# 4 Elementary Data Types

## 4.1 Integer types

The word *integer* is familiar from mathematics where it denotes the unbounded, ordered sequence of numbers:

$$\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots$$

The term is used in programming to denote something quite different: a specific data type. Recall that a data type is a set of values and a set of operations on those values. Let us begin by defining the set of values of the Integer data type.

Given a word of memory, we can simply define a set of values by the usual interpretation of the bits of the word as a binary value. For example, if a word with 8 bits contains the sequence 1010 0011, it is interpreted as:

$$(1 \times 2^7) + (1 \times 2^5) + (1 \times 2^1) + (1 \times 2^0) = 128 + 32 + 2 + 1 = 163$$

The range of possible values is 0..255, or in general $0..2^B - 1$ for a word of $B$ bits. A data type with this set of values is called an *unsigned integer* and a variable of this type can be declared in C as:

$\boxed{\text{C}}$

```
unsigned int v;
```

Note that the number of bits in a value of this type varies from one computer to another.[1] Today the most common word size is 32 bits and an (unsigned) integer value is in the range $0..2^{32} - 1 \approx 4 \times 10^9$. Thus, the set of mathematical integers is unbounded while integer types are limited to a finite range of values.

Since they cannot represent negative numbers, unsigned integers are mostly useful as representations of values read from external equipment. For example, a temperature probe may return 10 bits of information; the unsigned value in the range 0..1023 must then be translated into ordinary (positive and negative) numbers. Unsigned integers are also used to represent characters (see below). Unsigned integers should not be used for ordinary computation, because most computer instructions work on signed integers and a compiler may emit extra instructions for unsigned values.

The range of values that a variable can have may not fit into a single word, or it may fit into a partial word. Length specifiers can be added to indicate different integer types:

[1] In fact, unsigned int need not be the same size as a word of memory.

C

```
unsigned int v1;
unsigned short int v2;
unsigned long int v3;
```

In Ada, additional types such as Long_Integer are predefined along with ordinary Integer. The actual interpretation of length specifiers like long and short varies from one compiler to another; some compilers may even give the same interpretation to two or more specifiers.

The representation of signed numbers in mathematics uses a special character "–" followed by the usual representation of the absolute value of the number. This representation is not convenient for computer instructions to work with. Instead most computers represent signed integers in a notation called *two's complement*. A positive number is represented by a leading zero bit followed by the usual binary representation of the value. It follows that the largest positive integer that can be represented in a word with $w$ bits is only $2^{w-1}-1$.

To obtain the representation of $-n$ from the binary representation $B = b_1 b_2 \dots b_w$ of $n$:

- Take the logical *complement* of B, that is, change each $b_i$ from zero to one or from one to zero.
- Add one.

For example, the representations of $-1$, $-2$ and $-127$ as 8-bit words are:

$$\begin{array}{rclclcl}
1 &=& 0000\,0001 &\rightarrow& 1111\,1110 &\rightarrow& 1111\,1111 &=& -1\\
2 &=& 0000\,0010 &\rightarrow& 1111\,1101 &\rightarrow& 1111\,1110 &=& -2\\
127 &=& 0111\,1111 &\rightarrow& 1000\,0000 &\rightarrow& 1000\,0001 &=& -127
\end{array}$$

A negative value will always have a one in the high-order bit.

Two's complement is convenient because if you do ordinary binary arithmetic on values, the result is correct:

$$\begin{array}{rcl}
(-1)-1 &=& -2\\
1111\,1111 - 0000\,0001 &=& 1111\,1110
\end{array}$$

Note that the bit string 1000 0000 cannot be obtained from any positive value. It represents $-128$, even though its positive counterpart 128 cannot be represented as an 8-bit two's complement number. This asymmetry in the range of integer types must be kept in mind, especially when dealing with short types.

An alternative representation of signed numbers is *one's complement*, in which the representation of $-n$ is just the complement of $n$. The set of values is then symmetric, but there are two representations of zero: 0000 0000 called positive zero, and 1111 1111 called negative zero.

If you do not use a special syntax like unsigned in a declaration, the default is to use signed integers:

```
int i;                    /* Signed integer in C */
I: Integer;               -- Signed integer in Ada
```

## Integer operations

Integer operations include the four basic operations: addition, subtraction, multiplication and division. They can be used to form *expressions*:

```
a + b / c – 25 * (d – e)
```

Normal mathematical rules of precedence apply to integer operations; parentheses can be used to change the order of computation.

