INTRODUCTION TO Ada

DAVID PRICE

TP312
P25
(2)

INTRODUCTION TO ADA

David Price

Prentice-Hall, Inc.
Englewood Cliffs, N.J. 07632

Library of Congress Cataloging in Publication Data

Price, David, 1961–
Introduction to Ada.

Includes index.
1. Ada (Computer program language) I. Title.
QA76.73.A35P74 1984 001.64'24 83-16082
ISBN 0-13-477653-4
ISBN 0-13-477646-1 (pbk.)

Editorial/production supervision and
interior design: *Aliza Greenblatt*

Cover design: *Jeannette Jacobs*

Manufacturing buyer: *Gordon Osbourne*

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-477646-1 {P}
ISBN 0-13-477653-4 {C}

PRENTICE-HALL INTERNATIONAL, INC., *London*
PRENTICE-HALL OF AUSTRALIA PTY. LIMITED, *Sydney*
EDITORA PRENTICE-HALL DO BRASIL, LTDA., *Rio de Janeiro*
PRENTICE-HALL CANADA INC., *Toronto*
PRENTICE-HALL OF INDIA PRIVATE LIMITED, *New Delhi*
PRENTICE-HALL OF JAPAN, INC., *Tokyo*
PRENTICE-HALL OF SOUTHEAST ASIA PTE. LTD., *Singapore*
WHITEHALL BOOKS LIMITED, *Wellington, New Zealand*

PREFACE

This book is meant to serve as an introduction to the programming language Ada. The reader is not presumed to have extensive programming experience or advanced mathematical training. Only one prerequisite is essential: an interest in the Ada language. The aim of this book is to guide the reader through Ada programming concepts with a minimum of confusion or intimidation.

Though the development of Ada was sponsored by the U.S. Department of Defense, Ada is not limited to a handful of specific applications. It is, in fact, a powerful general-purpose tool. In many ways, it is a proper superset of other popular languages. At the time of its introduction, the uses projected for it ranged from controlling small computers embedded in machinery to maintaining the records of large businesses. A working knowledge of Ada, then, is likely to be an important stock-in-trade among professional programmers for many years to come.

Some aspects of the language may appear to be idiosyncratic or even cumbersome at first glance. They will be especially evident in the early chapters of this book, where most of the example programs will be quite small. The reason behind them lies in the fact that Ada is designed to simplify the production of large programs. Since the complexity of a program increases rapidly as the program becomes larger, Ada provides a number of facilities for dividing programs into smaller modules. These facilities help make Ada programs easier to read, to write, and to modify.

At this writing, most books about Ada are based on outdated descriptions of the language. This book, however, was written to be consistent with the draft Ada standard released in July 1982. Though minor editorial changes may follow in later standards, substantial revisions to this definition of the language are unlikely. Hence, the descriptions of Ada's features in this book are not in danger of becoming obsolete.

I am grateful to Charles Wetherall of Bell Laboratories for many insightful comments. (The remaining defects of the book are, of course, my responsibility.)

I am also grateful to James F. Fegen, Jr. and his associates at Prentice-Hall for their support of this project. Finally, I am grateful to my family and friends for their sympathetic patience.

DAVID PRICE
Midlothian, Virginia

CONTENTS

1 FUNDAMENTALS

1.1 SOME SHORT PROGRAMS IN ADA

Suppose we wanted to write a program in Ada for multiplying a pair of numbers. What would the completed program look like? This is obviously not the sort of problem to which a programmer would devote a great deal of attention in practice, but it will suit our purposes here quite well. In this section we will examine three programs for multiplying numbers; we will use these programs to become acquainted with the style and structure of the Ada language.

