text{ matrix} \end{array}$$

In addition, vector and matrix addition and multiplication can be done directly on such values.

Snobol, Icon

Early languages dealt almost exclusively with numbers. For work in fields such as natural language processing, Snobol (and its successor Icon) are ideally suited because their basic data structure is

[5]The strange notation is a historical artifact of the first computer on which Lisp was implemented.

the string. The basic operation in Snobol is matching a pattern to a string, and as a side-effect of the match, the string can be decomposed into substrings. In Icon, the basic operation is expression evaluation, where expressions include complex string operations.

An important predefined function in Icon is find(s1, s2) which searches for occurrences of the string s1 in the string s2. Unlike similar functions in C, find *generates* a list of all positions in s2 in which s1 occurs:

```
line := 0                       # Initialize line counter
while s := read( ) {            # Read until end of file
   every col := find("the", s) do
                                # Generate column positions
      write(line, " ", col)     # Write (line,col) of "the"
   line := line + 1
}
```

This program will write the line and column numbers of all occurrences of the string "the" in a file. If find does not find an occurrence, it will fail and the evaluation of the expression is terminated. The keyword every forces the repeated evaluation of the function as long as it is successful.

Icon expressions are not limited to strings which are sequences of characters; they are also defined on *csets*, which are sets of characters. Thus:

```
vowels := 'aeiou'
```

gives the variable vowel a value that is the set of characters shown. This can be used in functions like upto(vowels,s) which generates the sequence of locations of vowels in s, and many(vowels,s) which returns the longest initial sequence of vowels in s.

A more complex function is bal which is like upto except that it generates sequences of locations which are balanced with respect to bracketing characters:

```
bal('+-*/', '([', ')]', s)
```

This expression could be used in a compiler to generate balanced arithmetic sub-strings. Given the string "x+(y[u/v]−1)*z", the above expression will generate the indices corresponding to the sub-strings:

```
x
x + (y[u/v] - 1)
```

The first sub-string is balanced because it is terminated by "+" and there are no bracketing characters; the second is balanced because it is terminated by "*" and has square brackets correctly nested within parentheses.

Since an expression can fail, *backtracking* can be used to continue the search from earlier generators. The following program prints the occurrences of vowels except those that begin in column 1:

```
line := 0                   # Initialize line counter
while s := read( ) {        # Read until end of file
   every col := (upto(vowels, line) > 1) do
                            # Generate column positions
      write(line, " ", col) # Write (line,col) of vowel
   line := line + 1
}
```

The function find will generate an index which will then be tested by "¿". If the test fails (don't say: "if the result is false"), the program returns to the generator function upto to ask for a new index.

Icon is a practical language for programs that require complex string manipulation. Most of the explicit computation with indices is abstracted away, producing programs that are extremely concise relative to ordinary languages that were designed for numeric or systems programming. In addition, Icon is very interesting because of the built-in mechanism for generation and backtracking which offers a further level of control abstraction.

SETL

SETL's basic data structure is the set. Since sets are the most general mathematical structure from which all other mathematical structures are defined, SETL can be used to create generalized programs that are very abstract and thus very concise. The programs resemble logic programs (Chapter 17) in that mathematical descriptions can be directly executed. The notation used is that of set theory: $\{x \mid p(x)\}$ meaning the set of all x such that the logical formula $p(x)$ is true. For example, a mathematical specification of the set of prime numbers can be written:

$$\{n \mid \neg\exists m[(2 \leq m \leq n-1) \wedge (n \bmod m = 0)]\}$$

This formula is read: the set of numbers such that there does not exist a number m between 2 and $n-1$ that divides n without a remainder.

To print all primes in the range 2 to 100, we just translate the definition into a one-line SETL program:

```
print({n in {2..100} — not exists m in {2..n−1} — (n mod m) = 0});
```

What all these languages have in common is that they approach language design from a mathematical viewpoint—how can a well-understood theory be implemented, rather than from an engineering viewpoint—how can instructions be issued to the CPU and memory. In practice, these advanced languages are very useful for difficult programming tasks where it is important to concentrate on the problem and not on low-level details.

Data-oriented languages are somewhat less popular than they once were, partly because by using object-oriented techniques it is possible to embed such data-oriented operations into ordinary languages like C++ and Ada, but also because of competition from newer language concepts like functional and logic programming. Nevertheless, the languages are technically interesting and quite practical for the programming tasks for which they were designed. Students should make an effort to learn one or more of these languages, because they broaden one's vision of how a programming language can be structured.

