**Notes**

Freedman, R. S. (1982). *Programming concepts with the Ada language*. New York: Petrocelli Books.

* “Ada is a computer programming language that was designed to specifications by a community of programming language users. Ada was not specified by a computer vendor. The design specifications of Ada evolved over a six year period beginning in 1974.” (3)
* “Their goal was to design a language that would recognize the importance of reliability and maintenance, the ‘human side’ of software production, and the need for efficiency.” (3-4)
* “This user community was backed financially and politically by the United States Department of Defense.” (4)
* “At first glance [procedures and procedure calls seem] very similar to Pascal or PL/I. [They] should be. Ada was a language *engineering* effort, not an effort in language *science*. All ideas in Ada are used in one form or another in existing languages.” (5)
* “As you must have already guessed, comments are indicated by anything to the right of two hyphens.” (6)
* “However, we can observe at this point that every Ada program has a ‘declaration part,’ (‘visible part,’ ‘specification’) and an ‘action part’ (‘body part,’ ‘implementation’).” (6)
* “Ada can be used as a program design language (PDL) to describe software components. Ada can be used as a system design language (SDL) (or architecture language) to describe systems and their interfaces.” (6)
* “All identifiers in Ada must be defined (declared, specified) *someplace*. Ada, like other languages, reserves a special set of identifiers to indicate a particular meaning. These Ada *keywords* are ‘reserved words’; they cannot be arbitrarily redefined by a programmer. Keywords in Ada will be written in lower-case letters. This follows the practice of the LRM. Note that Ada does not distinguish between lower-case or upper-case letters.” (7)
* Keywords by category:
  + control structures
    - if
      * if, then, elseif, else, end if
      * case, is, when, when others, end case (8)
    - loop
      * for, in, loop, end loop, reverse, while, exit, when (9)
    - goto
      * goto (9)
    - block
      * declare, begin, end (9)
    - null
      * null (10)
  + arithmetic-logic structures
    - logic
      * and, or, xor, not
    - arithmetic
      * mod, rem (11)
  + data structure
    - array
      * array, of (11)
    - constant
      * constant (11)
    - type
      * type, is, subtype (12)
    - record
      * record, end record
    - numeric type
      * “for integer, fixed point, and floating point object specifications”
      * range, delta, digits (13)
    - access type
      * access (13)
    - private type
      * private, limited private (14)
  + subprogram structures
    - functions, procedures
      * procedure, in, out
      * function
      * return (14)
    - generic
      * generic (14)
  + package structures
    - package specification
      * package (15)
    - package body
      * package body (15)
    - private part
      * private (16)
    - separate
      * separate (16)
    - with use
      * with, use (17)
    - renaming
      * renames (17)
  + task structures
    - task specification and implementation
      * task, task body (17)
    - entry
      * entry (17)
    - rendezvous
      * select, accept, do, delay (18)
    - task stopping
      * terminate, abort (18)
  + exception structures
    - exception, raise (19)
  + implementation dependent structures
    - for, use, at (19)
  + pragma structures
    - pragma (19)
* “Subprograms are program units that perform some action. They consist of functions (a subprogram that *returns* a value) and procedures (a subprogram that locally performs some actions). A procedure may be given some parameters from another program unit. These parameters may be used only for input (and not modified), may be used only for output (where the parameter is assigned a value obtained by the local execution of the procedure), or may be used for both input and output. Ada calls these parameters **in**, **out**, and **in out** parameters respectively.” (23)
* “Note that the type of the parameter is explicitly indicated in the subprogram specification. For functions, the *type* of the value *returned* is also explicitly indicated. Also note that if **in**, **out**, or **in out** remains unspecified, **in** is assumed (as for the function).” (24)
* “The declaration (‘specification’) of a subprogram may be physically separate from its implementation (‘body’), where the actual programming actions are described. The separation between specification and body is an important feature of Ada programming structures: bodies may be *separately compiled*. This is an aid to program design.” (24)
* “We may also ‘group’ programming structures and data structures that are logically related into a ‘super’ programming structure. This *package* of data structures and programming structures enforced a certain component or modular discipline on a programmer. Ada packages (like subprograms) have a specification and body. A package *specification* typically contains type and subprogram declarations. The corresponding *body* would then contain the bodies of the previously declared subprograms.” (24)
* “The body [of the package] defines the actions; both body and specification can be separately compiled.” (25)
* “Packages may also contain statements that perform some actions; these statements are executed after the ‘begin’ in a package body.” (25)
* “Ada provides another way of packaging data and operations in the form of ‘templates’ of program units. This is accomplished by a generic subprogram specification (for generic functions and generic procedures) or a generic package specification. Generic program units act as a ‘mold’ from which particular program units (‘instances’) can be created.” (27)
* “Once our programmer constructs the mold, he can create as many instances of this mold as he likes.” (27-28)
* “How is a generic procedure used? First of all, the *instance* of the generic procedure is used (‘instantiation’)—the *instance* defines the *actions*. The generic procedure itself is merely a *specification* of a mold.” (28)
* “The phenomenon of subprogram names (and also names of other Ada entities) appearing the same, but being implemented differently is called ‘overloading.’” (29)
  + Ada can distinguish between overloaded procedures “by noting the *type* of the arguments.” (29)
* Generic procedures can occur with “one *generic formal parameter*.” (29)
* “It is also possible to use functions and procedures as well as types for generic formal parameters.” (29)
* “Also note that these functions (+, -, \* and some others) are overloaded in Ada as they are in most other languages (e.g. FORTRAN).” (31)
* “The phrase ‘**range** < >’ is an indication to Ada of an INTEGER type, where the range (at this point in time) is left unspecified. The box notation < > is used frequently in Ada to denote something that will be supplied later (in our case, the range of an INTEGER type).” (32)
* “All functions and procedures in Ada may be recursive.” (32)
* “A package is a collection of data types *and* allowable operations between objects defined from these types. . . . From the specification, a user sees *what* this package is (a collection of types and functions) and *how* to use it. The user of this package does not have to be supplied with the actual source code for the package body; all information regarding the *use* of the package is in the specification.” (34)
* “The ‘**with**’ clause allows the utilization of a separately compiled program unit. . . . In the Ada environment, these separately compiled units belong to a ‘Program Library.’” (35)
* “We utilize a ‘selected component notation’ (similar to the notations used with record data structures) when accessing a type, function or subprogram in another package. The ‘visibility’ of the package is made available with the ‘**with**’ clause.” (35)
* “For complicated situations (where packages can be nested within other packages), the selected components can generate long and confusing expressions. The easiest way to avoid this is by using the ‘**use**’ clause.” (35)
* “The ‘**use**’ clause makes all identifiers in the package name (that is referenced) ‘visible’ to all other program units. Use clauses can be used with any collection of Ada programs (they don’t have to be separately compiled). In this case, a use clause (without a with clause) appears in a declaration part of a program unit. In any case, if there is any ambiguity, selected component notation must be used. If one really dislikes the selected component notation, names of entities can still be ‘renamed’ (without using the use clause).” (36)
* “Renaming is a ‘shorthand’ for identifiers. It does not hide the old name. A type can be renamed by declaring a subtype with no constraints.” (36)
* “The with clause shows the concept of *using* separately compiled program units.” (37)
* “Separate compilation, like renaming, is an aid to readability and to programming methodology—especially for very large software systems.” (37-38)
* “Packages may also be generic. A generic package specification tells the Ada compiler that the corresponding package body is a mold that can be used for the creation (instantiation) of other packages.” (38)
* “Why is there such a concern for generic numeric types of arbitrary ranges and precisions? One reason is so mathematical libraries of *arbitrary precision* can be constructed out of generic packages.” (40)
* “A type is a collection of values with a *predefined* (or *inherited*) set of *operations* for those values.” (47)
* “In Ada, ‘objects’ are program entities with a certain value. The range of these values is specified by a type or subtype. All objects must be declared. Types that are not predefined in the language must also be declared.” (47)
* “Ada uses the ellipsis notation—the two dots (‘..’)—to denote a *range* of object values in object declarations and in loops. It is a very convenient mechanism for expressing this idea.” (48)
* “Enumeration types and integer types are referred to as ‘discrete’ types, because they can be used for indexing and for iteration over loops. Discrete types and ‘real’ types (fixed point types and floating point types) are referred to as scalar types. All scalar types have the property of being *ordered* (first, second, third, and so on), so that they can be defined over a *range* of values. A numeric type is an integer type or a real type.” (48)
* Enumeration types have the following predefined operations:
  + := (assignment)
  + = (equality test)
  + /= (inequality test)
  + in, not in (test for membership in a range or subtype)
  + <, <=, >, >= (tests for ordering) (50)
* “The predefined operations for the enumeration types . . . can be inhibited if we declare these types to be a private or limited private type. Private types only allow assignment, equality test and inequality test. Limited private types do not even allow these operations.” (52)
* “New types can be derived from old types in two ways:

1. By constructing a ‘new’ type from the old type that incorporates a *new* range of values and a *new* set of operations.

2. By constructing a ‘subtype’ by *constraining* the set of values (but *keeping* the operations).” (52)

* “Two other examples of enumeration types are the predefined type BOOLEAN and the predefined type CHARACTER.” (54)
* “Functions (operators) that return BOOLEAN values include the keywords **not**, **and**, **or**, **xor** (exclusive or). These functions are what they say. There are also ‘short circuit’ functions ‘**and then**’ and ‘**or else**.’” (55)
* “The values of the predefined type CHARACTER are from the 128 ASCII character set.” (55)
* “An object of a given type can be further specified to be a *constant*. The initial value of a constant cannot be modified.” (55)
* “Numeric types consist of integer and real types. We have already seen several examples of integer types. Type INTEGER is predefined. Objects of this type range from the smallest number in the implementation (SYSTEM.MIN\_INT) to the largest number in the implementation (SYSTEM.MAX\_INT). A predefined subtype of integer is subtype NATURAL.” (56)
* The type INTEGER and its derived objects have the following predefined operators:
  + := (assignment)
  + = (equality test)
  + /= (inequality test)
  + in, not in (membership tests)
  + <, >, <=, >=
  + +
  + –
  + \*
  + /
  + rem (remainder)
  + mod (modulus)
  + ABS (absolute value—note not a keyword)
  + \*\* (exponentiation for non-negative exponents) (57)
* “The predefined Ada language environment also provides a type SHORT\_INTEGER and a type LONG\_INTEGER (in addition to type INTEGER) which are both implementation defined.” (58)
* “The predefined floating point types denote a range of real numbers expressed with a certain number of (decimal) digits.” (59)
* Floating point types have the following predefined operations:
  + := (assignment)
  + = (equality test)
  + /= (inequality test)
  + in, not in (membership tests)
  + <, >, <=, >=
  + +
  + –
  + \*
  + /
  + ABS (absolute value)
  + \*\* (exponentiation for *integer* exponents) (60)
* “The predefined Ada language provides type FLOAT, type SHORT\_FLOAT and type LONG\_FLOAT (which are all implementation defined).” (60)
* “The predefined fixed point types denote a range of real numbers expressed within a certain tolerance, called a ‘delta.’” (61)
* Fixed point types have the following predefined operations:
  + := (assignment)
  + = (equality test)
  + /= (inequality test)
  + in, not in (membership tests)
  + <, >, <=, >=
  + +
  + –
  + \*
  + /
  + ABS (absolute value) (61-62)
* “Fixed point multiplication and division both have the following property: When any two fixed point numbers are multiplied, the type of the product must be specified (as in a conversion).” (62)
* “The predefined Ada language environment does *not* provide any predefined fixed point types.” (62)
* “All numberic [*sic*] types have the following properties:

1. Objects of a numeric type can be converted to any other numeric type. This can result in rounding: The conversion is always within the accuracy of the specified type. Conversions can go either way (but may result in an error condition if the result does not lie in the constraint range).

2. All numbers can be represented in any base, from 2 to 16. This is accomplished with the number sign notation (#). The base and the exponent are in decimal notation and the ‘mantissa’ is represented between two number signs.” (62-63)

* “Composite types consist of arrays and records. Objects of these types have ‘components.’ These components are all of the *same* type for arrays and (could be) of different types for records.” (63)
* “Array types are distinguished by the keyword ‘array.’” (64)
* “We refer to the components of A [an array] as A(1), A(2), A(3)[, and so on]. The *index* of objects of type A ranges over the numbers 1, 2, 3[, and so on].” (64)
* Arrays may be “constrained” by explicitly specifying an index range. Arrays may be “unconstrained” by not declaring an index range. (64)
* “The unconstrained array acts as a ‘mold’ to make other arrays.” (65)
* “An index of an array can be of any discrete type (an integer type or an enumeration type). The type of the components of an array can be of any type.” (65)
* “Ada provides the predefined type STRING as an array of CHARACTER.” (66)
* “Strings are written with the double-quote character (different from the apostrophe that is used for writing a character). However, they can also explicitly be written in component notation.” (66)
* “The bounds of an array object must be known when an array object is defined. However, a *constant* array object declaration can *omit* the index constraint; in this case, the bounds are those of the initial value.” (67)
* Array types have the following predefined operations:
  + := (assignment)
  + = (equality test for matching index constraints)
  + /= (inequality test for matching index constraints)
  + in, not in (membership tests)
  + <, >, <=, >= (order tests defined only for one dimensional arrays whose components have a discrete type (like STRING). The array index constraints must match. The order corresponds to ordinary lexicographic ordering.)
  + & (catenation—defined only for one-dimensional arrays.)
  + and, or, xor, not (defined only for one-dimensional arrays of BOOLEAN components with matching index constraints.) (67-68)
* “A record type consists of components which must be explicitly named.” (68)
* “We also see that record components are denoted with the ‘selected component’ notation (that we have already seen in connection with packages). The components in a record component can be of any type (they can even be other records).” (68)
* “A record type can have ‘discriminants’ and ‘variant parts.’ Discriminants are useful for constructing record objects with components having an arbitrary index constraint (consequently a discriminant must belong to a discrete type).” (69)
* “Note that the type of the discriminant must be explicitly declared.” (69-70)
* “A discriminant may also be initialized in the definition. When objects are then declared, we can use a ‘default’ value for an index.” (70)
* “Variant parts of a record type are useful for constructing records that may ‘vary’ from a certain desired structure.” (71)
* Record types have the following predefined operations:
  + := (assignment for matching components)
  + = (equality test for matching components)
  + /= (inequality test for matching components)
  + in, not in (membership tests) (73)
* “Access types are used for ‘dynamic storage allocation,’ whereby new objects are created during the execution of a program.” (73)
* “An access type is used to construct access objects. An access object ‘designates’ another object. When an access value is ‘allocated,’ two things happen:

1. an access value is established,

2. a new object is created.

Access values are used to ‘dynamically create’ objects during program execution time.” (73)

* “Ada establishes an access type definition in an idiomatic fashion.” (73)
* Ada allows for “recursively defined data structures.” (75)
* “The notation ‘**.all**’ indicates the entire object designated by an access type.” (77)
* Access types have the following predefined operations:
  + := (assignment)
  + = (equality test)
  + /= (inequality test)
  + in, not in (membership tests) (78)
* “A private type can be any type; its declaration is deferred to the ‘private’ part of a package. There are two kinds of private types: private and limited private.” (78)
* “A private type declaration may only occur in a package or generic specification. Private types are used for protection (frequently protecting a user from himself), for ‘information hiding’ (to increase program reliability by decreasing particular implementation dependencies), and for simply preventing unwarranted user modifications.” (78, 80)
* (Non-limited) Private types have the following predefined operations:
  + := (assignment)
  + = (equality test)
  + /= (inequality test)
  + in, not in (membership tests) (82)
* “A private type has the same allowable set of operations as an access type.” (82)
* “There are *no* predefined operations for a *limited private type* except for the membership tests **in**, **not in**. The only other *available* operations for a limited private type are defined in the package specification *where* the type is defined (these operations are *not* predefined).” (82)
* “Most of the predefined operations in Ada are ‘overloaded’ among objects of different types.” (82)
* “Ada distinguishes possible operator ambiguities by noting the type of the *operand*.” (82-83)
* “Overloading is an *explicit* fact of the Ada language for all identifier names (except reserved words). This is a very powerful mechanism; two programmers working on the same project should not be worried if they happen to declare the same identifier (if their types do not resolve any ambiguity, selected component notation can be used to distinguish these identifiers from one another). Consequently, we should not be surprised when we are told that Ada allows a redeclaration of all predefined operations (except assignment, inequality test, and the membership tests). The operations that a programmer can redefine are on the list of predefined operations for INTEGER, STRING, and BOOLEAN. They are =, <, >, <=, >=, +, -, \*, /, **rem**, **mod**, \*\*, &, **and**, **or**, **xor**, **not**. When these operators are redefined, they must appear in a character string format in the function declaration (e.g., ‘&’). Note that we can always redefine ABS since it is not a keyword. Also note that /=, **in**, **not in** are not on this list.” (84)
* “Certain conditions must be met when one wishes to redefine these predefined operations:

1. The following operators must be used in function specifications having *two* parameters.

=, <, >, <=, >=, +, -, /, **rem**, **mod**, \*\*, &, **and**, **or**, **xor**

2. **not** must be used in a function declaration having only one parameter. +, - can be used for either one-parameter or two-parameter functions.