Example 1.1

```
with TEXT_IO ; use TEXT_IO;
procedure FIRST_MULTIPLY is
    I,J : INTEGER ;
    package INT_IO is new INTEGER_IO(INTEGER) ;
    use INT_IO ;
begin
    I := 17 ;
    J := 60 ;
    PUT(I*J) ;
end FIRST_MULTIPLY ;
```

The program shown in example 1.1 prints the product of 17 and 60. The lines started by the words *with, package,* and *use* are included for "housekeeping" purposes; we will return to them in section 1.4. Aside from housekeeping, the program has three parts: the *header,* the *declarations,* and the *body*. The header simply indicates the program name. (This program has been given the name FIRST_ MULTIPLY.) The declarations list the programmer-defined entities that will be used inside the program. In this program, the names I and J are declared to represent integer variables. Finally, the body of the program lists the instructions that the computer will follow in executing the program.

In the header line, the name of the program is enclosed by the words *procedure* and *is*. These words are called *reserved words* in Ada because they are reserved for a particular purpose. The word *procedure,* for instance, indicates that we are reading the header line. Other reserved words are set aside for other situations. Although no typographical distinction is needed in an actual program, reserved words appear in lower case throughout this book to make the sample programs more readable.

Each variable in an Ada program is associated with a *data type*. The data type of a variable indicates, as you would expect, the type of data that the variable can hold. In this program, I and J are declared to have the type INTEGER. A variable of type INTEGER can store one integer number. In later chapters we will use data types that can represent other forms of data, such as rational numbers or matrices. Until then, our sample programs will use variables of type INTEGER.

Like the program name, the program body is surrounded by a pair of reserved words. The reserved word *begin* precedes the body; the reserved word *end* follows it. The first two lines of the program shown in example 1.1 assign values to the variables I and J. Statements in this form are called *assignment* statements. The two-character symbol ':=' is called the *assignment operator*.

The final statement in the body of this program is a PUT statement. PUT is a predefined "subprogram" that accepts a value and prints it. In this case, PUT has been given the expression I*J. When the PUT statement is reached, this expression will be evaluated; the current values of the variables will be substituted where they appear. Since I and J will equal 17 and 60 when the PUT is reached, the value of I*J will be 17*60, or 1020. Hence, the value 1020 will be printed.

Example 1.2

```
with TEXT_IO; use TEXT_IO;
program SECOND_MULTIPLY is
    I,J,PRODUCT : INTEGER ;
    package INT_IO is new INTEGER_IO(INTEGER) ;
    use INT_IO ;
begin
    I := 17 ;
    J := 60 ;
    PRODUCT := I * J ;
    PUT(PRODUCT) ;
end SECOND_MULTIPLY ;
```

The program shown in example 1.2 illustrates the use of expressions in assignment statements. When the third assignment is encountered, PRODUCT will be assigned the value of the expression. Any valid expression can appear in the right-hand part of an assignment statement, as long as both sides of the assignment have the same data type. For instance, if the variable on the left of the assignment operator is of type INTEGER, then the expression on the right should yield a value of type INTEGER. (We discuss the other arithmetic operations available for use with integers in the following section.)

The names that can be chosen for variables are subject to some constraints. Program names are subject to these constraints as well. In fact, variable names and program names fall into the same lexical category: both are examples of *identifiers*. Here are the rules governing identifiers:

1. The first character of an identifier must be a letter.
2. Subsequent characters of an identifier may include letters, digits, and underscores.
3. An identifier can have any number of characters.
4. A programmer-declared identifier cannot conflict with a reserved word. (Appendix A gives a list of Ada's reserved words.)

Here are some valid identifiers:

```
MY_NAME
Number
X
Count3
NEXT_LINE_OF_TEXT
```

Here are some invalid identifiers:

```
16BASE         --begins with a digit
Name&Age       --contains an invalid symbol
```

Notice that the fourth rule disallows variable names like BEGIN or END, which are already reserved by Ada. Importantly, differences in upper case and lower case letters do not make identifiers distinct from one another. Hence, the identifiers PAGE, Page, and pAGe are treated as if they referred to the same entity. All other distinctions are significant, however. The identifiers BIRTH_DAY and BIRTHDAY, for instance, are distinct because one has an underscore and the other does not.