1.4 Object-oriented languages

Object-oriented programming (OOP) is a method of structuring programs by identifying real-world or other objects, and then writing modules each of which contains all the data and executable statements needed to represent one *class* of objects. Within such a module, there is a clear distinction between the abstract properties of the class which are exported for use by other objects, and the implementation which is hidden so that it can be modified without affecting the rest of the system.

The first object-oriented programming language, Simula, was created in the 1960's by K. Nygaard and O.-J. Dahl for system simulation: each subsystem taking part in the simulation was programmed as an object. Since there can be more than one instance of each subsystem, a class can be written to describe each subsystem and objects of this class can then be allocated.

The Xerox Palo Alto Research Center popularized OOP with the Smalltalk[6] language. The same research also led to the windowing systems so popular today, and in fact, an important advantage of Smalltalk is that it is not just a language, but a complete programming environment. The technical achievement of Smalltalk was to show that a language can be defined in which classes and objects are the *only* structuring constructs, so there is no need to introduce these concepts into an "ordinary" language.

There is a technical aspect of these pioneering object-oriented languages that prevented wider acceptance of OOP: allocation, operation dispatching and type checking are dynamic (run-time) as opposed to static (compile-time). Without going into detail here (see the appropriate material in Chapters 8 and 14), the result is that there is a time and memory overhead to programs in these languages which can be prohibitive in many types of systems. In addition, static type checking (see Chapter 4) is now considered essential for developing reliable software. For these reasons, Ada 83 only implemented a subset of the language support required for OOP.

C++ showed that it was possible to implement the entire machinery of OOP in a manner that is consistent with *static* allocation and type-checking, and with fixed overhead for dispatching; the dynamic requirements of OOP are used only as needed. Ada 95 based its support for OOP on ideas similar to those found in C++.

However, it is not necessary to graft support for OOP onto existing languages to obtain these advantages. The Eiffel language is similar to Smalltalk in that the only structuring method is that of classes and objects, and it is similar to C++ and Ada 95 in that it is statically type-checked and the implementation of objects can be static or dynamic as needed. The simplicity of the language relative to the "hybrids", combined with full support for OOP, make Eiffel an excellent choice for

[6]Smalltalk is a trademark of Xerox Corporation.

a first programming language. Java[7] is both a programming language and a model for developing software for networks. The model is discussed in Section 3.11. The language is similar to C++ in syntax, but its semantics is quite different, because it is a "pure" OO language like Eiffel and requires strong type checking.

We will discuss language support for OOP in great detail, using C++, Java and Ada 95. In addition, a short description of Eiffel will show what a "pure" OOP language is like.

1.5 Non-imperative languages

All the languages we have discussed have one trait in common: the basic statement is the assignment statement which commands the computer to move data from one place to another. This is actually a relatively low level of abstraction compared to the level of the problems we want to solve with computers. Newer languages prefer to describe a problem and let the computer figure out how to solve it, rather than specifying in great detail how to move data around.

Modern software packages are really highly abstract programming languages. An application generator lets you *describe* a series of screen and database structures, and then automatically creates the low-level commands needed to implement the program. Similarly, spreadsheets, desktop publishing software, simulation packages and so on have extensive facilities for abstract programming. The disadvantage of this type of software is that they are usually limited in the type of application that can be easily programmed. It seems appropriate to called them *parameterized programs*, in the sense that by supplying descriptions as parameters, the package will configure itself to execute the program you need.

Another approach to abstract programming is to describe a computation using equations, functions, logical implications, or some similar formalism. Since mathematical formalisms are used, languages defined this way are true general-purpose programming languages, not limited to any particular application domain. The compiler does not really translate the program into machine code; rather it attempts to solve the mathematical problem, whose solution is considered to be the output of the program. Since indices, pointers, loops, etc. are abstracted away, these programs can be an order of magnitude shorter than ordinary programs. The main problem with descriptive programming is that computational tasks like I/O to a screen or disk do not fit in well with the concept, and the languages must be extended with ordinary programming constructs for these purposes.

We will discuss two non-imperative language formalisms: (1) functional programming (Chapter 16), which is based on the mathematical concept of pure functions, like sin and log that do not modify their environments, unlike so-called functions in an ordinary language like C which can have side-effects; (2) logic programming (Chapter 17), in which programs are expressed as formulas in mathematical logic, and the "compiler" attempts to infer logical consequences of these formulas in order to solve problems.

