get program—preferably a fast one—whose semantics match those of the source.

Chapters 3, 6, 7, 8, and 9 form the core of the rest of this book. They cover fundamental issues of language design, both from the point of view of the programmer and from the point of view of the language implementor. To support the discussion of implementations, Chapters 2 and 4 describe compiler front ends in more detail than has been possible in this introduction. Chapter 5 provides an overview of assembly-level architecture. Chapters 14 and 15 discuss compiler back ends, including assemblers and linkers. Additional language paradigms are covered in Chapters 10 through 13. Appendix A lists the principal programming languages mentioned in the text, together with a genealogical chart and bibliographic references. Appendix B contains a list of "Design and Implementation" sidebars. Appendix C contains a list of numbered examples.

1.8 Exercises

1.1 Errors in a computer program can be classified according to when they are detected and, if they are detected at compile time, what part of the compiler detects them. Using your favorite imperative language, give an example of each of the following.
(a) A lexical error, detected by the scanner
(b) A syntax error, detected by the parser
(c) A static semantic error, detected by semantic analysis
(d) A dynamic semantic error, detected by code generated by the compiler
(e) An error that the compiler can neither catch nor easily generate code to catch (this should be a violation of the language definition, not just a program bug)

1.2 Algol family languages are typically compiled, while Lisp family languages, which many issues cannot be settled until run time, are typically interpreted. Is interpretation simply what one "has to do" when compilation is infeasible, or are there actually some advantages to interpreting a language, even when a compiler is available?

1.3 The gcd program of Example 1.17 might also be written

```python
program gcd(input, output);
var i, j : integer;
begin
  read(i, j);
  while i <> j do
    if i > j then i := i mod j
    else j := j mod i;
  writeln(i)
end.
```
Does this program compute the same result? If not, can you fix it? Under what circumstances would you expect one or the other to be faster?

1.4 In your local implementation of C, what is the limit on the size of integers? What happens in the event of arithmetic overflow? What are the implications of size limits on the portability of programs from one machine/compiler to another? How do the answers to these questions differ for Java? For Ada? For Pascal? For Scheme? (You may need to find a manual.)

1.5 The Unix make utility allows the programmer to specify dependencies among the separately compiled pieces of a program. If file A depends on file B and file B is modified, make deduces that A must be recompiled, in case any of the changes to B would affect the code produced for A. How accurate is this sort of dependence management? Under what circumstances will it lead to unnecessary work? Under what circumstances will it fail to recompile something that needs to be recompiled?

1.6 Why is it difficult to tell whether a program is correct? How do you go about finding bugs in your code? What kinds of bugs are revealed by testing? What kinds of bugs are not? (For more formal notions of program correctness, see the bibliographic notes at the end of Chapter 4.)

1.9 Explorations

1.7 (a) What was the first programming language you learned? If you chose it, why did you do so? If it was chosen for you by others, why do you think they chose it? What parts of the language did you find the most difficult to learn?

(b) For the language with which you are most familiar (this may or may not be the first one you learned), list three things you wish had been differently designed. Why do you think they were designed the way they were? How would you fix them if you had the chance to do it over? Would there be any negative consequences—for example, in terms of compiler complexity or program execution speed?

1.8 Get together with a classmate whose principal programming experience is with a language in a different category of Figure 1.1. (If your experience is mostly in C, for example, you might search out someone with experience in Lisp.) Compare notes. What are the easiest and most difficult aspects of programming, in each of your experiences? Pick some simple problem (e.g., sorting, or identification of connected components in a graph) and solve it using each of your favorite languages. Which solution is more elegant (do the two of you agree)? Which is faster? Why?
Both scanners and parsers can be built by hand if an automatic tool is not available. Hand-built scanners are simple enough to be relatively common. Hand-built parsers are generally limited to top-down recursive descent, and are generally used only for comparatively simple languages (e.g., Pascal but not Ada). Automatic generation of the scanner and parser has the advantage of increased reliability, reduced development time, and easy modification and enhancement.

Various features of language design can have a major impact on the complexity of syntax analysis. In many cases, features that make it difficult for a compiler to scan or parse also make it difficult for a human being to write correct, maintainable code. Examples include the lexical structure of Fortran and the if... then... else statement of languages like Pascal. This interplay among language design, implementation, and use will be a recurring theme throughout the remainder of the book.

2.6 Exercises

2.1 Write regular expressions to capture

(a) Strings in C. These are delimited by double quotes ("), and may not contain newline characters. They may contain double quote or backslash characters if and only if those characters are “escaped” by a preceding backslash. You may find it helpful to introduce shorthand notation to represent any character that is not a member of a small specified set.

(b) Comments in Pascal. These are delimited by (* and *), as shown in Figure 2.6, or by { and }.

(c) Floating-point constants in Ada. These are the same as in Pascal (see the definition of unsigned number in Example 2.2 [page 41]), except that (1) an underscore is permitted between digits, and (2) an alternative numeric base may be specified by surrounding the non-exponent part of the number with pound signs, preceded by a base in decimal (e.g., 16#6.af#0e+2). In this latter case, the letters a..f (both upper- and lowercase) are permitted as digits. Use of these letters in an inappropriate (e.g., decimal) number is an error but need not be caught by the scanner.

(d) Inexact constants in Scheme. Scheme allows real numbers to be explicitly inexact (imprecise). A programmer who wants to express all constants using the same number of characters can use sharp signs (#) in place of any lower-significance digits whose values are not known. A base-ten constant without exponent consists of one or more digits followed by zero of more sharp signs. An optional decimal point can be placed at the beginning, the end, or anywhere in between. (For the record, numbers in Scheme are actually a good bit more complicated than this. For the