3. = can be redefined *only* for limited private types (or for a composite of a limited private type). (86-87)

* “Tasks are entities that may operate in parallel. The syntax structure of a task is a cross between a type and a subprogram. A task has a specification and a body. A task can be a ‘task type’: objects of a task type are tasks. A task must be *contained* in something (a block, subprogram, package, or another task body) but a task body may be compiled separately from this containing unit.” (91)
* “Ada tasks communicate with each other by means of an entry. This is a very powerful mechanism. Entries allow for task synchronization in a ‘rendezvous.’ A rendezvous occurs when two tasks (initially executing in parallel) meet together and execute concurrently. The rendezvous is over when the two tasks split up and become parallel independent entities again.” (95)
* “Note that an entry declaration is similar to a subprogram declaration.” (96)
* “Note that an entry call is similar to a subprogram call: The parameters can be given in either regular notation or in parameter association notation (=> ). Also note how we use the selected component notations in the entry call: The first component is the name of the (accepting) task and the second component is the name of the entry.” (97)
* “We note some details about entries and task communication. First, note that it should not be surprising that an accept statement is similar to a *subprogram declaration*, (since an *entry call* is similar to a *subprogram call*).” (99)
* “This situation uses a ‘selective wait with alternatives.’ Note the similarity between this form and the case statement. If the condition in the ‘when clause’ is true, then the corresponding accept statement *may* be selected for a rendezvous (*if* an entry call requested this rendezvous).” (102-103)
* “A delay statement can be considered a ‘rendezvous with the clock.’ A delay statement suspends task execution for the given time. A delay operates on objects of predefined type DURATION. SECOND is predefined; other objects (such as MINUTES) may be defined in terms of SECOND.” (105)
* “One way for a task to terminate is for it to reach the *end* of its task body. . . . While it may be of some use to have programs that normally run forever (for real-time systems), it is also equally desirable to terminate unwanted tasks so a particular subprogram or block can finish executing. This can be accomplished with two keywords: **terminate** and **abort**.” (105)
* “The **abort** keyword results in an ‘abnormal’ termination of any task. An **abort** statement unconditionally terminates a named task.” (105-106)
* “The terminate statement allows for a more conditional end.” (106)
* “Tasks are like subprograms because they perform some actions. They also are like types. Tasks, in fact, can be ‘task types.’ An object of a task type is a task (and the predefined operations for task types are the same as those for limited private types). . . . If tasks are very similar in their actions (as in a multiprocessing fault-tolerant redundant computer system) then task types make a system description easier (for a human) to understand.” (107-108)
* “A task type is a ‘factorization’ of the task entity.” (108)
* “For many real-time parallel operations implemented on minicomputers and microcomputers, entry calls are frequently specified as ‘hardware interrupts.’ . . . Frequently the hardware interrupts are specified by certain address that depend on the implementation.” (108-109)
* “A representation specification for an enumeration type specifies certain internal codes (for hardware interfacing). All representation specifications for types that occur in a package specification must occur in the private part of the package specification. For tasks and task types, the representation specification for the entries may be given in the task specification. Other representation specifications involve the length and alignment of record types as well as actual machine cord insertions.” (110)
* “A system design is a process that transforms system specifications (from requirements) into a representation of a system. A design is to a system as a map is to a territory. A system design is a *blueprint* of a system. A chief function of a system design is to communicate system concepts to other humans. Once a system design is frozen, it can be used as a ‘map’ for the actual system implementation. It is extremely important to construct not only a *correct* design, but a *maintainable* and *reliable* one as well. System specifications are rarely frozen; to create a *maintainable* and *reliable* system implementation, it is necessary to create a *maintainable* and *reliable* system design.” (115)
* “Ada can be an SDL as well as a PDL. The ‘manageable’ chunks are Ada packages, tasks and subprograms. The *interfaces* are shown clearly in the *specification* part of these program units. Note that the specifications of these program units are textually *distinct* from the bodies (implementation) of these program units. Note that the bodies are where the actual ‘action’ occurs. The *body* of an Ada program unit shows the *control structure* of the system: The body is the ADA PDL description of a system component.” (118)
* “The architectural design (SDL) and detailed component design (PDL) is used as a ‘map’ or ‘blueprint’ for actual system implementation. The goal of the system implementation is to produce correct and maintainable source code (in some implementation language) using the system design as a blueprint and using standard ‘programming’ techniques. The implementation language can be any programming language. The ‘best’ choice for an implementation language is a ‘high-level’ language (Which allows for maintainability and portability) that is translated into ‘efficient’ object code. If Ada were to be used as the design and implementation language, then the system design would be identical to the system itself, and maintaining this design would be identical to maintaining the system itself.” (118)
* “All packages (specifications and bodies) and subprograms (specifications and bodies) can be separately compiled. This allows a tremendous amount of design freedom and a true *simultaneous* top-down-bottom-up design that enforces a *modular* and *component discipline* on the system design and system implementer.” (127)
* “All tasks are activated just before the ‘**begin**’ in the immediately surrounding package (or after the declarative part in an immediately surrounding program unit).” (138)
* “The apostrophe has two more important uses [other than denoting character literals]. One is its use in the notation for a ‘qualified expression.’ A qualified expression is used to explicitly state the type of an object (to avoid ambiguity).” (142)
* “Qualified expressions also have some use for certain assignments . . . and for ‘code statements for machine code insertions.” (142)
* “The most important use of the apostrophe is for the identification of an *attribute*. An attribute denotes a certain predefined characteristic of an Ada program entity. There are 46 predefined Ada attributes; most of them reflect some implementation base characteristics.” (142)
* “Attributes allow somewhat greater control for particular problems. They are similar to certain ‘standard functions’ (in Pascal) or ‘built-in functions’ in (PL/I). (143)
* “Ada has certain facilities concerned with the handling of certain errors that occur during the *execution* of a program. These “exceptional situations’ (or exceptions, for short) cause a suspension of the action of a program. When a particular exception is ‘raised,’ program action can either halt or some other action can be taken (that depends on the exception) that ‘handles’ this exception.” (143)
* “Ada predefines the following exceptions:
  + CONSTRAINT\_ERROR is raised when a range constraint is violated.
  + NUMERIC\_ERROR is raised when a predefined numeric operation violates a range constraint.
  + SELECT\_ERROR is raised when no **accept** statement or **else** statement can be selected in a **select** statement.
  + STORAGE\_ERROR is raised when the available space allocated to a task or object is used up.
  + TASKING\_ERROR is raised when errors occur during task communication.” (143-144)
* “A user may define his own exceptions; they are declared just like objects.” (144)
* “When an exception is raised (either explicitly by the programmer in a raise statement, or implicitly for a predefined exception), execution is halted. All actions are then transferred to the ‘exception handler’ (if one exists) which occurs at the *end* of a block, subprogram, task body, or package body.” (145)
* “There are some tricky rules to observe when an exception is ‘propagated’ during declaration from one program unit to another program unit. . . . Where an exception is handled depends on *where* the exception situation occurs and *where* the handler is. Exceptions are usually propagated to a *containing* unit or a *calling* unit, if the units where the errors occur do not have handlers. Otherwise, execution is *abandoned*.” (145)
* “There are also some tricky rules to observe when exceptions occur during tasks communication. . . . Essentially, exception TASKING\_ERROR is raised when a calling task issues an entry call to a terminated task . . . . Exceptions occurring in a rendezvous are propagated to both tasks.” (145)
* “A pragma is a certain instruction to the Ada Compiler. Pragmas can also be defined by a particular Ada implementation. Ada has 12 predefined Pragmas. (146)
* “The Ada language (defined in package STANDARD . . .) does *not* contain any standard input-output features. Input-output is predefined by *other* Ada *packages*.” (147)
* “The input-output packages are:
  + Package LOW\_LEVEL\_IO. This package defines procedures that are used to send information to physical devices. Obviously, this package is highly implementation dependent.
  + Package TEXT\_IO. This package defines input-output facilities to place data in ‘human readable’ form. It contains the following *generic* packages:
    - generic package INTEGER\_IO (for integer types)
    - generic package FLOAT\_IO (for floating-point types)
    - generic package FIXED\_IO (for fixed-point types)
    - generic package ENUMERATION\_IO (for enumeration types)
    - It contains the following available procedures:
      * procedures for Boolean input-output
      * procedures for character and string input-output
    - The functions GET (for input) and PUT (for output) are overloaded. The usual range of formatting features is also available.
  + Generic package INPUT\_OUTPUT. This package is used for general user input-output (for file creation, modification, reading, writing). The generic feature allows the use of procedures on files of arbitrary type.
    - The parameters to the function PUT are:
      * the desired object that is to be output, and
      * other parameters that describe the output format. There are default values for these parameters. All Ada subprograms may have default parameters declared in their specification. This should not be surprising: The declarations in Ada for constants, objects, record components, discriminants, subprogram parameters, entry parameters and generic formal parameters may all contain default values.” (147-150)
* “The *basic* Ada character set includes the capital letters A-Z and the digits 0-9. In addition to the ‘blank’ character, the basic Ada character set also includes the ‘special characters’:

“ # % & ‘ ( ) \* + - . / : ; < = > \_ |” (150)

* “The use of these special characters may depend on a particular Ada implementation at a particular installation. Consequently, there are some special character ‘aliases’:
  + # (sharp) may be replaced by : (colon) in based numbers,
  + “ (double quote) may be replaced by % (percent) at both ends of a string, and
  + | (vertical bar) may be replaced by ! (exclamation). The | character is used to represent a disjunction of *choices* (not conditions) for aggregate components, variant parts of records, and case statements.” (151)

Pyle, I. C. (1981). *The Ada programming language: A guide for programmers*. London: Prentice Hall International.