It should be obvious that programs in an abstract, non-imperative language cannot hope to be as efficient as hand-coded C programs. Non-imperative languages are to be preferred whenever a

[7]Java is a trademark of Sun Microsystems, Inc.

software system must search through large amounts of data, or solve problems whose solution cannot be precisely described. Examples are: language processing (translation, style checking), pattern matching (vision, genetics) and process optimization (scheduling). As implementation techniques improve and as it becomes ever more difficult to develop reliable software systems in ordinary languages, these languages will become more widespread.

Functional and logic programming languages are highly recommended as first programming languages, so that students learn from the start to work at higher levels of abstraction than they would if they were introduced to programming via Pascal or C.

1.6 Standardization

The importance of standardization must be emphasized. *If* a standard exists for a language and *if* compilers adhere to the standard, programs can be ported from one computer to another. If you are writing a software package that is to run on a wide range of computers, you should strictly adhere to a standard. Otherwise your maintenance task will be extremely complicated because you must keep track of dozens or hundreds of machine-specific items.

Standards exist (or are in preparation) for most languages discussed here. Unfortunately, the standards were proposed years after the languages became popular and must preserve machine-specific quirks of early implementations. The Ada language is an exception in that the standards (1983 and 1995) were created and evaluated at the same time as the language design and initial implementation. Furthermore, the standard is enforced so that compilers can be compared based on performance and cost, rather than on adherence to the standard. Compilers for other languages may have a mode that will warn you if you are using a non-standard construct. If such constructs must be used, they should be concentrated in a few well-documented modules.

1.7 Computer architecture

Since we are dealing with programming languages as they are used in practice, we include a short section on computer architecture so that a minimal set of terms can be agreed upon. A computer is composed of a *central processing unit* (CPU) and *memory* (Figure 1.1). Input/output devices

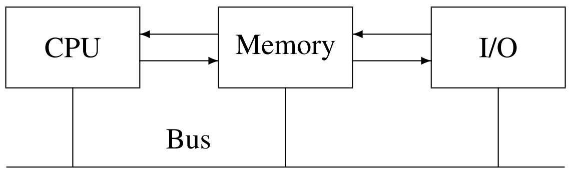

Figure 1.1: Computer architecture

can be considered to be a special case of memory. All components of a computer are normally connect together on a *bus*. Physically a bus is a set of connectors wired in parallel; logically a bus is a definition of the signals that enable the components to exchange data. As shown in the figure, modern computers may have more direct connections between the components to improve performance (by specializing the interface and by avoiding bottlenecks). From the point of view of the software, the only difference that need be considered is the rate at which data can be transferred between components.

The CPU contains a set of *registers* which are special memory locations in which computation is be done. The CPU can execute any one of a set of *instructions* which are stored in memory; the CPU maintains an *instruction pointer* which points to the location of the next instruction to be executed. Instructions are divided into the following classes:

- Memory access: Load the contents of a memory word into a register, and Store the contents of a register into a memory word.
- Arithmetic instructions such as add and subtract. These operations are performed on the contents of two registers (or sometimes between the content of a register and the content of a memory word). The result is left in a register. For example:

  ```
          add       R1,N
  ```

  adds the contents of the memory word N to the contents of the register R1 and leaves the result in the register.
- Compare and jump. The CPU can compare two values such as the contents of two registers; depending on the result (equal, greater than, etc.), the instruction pointer is changed to point to another instruction. For example:

  ```
          jump_eq   R1,L1
          ...
    L1:   ...
  ```

  causes the computation to continue with the instruction labeled L1 if the contents of R1 are zero; otherwise, the computation continues with the next instruction.

Many computers, called *Reduced Instruction Set Computers* (RISC), limit themselves to these elementary instructions; the rationale is that a CPU that only needs to execute a few simple instructions can be very fast. Other computers define more Complex Instructions (CISC) to simplify both assembly language programming and compiler construction. The debate between these two approaches is beyond the scope of this book; they have enough in common that the choice does not materially affect our discussion.

Memory is a set of locations that can be used to store data. Each memory location, called a memory *word*, has an *address*, and each word consists of a fixed number of bits, typically 16, 32, or 64 bits. The computer may be able to load and store 8 bit *bytes*, or double words of 64 bits.

It is important to know what kind of addressing modes can be used in an instruction. The simplest mode is immediate addressing which means that the operand is part of the instruction. The value of the operand may be the address of a variable and we will use the C notation in this case:

```
load     R3,#54        Load value 54 into R3
load     R2,&N         Load address of N into R2
```

Next we have the absolute addressing mode, which is usually used with the symbolic address of a variable:

```
        load    R3,54       Load contents of address 54
        load    R4,N        Load contents of variable N
```

Modern computers make extensive use of *index registers*. Index registers are not necessarily separate from registers used for computation; what is important is that the register has the property that the address of an operand of an instruction can be obtained from the content of the register. For example:

```
        load    R3,54(R2)   Load contents of addr(R2)+54
        load    R4,(R1)     Load contents of addr(R1)+0
```

where the first instruction means "load into register R3 the contents of the memory word whose address is obtained by adding 54 to the contents of the (index) register R2"; the second instruction is a special case which just uses the contents of register R1 as the address of a memory word whose contents are loaded into R4. Index registers are essential to efficient implementation of loops and arrays.