Example 1.3

```
with TEXT_IO; use TEXT_IO;
procedure THIRD_MULTIPLY is
   I,J,PRODUCT : INTEGER ;
   package INT_IO is new INTEGER_IO(INTEGER) ;
   use TEXT_IO, INT_IO ;
begin
   GET(I) ;      --read the first number
   GET(J) ;      --read the second number
   PRODUCT := I * J ;     --multiply them
   PUT(PRODUCT) ;     --print the result
end THIRD_MULTIPLY ;
```

The program shown in example 1.3 introduces two additional features of the language. First, the statements in the body are annotated with *comments*. A comment has no effect on program execution; its purpose is to make the program easier for a human reader to understand. To place a comment in an Ada program, we simply precede it with a pair of hyphens. A comment can be placed on the same line as a program statement or on a separate line altogether. Since the characters following the hyphens are ignored, a comment cannot be followed by another statement on the same line. For instance, the assignment that appears in the comment below will not be executed:

```
--now we will add.  X:= 5 + 3;
```

The second feature introduced in example 1.3 is the GET statement. Like PUT, GET is a predefined subprogram that can be invoked within another program. It accepts a value from the user and places the value in the specified variable. In example 1.3 it is used to fetch values for I and J.

The use of GET obviously makes THIRD__MULTIPLY far more flexible than SECOND__MULTIPLY. The latter can calculate the product of only one pair of numbers, while the former allows the user to enter a new pair each time the program is run. We will return to the subject of input and output in section 1.4 and again in later chapters.

Before going further, we should note the use of semicolons as punctuation in Ada programs. The semicolon is used in Ada to terminate statements. Regrettably, this rule is easier to state than to use in practice, because no comparable rule is available to define what a "statement" is in loose terms. The program header is not regarded as a separate statement, for instance, while the entries in the variable declarations are. The best route to familiarity with Ada's semicolon rules is simply to follow the usage shown in the sample programs.

1.2 INTEGER EXPRESSIONS

Ada provides ten arithmetic operators for use with integers. Three of the operators are *unary* operators; that is, they accept a single value and return a single value. The others are *binary* operators, meaning that they accept a pair of values. The unary operators are

+	for identity
−	for negation
abs	for absolute value

The binary operators are

+	for addition
−	for subtraction
*	for multiplication
/	for division
mod	for modulus

rem	for remainder
**	for exponentiation

The unary operators, along with the first three binary operators, are equivalent to their counterparts in conventional arithmetic. Here are some expressions written with these operators:

```
X * X              --square of X
-SIGN              --negative of SIGN
+I                 --same as I
4 + 6              --10
abs (4 - 6)        --2
```

When used with integers, the division operator differs from conventional division. This difference arises because an arithmetic operator in Ada always returns a value that has the same type as its operands. For operations like addition and multiplication, this is consistent with conventional arithmetic: the sum of two integers is always an integer, for instance, as is the product of two integers. With division, however, the rule creates a discrepancy, because integer division in Ada returns only the integer part of the quotient. Thus, the integer expression 5/2 yields 2, not 2.5.

The operators *mod* and *rem* are provided so that programmers can circumvent this discrepancy. These operators return the remainder from a division operation. Given a pair of positive integers *i* and *j*, we can find the remainder of i/j with either of the following expressions:

```
i rem j
i mod j
```

For positive integers, the following identity holds:

```
i mod j = i rem j = i - (i/j)*j
```

If the two integers have different signs, then the two operators have slightly different effects. When one operand is positive and the other is negative, the identity above does not hold for *mod*. The following identity holds instead:

```
i mod j = i - (i/j)*j + j
```