* “Ada is for programming embedded computer systems—that is, systems in which a computer is directly connected to some apparatus or plant which it monitors and/or controls. This means that Ada can be used for conventional programming (which actually accounts for the majority of embedded computer system programming) and also for the special technical requirements concerning input/output, timing relationships, contingency programming to cope with errors, and long-term maintenance.” (1)
* “This concern for maintenance underlies much of the style of Ada.” (1)
* “Ada gives special attention to the ease of reading and understanding programs—it is based on the realisation that it is more important to be able to read a program and understand it clearly than to be able to write it quickly or briefly. We therefore tend to use fairly long names and identifiers in an Ada program, and state the assumptions which the design of the program implies. The reason for this is that the writer of the program does his job once, but maintainers of the program may have to read the program many times throughout its life.” (1)
* “In an Ada development environment, many procedures will be held in a library, where they are available for use in other programs. In practice, all programs are likely to refer to the library for units defining many commonly required actions such as input/output and mathematical functions. A package called STANDARD is always available . . . . A program must begin by listing the units it needs; these will be extracted from the library by the translator.” (2)
* “The text of the Ada program consists mainly of two kinds of words and various punctuation marks. Words like **procedure**, **is**, **separate**, **end** (conventionally written in small letters) are reserved for special uses in Ada, and determine the main structure of the program. These are known as keywords. The other words . . . are invented by the programmer, to denote the various entities in the program; these words are technically called identifiers. They must always be different from the Ada keywords.” (5)
* “As well as the main text of the program, whose structure is prescribed by the Ada language, there may be comments on any line, introduced by a double hyphen. Comments may contain any characters without restriction, for the rest of the line. They are used by the programmer to give additional information to the reader of the program, but this is not checked in any way by the compiler.” (5)
* “Another special construct in an Ada program is called a **pragma**: this is a phrase used to give information to the compiler about translating the program.” (5)
* “Pragmas do not affect the meaning of a program, but may affect the way it is implemented (e.g. choice of optimisation).” (5)
* “Identifiers are the fundamental creation of the programmer: they name the entities which are needed for the particular program he is designing. An identifier is made up using letters and digits (linked by underline characters)—which must be different from the Ada key words, disregarding the case of the letters. Identifiers may not contain spaces, and may not spread over from one line to another.” (5)
* “The translation of an Ada program into its executable form is more than traditional compilation. It includes also the operations of linkage-editing and library module incorporation (which are done separately with other languages), and the provision of run-time facilities to implement the various semantic features of the language such as inter-task communication and dynamic storage allocation.” (6)
* “Translation of an Ada program involves compilation of the separate units, with cross-checking of interfaces and provision of other required units from a library.” (6)
* “An Ada program may be run on a single-computer system; this is the common way to presume its implementation. However, Ada does not require there to be only one computer, and a program with suitable restrictions (corresponding to the lack of direct communication between the processor in one computer and the store in a different computer) may be implemented in a multi-computer system.” (7)
* “A computer running an Ada program is dedicated to a specific task—controlling the system in which it is embedded. Consequently the Ada program in it is presumed to be the only program in the computer—not sharing facilities with others. (The Ada program may involve units written in other languages, provided that their interfaces and overall structure are consistent with Ada.) Thus Ada implies a single program but not necessarily a single computer.” (7)
* “A fundamental idea in Ada is that every data item has a particular type, which determines the possibilities for the values it may have.” (9)
* In common with other ‘strongly-typed’ programming languages, Ada requires the programmer to specify the type for every data item concerned in the program, so that its usage can be checked. Most checking is done when the program in Ada is compiled, and does not imply any run-time overhead. Ada includes some predefined types; but most of the types in a program will be invented by the programmer, suitable for the particular application involved.” (9)
* “A type, then, gives the set of possibilities for a data item—specifying not so much what its value currently happens to be, but what it might ever legitimately be. This is of great importance in program checking and maintenance: compilers make sure that the usage of every data item is consistent with its set of possibilities, and when a programmer needs to change a program, the type helps him to understand the purpose of a data item.” (9)
* “The simplest types are called scalars, and include numbers (integers and real), characters (ASCII, or any other specified character set), truth values (TRUE and FALSE) and enumeration types (in which the possible values are listed explicitly).” (9)
* “Ada has the predefined type BOOLEAN for this purpose [determining whether a condition is true]. It is actually an enumeration type, with only two possible values, TRUE and FALSE.” (10)
* “Numbers must always be given with a range, so that the lowest and highest possible values are stated.” (10)
* “Each implementation of Ada has a predefined type INTEGER which covers the whole range of values expressable [*sic*] in a machine word, also SHORT\_INTEGER and LONG\_INTEGER for implementation-determined ranges shorter and longer than a machine word.” (10)
* “For characters, there is a predefined character set CHARACTER.” (10)
* “Each implementation of Ada has a predefined type FLOAT, which covers the whole range of values expressable [*sic*] in a machine floating-point word, also SHORT\_FLOAT and LONG\_FLOAT for implementation determined floating point values shorter and longer than a machine word.” (11)
* “The idea of subtypes and constraints is to allow the programmer to state the intended set of values which might arise, both as an aid to his own thinking when writing the program, and to assist subsequent maintenance programmers who may have to change it. Subtypes are constructed by specifying an existing type by name, and, if required, giving a further constraint.” (12)
* “A record can have any number of components, of the same or different types.” (13)
* “The components in a record may be of any named type or subtype (but not the outer record type, or any type containing it). A constraint on the type or subtype may be given explicitly . . . .The type of a record component may be given as a named enumeration type . . . .” (14)
* “The components of a record may themselves be records (with the obvious restriction that they can not contain themselves).” (14)
* “The elements of an array may be of any named type. If the elements are required to be records, the element-type must be first declared and given a name. (It is not allowed to give the element-type-definition within the array definition.) The element-type may be constrained, either by an intermediate subtype declaration or by attaching a constrain to the element type in the array definition.” (16)
* “The components of an array may be records, and the components of a record may be arrays.” (16)
* “Formally, STRING is defined as a type in which the string length is not specified, but only its subtype (NATURAL). When the programmer introduces any data item as a string, he must specify the number of characters it is to contain, either by giving the initial value or the particular range for the index . . . .” (19)
* “String values are given by enclosing the appropriate sequence of characters in double quotes; if the double quote character itself is required, it must be written twice. To avoid mistakes, the string may not spread over more than one line.” (19)
* “Strings (and identifiers denoting strings) can be concatenated by using the operator & . . . .” (19)
* “Every data item (variable or constant) in a program has a particular type, which specified when it is introduced. A data item for a particular part of the subprogram (block or unit body) is introduced by an object declaration which gives its name, type or subtype, and optionally an initial value. An object is taken as a variable unless it is explicitly stated to be a constant.” (20)
* “Ada is a strongly typed language, which means that the types of all data values in the program are checked for consistency with their usage. Keeping types distinct has been found to be a very powerful means of detecting logical mistakes when a program is written and to give valuable assistance whenever the program is being subsequently maintained. In Ada, every type definition introduces a new type (even if it looks the same as another). Checks for type consistency are strictly applied.” (21)
* “Another way of interpreting the ‘strong typing’ rule is in terms of the abstract values which can occur. Each type defines a set of abstract values, and any object of that type can only take values from that set. Strong typing means that the abstract values for any type care absolutely distinct from those of any other type (even though their representations may be handled by the computer similarly). This means that any overlap between one type and another is logically impossible.” (22)
* “In a simple program each object has a single name and each name denotes a single object. In a more complicated program names and objects may be related in other ways. The same object may have several names (known as aliases) or no name at all. Each name is valid in a particular part of the program (over which it is said to be visible), and in distinct parts of the program it is possible for the same name to be used without confusion for completely different purposes.” (22)
* “Names are used to denote many kinds of entity in Ada: not only data objects (variables and constants) but also types, modules, subprograms, entries and exceptions. When the entity is introduced in the program, it is given an identifier (or possibly a character string in the case of a function subprogram, to overload an operator). This identifier is the local name for the entity, which can be used wherever the declaration is directly visible: principally the statements which follow the declaration. In other parts of the program, it is still possible that the entity can be referred to, but in this case a fuller name would have to be used, to establish which part of the program contains the required declaration.” (22-23)
* “An expression is a formula for calculating a value. The type of the value calculated, and the types of all constituents in the expression, are determined at compile time to give strong checking for inconsistencies.” (26)
* “Expressions are mainly used for numeric types but are also used to calculate truth values from logical expressions.” (26)
* “A floating point expression is evaluated approximately, depending on the precision of the constituents: the number of digits specified.” (28)
* “Expressions in Ada may deal with truth values (of type BOOLEAN), testing the truth of relations or conformance with constraints. (The types of the values in the expression are checked at compile time, depending on the operators used, but constraints may need run-time checking.) Logical expressions are used in situations where a truth value is required (such as in if statements0 as well as in expressions generally.” (29)
* “Any two values of the same scalar type may be compared, and the results taken as a truth value. The comparison is written using one of the six relational operators. Certain non-scalar values may also be compared.” (29)
* “Any two values of the same type (not necessarily scalar) may be tested for equality unless they have been declared to be **limited**.” (29)
* “We can test whether a value of a scalar type satisfies a range constraint. The constraint may be stated explicitly or as a subtype.” (30)
* “Truth values may be combined using the operators **not**, **and**, **or**, **xor** with their usual meanings. . . . In addition, there are ‘short circuit’ operators: **and then** and **or else**. These avoid the evaluation of the operand on the right when that is unnecessary.” (30)
* “A type which is declared as a derived type automatically has all the properties of the base type, including whatever operators are appropriate. A type that is declared as a private type automatically has the equality comparison operators (= and /=) unless it is limited.” (31-32)
* “Expressions are formed from operands and operators. An expression can consist of a literal value or the name of a variable or constant as special cases; in general it is one or more operands connected by operators. There is no limit in Ada to the size of an expression, but most programmers find that expressions longer than one line are hard to read and understand.” (32)
* “An expression is evaluated by evaluating the operands and combining them according to the operators, in an order defined by certain rules of precedence. The rules have been chosen to reflect common practice in mathematics and the conventions that have been established in other programming languages.” (32)
* “The operands within an expression can be of the following kinds:
  + Literals, giving values of scalar objects (numbers or enumeration literals)
  + Aggregates, giving values of compound objects (records or arrays). The values of the components are given by inner expressions, either in the natural order or with the corresponding filed names of index values.
  + Names of objects (variables or constants). An object may be scalar or compound.
  + Allocators, in which a value of an access type is obtained, denoting a newly created base object with a given value or constrain.
  + Function calls, in which a value is calculated in a subprogram. The value may be scalar or compound, and the function may have parameters which are given as variables or inner expressions.
  + Type conversions, whereby a value in one type is converted to the appropriate value in a closely related type. This is the way Ada combines the advantages of strong type checking with the flexibility of mixed expressions: the programmer has to state the type conversions explicitly.
  + Qualified expressions, for resolving possible ambiguities and ensuring that constraints are met.
  + Subexpressions, that is inner expressions enclosed in parentheses.” (32-34)
* “The operators which can be used to compose expressions are as follows [in order of precedence]:”
  + \*\* (exponentiation)
  + \*, /, mod, rem (multiplying and dividing)
  + +, -, not (unary)
  + +, - (adding and subtracting)
  + & (concatenation)
  + =, /=, <=, <, >, >= (relational)
  + In, not in (membership)
  + And, or, xor (logical)
  + And then (short circuit)
  + Or else (short circuit)” (35)
  + Adding and concatenation are same precedence
  + Relational and membership are same precedence
  + Logical and short circuit are same precedence (36)
* “The membership operators **in** and **not in** take a special kind of right operand: a type or a subtype rather than a data value. Otherwise all the operands are data values, and the result of ever operator is a data value.” (35)
* “In conformity with the ‘strong typing’ rule, the operators refer to particular types of operand, and combine them to form results of known types. These operators are defined in Ada for particular predefined types, and may be extended by explicit programming if required for other types.” (35)
* “The rules for evaluation are as follows:
  + Where an operand has operators on both sides of it, and the operators are of different precedence, the operator of the highest precedence is applied first. Where the operators are of the same precedence, the operator on the left is applied first.
  + A binary operator has operands on both sides. Unless the operator is a short circuit operator, both operands are evaluated (in either order) before the operator is applied. When the operator is a short circuit operator, the operand on the left is evaluated before the operator is applied; if necessary, the operand on the right is then evaluated.
  + A unary operator is applied to the resulting operand on its right, taking account of [the first rule].” (36)
* “The actions to be carried out in the program are specified by writing a series of statements, in the order they are to be executed.” (38)
* “Ada prescribes the basic form of statements, and how they can be combined to form more complicated actions. At any position in the program, it is likely that the programmer will need to use some of these basic statements but also some application-specific statements that call for the performance of actions whose details are stated elsewhere. Such statements are procedure calls . . . .” (38)
* “Each statement specifies an action to be carried out; a sequence of statements specifies a series of actions to be carried out, one after the other, in the order written. Statements are classified as either simple or compound: compound statements contain internal sequences of statements (which may in turn be simple or compound).” (38)
* “Any statement, whether simple or compound, may be labelled, to give it a name. The label is put in front of the statement, enclosed in double angle brackets.” (38)
* “Some compound statements (loop and declare) may be given a local name using a colon.” (39)
  + “This name is available only in the internal sequence of statements.” (39)
* “Every statement is written with a final semicolon.” (39)
* “A simple statement is either sequential or branching. Sequential statements are executed in their order of occurrence; after a sequential statement has been executed, the statement which follows it in the sequence (if any) is executed. A branching statement is normally the last of a sequence of statements: when it has been completely executed, the statement executed next is determined not by its position, but by some other rule, dependent on the particular statement type. A compound statement may have either characteristic, depending on the simple statements of which it is composed.” (39)
* “If a compound statement contains another compound statement, they are said to be nested. Ada does not limit the depth of nesting, but it can be difficult for the human reader to be sure of his context in a deeply nested structure.” (39)
* “The compound statements in Ada are bracketed by keywords, giving a clear indication of the start and finish of each sequence of statements. They include statements in which the actions to be carried out are selected according to particular criteria (**if** and **case** statements), or are repeated as required (**loop** statements), or have local declarations (**declare** statements) . . . . In addition, there are **accept** statements and **select** statements . . . .” (39)
* “A variable is a data item whose value may be changed. A new value is given to a variable by an assignment statement. The previous value of the variable (which might be involved in calculating the new value) is then completely lost. The variable may be of any type (including a record or an array as a complete entity); the new value given to it must of course be of the right type, and consistent with any constraints.” (40)
* “Note that the assignment gives the variable a complete new value. If it is of a compound type, all the components get new values.” (40)
* “In an **if** statement, the programmer states the condition to be tested, then the sequence of statements to be executed if the condition is true. If the condition is false, there may be other actions to be carried out, which may include testing further conditions. Thus the basic idea is to have a series of conditions to be tested in order, with corresponding actions to be taken on finding the first condition which is true, and finally actions to be carried out if none of the conditions are true.” (40-41)
* “The keywords used to identify these parts of the statement are **if** at the beginning, before the first condition, **elsif** before each subsequent condition (zero or more times); each condition is followed by **then** and a sequence of statements. The whole of the statement is always terminated by **end if**;” (41)
* “In a **case** statement, the program gives an expression whose value (which must be of a discrete type, i.e. integer or enumeration) determines which actions are required. The programmer states the possible expected value (or values) as choices for the discriminating expression, then the series of statements to be executed if the expression has that value. Several values may be given for each choice, but the values given for the different choices must of course be distinct.” (41-42)
* “The choice **others** means any possible values of the discriminant which are not given previously. It may only be given as the last choice.” (42)
* “The basic difference between **if** and **case** is that in an **if** statement the conditions are tested on after the other, whereas in a **case** statement the expression’s value determines one of the various possibilities directly.” (42)
* “An action may need to be repeated, for example to carry out some operations regularly to a series of items, or to iterate some improving operation. The unit of repetition is written as a sequence of statements, forming the body of the loop, introduced by a suitable phrase which may show a condition for stopping the repetitions.” (42)
* “The basic loop simply groups the statements to be repeated, without implying any termination; this is frequently needed in large program structures . . . .” (42)
* “The repetitions of a loop may be controlled by either counting or testing a condition; these are expressed by the iteration specification at the start of the basic loop. Independently, during the course of execution of the loop, a condition might arise requiring exit from the loop.” (43)
* “Control by a condition is expressed by a **while** clause.” (43)
* “The condition is tested before starting each repetition: it may indicate that the body of the loop is not to be executed at all.” (43)
* “Control by counting is expressed by a **for** clause.” (43)
* “Unless otherwise specified the loop parameter counts forwards through the range. However, if the keyword **reverse** is put in, the loop parameter counts backwards through the range.” (44)
* “Higher order actions are constructed out of more elementary ones, and made available as procedures (or entries) either from libraries which have already been written, or by the programmers writing the current program.” (44)
* “Procedures and entries can have parameters, and each time one is called to carry out its action, appropriate values and variables must be specified for its parameters. Every procedure and every entry has a specification that states exactly what kinds of parameters it needs, if any.” (44)
* “The call statement gives the name of the procedure or entry to be executed, followed by the actual parameter values and/or variables to be used. To call a procedure or entry with no parameters, just write its name.” (44)
* “To call a procedure or entry which requires parameters, put the required values or variables in parentheses after the name.” (44)
* “The same notation is used to call procedures and entries, because from the point of view of the caller they both carry out some action and then return. The difference is in the way they may interact with the rest of the program: a procedure in Ada is reentrable, so may be also executed in parallel by several tasks concurrently; an entry is not reentrable, and may only be accepted by one task at a time to carry out its action.” (45)
* “A declare statement is a compound statement containing a sequence of statements, optionally preceded by local declarations and optionally followed by local exception handlers. This is also called a block. The local declarations apply within the block but not outside it in the rest of the program.” (45)
* “The declaration statement is introduced by the keyword **declare** if there are any local declarations, which would be the normal case. (If there are no declarations, omit the word **declare**.) After the local declarations, the start of the sequence of statements is marked by the keyword **begin**. If any exception handlers are required they are written after the main statements, introduced by the keyword **exception**.” (45)
* “There are statements whose successor in execution is not necessarily that which follows it in sequence, but some other, determined by the particular statement type. The statements concerned are exit, return, raise and goto.” (46)
* “An exit statement specifies explicit termination of a loop. Within a loop, whether or not there is an iteration specification given by a **for** or **while** clause, the repetitions may be stopped by execution of an exit statement; the whole of the loop statement is then deemed complete.” (46)
* “A loop statement may of course be one of the statements in the body of another loop, and so on. The loops are said to be nested. If an exit statement is written simply, then it terminated repletion of the smallest loop enclosing it; however, any required loop of the next may be terminated by giving the loop a local name and writing that name in the exit statement.” (46)
* “A return statement specifies the end of execution of a subprogram body, and indicates completion of the corresponding subprogram call.” (47)
* “If the subprogram is a function, then the return statement must specify the value to be delivered by the subprogram; it must be of the right type as determined by the subprogram specification.” (47)
* “The [return] statement may be in the body of an inner condition, loop or declare statement—if so it automatically terminates the inner statements by terminating the subprogram body.” (47)
* “A procedure which does not return a value may contain a return statement with no value attached . . . but in this case the return statement is not essential: reaching the normal end of the sequence of statements in the body equally indicates completion of the corresponding procedure call statement.” (47)
* “A raise statement specifies that an exception situation has been detected, so that normal sequential execution of statements cannot continue.” (47)
* “A goto statement specifies its success statement explicitly, by giving a statement label.” (48)
* “The pair of statements connected by a goto and a label must be fairly close together in an Ada program. Specifically, they must both be in the same unit body, that is the same subprogram, package body (initialisation) or task body.” (48)
* “Within that body, if one [goto] is in an accept statement the other must also be in the same accept statement; if one is an exception handler the other must also be in the same exception handler.” (49)
* “The goto statement must be in a sequence of statements (usually at the end of the sequence), and this sequence will usually be in a compound statement which is part of an enclosing sequence of statements. The corresponding labelled statement must be one of the statements in a sequence containing or enclosing the goto statement.” (49)
* “The statements in a sequence are normally executed as rapidly as possible: as soon as each statement has been completed, the next statement in the sequence may be started. If you want to insert a timed delay in the sequence, us a delay statement.” (49)
* “The amount of the delay required must be given in seconds; it can be a constant, a variable or an expression. (A negative value is treated as zero.) The effect of such a statement is to delay starting execution of the next statement in the sequence for at least the specified duration. . . . Note that a delay statement gives a minimum time: it does not imply any upper limit to the interval before the next statement is started. . . . Note also that a delay is for a particular period of time (which may of course have been calculated in advance).” (49)
* “The delay statement is the ordinary method of achieving timed sequential control . . . .” (49)
* “Sometimes the program structure requires you to write a sequence of statements even if nothing has to be done—for example in an arm of a case statement. The **null** statement is a positive indication that no action is required. It is used to avoid the error of accidentally omitting a sequence of statements that should be there.” (50)
* “A subprogram is a program unit for describing an action. There are three important aspects of subprograms: how to specify their interfaces; how to describe in detail the action concerned; and how to call for that action to be carried out.” (52)
* “Subprograms include procedures and functions; in many ways they are also similar to entries. Procedures are actions to achieve a particular effect; functions are actions to calculate a value (usually with no other effect on their environment). A function may be called either as a named function or by an operator in an expression.” (52)
* “Each subprogram has a specification (which gives its signature) and a body (which also includes the specification, and gives details of how the action of the subprogram is to be achieved). In most cases it is not necessary to give the specification apart from the body.” (52)
* “In the case of a subprogram, the specification tells you exactly what kinds of parameters the subprogram needs, and what kind of effect it has.” (52)
* “The specification gives the name of the subprogram and all the necessary information about its parameters.” (52)
* “A subprogram may be specified as a function or a procedure; this indicates the intention of the subprogram: a procedure is intended to achieve an effect, a function is intended to calculate a value. There are special rules concerning a subprogram specified to be a function—any parameters it has must all be of mode **in**, and it must deliver a result; its effect on its environment must be solely determined by its parameters.” (53-54)
* “In most cases the name of a subprogram is an identifier. However, it is possible to introduce an additional meaning for an operator by defining a function with the operator symbol (as a character string) given as its name. This is called overloading the operator. Subprograms named by an identifier may have any number of parameters (including zero). Subprograms named by an operator symbol must be functions and must have the number of parameters appropriate for that operator symbol (one or two).” (54)
* “A subprogram may have parameters, each of which is specified with an identifier, mode, type and possibly a default value.” (54)
* “The direction in which the value is passed [to a subprogram] is called the mode of the parameter, and is written **in**, **out**, or **in out**. Mode **in** is implied if none is stated explicitly: the calling program passes a value **in** to the procedure parameter. For a parameter mode of **in**, the corresponding call must provide a value (of the correct type) which is given as an expression, including a variable or a constant. For a parameter of mode **in out**, the value is passed in at the beginning, and out at the end of the subprogram. The mode **out** means that the procedure passes a value out to the parameter (but does not use its previous value).” (54-55)
* “The values and variables given as actual parameters when a subprogram is called must match those specified. The correspondence between the actual parameters given and the formal parameters required may be established either by the order in which they are written or by using the formal parameter names. For association by position, the first actual parameter given corresponds to the first formal parameter required and so on. For association by name, the actual parameter is given with the name of the formal parameter it matches, with different parameters given in any order.” (55)
* “A subprogram can be specified with defaultable parameters. A subprogram call need not then give the corresponding actual parameter, and the default from the specification will be used.” (55)
* “Defaults may only be used with **in** mode parameters.” (56)
* “A subprogram can calculate a value for use in an expression. Such a subprogram is a function.” (56)
* “In Ada, there can be several subprograms with the same name which are distinguished by the properties of their parameters. Using the same name for distinct meanings in the same context is called overloading. This is useful when substantially the same action can be applied to objects of different types.” (56)
* “The choice of the procedure called is made by the compiler based on type matching. There is no overhead at run-time to select the correct procedure.” (57)
* “The properties of the parameters which distinguish different subprograms with the same name are their order, names, modes and types, together with the type of the result. All of this information, together with the subprogram name, is know [*sic*] as its signature.” (57)
* “The body of a subprogram gives the details of what is to be done whenever the subprogram is called. The body begins with the specification of the subprogram; it may then have local declarations (introducing entities which are valid for the subprogram body but nowhere else in the program). The main part consists of a **begin** block which gives the sequence of statements to be executed, and optionally an exception handler.” (58)
* “A subprogram body is very similar to a block, except that it starts with the specification instead of the word **declare**. After the specification there may be local declarations, and there always is an executable part introduced by the word **begin** which may be defended by exception handlers.” (59)
* “The formal parameters in the specification, together with the local declarations, establish entities for the subprogram body. Formal parameters have the force of object declarations with the specified type or subtype; those with the mode **in** or **in out** are effectively initialised to the value of the corresponding actual parameter or default value (and those with the mode **out** have no significant value initially). Parameters with mode **in out** or **out** have significant final values—the value of the parameter at the end of the subprogram is assigned to the variable in the caller stated as the corresponding actual parameter.” (59)
* “For a function, the specification contains a **return** part, and the statements in the body must include a **return** statement with an expression of the type specified, to define the result of the subprogram. For a procedure the specification does not contain a **return** part, but a **return** statement may be used (without an expression) to mark the end of execution of the body.” (59-60)
* “A subprogram body occurs either in the declarative part of some larger program unit or on its own as a separate compilation unit. If it is in a declarative part of a subprogram or block, the specification given with the body is usually sufficient and does not have to be repeated. However, there are two cases where a specification must be given by itself early in the declarative part, and also again with the body late in the declarative part. The cases are
  + When two or more subprograms include calls of one another: a specification must be given before any call of the subprogram.
  + When a subprogram is defined as part of a package: the specification must be given in the visible part of the package.” (60)
* “Ada is designed to permit defensive, or fault-tolerant programming. This is in contrast with traditional styles of programming, in which there is an implicit assumption that everything is correct—the program, the translator, and the computer hardware (with only some concessionary provision for faults in peripherals or operator behaviour).” (65)
* “There are a number of exceptions predefined in Ada, for situations which are logically possible in the language. . . . In addition to the predefined exceptions, any number of further exceptions may be declared, for particular exceptional situations relevant to the program.” (65)
* “The action of indicating that there is an exceptional situation is called raising an exception. Some constructs in a program may implicitly raise an exception; the program may also contain explicit statements to raise exceptions.” (64-65)
* “For each exception, whether predefined or explicitly declared in the program, the programmer may declare handlers to carry out whatever recovery action is necessary and then to resume the normal sequence of execution of the program. The place in the program where normal execution is resumed depends on the position of the corresponding exception handler—the handler always comes at the end of a block and completion of the handler has the same effect as normal completion of the block.” (65)
* “Exceptions are dealt with according to the dynamic block structure of a program. As each statement is executed, it may involve the execution of statements elsewhere in the program (by subprogram calls); in this sense all the blocks of a program are dynamically nested during execution. When an exception arises, the dynamic block structure is ‘unwound’ in the course of recovery and resumption of normal processing, from the smallest block enclosing the offending statement to the first dynamically enclosing block containing a handler for the relevant exception.” (65)
* “Each exception is denoted by an identifier, which must be introduced by a suitable declaration.” (65)
* “Exception declarations may occur in the visible part of a package specification (where most kinds of declaration are allowed), but since they are not data objects they may not occur as procedure parameters.” (65)
* “An exception, whether predefined or user-defined, may be raised anywhere in the scope of its declaration by a **raise** statement.” (66)
* “Within an exception handler, the raise statement can be given without an exception name. . . . The effect of this statement is to reraise the exception which caused the handler to be entered (even if that exception cannot be named).” (66)
* “A predefined exception is raised implicitly if the corresponding fault is detected in the program. An exception, whether predefined or user-defined, may also be raised implicitly by a call of a subprogram. This happens if an exception arises during execution of the subprogram body and there is no handler for it within the subprogram.” (67)
* “Although exceptions can only be explicitly raised in the scope of their declarations, the implicit raising of exceptions can occur in situations outside that scope, as a result of propagation.” (67)
* “Whenever an exception is raised, the current block is executed no further, and a handler is executed instead.” (67)
* “A handler for an exception, to carry out the specific recovery action after it has arisen and before normal processing can continue, may be written in any block in the scope of the declaration for the exception. Exception handlers are always written at the end of a block: the word **exception** is put after the last normally executable statement, and introduces the exception part of the block. This consists of a series of handlers, written like alternatives in a **case** statement, with exception names as the selectors. There can be several handlers for one exception, in different blocks.” (67)
* “Each particular handler is introduced by a **when** clause giving the relevant exception name (possibly several exception names if the same recovery action is to serve for more than one exception), then after the => sign, a sequence of statements forming the body of the handler.” (67)
* “The sequence of statements in a handler may be any statements allowed in the current block. In particular, if the block forms the body of a subprogram, the statements may refer to the parameters of the subprogram, and would normally contain a return statement. Further, if the subprogram returns a value (as in a function or operator) then each handler should finish with a return statement giving a value of the correct type. If an exception arises during execution of an exception handler, the action is as though the exception had arisen in the associated main block without the current handler.” (68)
* “While the several handlers in the exception part of a single block must of course refer to distinct exceptions, it is possible for there to be different handlers for the same exception in different blocks (whether disjoint or nested).” (68)
* “Whenever an exception arises . . . the normal course of execution of the program ceases and a handler is sought for the exception: one with the exception name in its **when** clause, or one introduced by **when others**. As soon as a matching handler is found, its sequence of statements is executed instead of the remainder of the block containing it.” (69)
* “In the first instance a matching handler is sought in the block currently being executed. If there is none (particularly if the block has no exception part), or if the handler itself detects an exception, then the exception is raised implicitly in the dynamically enclosing block. This is called propagating the exception. If the handler raises another exception (by an explicit raise statement or any statement which implicitly raises an exception) then this exception is propagated to the dynamically enclosing block.” (69)
* “Propagation is automatic through a compound statement to the enclosing sequence of statements. In the case of propagation through an **accept** statement, an exception is also raised in the associated task. This continues through nested statements until a matching handler is found or the sequence of statements is that of a main program or task.” (69)
* “If the exception arises (directly or by propagation) in the sequence of statements which form a subprogram body, then the dynamically enclosing block is that containing the current call of that subprogram—either a procedure call statement or a function call or operator call in an expression.” (70)
* “If an exception arises in a declaration, it is propagated back from the block containing the declaration to the statement or declaration that called it, until a statement is reached; a handler is then sought in the block containing that statement.” (70)
* “If the exception arises in the initialisation part of a package (and there is no local handler), then the unhandled exception is considered to have arisen in the package declaration, so is propagated to the enclosing block.” (70)
* “If the package is a separately compiled main unit, this prevents execution of the whole program: the initialisation cannot be carried out. If the package is a subunit, then the exception is propagated to the unit containing the corresponding stub from which it is propagated as a declaration. If the exception arises (directly or by propagation) in the sequence of statements which are the substance of an **accept** statement, then in addition to propagating the exception to the block of which the **accept** statement is part, an exception is also raised in the task which is currently in rendezvous through the corresponding **entry** declaration. This is at the point of the statement in the task which calls the entry.” (70)
* “If the exception arises in the body of a task, then an unhandled exception is not propagated: it is deemed to be **null**. The task is terminated and no further action taken.” (70)
* “There is inevitably a certain amount of overhead associated with exception handling, particularly to propagate an exception. The aim in Ada is to have no overhead in normal execution (in particular no overhead on entering a program unit containing exception handlers) but to tolerate some overhead after an exception has arisen. This ensures that performance is only lost when it is necessary to achieve safety.” (70)
* “A simple use of exceptions is to check for consistency as the program runs, by the use of assertions.” (71)
* “The most common places for assertions are on entry to subprograms, to check that the values provided for **in** parameters are suitable, on completion of loops, and after a set of alternatives (to state properties which should now hold). If the assertion is found during execution to be false, the sequence of statements containing it is not executed further, and the exception ASSERT\_ERROR is raised. Assertions may also be useful on exit from a procedure, to confirm that the results satisfy required relationships.” (71)
* “The following exceptions are defined in the standard language environment:
  + CONSTRAINT\_ERROR—The current value is inconsistent with the current requirement. This exception is raised whenever an assignment is attempted with a value out of the range for the variable, an array component is referenced with an index value out of range, a record component is referenced with a field selector for the wrong variant, an access type object is referenced with an access value of null or similar situations (for example when matching subprogram parameters).
  + NUMERIC\_ERROR—The current value is not within the implemented range. This exception is raised if the result of a calculation is toolarge [*sic*] or mathematically infinite. The actual limits for detecting this situation depend on the implementation, and there is no guarantee that the exception will be raised anyway.
  + SELECT\_ERROR—Execution is blocked by a select statement. This exception is raised if a select wait statement is met, with all its select alternatives closed, and no else part. Such a statement can never complete its execution.
  + STORAGE\_ERROR—Storage is needed which cannot be provided. This exception is raised if the program requires more storage space than is available, by execution of an allocator or use of dynamic storage in a task.
  + TASKING\_ERROR—The communication with the required task cannot occur. This exception is raised if an entry call is met, and the task containing the corresponding accept statement is terminated before it can accept the entry.” (72-73)
* “A program in Ada is written as a number of logical pieces, each of which covers one aspect of the problem. A package is the general form of one such logical piece; it defines a set of facilities which the rest of the program may use. These might either be specifically designed for the program currently being written, or already in existence because a similar problem has arisen previously. The fundamental idea is to separate the use of a facility from its provision. In this way a program can be arranged to show what is meant to happen without a clutter of detailed implementation.” (75)
* “Several parts of the program may need to refer to the same common data. The declarations for these variables and constants logically belong together, and such a collection is one form of a package. Any part of the program that needs to refer to this data can then do so, without having to repeat all the declarations.” (75)
* “A package may be given among the declarations within the program (when it applies like other declarations), or compiled separately as a library unit. In the latter case, any other compilation unit may refer to the library unit by giving its name in a context specification.” (77)
* “The entities declared in any package can be used elsewhere in the program after they have been declared, as determined by the following rules:
  + If the package specification is in the declarative part of some program unit, then its scope extends throughout that program unit, after the specification. (If the package needs a body, then that or the stub must also be in the same declarative part.)
  + If the package specification is reduced to a stub in the declarative part of some program unit, with the specification expanded as a separate compilation unit (which in this case is a subunit), then its scope extends throughout the program unit containing the specification stub, after the stub.
  + If the package specification is a separate compilation main unit, (i.e. not having a specification stub within another unit) then its scope extends over all other compilation units that specify the package name in their context specification (i.e. state that they must be compiled **with** the given package).” (78)
* “The entities declared in the package can be used in ordinary statements and declarations, with names like components of a record.” (79)
* “A package is also a good way to connect the definition of an application-dependent type with the objects of that type which the program can use . . . .” (79)
* “We frequently have a set of subprograms which are closely associated; there might also be some data or special types which they use. In this case the package is written in two pieces—a specification and a body. The purpose of this is to show clearly what information the user of the package needs to know, without burdening him with implementation details that may be irrelevant. The specifications of the subprograms give all the information needed to call them. The package body contains all the corresponding subprogram bodies.” (80)
* “There are some circumstances when we want a special type for the problem, and subprograms to work with values of that type, with a means of ensuring that no other operations are applied to the corresponding objects.” (81)
* “In this case we have the package specification and package body . . . , but the package specification is itself split into a visible part and a private part. The purpose is again to show clearly the information needed by the user of the package; the visible part contains the declaration of the special type and the subprogram specifications, while the private part gives details of the type and values for any constants needed, together with any representation specifications. Several private types may be declared in the same package.” (81)
* “Values of a private type can be assigned and compared with one another but none of their internal details can be used outside the package body.” (82)
* “Sometimes the values of the special type must be kept unique, so that at most one object has any given value. . . . The word **limited** . . . makes any objects of the given type eligible only for use as parameters to subprograms which work with that type.” (82)
* “A package specification can occur either among the declarations in a program unit or as a separate compilation unit. It makes a collection of entities available for other parts of the program to use.” (82)
* “The main part of a package specification is its visible part, which gives the public information about the entities it contains. A package specification may also contain a private part, which gives further details about types and constants. (The private information is needed by the translator, but not by the programmer using the package.)” (82-83)
* “All kinds of program entity can be declared in a package: constants, variables, types, subprograms, exceptions, tasks and inner packages (for more closely associated entities).” (83)
* “As a package is a program unit, it may contain smaller program units and be itself contained in a larger program unit, corresponding to the logical structure of the problem.” (83)
* “A package body is distinct from a package specification; every package needs a specification, but sometimes a body is not necessary. If the package provides any subprogram, task, or inner package, their specifications must be collected in the package specification, and their corresponding bodies written in the package body. If the package contains any variables whose initial values must be calculated by executing statements, the initialisation must be written in the package body.” (83)
* “A package body need not occur immediately after its specification—it may even be written as a separate compilation unit. The body of a package consists of a declarative part and an optional executable part. The declarative part contains the bodies of all subprograms specified, and also any other declarations they need: local variables or procedures, etc. Anything declared in the body which was not in the specification part is entirely restricted to the body: it cannot be used in any other part of the program.” (83)
* “Subprograms in the [package] body may be used to implement tricky aspects of the package—without risk of misuse or interference by other parts of the program. . . . In this way, special or privileged actions may be confined to restricted sections of the program. This feature of Ada makes it possible to ensure safe use of potentially dangerous facilities, for example in library packages.” (84)
* “Variables declared in a package specification or body retain their values through-out the lifetime of the package—their values are not lost between calls of subprograms in the package. These are sometimes called ‘own’ variables (of the package). Those in the body are available for communication between one of the subprograms and another.” (84)
* “The executable part of the package body is executed when the package declaration is elaborated, and is used to initialise objects in the package (whether declared in the specification or the body). The body may also contain exception handlers, in case an exception during initialisation; such a handler does not apply after initialisation of the package.” (85)
* “It is inherent to Ada that the tasks are expected to ooperate, being aware of one another’s existence, and being designed to work with one another to solve an overall problem. If several tasks have to share to [*sic*] resource, they are expected to be programmed to use the resource responsibly; in particular, if any task acquires a resource that other tasks would need, it should release it in a reasonable time. Failure to do this would be a fault in the program, to be detected and put right before operational use.” (86-87)
* “Every task in Ada is written in the declarative part of some enclosing program unit, which is called its parent. The execution of the parent determines the overall start and finish of execution of the task. If several tasks are written in the same declarative part, they are executed in parallel with one another (and in parallel with the body of the parent unit). We call these sibling tasks. The parent unit may be a subprogram, a declare block, a package or another task.” (87)