The result of an operation on a signed integer must not be outside the range of values; otherwise, overflow will occur as discussed below. For unsigned integers, arithmetic is cyclic. If short int is stored in one 16-bit word, then:

C

```
unsigned short int i;       /* Range of i = 0..65535 */
i = 65535;                  /* Largest possible value */
i = i + 1;                  /* Cyclic arithmetic, i = 0 */
```

The designers of Ada 83 made the mistake of not including unsigned integers. Ada 95 generalizes the concept of unsigned integers to *modular types*, which are integer types with cyclic arithmetic on an arbitrary modulus. A standard unsigned byte can be declared:

Ada

```
type Unsigned_Byte is mod 256;
```

while a modulus that is not a power of two can be used for hash tables or random numbers:

Ada

```
type Random_Integer is mod 41;
```

Note that modular types in Ada are portable since only the cyclic range is part of the definition, not the *size* of the representation as in C.

## Division

In mathematics, division of two integers $a/b$ produces two values, a *quotient* $q$ and a *remainder* $r$ such that:

$$a = q * b + r$$

Since arithmetical expressions in programs return a single value, the "/" operator is used to return the quotient, and a distinct operator (called "%" in C and rem in Ada) returns the remainder. The expression 54/10 yields the value 5, and we say that the result of the operation has been *truncated*. Pascal uses a separate operator div for integer division.

The definition of integer division is not trivial when we consider negative numbers. Is the expression −54/10 equal to $-5$ or $-6$? In other words, is truncation done to smaller (more negative) values or to values closer to zero? One choice is to truncate towards zero, since the equation for integer division can be maintained just by changing the sign of the remainder:

$$-54 = -5 * 10 + (-4)$$

There is another mathematical operation called *modulo* which corresponds to truncation of negative numbers to smaller (more negative) values:

$$\begin{aligned} -54 &= -6 * 10 + 6 \\ -54 \text{ mod } 10 &= 6 \end{aligned}$$

Modulo arithmetic has applications whenever arithmetic is done on finite ranges, such as error-correcting codes used in communications systems.

The meaning of "/" and "%" in C is implementation dependent, so programs using these integer operators may not be portable. In Ada, "/" always truncates towards zero. The operator rem returns the remainder corresponding to truncation towards zero, while the operator mod returns the remainder corresponding to truncation towards minus infinity.

## Overflow

An operation is said to *overflow* if it produces a result that is not within the range of defined values. The following discussion is given in terms of 8-bit integers for clarity.

Suppose that the signed integer variable i has the value 127 and that we increment i. The computer will simply add one to the integer representation of 127:

$$0111\,1111 + 0000\,0001 = 1000\,0000$$

to obtain $-128$. This is not the correct result and the error is called overflow.

Overflow may cause strange bugs:

$\boxed{\text{C}}$

```
    for (i = 0; i ¡ j*k; i++) ...
```

If the expression j*k overflows, the upper bound may be negative and the loop will not be executed.

Suppose now that the variables a, b and c have the values 90, 60 and 80, respectively. The expression (a–b+c) evaluates to 110, because (a–b) evaluates to 30, and then the addition gives 110. However, the optimizer may choose to evaluate the expression in a different order, (a+c–b), giving an incorrect answer because the addition (a+c) gives the value 170 which overflows. If your high-school teacher told you that addition is commutative and associative, he/she was talking about mathematics and not about programming!

Some compilers have the option of checking every integer operation for overflow, but this may be prohibitive in terms of run-time overhead unless the detection of overflow is done by hardware. Now that most computers use 32-bit memory words, integer overflow is rarely a problem, but you must be aware of it and take care not to fall into the traps demonstrated above.

### Implementation

Integer values are stored directly in memory words. Some computers have instructions to compute with partial words or even individual bytes. Compilers for these computers will usually map short int into partial words, while compilers for computers that recognize only full words will implement all integer types identically. long int will be mapped into two words to achieve a larger range of values.

Addition and subtraction are compiled directly into the corresponding instructions. Multiplication is also implemented as a single instruction but requires significantly more processing time than addition and subtraction. Multiplication of two words stored in registers R1 and R2 produces a result that is two words long and will require two registers to store. If the register containing the high-order value is not zero, the result has overflowed.

Division requires the computer to perform an iterative algorithm similar to “long division” done by hand. This is done in hardware and you need not concern yourself with the details, but if efficiency is important it is best to avoid division.

Arithmetic on long int takes more than twice the amount of time as it does on int. The reason is that an instruction must be used to transfer a possible “carry” from the low-order words to the high-order words.