Here are some expressions in which these operators appear:

```
7/3             --equals 2
7 rem 3         --equals 1
7 mod 3         --equals 1
-7/-3           --equals 2
-7 rem -3       --equals 1
-7 mod -3       --equals 1
-7/3            --equals -2
-7 rem 3        --equals -1
```

```
    -7 mod 3         --equals 2
    7/-3             --equals -2
    7 rem -3         --equals 1
    7 mod -3         --equals -2
```

When used with integers, the exponentiation operator requires that the right operand (i.e., the exponent) be a nonnegative value. If the exponent is zero, then the operation returns the value 1, as with ordinary exponent operations. Here are some examples of expressions in which the exponentiation operator is used:

```
    X ** 2      --square of X
    J ** 0      --equals 1 for any J
    2 ** BITS
    M ** N
    A*(X**2) + B*X + C
```

Expressions in Ada are evaluated according to rules of operator precedence similar to the standard precedence rules of algebra. Each operator has a level of precedence that determines the order in which the components of an expression will be processed. Four levels of precedence are defined for the integer operators:

```
    **
    * / mod rem
    + - abs       --unary
    + -           --binary
```

When an integer expression is encountered, the exponentiation operations are performed first; binary additions and subtractions are performed last. When an expression contains several operators on the same level of precedence, they are evaluated from left to right. Thus,

```
    10 + 6 * 2        --equals 22, not 32
    2 ** 3 + 1        --equals 9, not 16
    10*3/2            --equals 15, not 10
```

Parentheses can be inserted in an expression to override the precedence rules. Thus,

```
    (10 + 6) * 2        --equals 32
    2**(3 + 1)          --equals 16
    10 * (3/2)          --equals 10
```

When writing large integers, we can use two options provided by Ada to make them more readable. First, we can insert underscores in them to separate groups of digits. These embedded underscores have no effect on the value of the number. Hence, the following integers all have the same value:

```
    1234567
    123_4567
    1_234_567
```

Second, we can use scientific notation. A number written in scientific notation has two parts: a *mantissa* and an *exponent*. In the form of scientific notation provided by Ada, the mantissa and the exponent are separated by an *E* or an *e*. A number written in scientific notation is equal to the mantissa multiplied by ten to the power of the exponent. For example:

```
5e0         --equals 5
5E1         --equals 50
5E2         --equals 500
0e2         --equals 0
```

An integer written in scientific notation cannot have a negative exponent. Ada imposes this restriction because raising a number to a negative power generally yields a noninteger result. Of course, no such restriction applies to the mantissa.

1.3 OBJECT DECLARATIONS

In Ada, the term *object* refers to a place where values can be stored. A variable is one kind of object; other kinds of objects include *constants* and *numbers*. This section will examine declarations for all three. We have seen some examples of variable declarations already. In general, a variable declaration takes the form of a list of identifiers followed by the name of the desired type. Variable declarations may also specify the value that the variables will initially have. Here are some examples of variable declarations:

```
SUM : INTEGER;
P, Q : INTEGER;
SALES1, SALES2, SALES3 : INTEGER := 0;
```

The first declaration creates an integer variable named SUM. The second creates integer variables named P and Q. The third creates integer variables named SALES1, SALES2, and SALES3 and assigns them an initial value of 0. The practice of initializing a variable within its declaration offers several advantages. First, it eliminates the need to write an additional assignment in the body of the program. Second, it makes the initial value of the variable easier for a human reader to find. Third, and most usefully, it prevents "undefined value" errors—i.e., it eliminates the possibility that the program will attempt to access the variable before it has been given a value.