* “Whenever the parent unit comes to be executed, all the tasks in is [*sic*] declarative part are started, and each is executed in its own sequential order, independent of the order of the others (unless there are explicit statements to relate them). Each task may came [*sic*] to the end of its statements and thus finish its execution, or it may be brought to an end by the influence of other tasks. Only when all the tasks declared in the parent unit have finished (and the parent itself has reached its end) is the parent deemed to have finished its execution. Thus the execution of tasks is fully nested.” (87)
* “The normal structure of a multi-task program is to write the several tasks that do the actions required, and declare them as siblings in the same parent; the body of the parent is responsible for overall control, in particular to ensure that the tasks are running properly, and to close them down when necessary.” (87)
* “Passive tasks provide some service to the others, such as buffering messages between two active tasks. Each passive task may provide a set of services (with appropriate names), and the active tasks call for these services as they need to use them.” (87-88)
* “A task is written with a specification and a distinct body (like a package).” (88)
* “The specification gives the task name and lists any services it provides. Services are specified like procedures, but introduced by the word **entry**. Sibling tasks (or the parent body) may call for these services. The body gives local declarations and the sequence of statements to implement the services provided by the task, when it accepts a call to on of its entries.” (88)
* “To be effective, tasks must be able to communicate with one another. In accordance with the normal visibility rules of Ada, the statements (and declarations) inside a task body may refer to entities declared outside it, for example other declarations in the parent unit. Thus one task may refer to entities declared in the specification of a sibling task, as well as other declarations of its parent.” (89)
* “An entry may have parameters, for communicating data values between tasks.” (89)
* “There may be faults during the execution of a task (as in any other part of a program), and the mechanism of exception handling allows appropriate actions to be taken after detected errors. Additional modes of control are possible, however, by one task influencing another: typically a parent controlling its children.” (91)
* “Every task automatically has a special exception that may be raised by another (controlling) task. This may, for example, be used to extract the task from an infinite loop. The handler for this exception should release any shared resources acquired by the task, make a report if necessary, and safely close down any equipment associated with the task. The special exception takes the task name with the attribute FAILURE.” (91)
* “The normal method of communication between tasks in Ada is called a rendezvous. This means that two tasks coincide for a period of time, after which they resume their parallel execution. A rendezvous joins together a task calling for a service and the task providing that service; the caller is normally an active task and the provider a passive task.” (92)
* “The task calling for the service does so by executing an entry call statement (which is written like a procedure call statement, but with the entry name instead of a procedure name); this says it wishes to have the rendezvous. The task providing the service expresses its readiness to carry out the service by reaching an accept statement; this says it is ready for the rendezvous.” (92)
* “Whichever the tasks is first at the rendezvous, it waits for the other to arrive.” (92)
* “When both calling and accepting tasks have arrived, the rendezvous action takes place (as specified in the accept statement). At the end of the rendezvous, the calling task continues after the entry call statement and in parallel the accepting task continues execution after the accept statement.” (92)
* “If several tasks call the same entry, the accepting task carries out the rendezvous with the first to arrive. The accepting task usually contains a loop so that the accept statement is repeated. This has the effect of dealing with the entry calls one at a time, in their order of arrival.” (92)
* “The action to be taken at the rendezvous is specified in an accept statement (in the accepting task), and the specification for that task makes the rendezvous visible by declaring it as an entry. Every entry in the specification must have a corresponding accept statement (perhaps more than one) in the body. . . . Note that the calling task cannot be doing anything else during the rendezvous, so any actions involving variables of the calling task (through parameters) are properly synchronised.” (92-93)
* “The parameters of the accept statement allow data values to be communicated between two tasks, in both directions: **in**, (default) at the beginning of the rendezvous, and **out** at its end (possibly both).” (93)
* “As well as the rendezvous method, tasks may communicate by sharing variables. By virtue of the visibility rules, an object declared in the parent task may be used in the children, and this can lead to trouble. Distinct sibling tasks may refer to the same variable while they are executing in parallel, so that one task may be accessing it while another is changing it.” (93)
* “the junction of a calling task and an accepting task to form a rendezvous is not symmetrical, and it is useful to appreciate the implications of the difference.
  + The calling task must know the name of the accepting task (and its entry point), but not vice versa. Thus a task with entries offers a service to other tasks, limited only by the visibility rules.
  + An accepting task may wait for the first of several entries to be called, whereas a calling task must state one specific entry it wishes to call. Thus there is greater non-determinism in the accepting task.
  + Failure of a task in rendezvous is not passed from the calling to the accepting task, but is passed from the accepting to the calling task.” (94)
* “The normal pattern for Ada tasks then is for the declarative part of some program unit (the parent) to contain passive and active tasks, with the body of the parent controlling them. The passive tasks provide the facilities needed for communication between the active tasks.” (95)
* “From the point of view of the active task, an entry call is just like a procedure call apart from timing: a procedure is started immediately but an entry may have to wait for the passive task to be prepared to accept it. There are two programming features that may be used to deal with this, both varieties of **select** statement.” (95)
* “If a task makes an entry call as an ordinary statement, it waits (perhaps indefinitely) until the called task is prepared to accept that entry. We can use a select statement [in conjunction with a delay statement] to limit the period of time it is prepared to wait.” (95-96)
* “Conditionally calling a rendezvous allows a task that calls an entry to have an alternative action if the called task is not prepared to accept the call immediately.” (96)
* “An interrupt is considered in Ada to be a hardware-generated entry call. The programmer writes the necessary interrupt handler as an accept statement in a handling task. The Ada entry is linked with the appropriate interrupt entry address by giving its representation specification.” (97)
* “The priority of a task may be given by a pragma; the possible values for priorities depend on the target computer used.” (97)
* “Between the interrupt handlers and the higher levels of a program, it is often useful to have distinct tasks to deal with particular aspects of the low-level device control.” (98)
* “There can be many identifiers in an Ada program, denoting variables, constants, types, procedures, exceptions, tasks, etc., and each applies throughout a particular part of the program. This is true both for the predefined identifiers such as INTEGER and CONSTRAINT\_ERROR as well as for identifiers defined by the programmer such as X, I, OBJECT, ASSOCIATED\_TRACK. The meaning of an identifier is determined by the way it is introduced—usually in a declaration at the head of a block or module. The positions in the program where it may be used depend on the position of the declaration. The same applies also to the operator symbols such as + \* (which may be overloaded by the programmer), and to enumeration literals (which may be characters).” (101)
* “A program in Ada has a logical and a physical structure. These are closely related but not identical. The logical structure consists of ‘program units,’ which may be nested inside one another. The physical structure consists of ‘compilation units’ which are always separate from one another.” (101-102)
* “A program unit can be a subprogram, package or task. For each kind of program unit there may be a specification and a body. Any kind of program unit may contain units of the same or different kinds inside it. Each compilation unit is a distinct program unit, with an appropriate context specification at its head.” (102)
* “The program as a whole consists of a collection of compilation units, all separate, and each compilation unit consists of a program unit that may contain other program units inside it, nested to any depth. It is possible for the text inside one compilation unit to refer to other compilation units, provided that the other compilation units are mentioned in the introductory context specification.” (102)
* “Because it may happen that the same identifier is used for different purposes in different parts of the program, we need to be able to associate any identifier used in the program with the proper meaning. We say that a declaration is ‘visible’ in those parts of the program where its identifier can be used. The visibility rules ensure that there is no confusion: each use of the identifier has the meaning given by the corresponding declaration.” (102)
* “Each identifier (other than a statement label, block or loop identifier, or a loop parameter) which is used in a program must be declared somewhere in the program, or in the library packages it uses or the standard environment. The statement labels, block and loop identifiers and loop parameters are implicitly declared by their occurrence . . . .” (102)
* “For each declaration, there is a defined region of the program over which the declared identifier has the meaning given in the declaration; this is called the scope of the declaration. Within this scope, there is a particular region where the identifier may be immediately used, without any additional information. In this region, the declaration is said to be ‘directly visible.’” (104)
* “Within the scope but not in the region of direct visibility, the declaration is said to be hidden; it is possible by appropriate additional information (of various kinds) to arrange for one or more hidden declarations to be made visible. The identifiers they declare can then be used with the declared meanings.” (104-105)
* “The regions of the program over which a declaration is relevant, both for scope and for direct visibility, are related to the program structure.” (105)
* “Scopes are mainly determined by the block and module structure of a program. The block structure relates to subprogram bodies and declare statements; module structure relates to packages and tasks. Within the major units determined by the program structure, certain constructs introduce special regions relevant to enumeration types, records and procedure parameters.” (105)
* “The classical form of block structure, which is available in Ada, provides for declarations to be valid within the part of the program immediately following them. We have declare statements, which are blocks in the executable part of the program, and subprograms, which are blocks in the declarative part of a program.” (105)
* “The declare statement is the simplest form of block structure in Ada. The word **declare** introduces one or more declarations, which are followed by **begin** then a sequence of statements, possibly with some exception handlers, and finally the word **end**. Each of the declared identifiers may be used in any subsequent declaration in the block, and throughout the sequence of statements and exception handlers. Thus any identifier introduced in the declarative part of the block has that meaning for all the executable part of the block. Its scope is all the block after the declaration.” (105)
* “The identifiers that are introduced in the declarative part of any block must all be distinct (with the exception of overloaded enumeration literals and procedures). However, since the declare statement occurs in a positon where many identifiers may already be visible, there is the possibility that an identifier declared inside the block is the same as an identifier visible in the context of the declare statement. In this case, the declaration at the head of the block takes precedence for the statements in the block, hiding the outer declarations for the same identifier. So in general, if the program has a nested block structure with declare statements in the executable part of larger declare statements, an identifier used in the program is interpreted according to the smallest enclosing block in which it is declared. (It is possible to have a block without any declarations, in which the statement starts with the word **begin** rather than **declare**; such a construct allows a particular set of statements to be guarded by specific exception handlers.)” (105-106)
* “An outer declaration that is hidden by a declaration for the same identifier in an inner block is still in scope, but its full name must be used to identify it.” (106)
* “A procedure or a function declaration is another form of block structure, which is basically similar to a declare statement but has the added feature of formal parameters.” (107)
* “The formal parameters of the procedure or function are introduced in the subprogram heading, and have the same effect on the subprogram body as declarations which may occur between the heading and the word **begin** at the start of the executable part of the subprogram. Thus the identifiers introduced as formal parameters may be used in any subsequent declaration and throughout the sequence of statements forming the executable part. The formal parameters are allowed only to be data objects, that is constants or variables. (they may not be types or other program objects such as procedures, exceptions, packages or tasks.) The identifiers which are introduced as the formal parameters must be all distinct, from one another and from any local declarations. All these identifiers are visible in the subprogram body.” (107)
* “Whereas blocks in a program serve the primary purpose of limiting the visibility of identifiers declared in them, modules (whether packages or tasks) serve the complementary purpose of expanding the visibility of identifiers declared in them. Certain of the declarations in a module have a scope which is greater than the module containing them. Identifiers which may have such an expanded visibility must be declared in the public part of the module—the specification part. We call these the public declarations. All other identifiers declared in the module, namely those declared in the implementation part or the private part of the specification, can never be visible outside the module, and are treated in the same way as identifiers in a block structure. (Note however that there may be separately compiled subunits of the module body, which although physically separate are logically inside the module.)” (108)
* “Throughout the scope of the package declaration, its public declarations are also in scope but they are hidden. They may be made visible throughout any specified program unit (provided that there is no ambiguity )by a use clause. Making the public identifiers visible throughout a unit is called exposing them.” (108)
* “Whether or not there is a use clause, any public identifier in scope may be accessed throughout the scope of the module by component selection, since there is then no possible ambiguity. In a context where the module name is visible, a component selection (consisting of the module name, a period, then the public identifier) constitutes the name of the entity which was declared in the visible part of the module.” (109)
* “For most declared entities (including variables, constants, types, exceptions, modules) where the identifier stands alone, this requires that distinct identifiers must be used for the different entities declared together. This rule is relaxed for the values of an enumeration type: each individual enumeration type must have a distinct set of values, but the identifiers introduced as the values of one type do not have to be distinct from the identifiers introduced as the values of another type; such identifiers are said to be overloaded. The rule is also partially relaxed for subprograms, which can have parameters (and each subprogram has a specific number of parameters, which may be zero). Since their identifiers are always used with a list of parameters of specific types, it is possible for several subprograms to have the same identifier, provided that their parameters are sufficiently different. This is called overloading the subprogram . . . .” (110)
* “In each of the following, where several identifiers are introduced together, they must all be different:
  + Values of an enumeration type.
  + Components of a record type.
  + Entities introduced in a block or module (taking both specification and body together), apart from overloaded subprograms. These entities include formal parameters, local declarations, block and loop identifiers and statement labels.” (110)
* “Between any one set of declarations and another, an identifier may be repeated, and the rules of visibility determine which meaning is implied by any use of the identifier. It is illegal to use an identifier where no declaration for it is visible—it then has no meaning.” (110-111)
* “Each enumeration type declaration gives the set of identifiers (or characters) that are the possible values in the enumeration type. These identifiers or characters are known as enumeration literals; any such identifier is normally available for use in expressions of the appropriate enumeration type, provided that no other declaration is given for it. For any individual enumeration type, the possible values must of course all be distinct, but different enumeration types need not have disjoint possible values.” (111)
* “An identifier which is a possible enumeration value for more than one enumeration type in the same context is said to be overloaded. In this case, the identifier is not immediately available for use in an expression: it is excluded from the context in which it is overloaded. However, the enumeration type name (which can not be overloaded) is available for use, to introduce a qualified expression.” (111)
* “Within this qualified expression the enumeration value may be given, as any possible ambiguity has been eliminated. Thus the qualified expression constitutes a local context, in which the identifier gives the value in the specified enumeration type.” (111-112)
* “The identifiers of record components are not necessarily visible in the context of the declaration of either the record type or any of the record objects; however, they become visible in the local contexts of component selections and record aggregates. The type of the record must be visible for any possibility of access to components of an object of that type.” (112)
* “A record type declaration introduces the identifiers and types for its components. These identifiers are not directly visible in the context of the type declaration. An object of that record type may be declared wherever the type declaration is visible, and in that context the type identifier may not be redeclared. There may be an inner context in which the record type declaration is redeclared but the record object is still visible: in this context, all the constituents of the data object are hidden (as though it were of a private type).” (113)
* “The association of a component with a value may be by position or by name, using the same notation as for subprogram parameters. This is a special context where the component identifiers are visible on the left hand sides of name-associated components, but not on right hand sides or in position-associated components, for which the outer context applies.” (113)
* “In a context where both the object identifier and the type identifier are visible, components of the record may be accessed by ‘component selection.’ The record object name collowed by the component identifier constitutes the name of the component, and has all the properties of an object of the appropriate type—including the possibility of having components itself, if the component is a record within a record.” (113)
* “Within any context where a subprogram name is visible, the identifiers for the formal parameters of the subprogram are not directly visible, but may become visible in the local context of a subprogram call, for association by name with actual parameters.” (114)
* “A subprogram call consists of the procedure or function name followed by the list of parameter associations. This list constitutes a special context, in which the identifiers for the formal parameter of the subprogram are visible on the left hand sides of name-association parameters. They are not visible on the right hand sides of name-associations or in positional parameters.” (114)
* “A subprogram (procedure or function) is distinguished by the combination of its declared identifier and the properties of its parameters. The relevant parameter properties are their formal identifiers, modes (**in**, **out** or **in out**), and types, with the type of the result, if any. This combination is called its signature. As a consequence of the parameter properties being involved in the signature, it is possible for a single subprogram identifier to be ‘overloaded,’ so that it can be declared with different bodies for different formal parameters.” (115)
* “The signature of the subprogram is used (at compile-time) to find which is the appropriate subprogram declaration for each subprogram call, and is composed of the information which must be specified in the call. The resolution of subprogram overloading is done at compile time, so there is no run-time overhead.” (115)
* “In order to give this information unambiguously, each subprogram parameter must have its type determinable (implying the need for a qualified expression if overloaded literals are used, such as numerical constants with user-defined types).” (115)
* “The physically separate program units are called compilation units. Ada includes facilities for compile-time parameterization: as well as conventional run-time parameters for subprograms, both subprograms and packages may be written with compile-time parameters; they are then called generic program units.” (118)
* “An Ada program is written as a collection of separately compiled program units, with compile-time type-checking across boundaries. Each unit may provide a set of facilities, and may use the facilities of other units. This kind of program structure determines the order in which units must be compiled: definition before usage. The context in which a unit is to be used must clearly be specified first if the unit is sensitive to that context. Library units are context insensitive so can [*sic*] be compiled before any unit which uses them. Ada separates the implementation part of a module from its specification part, so the specification parts can be compiled in the required order but the implementation parts subsequently in whatever order is convenient.” (119)
* “A program in Ada is formed from a number of separate compilation units, which are translated individually and then linked together to form the target program load.” (119)
* “Although compilation units are translated separately, consistency checks are applied to ensure that their interfaces are compatible. . . . In order to achieve this degree of security at compilation time, the translation of a compilation unit is not done in isolation but is associated by the compiler with information from other compilation units which are relevant. Similarly, after a compilation unit has been compiled, information is maintained for subsequent use in the translation of other compilation units which may depend on it.” (119)
* “The fundamental idea in Ada is that each compilation unit, although compiled individually, is considered to be defined in some context which determines the visibility of declarations between one compilation unit and another. A compilation unit may use identifiers declared in that context, but of course need not do so.” (120)
* “A basic distinction is drawn between compilation units that are effectively at the outermost level of declarations in a program, and those which are not. The former are called library units, the latter sub-units. A sub-unit must state which already existing unit effectively contains it; this is called its parent unit. A library unit makes no assumptions about declarations in the context of its definition; it could then be taken into other contexts as required. Tasks may only be sub-units; subprograms and packages may be library units or sub-units.” (120)
* “The identifiers declared in a separately compiled unit are visible in the context of the declaration and any context in which they are exposed, but nowhere else.” (120)
* “A library unit has an effective context which is the whole program, i.e. all other library units. A subunit has a context which is which is [*sic*] explicitly specified as its parent unit: another unit (either a library unit or another subunit). Thus a program as a whole consists of a number of compilation units, of which some are library units and the rest are subunits.” (120-121)
* “Library units are effectively global for the whole program, but subunits belong to particular contexts.” (121)
* “The parent unit must always contain the complete specification of the subunit, whether it is a package, task or subprogram.” (121)
* “If the subunit is a module, the parent unit must contain the specification explicitly.” (121)
* “The default context for a compilation unit is the package STANDARD, allowing the unit to access the predefined declarations for ITNEGER, NATURAL, BOOLEAN, FLOAT, etc.” (122)
* “A compilation unit may access the declarations of other compilation units if they are explicitly stated in the context specification at its head, in with clauses.” (122)
* “Any number of library units may be specified in the context specification. All the public declarations of these units are brought into scope for the unit being compiled.” (123)
* “The separate compilation units which form an Ada program may be submitted for translation in any logical order.” (124)
* “The underlying principle is that if one compilation unit requires information from another compilation unit, then the unit providing the information must be translated before that requiring the information.” (124)
* “The with clause is the main influence on order of compilation. For each compilation unit governed by such a clause, the list units’ specifications must have already been compiled before the unit naming them in a with clause, so that their declarations may be accessible.” (124)
* “If a compilation unit does not have a context specification, then there is no constraint on its order of compilation. This allows library modules to be compiled in advance of programs which use them.” (124)
* “The parent unit of a subunit contains the specification of the subunit, so may use all its facilities. The other declarations in the outermost level of the parent unit may also be used in the body of the subunit, just as though it were textually present in the parent.” (124)
* “If there is a change to the program . . . [and] one or more compilation units are immediately affected and the new versions of them must be recompiled.” (125)
* “There are circumstances in which other compilation units not immediately concerned are nethertheless [*sic*] possibly affected; these must also be recompiled, to make sure that all the consistency checks for an Ada program still hold.” (125)
* “The guiding principle is to maintain software interfaces as far as possible, and explicitly check changes. If a change does not affect an interface, then parts of the program beyond the interface need no attention.” (125)
* “If a separate subprogram is changed so that its body is different but its specification remains the same, then recompilation of the subprogram has no effect on the other compilation units of the program.” (125-126)
* “However if a subprogram is changed and given a different specification in any respect then all other compilation units which refer to this subprogram are invalidated, and must be recompiled before the program load can be linked together again.” (126)
* “Similarly, if a module body is changed, there are no consequential effects, but if a module specification is changed in a recompilation, then the corresponding module body must also be recompiled.” (126)
* “Any compilation unit carrying a **with** clause is also invalidated by recompilation of any of the units mentioned in this context specification. A subunit is invalidated by recompilation of the unit containing its stub.” (126)
* “A universally valuable technique in programming is to write once a piece of code which will be used many times. This underlies the ideas of loops, procedures and libraries in Ada as in other programming languages; generic program units are another case of the idea in Ada. A generic program unit is a compile-time parametric subprogram or package, that can be written once (usually as a library unit) and then used as many times as needed, as the program is being compiled. The generic program unit forms a model or pattern that may involve compile-time parameters.” (127)
* “A generic unit is not itself directly executable, but instances of the model have the properties of the appropriate program unit. Only subprograms and packages may be made generic.” (127)
* “Generic program units are pieces of program which define patterns for subprograms or packages, such that the patterns can be filled out to form proper subprograms or packages by the program translator. The act of making a particular instance of the unit given generically is called instantiation.” (127)
* “Note that all the processing to do with generic program units takes place at translation-time. After a unit has been compiled, there is no difference in meaning between one which was written individually and one which was an instance of a generic unit.” (127)
* “Generic parameters offer a wider range of possibilities than subprogram parameters, because they are handled at translation time rather than execution time. Generic parameters may be data values or data objects, or types or subprograms.” (127)
* “Whenever an instance of a generic unit is required in a program, actual entities must be specified for the generic parameters. This corresponds to giving actual subprogram parameters in a subprogram call, but takes place at compile time rather than run-time.” (127)
* “When the generic program unit is instantiated, actual parameters are given which replace the generic parameters in the body. There are two major differences from the formal parameters of a procedure:
  + The substitution is at compile-time (so that generics are in this respect more like macros than procedures); consequently
  + The kinds of entity which can be parametrized [*sic*] generically are more varied than those for procedures.” (129)
* “If a generic parameter is given as a data item, it may be of mode **in** (the default), meaning a constant determined at compile-time, or of mode **in out**, meaning an object whose identity is determined at compile-time.” (129)
* “If a generic parameter is given as a type, then the generic formal parameter must indicate what kind of type it is, to permit strict checking. The kinds of type recognised are indicated in the heading of the generic unit when introducing the identifier:
  + ( <> ) -- discrete (enumeration or integer)
  + range <> -- integer
  + delta <> -- fixed point
  + digits <> -- floating point
  + for scalar types, or the actual definition of an array type, access type, or private type.” (129)
* “If a generic parameter is given as a subprogram, the full specification of the subprogram must be stated (introduced by the word **with**). The parameter may be provided with a default, in which case the corresponding parameter may be omitted when the program unit is instantiated. The default is shown by writing **is** after the subprogram specification. Two forms of default are allowed: either the name of a subprogram visible at the position where the generic program unit is declared, or the subprogram with the same name as the formal parameter, visible at the position where the generic program unit is instantiated. For the first default write the required name after **is**, for the second write **is** <>.” (130)
* “A principal use for generic program units is to specify the pattern of a package for dealing with entities of different types, where similar but not necessarily identical actions are needed.” (130)
* “A generic program unit is written by prefixing a generic part to a subprogram specification or a package specification. The generic part gives the generic parameters, which may then be used within the specification and corresponding body. The generic program unit would usually be a library unit, in which case the corresponding body would be written as another library unit. However, it may be written wherever a declaration is allowed, in accordance with the ordinary Ada rules about specification and body.” (130-131)
* “Ada allows the programmer to deal with input/output on any peripherals which may be connected to a computer . . . .” (138)
* “No special features or additional program structures are needed to deal with these [communication stimuli] in Ada, but the programmer needs the ability to specify data structures and their representations.” (138)
* “The facilities required for input/output are achieved by making extensive use of the module (package and task) features, rather than by having many special features in the language.” (138)
* “Input/output is necessarily machine dependent, and the aim of this style of programming in Ada is to keep the machine dependency in as limited a context as possible.” (138)
* “Each input/output operation consists of a type conversion between an internal value and a value at the shared interface between the computer and the peripheral. The program has to treat concrete representations, and relate the desired internal types to the hardware-determined types of the physical devices.” (138-139)
* “The Ada Language Reference Manual defines such common Input/Output facilities in two packages: INPUT\_OUTPUT (a generic package) for general handling of files, and TEXT\_IO (a package which contains inner generic packages) for handling streams of characters intended for communicating with people.” (139)
* “The physical configuration of peripherals is specified by values in a data structure determined by the computer architecture (e.g. channel number, subchannel number, device address; or unibus address of principal control and status word). The hardware/software interface is specified by another data structure determined by the device controller (e.g. structure of control registers, layout of buffer registers).” (139)
* “Ada treats input/output as the transfer of sequences of data values between the program and some external sources or destinations. There can be a number of logically independent sequences being operated on concurrently. They are called files, and are distinguished as values in special types which are defined in the package INPUT\_OUTPUT.” (139)
* “Ada requires the programmer to state two fundamental facts about each input/output activity, which of their nature seem to imply a restriction on the kinds of input/output that can be carried out. Each input/output sequence (file) has a particular direction and a particular element-type.” (139)
* “The direction indicates the permissible transfer operations: either reading data values into the program from the external source, or writing data values from the program to the external destination, or both reading and writing data values between the program and the external file.” (139)
* “A file available for read-only access must be given as an IN\_FILE; a file available for write-only access must be given as an OUT\_FILE; a file available for both reading and writing must be given as a INOUT\_FILE. The purpose of this distinction is to allow checks at compile time that reading or writing are intended.” (140)
* “Specifying the element-type for a file is the way of stating the type of the data values which will be transferred. Note that this has to be fixed in the program and cannot change during execution. This rule is a consequence of the ‘strong typing’ style of programming in Ada. Since each data object has to have a particular type known at compile-time, any data object whose value is obtained by input (or which gives its value for output) must have a particular type, and consequently all the data values in the sequence which might be transfered [*sic*] to it during input/output must be of the same type.” (140)
* “Although the element-type for a file must be fixed, there is no restriction on using the file to hold values of different types, provided that they can all be represented as sequences of elements of the common element-type. It is then a matter of programming appropriate conversion procedures to be used in association with the actual data transfer operations.” (140)
* “Ada does not specify which particular external source or destination is to be used for each file: the way of identifying the external partners and their connection is so machine-dependent that it has to be programmed explicitly.” (140)
* “The package INPUT\_OUTPUT in Ada presumes a system in which files have permanent names, used to identify them when several programs may access the same file. The package is generic, with the type of the elements in the file as the parameter. For any particular element-type . . . the package must be instantiated . . . .” (141)
* “If it is impossible or illegal to create a file with the name given, an exception is raised. STATUS\_ERROR means that the file variable is already in use, so it is not available for a new file; NAME\_ERROR means that the permanent name specified is already in use or is not allowed for some reason.” (142)
* “If the OPEN operation cannot be carried out, an exception is raised. . . . STATUS\_ERROR means that the file variable is already in use. . . . NAME\_ERROR means either that there is no file in existence with the specified name, or that it exists but access to it is prohibited.” (142)
* “No other operations may be carried out on the file until it has been opened (creation implies opening)—if any is attempted, the exception STATUS\_ERROR is raised.” (142-143)
* “Other exceptions may be raised during the course of input/output operations once a file has been opened: DEVICE\_ERROR means that there are difficulties in the underlying system, which prevent the operation being completed; USE\_ERROR means that the specified operation is physically impossible or prohibited.” (143)
* “CLOSE causes the file . . . to be closed but to remain in existence with its permanent name for future use. DELETE causes the file . . . to be closed and all information about it deleted, including its permanent name. The permanent name of a file may be discovered by using the function NAME. . . and the current state (open or closed) may be tested using the function IS\_OPEN.” (143)
* “The primary operations of transfering [*sic*] data to or from a file are done by the procedures READ and WRITE. Each transfers a single element of the appropriate type.” (143)
* “Note that the use of READ and WRITE is checked at compile time for compatibility of the direction of transfer: READ may be used only with an IN\_FILE or INOUT\_FILE; WRITE may be used only with an OUT\_FILE or an INOUT\_FILE.” (144)
* “In the simplest case, each WRITE adds one new element to the file, until an eventual CLOSE marks the end of the file. If a READ operation is executed when there are no more elements in the file, the exception END\_ERROR is raised.” (144)
* “The READ operation implies checking that the next element in the file is a proper value of the declared element-type for the file. If the element is not of the proper type then the exception DATA\_ERROR is raised.” (144)
* “In the presence of repositioning operations, each READ or WRITE applies to the current ‘next’ position in the file for the appropriate direction of transfer. . . . If the position has been set to a value outside the range of existing positions for the file, a READ operation will raise the END\_ERROR exception In [*sic*] similar circumstances a WRITE operation is carried out with automatic adjustment of the file size. Any intermediate positions have indeterminate values—a subsequent READ might give an arbitrary value or might raise DATA\_ERROR.” (144)
* “At any stage during processing, there is a current read position and a current write position, which establish the next element to be read or written. Unless specific instructions are given, a series of read or write operations apply to consecutive positions in the file.” (145)
* “The package INPUT\_OUTPUT provides procedures to manipulate the position of a file, but recognises that particular operations may be physically impossible. There is no systematic distinction between files which can or cannot have particular operations applied to them: there is no check at compile-time on the use of positioning procedures, but if such a procedure is executed on a file which cannot substain the operation, the exception USE\_ERROR is raised.” (145)
* “The positions in the file are counted from 1, with one position occupied for each element of the file.” (145)
* “When a file has just been created, its size is 0 and its position is 1. As each element is written to it, its size and position both increase by 1. . . . When a file that already exists is opened, its size is the same as when it was last closed; the newly opened file is set at position 1.” (145)
* “The position of a file is changed by reading from it or writing to it, or (if the external partner has the ability) by reposition. The procedures SET\_READ and SET\_WRITE change the position number, making the element at the specified position be the next one available to be read or written.” (145)
* “The procedures RESET\_READ and RESET\_WRITE set the current read and write positions to 1 . . . .” (146)
* “Ada includes a package providing facilities for input/output of strings of characters, such as are needed for communication between a computer and an operator or programmer.” (146)
* “The [text] package presumes that a particular source and destination are by convention used as standard input and standard output for each program. The identification of these standard partners is not specified by Ada, but is presumed to be arranged by the implementation, in other words by the operating system. The package is called TEXT\_IO, and uses the facilities of the standard file package INPUT\_OUTPUT, with the addition of extra facilities to cover textual layout and input/output of values in the various data types in Ada.” (146)
* “The standard input and output streams are automatically opened when the program begins execution. If any other files are to be used with TEXT\_IO, they must be opened or created using the facilities of CHARACTER\_IO, with IN\_FILE and OUT\_FILE.” (146-147)
* “All text output is done by the procedure PUT, which can take parameters of different types for the values to be output, together with parameters defining the layout.” (147)
* “The procedure PUT can be used directly to output characters, strings, or truth values (BOOLEAN). The appropriate inner package must be instantiated before it can be used for integers, floating point, fixed point or enumeration types.” (148)
* “All text input is done by the procedure GET, which has an **out** parameter (i.e. variable) to receive the value input. The type of the parameter determines what kind of conversion (if any) is done as the value is taken in. GET takes the characters from the input stream to make a value of the required type, automatically going over line boundaries as necessary.” (148)
* “If an error or inconsistency is detected during input, an exception is raised. If the characters read are not consistent with the type of the parameter given then DATA\_ERROR is raised. If the value of an integer or approximate number read is outside the range implemented, then NUMERIC\_ERROR is raised. If the value is outside the range constraint of a subtype, then CONSTRAINT\_ERROR is raised.” (149)
* “This is free-format input, and the values are not dependent on the number of spaces or lines between them.” (149)
* “This [text] package provides facilities for fixed format input, but only of character strings.” (150)
* “There are no automatic facilities for discovering what type this string might refer to (and because of overloading, the value may not be unique); but when the program has decided what type is required—say INTEGER—the appropriate value in that type can be obtained by using the attribute VALUE in a conversion function: I := INTEGER’VALUE(S);” (150)
* “The printable characters are considered to be arranged as a series of lines of text, each line consisting of a number of columns. Line lengths may be fixed or may be indicated by the control characters.” (150)
* “The effect of the [text] package is defined only for the printable characters: Ada does not specify what happens with control characters.” (150)
* “If the column number gets beyond the line length, LAYOUT\_ERROR is raised.” (151)
* “The procedures PUT and GET can work with either an implied file or an explicitly specified file. If the file is specified, it must have been declared as an IN\_FILE for GT, and an OUT\_FILE for PUT. (Note that there are no INOUT\_FILES with TEXT\_IO.) If the file is not specified, an implied input or output stream is used.” (151)
* “The implied streams are initially set to the standard input and output streams, but subsequently may be set by procedures SET\_INPUT and SET\_OUTPUT.” (151)
* “The type specifies more than the range of possible values—it also determines the way the data values are represented, and which operators are valid for a data item. Ada has a fixed set of operator symbols; their meanings with certain elementary types are predefined, but the program may give additional meanings for an operator symbol by defining it with other types.” (163)
* “Every type has a representation (such as a series of distinct bit patterns for an enumeration type). Different types sometimes have the same representation, but are nevertheless kept distinct by the translator. The representation is usually chosen by the translator, but in some circumstances (particularly in connection input/output) it may be necessary to specify explicit details.” (163)
* “The fundamental ‘strong typing’ idea means that programs should contain distinct types for all the different kinds of data item with which they deal. Ada allows new types to be introduced easily. Derived types take their properties from already existing types, so that there are similar values in the new type, and the operations applicable to them are the same. Abstract data types are more general, in that the new type has an explicitly specified set of properties—the programmer gives the complete set of operations applicable to values of the new type. Variant records are more flexible, in that different values of the type may have different components.” (164)
* “We can construct a new type as a derivative of another, so that the new type ‘inherits’ all properties of the original one (in particular its literals and the operators which may be applied to it) yet nevertheless the values in the new type are treated as distinct, and assignments between different types are prevented.” (164)
* “A derived type is introduced by writing the word **new** before the original type name, and then possibly putting a constraint. The derived type is given a name by introducing it in a type declaration: **type** NEW\_TYPE **is new** OLD\_TYPE; to allow objects and parameters to be introduced for this type.” (165)
* “Any type may be used as the basis for derived types, including arrays and records.” (165)
* “. . . in Ada using an abstract data type, in which the type is given a name and operations on objects of the type are defined, but nothing is shown about the implementation details for the type.” (165)
* “An abstract data type in Ada is said to be private , and is declared thus:” **type** NEW\_TYPE **is private**; (166)
* “The private type must always be introduced in the visible part of a package module, and the same visible part must contain the declarations for all the other information related to that type—any constant values of the type that may be used, and the subprograms that can manipulate objects of that type. Several abstract data types may be introduced in the same module if required, by appropriate private type declarations.” (166)
* “Objects of a **limited private** type may not be assigned (copied) using assignment statements, nor may they be compared with one another, unless suitable relational operators are given in the same visible part as the type definition.” (167)
* “A record may have variants in which certain components are only present in particular circumstances.” (167)
* “Variants allow the value of the record to take account of the possibility that certain components may sometimes be ‘not applicable’ or ‘not relevant.’” (167)
* “In general, a record consists of a fixed part (containing one or more components) and possibly a number of variant parts, which are mutually exclusive. Each variant part contains any number of components (which may themselves be records with variants). The several variants are distinguished by the value of a special component called the discriminant . . . . The discriminant is introduced as a special component in the heading of the type declaration; all the variants are collected together in a **case** clause which uses values of the discriminant to select each particular variant.” (168)
* “A record may contain several variable-length arrays, but the bounds on their indices must be simple (not expressions), and all variable bounds must be written in the heading as discriminants. This is to ensure that each object of the record type is allocated the right amount of space when it is declared.” (169)
* “A range constraint may be given for [an integer type and] a floating point type and must be given for a fixed point type. It is needed to determine the representation for values of the type. The actual range implemented will cover the specified range, but need not exactly match it. A value outside the specified range will raise CONSTRAINT\_ERROR. A value outside the implemented range will raise NUMERIC\_ERROR.” (172)
* “For a real type (i.e. fixed or floating point), we use accuracy constraints. The sets of abstract values concerned are approximations to mathematical real numbers. A fundamental distinction is made in Ada between approximations which are relative and those which are absolute. This is shown by the accuracy constraint given in the declaration. You either specify the number of decimal digits of precision needed (indicating that the approximation is relative) or the magnitude of the acceptable error bound for the approximation (indicating that the approximation is absolute).” (173)
* “Every object declared with absolute precision must have its range specified.” (174)
* “There are no predefined fixed point types in Ada: any absolute precision quantities needed must have an explicit type definition.” (174)
* “A type defined as a record may have discriminant constraints.” (174)
* “The discriminant constraint is written after the name of the type (or subtype) to which it applies, and consists of an aggregate specifying the required value of the discriminant.” (175)
* “A discriminant constraint may also be applied to an access type, where the base type is a record with variants.” (175)
* “A type defined as an array may have index constraints. These give the lower and upper bounds (and hence the type) of each index of the array. Index constraints are particularly important with array parameters for subprograms, and for arrays whose sizes are not fixed in the type definition.” (176)
* “Since array indexes may be of any discrete type (not just integers), an index constraint must be of the same type as is given in the array definition.” (176)
* “An index constraint is written after the name of the type (or subtype) to which it applies, and consists of one or more discrete ranges that specify the bounds for the array indices.” (176)
* “In general, if a derived type definition is given with constraints, the effect is as though a new type had been set up by definition, then the constraints applied to that.” (178)
* “An access type is used for a collection of objects, all of the same type (usually a record) where the number of objects currently relevant can change during the execution of the program, and it is necessary to note relationships between one object in the collection and another. . . . The type of the individual objects in the collection is called the base type.” (179)
* “The access type both establishes the collection, and introduces the possibility of values denoting particular members of the collection. Each access type establishes a collection of a particular base type; any value of that access type can denote one of the objects currently in that collection, or be **null** denoting no object at all.” (179-180)
* “A new object in the collection is formed by allocation in an expression. An allocator consists of the word **new** followed by an appropriate typed expression to give the value of the allocated object. Each new object is different from all previously allocated objects in the collection, even though the value of the object may be the same as that of another object. The allocator both creates the new object, and gives an access value denoting that object as its result.” (183)
* “In principle, the number of objects in a collection for an access type may change during execution without limit. In practice, because a finite amount of storage must be reserved for the collection, there is a limit to the number of new objects which may be allocated in each collection.” (192)
* “If the program tries to allocate an object when there is no space available for it in the collection, then the exception STORAGE\_ERROR is raised.” (192)