Cache and virtual memory

One of the hardest problems confronting computer architects is matching the performance of the CPU to the performance of the memory. CPU processing speeds are so fast compared with memory access times, that the memory cannot supply data fast enough for the CPU to be kept continuously working. There are two reasons for this: (1) there are only a few CPU's (usually one) in a computer, and the fastest, most expensive technology can be used, but memory is installed in ever increasing amounts and must use less-expensive technology; (2) the speeds are so fast that a limiting factor is the speed at which electricity flows in the wires between the CPU and the memory.

The solution is to use a hierarchy of memories as shown in Figure 1.2. The idea is to store unlim-

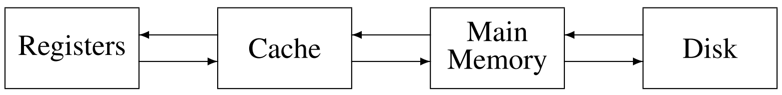

Figure 1.2: Cache and virtual memory

ited amounts of program instructions and data in relatively slow (and inexpensive) memory, and to load relevant portions of the instructions and data into smaller amounts of fast (and expensive) memory. When the slow memory is disk and the fast memory is ordinary RAM (Random Access Memory), the concept is called *virtual memory* or *paged memory*. When the slow memory is RAM and the fast memory is RAM implemented in a faster technology, the concept is called *cache memory*.

A discussion of these concepts is beyond the scope of this book, but the programmer must be aware of the potential effect of cache or virtual memory on a program, even though the maintenance of these memories is done by the computer hardware or operating system software, and is totally transparent to the programmer. Instructions and data are moved between slower and faster memory in blocks, not in single words. This means that a sequence of instructions without jumps, and a sequence of consecutive data accesses (such as indexing through an array) are likely to be much more efficient than jumps and random accesses, which require the computer to move different blocks between levels of the memory hierarchy. If you are attempting to improve the efficiency of a program, you should resist the temptation to write portions in lower-level languages or assembly language; instead, attempt to rearrange the computation taking into consideration the influence of the cache and virtual memory. The rearrangement of statements in a high-level language does not affect the portability of the program, though of course the efficiency improvement may not carry over to another computer with a different architecture.

1.8 * Computability

In the 1930's, even before digital computers were invented, logicians studied abstract concepts of computation. Alan Turing and Alonzo Church each proposed extremely simple models of computation (called *Turing machines* and *Lambda calculus*, respectively) and then advanced the following claim (known as the *Church-Turing Thesis*):

> Any effective computation can be done in one of these models.

Turing machines are extremely simple; expressed in the syntax of C, there are two data declarations:

```
char tape[...];
int current = 0;
```

where the tape is potentially infinite.[8] A program consists of any number of statements of the form:

```
L17:  if (tape[current] == 'g') {
          tape[current++] = 'j';
          goto L43;
      }
```

Executing a statement of a Turing machine is done in four stages:

- Read and examine the current character[9] on the current cell of the tape.
- Replace the character with another character (optional).
- Increment or decrement the pointer to the current cell.

[8]This does not mean that an infinite number of memory words need be allocated, only that we can always plug in more memory if needed.

[9]In fact, it is sufficient to use just two characters, blank and non-blank.

- Jump to any other statement.

According to the Church-Turing Thesis, any computation that you can effectively describe can be programmed on this primitive machine. The evidence for the Thesis rests on two foundations:

- Researchers have proposed dozens of models of computation and all of them have been proven to be equivalent to Turing machines.
- No one has ever described a computation that cannot be computed by a Turing machine.

Since a Turing machine can easily be simulated in any programming language, all programming languages "do" the same thing.

1.9 Exercises

1. Describe how to implement a compiler for a language in the same language ("bootstrapping").
2. Invent a syntax for an APL-like matrix-based language that uses ordinary characters.
3. Write a list of interesting operations on strings and compare your list with the predefined operations of Snobol and Icon.
4. Write a list of interesting operations on sets and compare your list with the predefined operations of SETL.
5. Simulate a (universal) Turing machine in several programming languages.