## 4.2 Enumeration types

Programming languages such as Fortran and C define data in terms of the underlying computer. Real-world data must be explicitly mapped onto data types that exist on the computer, in most cases one of the integer types. For example, if you are writing a program to control a heater you might have a variable dial that stores the current position of the dial. Suppose that the real-world dial has four positions: *off*, *low*, *medium*, *high*. How would you declare the variable and denote the positions? Since the computer does not have instructions that work on four-valued memory words, you will choose to declare the variable as an integer, and you will choose four specific integers (say 1, 2, 3, 4) to denote the positions:

C

```
int dial;                  /* Current position of dial */
if (dial ¡ 4) dial++;      /* Increment heater level*/
```

The obvious problem with using integers is that the program becomes hard to read and maintain. You must write extensive documentation and continually refer to it in order to understand the code. The first improvement that can be made is to document the intended values internally:

C

```
#define  Off        1
#define  Low        2
#define  Medium     3
#define  High       4
```

```
int dial;
if (dial ¡ High) dial++;
```

Improving the documentation, however, does nothing to prevent the following problems:

C

```
dial = −1;              /* No such value */
dial = High + 1;        /* Undetected overflow */
dial = dial * 3;        /* Meaningless operation */
```

To be precise, representing the four-position dial as an integer allows the programmer to assign values that are not within the intended range, and to execute operations that are meaningless for the real-world object. Even if the programmer does not intentionally create any of these problems, experience has shown that they frequently arise from misunderstandings among software team members, typographical errors and other mistakes typical in the development of complex systems.

The solution is to allow the program designer to create *new* types that correspond exactly to the real-world objects being modeled. The requirement discussed here—a short ordered sequence of values—is so common that modern programming languages support creation of types called *enumeration types*. In Ada, the above example would be:

Ada

```
type Heat is (Off, Low, Medium, High);
Dial: Heat;

Dial := Low;
if Dial ¡ High then Dial := Heat'Succ(Dial);

Dial := −1;                      -- Error
Dial := Heat'Succ(High);         -- Error
Dial := Dial * 3;                -- Error
```

Before explaining the example in detail, note that C has a construct that is superficially the same:

C

```
typedef enum {Off, Low, Medium, High} Heat;
```

However, variables declared to be of type Heat are still *integers* and none of the above statements are considered to be errors (though a compiler may issue a warning):

C

```
Heat dial;

dial = −1;              /* Not an error! */
dial = High + 1;        /* Not an error! */
dial = dial * 3;        /* Not an error! */
```

In other words, the enum construct is just a means of documentation that is more convenient than a long series of define's, but it does not create a new type.

Fortunately, C++ takes a stricter interpretation of enumeration types and does not allow assignment of an integer value to a variable of an enumeration type; the above three statements are in error. However, values of enumeration types can be implicitly converted to integers so the type checking is not complete. Unfortunately, C++ provides no operations on enumeration types so there is no predefined way to increment a variable of enumeration type. You can write your own function that takes the result of an integer expression and then explicitly converts it back to the enumeration type:

C++

```
dial = (Heat) (dial + 1);
```

Note the implicit conversion of dial to integer, followed by the explicit conversion of the result back to Heat. The "++" and "−−" operators can be overloaded in C++ (Section 10.2), so they can be used for defining operations on enumeration types that are syntactically the same as those of integer types.

In Ada, the type definition creates a new type Heat. The values of this type are *not* integers. Any attempt to exceed the range or to use integer operations will be detected as an error. If your finger slips and you enter Higj instead of High, an error will be detected, because the type contains exactly the four values that were declared; if you had used integers, 5 would be as valid an integer as 4.

Enumeration types are types just like integers: you can declare variables and parameters to be of these types. However, they are restricted in the set of operations that can be performed on values of the type. These include assignment (:=), equality (=) and inequality (/=). Since the set of values in the declaration is interpreted as an ordered sequence, the relational operators (<,>,>=,<=) are defined.

Given an enumeration type T and a value V of type T, the following functions called *attributes* are defined in Ada:

- T'First returns the first value of T.
- T'Last returns the last value of T.
- T'Succ(V) returns the successor of V.
- T'Pred(V) returns the predecessor of V.
- T'Pos(V) returns the position[2] of V within the list of values of T.
- T'Val(I) returns the value at the I'th position in T.

Attributes make the program robust to modifications: if values are added to an enumeration type, or if the values are rearranged, loops and indices remain unaltered:

[2]The position of a value is defined by its position (starting from zero) in the declaration of the type.

Ada

```
for I in Heat'First .. Heat'Last – 1 loop
    A(I) := A(Heat'Succ(I));
end loop;
```

Not every language designer believes in enumeration types. Eiffel does not contain enumeration types for the following reasons:

- A desire to keep the language as small as possible.
- The same level of safety can be obtained using assertions (Section 11.5).
- Enumeration types are often used with variant records (Section 10.5); when inheritance (Section 14.3) is used correctly there is less need for enumeration types.

Enumeration types should be used whenever possible in preference to ordinary integers with lists of defined constants; their contribution to the reliability of a program cannot be overestimated. C programmers do not have the benefit of type-checking as in Ada and C++, but they should still use enum for its significant contribution to readability.