Barnes, J. G. P. (1989). *Programming in Ada*, 3rd ed. Wokingham, England: Addison-Wesley Publishing.

* “Ada is a high level programming language originally sponsored by the US Department of Defense for use in the so-called embedded system application area.” (1)
* “The story of Ada goes back to about 1974 when the United States Department of Defense realized that it was spending far too much on software. It carried out a detailed analysis of how its costs were distributed over the various application areas and discovered that over half of them were directly attributed to embedded systems.” (1)
* “Further analysis was directed towards the programming languages in use in the various areas. It was discovered that COBOL was the universal standard for data processing and FORTRAN was a similar standard for scientific and engineering computation. Although these languages were not modern, the fact that they were uniformly applied in their respective areas meant that unnecessary and expensive duplication was avoided.” (1)
* “The situation with regard to embedded systems was, however, quite different. The number of languages in use was enormous. Not only did each of the three Armed Services have their own favourite high level languages, but they also used many assembly languages as well. Moreover, the high level languages had spawned variants. It seemed that successive contracts had encouraged the development of special versions aimed at different applications. The net result was that a lot of money was being spent on an unnecessary number of compilers. There were also all the additional costs of training and maintenance associated with a lack of standardization.” (1-2)
* “It was therefore realized that standardization had to be established in the embedded systems area if the costs were to be continued. The ultimate goal was, of course, a single language.” (2)
* “The first step in moving towards the development of a single standard was the writing of a document outlining the requirements. The first version was known as Strawman and was published in early 1975. After receiving comments from various sources it was refined and became Woodenman. A further iteration produced Tinman in June 1976. This was quite a specific document and identified the functionality required of the language.” (2)
* “At this point the requirements document was revised and reorganized to give Ironman. Proposals were then invited from contractors to design a new language starting from one of the recommended bases [Pascal, PL/I, and ALGOL 68]. Seventeen proposals were received and four were chosen to go ahead in parallel and in competition. The four contractors with their colour codings were CII Honeywell Bull (Green), Intermetrics (Red), Softech (Blue) and SRI International (Yellow). The colour codings were introduced so that the resulting initial designs could be compared anonymously and hopefully therefore without bias.” (2)
* “The requirements were also revised [during the competition] in the light of feedback from the initial designs and became the final document Steelman.” (3)
* “The final choice of language was made on 2 May 1979 when the Green language developed at CII Honeywell Bull by an international team led by Jean Ichbiah was declared the winner.” (3)
* “The DoD then announced that the new language would be known as Ada in honour of Augusta Ada Byron, Countess of Lovelace (1815-52). . . . In a very real sense she was . . . the world’s first programmer.” (3)
* “. . . the preliminary Ada design was revised and this resulted in the publication in July 1980 of the first definitive version of the language. It was then proposed to the American National Standards Institute (ANSI) as a standard.” (3)
* “The ANSI standardization process occupied over two years and resulted in a certain number of changes to Ada. Most of these were small but often of subtle significance especially for the compiler writer. The ANSI standard *Language Reference Manual (LRM)* was finally published in January 1983 . . . .” (3)
* “ANSI then proposed to the International Standards Organization that Ada become an ISO standard. . . . Ada became ISO standard 8652 in 1987.” (3)
* “Ada is a large language since it addresses many important issues relevant to the programming of practical systems in the real world. It is, for instance, much larger than Pascal, which, unless extended in some way, is really only suitable for training purposes (for which it was designed) and for small personal programs.” (9)
* “Some of the key issues in Ada are
  + Readability—it is recognized that professional programs are read much more often than they are written. It is important therefore to avoid an over-terse notation such as in APL which, although allowing a program to be written down quickly, makes it almost impossible to be read except perhaps by the original author soon after it was written.
  + Strong typing—this ensures that each object has a clearly defined set of values and prevents confusion between logically distinct concepts. As a consequence many errors are detected by the compiler which in other languages (such as C) would have led to an executable but incorrect program.
  + Programming in the large—mechanisms for encapsulation, separate compilation and library management are necessary for the writing of portable and maintainable programs of any size.
  + Exception handling—it is a fact of life that programs of consequence are rarely perfect. It is necessary to provide a means whereby a program can be constructed in a layered and partitioned way so that the consequences of unanticipated events in one part can be contained.
  + Data abstraction—as mentioned earlier, extra portability and maintainability can be obtained if the details of the representation of data can be kept separate from the specifications of the logical operations on the data.
  + Tasking—for many applications it is important that the program be conceived as a series of parallel activities rather than just as a single sequence of actions. Building appropriate facilities into a language rather than adding them via calls to an operating system gives better portability and reliability.
  + Generic units—in many cases the logic part of a program is independent of the types of the values being manipulated. A mechanism is therefore necessary for the creation of related pieces of program from a single template. This is particularly useful for the creation of libraries.” (9-10)
* BNF Syntax Description p. 415-425