The initial value specified in a variable declaration can be an expression. If the expression includes other variables, then those variables must be declared and initialized first. Here is a sequence of declarations in which this requirement is met:

```
MASS : INTEGER := 10;
ACCEL : INTEGER := 5;
FORCE : INTEGER := MASS * ACCEL;
```

In this case, FORCE will be initialized to 50. Transposing the declarations for FORCE and ACCEL, however, would make the sequence invalid. Here is another sequence of declarations that meets the requirement:

```
POS1 : INTEGER := -3;
POS2 : INTEGER := 7;
LENGTH : INTEGER := abs (POS1-POS2);
```

A constant declaration associates an identifier with a value that remains unchanged during program execution. Constant declarations have the same form as variable declarations, except that the type identifier is preceded by the reserved word *constant*. Here are some examples of constant declarations:

```
Freezing_Point : constant INTEGER := 0;
SPEED_LIMIT : constant INTEGER := 55;
CAPACITY : constant INTEGER := 2e5;
LOAD : constant INTEGER := CAPACITY/3;
```

As with variable declarations, the value specified in a constant declaration may be an expression. Variables and constants can appear interchangeably in constant and variable declarations, as long as the declarations are made in a valid order. Unlike a variable, however, a constant cannot be assigned a new value in the body of the program.

A number declaration is similar to a constant declaration, with two differences. First, a number declaration does not include a type identifier; the type of the value is determined by inspection. Second, the value specified in a number declaration must be given as a *static expression*. A static expression is one that can be evaluated before the program runs. The only values that can appear in a static numeric expression are literals, constants declared with static expressions, and declared number identifiers. A literal is an explicit value, like "−10" or "3e4." The only operations that can be performed in a static expression are the predefined operations, such as addition and multiplication. Here are some examples of valid number declarations:

```
DOUBLE_BYTE : constant := 2 ** 16;
Line_Size : constant := 80;
Lines : constant := 24;
Page_Size : constant := Line_Size*Lines;
```

Constant and number declarations are useful when a certain value is likely to be used many times throughout a program. First, associating such a value with a symbolic identifier makes the program easier to understand. When we read a program for the first time, we will doubtless find it more comprehensible if the programmer used identifiers instead of unexplained literals. Second, constant and number declarations make a program easier to transport and modify. If we want to alter a machine-dependent value, say, we can do so more quickly and reliably if the value has been associated with an identifier. It is obviously far simpler to modify one line—a constant declaration—than to update every occurrence of the value in the program body.

In Ada parlance, the action of associating an identifier with an entity in the program is called *elaboration*. Just as statements are *executed* and expressions are *evaluated,* declarations in an Ada program are said to be *elaborated*. Taken together, the declarations in an Ada program are termed the *declarative part* of the program.

1.4 INPUT AND OUTPUT

The example programs that appeared earlier in this chapter included a number of cryptic statements. One of them preceded the program headers:

```
with TEXT_IO; use TEXT_IO;
```

The others were placed in the declarative part:

```
package INT_IO is new INTEGER_IO(INTEGER);
use INT_IO;
```

These statements are needed so we can perform input and output in the program body. The first one informs Ada that we will be using input–output subprograms like GET and PUT. The second and third are used to "awaken" the subprograms. To be precise, we should call this awakening process *instantiation*. We will not concern ourselves with the details of instantiation until much later. For now, all we need to know is how to make it work.

A separate instantiation is needed for each data type that will be involved in input or output. In the examples, we used only the type INTEGER. Suppose we are also using GET or PUT with an integer type called WXYZ. We would then write this in the declarative part:

```
package INT_IO is new INTEGER_IO(INTEGER);
package WXYZ_IO is new INTEGER_IO(WXYZ);
use INT_IO, WXYZ_IO;
```

The pattern should now be evident. First, we chose an identifier for each instantiation. (Any identifier is suitable.) For the instantiation of INTEGER input–output we used INT_IO. For WXYZ we used WXYZ_IO. We then placed each one in a different *package . . . is* statement as shown. Finally, we listed them in a *use* clause. Ada's input–output subprograms are now available for the given types.

As used here, GET and PUT control whatever device has been designated the *default file*. We will assume that the default file is a keyboard terminal with a screen or a printer of some kind. The following statement, then, will accept a value from the terminal:

```
GET(USER_ID);
```