Wilson, L. B., & Clark, R. G. (1988). *Comparative programming languages*. Wokingham, England: Addison-Wesley Publishing.

* “An Ada compiler must be validated using a specially constructed suite of test programs before it can be called an Ada compiler.” (8)
* “The US Department of Defense, the largest user of computers in the world, decided to sponsor the development of a new programming language in the mid 1970s. The main reason for this was that it was dissatisfied with the conglomeration of different languages used in its computer systems and it wanted a standard programming language for embedded computer systems. The latter are computer systems used to control part of a larger system, such as an industrial plant, an aircraft or a hospital life support system. These applications are normally very large, highly complex, contain a high degree of concurrency and change with time. The other important requirement of such systems is high reliability combined with the ability to recover from errors.” (52)
* “Although Ada is Pascal based, it is a much larger and more complex language. It not only extends Pascal constructs but contains features that have no analogue in Pascal. One of the key features of Ada is the package, which is designed for the description of large software components. A package has affinities both with the class concept of SIMULA 67 and the module of Modula-2. It contains type definitions and data objects in addition to operations for manipulating these objects. So, like the class concept, it is a further abstraction of the well-known procedure idea. Packages aid information hiding because both the representation of data and the implementation of the operations may be hidden from the user. Ada also has the notion of a library of packages, thereby enabling the programmer to create a system from combinations of existing packages, rather than writing a new program from scratch.” (52)
* “Apart from a broad range of built-in data types, there is a powerful set of data typing mechanisms. Ada is a strongly typed language, which means that all type checking can be done at compile time. Sequencing and control statements are similar to those used in Pascal . . . but the **goto** statement is only given in a restricted form. An interesting feature of Ada is the loop structure which includes an **exit when** construct. This allows general loops with a test at the beginning, like the traditional **while** statement, at the end, like the **repeat…until** statement, or in the middle of the loop. Procedures and functions are again similar to those in Pascal, but the parameter-passing mechanisms using **in**, **out** and **in out** correspond to ALGOL W’s value, result and value-result.” (52-53)
* “The task facility in Ada is included to permit parallel processing and there is a *clock* data type to exercise control and allow the programmer to handle real-time applications. Ada also contains an extensive set of features for interrupt and exception handling. These are on the same lines as the PL/I ON-conditions, but are more powerful and somewhat better designed. “ (53)
* “Ada has generalized Pascal’s constant declarations and brought them more in line with variable declarations. In Ada, both variables and constants can be given values within a declaration. Hence, the effect of the Ada declaration:

*size* : *integer* := 20;

is to declare an integer variable whose initial value is 20 whereas the effect of the declaration:

*size* : **constant** *integer* := 20;

is to declare an integer constant whose value is always 20. The values in such variable and constant declarations may be given by expressions that cannot be evaluated until run time. Consequently, in Ada, the name-value binding of a constant may have to be delayed until block entry.” (68-69)

* “In Ada, a distinction is made between new types (called **derived types**) and new **subtypes**. The Ada declaration:

**subtype** *index* **is** *integer* **range** 1 .. 10;

*count* : *index*;

has a similar effect to the earlier declaration in Pascal. Hence, provided its value does not go out of range, an object of subtype *index* can be used anywhere that an object of type *integer* can be used. The type declaration:

**type** *new­\_index* **is new** *integer* **range** 1 .. 10;

*new\_count* : *new\_index*;

on the other hand, creates a new integer type called *new\_index*. A consequence of this type declaration is that for each operator available on objects of type *integer* there is now a corresponding, but distinct, operator on objects of type *new\_index*.” (71)

* “Explicit conversion between [two types] can be made through the use of what is known as a **type mark**. In the expression:

*new\_index(count) + new\_count*

the type mark *new\_index* is used to convert the value of *count* from type *integer* to type *new\_index*.” (72)

* “… Ada provides distinct Boolean operators for cases where short-circuit evaluation is desired. Ada’s method is to introduce two further logical operators (**and then**, **or else**) with the same precedence as **and** and **or**. These operators function as follows:

(1) *a* **and then** *b*, where *b* is only evaluated if *a* is true.

(2) *a* **or else** *b*, where *b* is only evaluated if *a* is false.

The && operator [from C] correspond to the Ada **and then** operator while || [from C] correspond to **or else**.” (95)

* “Languages such as Ada and Modula-2 … forbid mixed-mode expressions.” (95)
* “Hence, to add an integer to a real in a language like Ada or Modula-2, the type conversion must be done explicitly.” (95)
* “As an example, consider the typical Pascal or Ada assignment statement:

*x* := *a* + *b* \* *c*;

If the type of *x* is the same as that of the result obtained by evaluating the expression on the right-hand side, then there are no problems. However, some languages state that if the two types are not the same, then it is a compile-time error. Such a strict interpretation of assignment compatibility is common in modern languages like Ada.” (97)

* “In Ada, each **if** is paired with a closing **end if**; for example:

**if** C1 **then**

**if** C2 **then** S1;

**end if**;

**else** S2;

**end if**;” (102)

* “Some languages, notably Ada and Modula-2, have an **elsif** construction to assist in the presentation and understanding of cascaded conditional statements.” (103)
* “More recent languages, notably Ada, have made two changes to the basic **case** statement. The first is the inclusion of a default condition. . . . Such a default case is covered in Ada by the compulsory inclusion of **others**, so the programmer must explicitly consider what action is required if an unspecified value is encountered. . . . Secondly, Ada allows ranges of values to be used for each constant in the case list while in Pascal each value must be explicitly defined.” (103)
* “This is a more general form of statement and includes both the **while** and **repeat** statements as special cases. In its Ada form it consists of:

**loop**

*sequence of statements*

**end loop**;

The sequence of statements can include an **exit** statement:

**exit when** C;

Furthermore, it is possible to have more than one exit from a loop.” (105)

* “Hence, it is not necessary for Ada to include any iterative statement other than the **loop** statement. However, it does include a **while** statement, presumably on the grounds that it is part of most programmers[‘] armoury and that they would be lost without it.” (106)
* “ALGOL W and Ada take the strong line that the control variable [for a **for** statement] only has scope within the **for** statement and is not available outside it. In Ada, the **for** loop forms a block in which the control variable is implicitly declared.” (110)
* “Languages such as Pascal and Ada, which allow discrete types such as characters or enumeration types, also permit the use of these types in **for** statements.” (110)
* “The **exit** statement in Ada and Modula-2 is a controlled jump to the end of a loop and Ada goes further than Pascal in not allowing jumps out of procedures.” (112)
* “The principle idea in Ada is to provide each group of statements with an exception handling capability.” (114)
* “Exception handlers are associated with a statement sequence in what is called a **frame**, which always includes the structure:

**begin**

*statements*

**exception**

**when** *overflow* => …;

**when** *help* => …;

**end**;

A frame can be a subprogram body, a package body or a block.” (115)

* “If, during the execution of the statements in a frame, an exception is raised, then control is passed to the appropriate handler at the end of the frame. When the actions in the exception handler have been completed, the frame is left.” (115)
* “If an exception is raised in a frame that does not have an exception handler for the condition, the exception is propagated to the next level. If the frame concerned is a subprogram body, the exception is passed to the frame containing the subprogram call. This is a dynamic process with the exception being raised again in the calling frame if it is not handled. A similar mechanism is used for blocks and package bodies.” (115)
* “Ada has both predefined and user-declared exceptions. The predefined exceptions . . . are essential for an embedded language like Ada so that errors that cannot be checked at compile time are raised as an exception at run time. In contrast, user-declared exceptions, such as *overflow* and *help*, are activated only by the use of a raise statement. However, the raise statement can also be used to activate the predefined exceptions. Furthermore, the programmer is also provided with the facility to write exception handlers for the predefined exceptions.” (115)
* “Finally, Ada does not allow parameter passing to exception handlers. This is because exceptions are considered to be irregular conditions and the frame in which they occur should provide the appropriate exception handling operations before passing control back to the calling frame.” (115)
* “… Ada allows an ordinary procedure . . . to be the main program.” (124)
* “… Ada allow[s] the nesting of blocks that are not subprograms.” (127)
* “Ada has three different parameter modes called **in**, **out** and **in out** . . . .” (130)
* “Ada allows more than one subprogram with the same identifier to be declared in the same scope, but they must have a different parameter profile so that they can be differentiated from one another.” (137)
* “Another feature of Ada is that **in** mode parameters may be given a default initial value. Hence, the *increment* procedure could be written as:

**procedure** *increment*(*item* : **in out** integer; *by* : **in** *integer* := 1) **is**

**begin**

*item* := *item* + *by*;

**end** *increment*;” (140)

* Example function (p. 142):

**function** *add*(*a*, *b* : **in** *integer*) **return** *integer* **is**

**begin**

**return** *a* + *b*;

**end** *add*;

* Ada and ALGOL 68 allow structured values to be returned from functions (143).
* “Ada, in fact, does not allow **out** or **in out** parameters with functions . . . .” (143)
* “Subprograms may not be passed as parameters in Ada, although a similar effect is achieved by what are called **generics**.” (155)
* “A generic subprogram is a **template** from which an actual subprogram may be obtained by what is called **instantiation**.” (155)
* “Generic instantiation occurs at compile time and results in two completely different . . . functions being created.” (155)
* “Many other languages, such as . . . Ada, have followed ALGOL 60 in allowing semi-dynamic arrays. The advantage of using this method is that arrays can be created to exactly the size required by the particular problem.” (164)
* “Ada has **constrained** and **unconstrained** array types.” (166)
* “Ada allows the specification of array values, called **array aggregates**, which are especially useful in cases where it is necessary to initialize an array.” (168)
* “In Pascal and Ada, a two-dimensional array is an array of arrays and so the concept of a row slice is a natural one. A row of a two-dimensional array can be passed as a one-dimensional array parameter to a procedure or function, but it is not possible to treat a column as a slice. In Ada, a slice can be part of a one-dimensional array.” (168)
* “In Ada, arrays or records may also be returned [from functions].” (169)
* “Ada has **record aggregates** and so values can be assigned to complete records.” (172)
* “In some languages, such as . . . Ada, a facility is provided where a single record can have several **variants**.” (172)
* “Ada allows similar variant records, although with syntactic differences [from Pascal], and the insecurities of the Pascal construction are removed by not allowing the tag field to be changed independently of the dependent fields in the variant part.” (173)
* “In Pascal, Ada, Modula-2 and C, strings are simply arrays of characters and since arrays are not dynamic in these languages, the length of a string cannot be changed once it has been declared.” (179)
* “Similarly, ‘x’ and “x” in Ada are character and string constants, respectively.” (180)
* “An Ada module is called a **package**. The interface with the outside world is called the **package specification** and the implementation details are held in a separate **package body**.” (193)
* “A package specification is an example of an Ada **program unit** while its corresponding body is called a **secondary unit**. An Ada program consists of a series of program and secondary units. One of the program units, the main program, must be a procedure. Variables declared in a program or secondary unit are allocated space at load time, which allows the separation of the scope and extent of a variable in a way not possible in languages such as Pascal.” (194)
* “When dealing with large systems composed of hundreds of subprograms, it is important to ensure that a change in one subprogram does not necessitate the recompilation of all other subprograms while, if the system is to be secure and reliable, it is important to have full type checking. Ada and Modula-2 solve this problem by replacing *independent* compilation [as in FORTRAN] by *separate* compilation.” (201)
* “The packages . . . , the package bodies . . . and the [main] procedure . . . can all be compiled separately and, together with their context clauses, are called **compilation units**. These units are not totally independent, since some depend on information specified in others, and their interdependence is given by the context clauses. A package body is always dependent on its specification as identifiers declared in a specification are available in the corresponding body.” (202)
* “No unit may be compiled before any of the units on which it depends.” (202)
* “Associated with each Ada program is a library file, called the **program library**, which contains information on the compilation units, including the time at which they were compiled.” (202)
* “Modules in Ada and Modula-2 may be nested within other modules or subprograms.” (203)
* “. . . Ada uses static binding . . . .” (216)
* “Unlike Modula-2, Ada and occam have built-in features to support true concurrency involving multiprocessors with no shared memory. In Ada, each process is described by means of a task and, like packages, tasks have both a specification and a body.” (234)
* “As two Ada tasks can be declared within the same procedure, they can share access to non-local variables and, therefore, communicate through shared variables. However, the preferred method of inter-task communication in Ada is to avoid the use of shared variables and to use the rendezvous mechanism.” (234)
* “A task is entered by means of an **entry call**. An entry is similar to a procedure, the difference being that an entry call cannot proceed until the called task is ready to accept the call—that is, when there is a rendezvous. An entry call is, therefore, used both for inter-task communication and for synchronization.” (234)
* “. . . the body of the **accept** statement acts as a critical region where access to shared variables can take place.” (237)
* “Once the body of the **accept** statement has been executed, the rendezvous is over and both tasks are free to continue in parallel.” (237)
* “In the case of Ada, the called task does not know the identity of the caller and an **accept** statement can accept entry calls from several different tasks.” (238)
* “The main disadvantage of the rendezvous approach is that it often requires a larger number of processes than that required when monitors are used.” (239)
* “In Ada . . . the tasking overhead is high as the language is much more complex [than occam] and is implemented on general-purpose computers . . . .” (239)
* “Ada was specifically designed for use in real-time embedded systems.” (242)
* “In general, real-time programming has all the problems of concurrent programming with the added problem of time. To help solve such problems, Ada and occam allow different processes to be given different priorities . . . .” (242)
* “Time can be introduced into an Ada program through the **delay** statement.” (243)
* “I/O in Ada is provided by a hierarchy of predefined packages called *low\_level\_io*, *direct­\_io*, *sequential\_io* and *text\_io*. All validated Ada systems must provide an implementation of these packages . . . .” (308)
* “High-level text I/O in Ada is provided by the package *text\_io*, which provides facilities at a similar level to Pascal I/O. This package defines procedures called *get* and *put* to read and write characters or strings. The *get* and *put* routines to read and write integers, fixed-point numbers, reals and enumerated types are declared in the local generic packages *integer\_io*, *fixed\_io*, *float\_io* and *enumeration\_io*, which are declared within the package *text­\_io*. However, these packages must be instantiated before such I/O can take place."”(308)
* “As Ada allows procedures and functions to be overloaded, the type of the parameter determines which version of *get* or *put* is required. The drawback of this approach is that generic instantiation has to be introduced at the beginning of an Ada course.” (309)
* “A difference between Pascal and Ada is that the Ada I/O routines all follow the ordinary Ada rules for procedures and functions, while Pascal I/O routines only look like ordinary subprograms and can have a variable number of parameters of differing types.” (309)
* “As with Pascal, Ada lines are terminated by an end-of-line indicator, which can only be written by calls of procedures such as *new\_line*. The Ada function corresponding to the Pascal *eoln* function is *end\_of\_line*. In addition, in Ada, lines are grouped into pages. End-of-page indicators are written by calls of *new\_page* and are recognized by calls of the Boolean function *end\_of\_page*.” (309)
* “In Ada, logical file names are declared to be of type *file\_type*.” (309)
* “Ada files may have the mode *in\_file* or *out\_file* where *file\_mode* has been declared in *text\_io* . . . .” (309)
* “To close the file associated with *new\_file*, the command is *close(new\_file)* and to delete it, the command is *delete(new\_file)*.” (310)
* “An Ada program is a collection of modules (subprograms or packages) whose interdependency is given by context clauses. One of the modules is the main program and it must be a procedure. A module is compiled in the context of those modules from which it imports information. A package is in two parts: a specification, which gives its interface with the rest of the program, and a hidden implementation part. This allows separate compilation.” (335)
* “The predefined types are *integer*, *float*, *character*, *Boolean* and *string*. Scalar user-defined types are the derived subrange types and the enumeration types. Structured types can be defined using arrays, records and pointers. Abstract data types can be implemented using packages and parameterized abstract data types using generic packages.” (335-336)
* “Arrays and records can be passed as parameters and returned as function values. Fixing the size of an array can be delayed until block entry. Dynamic data structures are created using records that have one or more fields of a pointer type. Strings are arrays of *character*.” (336)
* “Structure control statements have explicit terminators. The conditional statements are **if** and **case**. The iterative statements are **while**, **for** and a **loop ... end loop** construct that may be left using an **exit** statement. Exception handling is also supported.” (336)
* “Parallel processing is achieved using tasks. Message passing is used both to pass information and to achieve synchronization.” (336)
* “I/O is supported by a hierarchy of packages . . . . At the highest level, text I/O is supported while the lowest level supports direct access to peripheral devices.” (336)

Watt, D. A. (1990). *Programming language concepts and paradigms*. London: Prentice Hall International.

* “Pascal’s powerful successor, *Ada*, introduced packages and generics—designed to aid the construction of large modular programs—as well as high-level forms of exceptions and concurrency. Like PL/I, Ada was intended by its designers to become *the* standard general-purpose language.” (5)
* “In Pascal and Ada, recursive types must be defined in terms of pointers.” (23)
* “In Ada, an array type incorporates the *type* of the index bounds but not their actual values.” (41)
* “An ***activation*** of a block is a time interval during which that block is being executed. In the Algol-like languages, including Pascal and Ada, a lifetime of a local variable *V* corresponds exactly to an activation of the block containing *V*’s declaration. In the special case where *V* is a global variable, *V* is alive throughout the program’s execution time.” (44)
* “In Ada, a variable may be initialized as part of its declaration.” (45)
* Ada opts for static binding. (72)
* Ada provides only eager evaluation. (101-102)
* “In Ada, a package that specifies and implements a single object can easily be made into a *generic* *package*, which then defines an object class.” (115)
* Ada makes “overloading or polymorphism uniformly available for both built-in and defined abstractions.” (129)
* “Ada recognizes a much wider variety of subtypes than Pascal. We can define subranges of all primitive types, including real types. We can define a subtype of an array type by freezing its index bounds. We can define a subtype of a record type by freezing specially designated components called *discriminants*.” (142)
* “Ada’s *exit* sequencer can be used to terminate loops only.” (154)
* Ada’s rendezvous mechanism was inspired by Hoare’s CSP language. (183)
* “Ada was the first major programming language to incorporate structured concurrent programming. A *task* module is in some ways similar to a package. The body of a task module may be executed as a concurrent process. Global variables are accessible by a task body, as a normal consequence of Ada’s scope rules, so tasks may interact by their effect on shared variables. However, Ada provides no support for such interactions: it lacks a built-in mutual exclusion construct, and the programmer is responsible for ensuring that sharing of variables has no harmful effects.” (183)
* “Instead, the preferred mode of communication between tasks is the rendezvous, which is encouraged by a rich set of language features. A task may export some *entries*, which resemble remote procedures. . . . The sender task communicates by means of an *entry call*, which is syntactically similar to a procedure call, and which passes arguments to the nominated entry. The receiver task communicates by means of an *accept* command, a composite command that has access to the arguments from the entry call. The sender is blocked until the entry call is served, and waits in a first-com first-served queue. The receiver is blocked at an *accept* command for an entry with no outstanding calls. Additional language features allow for bounded nondeterminism in accepting calls, and offer the option not to wait if no rendezvous can be achieved.” (183)
* “Nondeterministic choice is provided by the *selective wait* command, which allows a task to respond to the first of a selection of entry calls (or to any one of them if several are outstanding). Acceptance can be made conditional in each case. This makes it easy to write monitor-like tasks, with no need for a mutual exclusion mechanism.” (185)
* “To classify Ada as a Pascal-like language might be misleading. Only about half of Ada is directly comparable to Pascal (although that half corrects most of Pascal’s design flaws and is notationally richer). The other half supports important concepts such as encapsulation, generics, exceptions, and concurrency, and in these respects Ada extends far beyond Pascal.” (196)
* “In Ada we can write nontrivial expressions of every type. In particular, aggregates allow records and arrays to be constructed from their components. However, Ada has no conditional expressions or block expressions, and a function body is a command (in effect). Thus Ada, like Pascal, forces the programmer to resort to commands for any computation that is too complicated for its limited repertoire of expressions.” (197)
* “Ada was the first major language to include a secure form of exception handling.” (197)
* “Assignment and equality tests are operations applicable to all Ada types, including abstract types—but not abstract types declared as *limited private*. When defining a recursive type, the programmer will normally declare it as limited, in order to avoid inconsistency in the results of assignment and equality tests.” (200)
* The division of declaration from body “supports separate compilation of modules.” (201)

Burns, A., & Wellings, A. (1995). *Concurrency in Ada*. Cambridge, UK: Cambridge University Press.

* “An Ada block consists of (i) the declarations of any entities (types, objects, procedures, etc.) that are local to the block (if there are no such entities, then this declaration part may be omitted), (ii) a sequence of statements and (iii) a collection of exception handlers (these handlers may be omitted . . . .):

**declare**

<declarative part>

**begin**

<sequence of statements>

**exception**

<exception handlers>

**end**;” (2)

* “A block itself may be declared at any place in the program where a statement may be written. Blocks, which may be named, can contain other blocks and they therefore form a useful method of providing decomposition within a program unit. When a block is executed, the declarative part is said to be elaborated and the sequence of statements executed; in both operations it may be necessary to evaluate expressions.” (2)
* “The scope rules for blocks in Ada are somewhat different from those in other languages. They can be described informally as follows:

1. The scope of an identifier is the block in which it is declared and all blocks enclosed within.

2. An identifier is in scope immediately following its declaration.

3. The visibility of an identifier matches its scope unless an identical identifier is declared in an inner block, in which case the original identifier is not directly visible.” (2)

* Example (2-3):

Top: **declare**

A : Integer := 42;

B : Integer := A + 3;

**begin**

A := A + B;

**declare**

A : Integer := 10;

C : Integer;

**begin**

C := B;

C := A;

C := Top.A;

**end**;

B := A;

**end** Top;

* “With all data types it is permissible to define a subtype that restricts the range of values of the parent type. It is not permitted, however, to use a subtype to restrict the operators that are associated with that type.” (3)
* Example (3):

**declare**

**type** Penny **is range** 1 .. 100;

-- a new type constructed from the

-- underlying (Universal) integer type

**type** New\_Int **is new** Integer;

-- a new type derived from Integer

**type** Small\_Int **is new** Integer range -32 .. +31;

-- a new constrained type derived from Integer

**type** Day **is** (Monday, Tuesday, Wednesday, Thursday, Friday,

Saturday, Sunday); -- an enumeration type

**subtype** Pos\_Int **is** New\_Int **range** 1 .. New\_Int’Last;

**subtype** Weekday **is** Day **range** Monday .. Friday;

Start : Day := Monday;

A : New\_Int;

B : Pos\_Int;

C : Integer;

**begin**

A := B; -- legal.

B := A; -- legal, but the value may be out of range

-- and cause an exception to be raised during execution

A := C; -- illegal, this is a type clash as A and C are different types:

-- A is New\_Int and C is Integer.

**end**;

* “Attributes are used throughout the language to provide information about types and objects.” Attributes are usually denoted by ‘ followed by the attribute name (e.g., ‘Last). (4)
* “Ada facilitates object-oriented programming by providing a form of inheritance (via type extension) and run-time polymorphism (via run-time dispatching operations). If a type is to be extended, it must be declared as being a *tagged* record type (or a tagged private type).” (4)
* “All types derived in this way (including the original *root*) are said to belong to the same *class* hierarchy. When a type is extended, it automatically inherits any primitive operations (those defined with the type) available for the parent type. However, these may be overridden with new versions.” (5)
* “Procedures and functions are known collectively as subprograms. They are specified by giving a name, a complete description of the parameters and, if the subprogram is a function, the type of the returned object.” (5)
* “The parameters [of subprograms] must have their type and their mode specified. Three such modes are allowed:
  + **in**—within the subprogram the parameter acts as a local constant—a value is assigned to the formal parameter on entry to the procedure or function. This is the only mode allowed for functions. It is the default mode.
  + **out**—within the subprogram the value of the parameter can be written and read—a value is assigned to the calling parameter upon termination of the subprogram.
  + **in out**—within the subprogram the parameter acts as a variable—upon entry to the subprogram a value is assigned to the formal parameter; upon termination the value attained is passed back to the calling parameter.” (6)
* “Interestingly, these parameter modes do not dictate the method of implementation, which could be either by copy or by reference. There are, however, some restrictions. Non-composite types (e.g. scalars such as integers and Booleans) must be passed ‘by copy.’ Also, certain more complex types (e.g. tasks and protected types) must be passed ‘by reference.’” (6)
* “A subprogram body consists of a repeat of the specification plus whatever declarations and statements are needed to ‘implement’ the specification.” (6)
* Example (5):

**function** Minimum(X, Y, Z : Integer) **return** Integer **is**

**begin**

**if** X <= Y then

**if** X < Z **then**

**return** X;

**else**

**return** Z;

**end if**;

**elsif** Y <= Z **then return** Y;

**else** **return** Z;

**end if**;

**end** Minimum;

* Example (5-6):

**procedure** Quadratic(A, B, C : **in** Float; R1, R2 : **out** Float; OK : **out** Boolean) **is**

Z : **constant** Float := B\*\*2 – 4.0\*A\*C;

**begin**

**if** Z < 0.0 **or** A = 0.0 **then**

OK := False;

R1 := 0.0;

R2 := 0.0;

**end if**;

OK := True;

R1 := (-B + Sqrt(Z))/2.0\*A);

R2 := (-B – Sqrt(Z))/2.0\*A);

**end** Quadratic;

* “Parameters of mode ‘in’ can have default expressions, the values of which are used if a call of the subprogram does not contain an actual matching parameter. Calls, in general, can make use of the usual positional notation or employ named notation to remove positional errors and increase readability.” (7)
* Example (7):

**function** Wage(Grade : University\_Post;

Hours : Float := 40.0;

Overtime : Float := 0.0) **return** Float;

Calls:

Pay := Wage(Secretary, 40.0, 0.0);

Pay := Wage(Secretary); -- uses default values.

Pay := Wage(Grade => Secretary); -- uses name notation.

Pay := Wage(Professor, Hours => 80.0);

Pay := Wage(Grade => Lecturer, Overtime => 25.0);

* “All subprograms can be called recursively and are reentrant.” (7)
* “Indirect access to program entities (i.e. objects and subprograms) can be obtained through the use of access types. Such types can be used to access subprograms, ordinary objects and objects created by allocators.” (7)
* “One of the useful applications of this feature [access] is the ability to pass subprograms (via access types) as parameters to other subprograms.” (8)
* “Dynamic object creation comes from the use of an access type and an allocator:

**type** Some\_Data **is**

**record**

…

**end record**;

**type** Some\_Data\_Pointer **is access** Some\_Data;

Srp : Some\_Data\_Pointer; -- the ‘value’ of Srp is **null**

Srp := **new** Some\_Data; -- a new object is created; it is identified as Srp.**all**” (8)

* Access types may be used to reference ordinary objects by using aliases. Access types may also be made read-only by adding **constant** to the declaration. (8)
* “Access types can be passed as parameters to subprograms . . . .” (9)
* “The package is the single most important construct in Ada. It serves as the logical building block of large, complex programs and is the most natural unit of separate compilation. In addition, the package provides for data hiding and the definition of abstract data types. A package definition has two parts, the specification (which itself may consist of a private as well as a visible part) and the body. The body contains the code necessary to implement the specification. Its inner details are hidden from the rest of the program, and, in terms of program development, the body of a package will often be constructed later and take the form of a separately compiled unit.” (9)
* “The general form of a package is

**package** <Name> **is**

-- visible declarations of constants, types, variables and subprograms

**private**

-- hidden type and constant declarations, if any

**end** <Name>;

**package body** <Name> **is**

--internal declarations

**begin**

--sequence of statements for initialising the package

**end** <Name>;” (9-10)

* Example package (10):

**package** Complex\_Arithmetic **is**

**type** Complex **is private**;

**function** “+”(X, Y : Complex) **return** Complex;

**function** “-“(X, Y : Complex) **return** Complex;

**function** “\*”(X, Y : Complex) **return** Complex;

**function** “/”(X, Y : Complex) **return** Complex;

**function** Comp(A, B: Float) **return** Complex;

**function** Real\_Part(X : Complex) **return** Float;

**function** Imag\_Part(X : Complex) **return** Float;

**private**

**type** Complex **is**

**record**

Real : Float;

Imag : Float;

**end record**;

**end** Complex\_Arithmetic;

* “Overloading is allowed for all subprograms in Ada as long as the meaning is unambiguous.” (10)
* “If a type has been designated as being private, then the following operations are allowed in the rest of the program where the package is in scope:

1. Objects of that type can be declared.

2. Subprograms supplied with the type can be called.

3. Two objects of the type can be compared for equality.

4. Values of that type can be assigned to objects.

5. Subprograms can be defined with parameters of the type.” (10-11)

* “If it is desirable to remove the possibility of assignment and (predefined) equality tests, then the type can be declared as being ‘limited private.’ If the type is to be extended, it may be defined as ‘tagged private’ or ‘tagged limited private.’” (11)
* “The ‘with’ clause names the precompiled library unit (or units) upon which [the main program] is dependent. Only those units upon which there is a direct dependence should be specified. Library units can themselves have ‘with’ clauses, thereby supporting a hierarchy of dependencies. Clearly, the recompilation of any unit necessitates the recompilation of all units that depend upon it.” (12)
* “The ‘use’ clause provides direct visibility of declarations that appear in the visible part of the named package. If the ‘use’ clause were omitted . . . then the names of all objects from the package would need to be prefixed by the package name [and a period].” (12)
* “Separate compilation can also be achieved by the use of subunits which are either package, subprogram, task (or protected object) bodies. A subunit is removed from its immediate surrounding unit and compiled later:

**package body** Complex\_Arithmetic **is**

**function** Comp(A, B : Float) **return** Complex **is separate**;

…

The separate compilation of this subunit (which must name its parent unit) would take the form

**separate** (Complex\_Arithmetic)

**function** Comp(A, B : Float) **return** Complex **is**

**begin**

**return** (A, B);

**end** Comp;” (12)

* “Subunits provide for the top-down construction of programs; they also have the facility for providing access to additional library units.” (12)
* “Given the declaration of a package with a private part, a child package is allowed access to these private declarations.” (13)
* “The package construct provides both for decomposition and abstraction and it therefore has an important role in any design method. Module specification will invariably lead to package specification, and the ‘implementation’ of package bodies gives a natural partition to project work. The link between package specification and implementation is, however, merely a syntactical one: there are no formal means by which the semantics of the implementation can be specified using standard Ada. This shortcoming has led to the development of a number of methods for specifying the semantics of package and subprogram bodies. Two general approaches have been suggested: one involves the use of formal comments, the other uses additional subprograms to state precondition, postcondition and invariant properties of the implementation. Formal comments are, in effect, extensions to the language definition and may use Ada-like syntax or algebraic axioms. Subprograms have the advantage that they can be used during unit testing to check that the intended conditions are satisfied.” (13)
* “The idea of a package is not new in programming languages. In fact, the class construct in SIMULA is probably the first example of such an abstraction mechanism. Following SIMULA, a number of Pascal derivatives have introduced some form of class/module structure; for example, Modula-2 has a module structure. The main criticism that can be levelled at the Ada package is its use of open scope rules, whereby all objects visible at the point of package declaration are automatically accessible outside the package.” (15)
* “An essential feature of library units should be a generality which will allow them to be used over a wide range of applications. This generality can be achieved in a limited sense by the appropriate choice of subprogram parameters (with default values). However, the strong typing model of Ada prevents flexibility. Generics and type extensibility make possible the production of reusable software components and they represent important features of the Ada language.” (16)
* “A generic is a template (with parameters) from which instances of subprograms and packages can be constructed. This construction, or *instantiation* as it is usually called, involves the association of formal and calling parameters at compile time.” (16)
* “Two broad classifications of exception can be isolated:
  + *Error conditions*—arithmetic overflow, storage exhaustion, array-bound violation, subrange violations, peripheral time-outs, etc.
  + *Abnormal program condition*—errors in user input data, need for special algorithms to deal with singularities, etc.” (17)
* “These exception handlers are separated, textually, from the place at which the error is raised so that the normal behaviour of the program is not obscured. The raised exception is, therefore, an up-market goto statement in many ways, and its widespread use can lead to the kind of unmaintainable spaghetti code that high level languages are designed to avoid. Because of this, Ada contains a limited form of exception handling that should really be used only for error conditions. For example, the Ada model of exceptions does not allow for an automatic return to the point of the error from within the exception handler.” (17)
* “Any program block (or subprogram) in Ada may contain handlers that can catch exceptions raised during the execution of that block.” (17)
* “When an exception is raised, the appropriate exception handler is executed and the block terminates. If no handler is to be found in the local block, then the exception is propagated to containing blocks (in the absence of subprogram calls) until it is handled or, if there is no such handler in the main program block (or task), the program (or task) itself terminates. Where an exception is raised and not handled in a subprogram, the subprogram is terminated and the named exception is raised again at the point of call of the subprogram. In this way, the exception is propagated through the dynamic chain of subprogram calls. If a user-defined exception goes out of scope due to its propagation, then the exception becomes anonymous and can only be caught with an ‘others’ handler.” (18)
* “For run-time errors, Ada recognises three separate classes:
  + those that must raise an exception;
  + those that have a bounded number of possible effects; and
  + those that can lead to totally erroneous behaviour.” (20)
* “Ada provides for the direct programming of parallel activities. Within an Ada program there may be a number of tasks, each of which has its own thread of control.” (22)
* “Ada 83 only supported the single notion of a **task**; thus active and all control entities were encoded in tasks. The current version of Ada [Ada 9X], by comparison, has introduced a new abstraction for resource entities—the **protected type**.” (26)
* “Task objects and types can be declared in any declarative part, including task bodies themselves. For any task type, the specification and body must be declared together in the same unit, with the body usually being placed at the end of the declarative part. The entries are in scope immediately after the task specification that defines them.” (70)
* “A task is said to be *created* by its elaboration. The execution of a task object has three main active phases:

Phase 1

*Activation*—the elaboration of the declarative part, if any, of the task body (any local variables in the body of the task are created and initialised during activation).

Phase 2

*Normal execution*—the execution of the statements within the body of the task.

Phase 3

*Finalisation*—the execution of any finalisation code associated with any objects in its declarative part.” (76)

* “A tasks, in general, indicate its willingness to begin finalisation by executing its ‘end’ statement. A tasks may also begin its finalisation as a result of an unhandled exception, or by executing a select statement with a terminate alternative, or by being aborted. A finished task is called *completed* or *terminated* depending on whether it has any active dependents.” (76)
* “For static tasks, activation starts immediately after the complete elaboration of the declarative part in which they are defined.” (76)
* “Dynamic tasks are activated immediately after the evaluation of the allocator (the **new** operator) which created them.” (77)
* “If an exception is raised during a task’s activation, then the task itself cannot handle the exception. The task is prevented from executing its body and hence becomes completed or terminated. If any objects have already been created, then they must be finalised. As the task itself cannot handle them, the language model requires the parent (creator) task or scope to deal with the situation: the predefined exception Tasking\_Error is raised. In the case of dynamic task creation, the exception is raised after the statement which issued the allocator call. However, if the call is in a declarative part (as part of the initialisation of an object), the declarative part fails and the exception is raised in the surrounding block (or calling subprogram).” (79)
* “Ada is a block structured language in which blocks may be nested within blocks. A Task can be declared in any block; therefore it is possible for tasks to be declared within tasks (or blocks) which themselves are declared within other tasks (or blocks). This structure is called a task hierarchy.” (80)
* “Ada forces the master task to wait for finalisation and termination of its dependents before it itself can finalise any variables (of which it is the master) and terminate. When a master has finished its execution but cannot terminate because its dependents are still executing, the master is said to be *completed*. Of course, any dependent task cannot themselves terminate until all their dependent tasks have also terminated.” (84)
* “All task types in Ada are considered to be limited private. It is therefore not possible to pass a task, by assignment, to another data structure or program unit.” (86)
* “The rendezvous model of Ada is based on a client/server model of interaction. One task, the server, declares a set of services that it is prepared to offer to other tasks (the clients). It does this by declaring one or more public **entries** in its task specification. Each entry identifies the name of the service, the parameters that are required with the request and the results that will be returned.” (90)
* “A client task (also named the *calling task*) issues an ‘entry call’ on the server task (or *called task*) by identifying both the server and the required entry.” (90)
* “For the communication to occur between the client and the server, both tasks must have issued their respective requests. When they have, the communication takes place; this is called a rendezvous because both tasks have to meet at the entry at the same time. When the rendezvous occurs, any **in** (and **in out**) parameters are passed to the server task from the client. The server task then executes the code inside the **accept** statement. When this statement finishes (by encountering its **end** statement), any **out** (and **in out**) parameters are passed back to the client and both tasks proceed independently and concurrently.” (91)
* “For the simple rendezvous case, the server is obliged to wait for a call; whilst it is waiting it frees up any processing resource it is using; a task which is generally waiting for some event to occur is usually termed *suspended*. Similarly, if a client issues a request and the server has not indicated that it is prepared to accept the request, then the client must wait.” (92)
* “A formal description of the entry statement is given below:

entry\_declaration : :=

**entry** defining identifier [(discrete\_subtype\_definition)] parameter\_profile;

The parameter profile is the same as for Ada procedures (**in**, **out**, **in out**—with **in** begin the default). Access parameters are not permitted and default parameters are allowed.” (92)

* “For each and every entry defined in a task specification there must be at least one accept statement in the corresponding task body. . . . A task body, however, may contain more than one accept statement for the same entry. It should be noted that the accept statement must be placed directly in the task body; it cannot be placed in a procedure which is called by the task body. Such a procedure could be called from more than one task!” (94)
* “In general, therefore, the accept statement has the following form:

**accept** Entry\_Name(Family\_Index)(P : Parameters) **do**

-- sequence of statements

**exception**

-- exception handling part

**end** Entry\_Name;” (94)

* “The code it [the accept statement] contains should be only that which is necessary for the rendezvous. If it contains extra statements, then the calling task will be held up unnecessarily.” (95)
* “Clearly, it is possible for some exceptions to propagate outside the accept statement. If this happens, the rendezvous is terminated and the exception is raised again in *both* the server (called) and the client (calling) tasks.” (104)
* “An exception may be raised during elaboration of a declarative part.” (236)
* “An exception may be raised during the activation of a task. . . . The task is unable to handle this exception and there becomes completed.” (236)
* “It is a bounded error for an exception to be raised during any finalisation routine. If one does occur during the finalisation of a task, or a master block containing tasks, then the most likely consequence will be that the exception will be lost and the finalisation of other objects continued.” (237)
* “The model is essentially that if an exception is not handled within the rendezvous, it is propagated to both the server and the client task.” (237)
* “The Program\_Error exception may be raised when a task interacts with a protected object.” (237)
* “If an exception is propagated from an interrupt handler that is invoked by an interrupt, the exception has no effect.” (238)
* “The abort statement is intended for use in response to those error conditions where recovery by the errant task is deemed to be impossible. Tasks which are aborted are said to become *abnormal*, and are thus prevented from interacting with any other task. Ideally, an abnormal task will stop executing immediately. However, some implementations may not be able to facilitate immediate shut down, and hence all . . . [that is required] is that the task terminate before it next interacts with other tasks.” (239)
* “After the task has been marked as abnormal, execution of the task body is aborted. This means that the execution of every construct in the task body is aborted, unless it is involved in the execution of an *abort-deferred operation*. The execution of an abort-deferred operation is allowed to complete before it is aborted.” (239)
* “Ada 83’s concurrency facilities were widely criticised, and during the Ada 95 development process some 25 major ‘revision issues’ were raised on the tasking model by the team responsible for generating the requirements for the new language.” (380)
* “Although, in general, Ada 83 has a coherent tasking model, there were some minor issues with the ease with which some programming paradigms (expressible within the model) could be written. For example, it was easy to write a function which could return an instance of a task local to the function. However, the task termination rules required the returned task to terminate before the function could return. Another problem was that representation clauses and priority levels were associated with a task type and not the task object. These minor irritations have been corrected in Ada 95.” (381)
* “One of the main concerns with Ada 83’s tasking model was its poor support for real-time systems. The priority and time models were vague and weak, and really not adequate for implementing hard real-time systems. Priority inversion was inevitable because of FIFO entry queues. Periodic tasks were not well supported and relied on the delay statement for their implementation (this suffered from race conditions). Concern was also expressed that the language defined a particular model of scheduling and it was difficult for applications to program their own paradigms.

Ada 95 has resolved all these issues, and now supports a consistent set of facilities for programming all classes of real-time systems.” (381)

* “Ada 83 made a brave attempt to address interrupt handling by providing a mechanism by which an entry call could be associated with an interrupt and by allowing the specification and manipulation of device registers through representation clauses. However, as mentioned above, by tying these facilities to the task type rather than the task object, it made interrupt handling of identical devices very cumbersome. Furthermore, many application programmers felt that the model of an interrupt as an entry call was too complicated and that an interrupt was better thought of as a procedure call.

Ada 95 has a well integrated set of facilities for handling interrupts. It allows for a protected procedure to be called by an interrupt, and has removed the restrictions associated with representation clauses.” (381)

* “Ada 83 was originally designed when distributed and parallel systems were in their infancy. Consequently, the distribution model was almost non-existent. Furthermore, problems were to be found with implementations of the language on highly parallel computers; there is no way within the language to specify fine-grain parallelism, for example, parallel loops.

Ada 95 has made some effort to correct these problems, but still suffers from a lack of expressive power in this area.” (381-382)

* “There were various problems in Ada 83 on the topic of sharing variables between tasks. The exact effect of pragma Shared was not clear, and there was confusion over the notion of synchronisation points within a task and between tasks.

Ada 95 has resolved all these issues.” (382)

* “The Ada 83 tasking model was synchronous, and whenever it was necessary for tasks to interact in an asynchronous manner extra agent tasks had to be created. This led to an proliferation of tasks in an application and a perceived loss of efficiency. Promises of optimised task structures (such as monitor or buffer tasks) were never fulfiled by compiler vendors in a standard manner (if at all). Consequently, it was impossible to write portable programs which relied on these optimisations. Furthermore, it was not possible to attract the attention of a task and alter its flow of control. The task had either to be aborted or was forced to poll for the asynchronous communication.

Ada 95 has addressed the issue of asynchronous systems on two fronts. Protected objects allow data-oriented asynchronous communication and the asynchronous select statement allows one task to alter the flow of control in another asynchronously.” (382)

* “Perhaps one of the most common complaints against Ada 83 was that the implementations were inherently inefficient due to the complexity of the tasking model. An often articulated requirement was for an efficient mechanism to support mutually exclusive access to shared data.” (383)
* “The upward compatibility requirement on Ada 95 has made it impossible to simplify the tasking model. However, the language designers have made significant efforts to facilitate the production of efficient tasking paradigms by:

- providing data-oriented communication and synchronisation via protected objects;

- allowing alternative tasking models to be implemented in the Real-Time Systems Annex;

- sanctioning ‘subsets’ of tasking via the introduction of a pragma Restricted;

- requiring the implementation to recognise entry-less protected types for special optimisation.” (383)

Ichbiah, J., Barnes, J., Firth, R., & Woodger, M. (1991). *Rationale for the design of the Ada programming language*. Cambridge, UK: Cambridge University Press.

* “Programs are written by human programmers, and read by their authors or by other programmers for checking and maintenance purposes; they are also processed by compilers and other automatic tools. The need to accommodate these various forms of communication permeates every level of consideration of a programming language, including the most immediate levels where we are only concerned with the physical appearance of a program text.” (5)
* “The lexical and textual structures of a programming language are of course important for ease of program compilation, and for compilation-time detection of errors. The importance of lexical and textual structures is even greater for ease of reading and understanding programs—in particular, for detection of logical errors—and for ease of teaching the language. We believe that our understanding of programs can be greatly simplified if our intuition is able to rely on textual forms that convey the logical structure of the program. This is the justification for giving major consideration to readability and teachability in the design of lexical and textual structures in Ada; moreover, special attention has been devoted to structural analogies.” (5)
* “Reserved words are special identifiers that are reserved for special significance in the language. There are 63 such words [in Ada 83]. Many of them play an important role in the definition of the overall syntax of the major program units of the language . . . . Other reserved words play a syntactic role at a more detailed level . . . . Finally, seven of them are used as operators [and, or, xor, not, abs, rem, mod].” (6)
* “Separate compilation of program units is a practical necessity. Its basic goals are to permit the separation of large programs into simpler, more manageable parts, and to create libraries of program units. Separate compilation helps to reduce compilation costs and to simplify the development and management of program corrections and modifications.” (191)
* “The physical separation of program texts may be viewed as a support facility for the structured programming concept of refinement. It may also be used to conceal the text of a subprogram body from users who are only allowed to call the subprogram. . . . Finally this physical separation facilitates the construction of libraries and reusable software components.” (191)
* “Central to the definition of Ada is a concern for the general structure of a program, the rules defining the visibility of identifiers at various points of a program, and the facilities offered for separate compilation. A major goal in this design was to give the programmer precise control over his name space: the set of names that he may define and use. It is important to be able to introduce new names without having to bother about possible conflicts with existing names. This requires the ability to control the inheritance of names that are defined in other contests. . . . the notion of package is essential to achieve this kind of control. Another goal was to provide the same visibility rules for all program units, whether they are separately compiled or not.” (213)
* “The subjects of general program structure and visibility rules are connected in many ways—in particular because of the possibility of nesting program units. They also interact with the facilities offered for separate compilation.” (213)
* “The overall structure of an Ada program text (a compilation unit) is similar to that of an Algol 60 or Pascal text: it appears as a nested structure of program units—subprograms, packages, task units, and generic units—and block statements.” (213)
* “Nesting is achieved through declarative parts: A declarative part may contain bodies of program units, and each of these may in turn contain a declarative part; furthermore, a sequence of statements may contain a block statement that contains a declarative part.” (213)
* “Thus an Ada program appears as a collection of nested declarative regions. A given declarative region may include the declarations of inner program units, in which case it will also include the bodies of these program units. Each of these bodies again defines a declarative region which may in turn declare other inner program units.” (214)
* “In general it is possible to provide the definition of program units—especially packages—in two textually distinct parts:

(a) the specification, which defines the logical interface (between definition and use) of the program unit

(b) the body, which describes a particular realization of the specification.

This possibility has far reaching implications, in that it provides a single basis for achieving several different objectives, notably textual clarity, abstraction, and separate compilation.” (214)

* “These two constructs—the declaration and the body—jointly define the procedure. In cases where the advantages of separate specification are not essential, the procedure declaration may be omitted. In any case, the specification of the parameters must always be given in the body for reasons of readability, and also because of the possibility of overloading . . . .” (215)
* “A similar separation is provided for packages. A package declaration provides the interface to the user: the visible part. . . . The declaration provides the user with the specification of the name of a type . . . and also with the specification of the associated procedures . . . . This constitutes the logical interface of the package.” (215)
* “The package implementation is always provided as a textually distinct package body . . . .” (215)
* “A similar separation is also used for task units and for generic units.” (216)