### Implementation

I will let you in on a secret and tell you that the values of an enumeration type are represented in the computer as consecutive integers starting from zero. The Ada type checking is purely compile-time, and operations such as "¡" are ordinary integer operations.

It is possible to request the compiler to use a non-standard representation of enumeration types. In C this is specified directly in the type definition:

C

```
typedef enum {Off=1, Low=2, Medium=4, High=8} Heat;
```

while in Ada a representation specification is used:

Ada

```
type Heat is (Off, Low, Medium, High);
for Heat use (Off => 1, Low => 2, Medium => 4, High => 8);
```

## 4.3 Character type

Though computers were originally invented in order to do numerical calculations, it quickly became apparent that non-numerical applications are just as important. Today non-numerical applications, such as word-processors, educational programs and databases, probably outnumber mathematical applications. Even mathematical applications such as financial software need text for input and output.

From the point of view of a software developer, text processing is extremely complicated because of the variety of natural languages and writing systems. From the point of view of programming languages text processing is relatively simple, because the language assumes that the set of characters forms a short, ordered sequence of values, that is, characters can be defined as an enumeration type. In fact, except for languages like Chinese and Japanese that use thousands of symbols, the 128 or 256 possible values of a signed or unsigned 8-bit integer suffice.

The difference in the way that Ada and C define characters is similar to the difference in the way that they define enumeration types. In Ada, there is a predefined enumeration type:

Ada

```
type Character is (..., 'A', 'B', ...);
```

and all the usual operations on enumeration types (assignment, relations, successor, predecessor, etc.) apply to characters. In Ada 83, Character is limited to the 128 values of the American standard ASCII, while in Ada 95, the type is assumed to be represented in an unsigned byte, so the 256 values required by international standards are available.

In C, char is just a very small integer type and all the following statements are accepted since char and int are essentially the same:

C

```
char c;
int i;
c = 'A' + 10;           /* Convert char to int and back */
i = 'A';                /* Convert char to int */
c = i;                  /* Convert int to char */
```

char is a distinct type in C++, but is convertible to and from integers so the above statements remain valid.

For non-alphabetic languages, 16-bit characters can be defined. These are called wchar_t in C and C++, and Wide_Character in Ada 95.

The only thing that distinguishes characters from ordinary enumeration or integer types is the special syntax ('A') for the set of values, and more importantly the special syntax that exists for arrays of characters called *strings* (Section 5.6).

## 4.4 Boolean type

Boolean is a predefined enumeration type in Ada:

Ada

```
type Boolean is (False, True);
```

The Boolean type is extremely important because:

- Relational operators (=, >, etc.) are functions which return a value of Boolean type.

- The if-statement takes an expression of Boolean type.
- Operators for Boolean algebra (and, or, not, xor) are defined on the Boolean type.

C does not define a separate type; instead integers are used with the following interpretation:

- Relational operators return 1 if successful and 0 if not.
- The if-statement takes the false branch if the integer expression evaluates to zero, and the true branch otherwise.

There are several methods for introducing Boolean types into C. One possibility is to define a type which will allow declaration of functions with Boolean result:

C

```
typedef enum {false, true} bool;

bool data_valid(int a, float b);

if (data_valid(x, y)) . . .
```

but of course this is purely for documentation and readability, because a statement like:

C

```
bool b;
b = b + 56;                 /* Add 56 to "true" ?? */
```

is still accepted and can cause obscure bugs.

In C++, bool is predefined *integer* type (not an enumeration type), with implicit conversion between non-zero values and a literal true, and between zero values and false. A C program with bool defined as shown above can be compiled in C++ simply by removing the typedef.

Even in C it is better not to use the implicit conversion of integer to Boolean, but rather to use explicit equality and inequality operators:

C

```
if (a+b == 2) . . .         /* This version is clearer than */
if (!(a+b-2)) . . .         /* . . . this version. */
if (a+b != 0) . . .         /* This version is clearer than */
if (a+b) . . .              /* . . . this version. */
```

Finally, C supports Boolean algebra using short-circuit operators which we will discuss in Section 6.2.

## 4.5 * Subtypes

In the previous sections we have discussed integer types which allow computations to be performed on the large range of values that can be represented in a memory word, and enumeration types which work with smaller ranges but do not allow arithmetic computation to be done. However, in many cases we would like to do computations on smaller ranges of integers. For example, there should be some way of detecting errors such as:

```
Temperature: Integer;
Temperature := -280;      -- Below absolute zero!

Compass_Heading: Integer;
Compass_Heading := 365; -- Compass shows 0..359 degrees!
```

Suppose we try to define a new class of types:

```
type Temperatures is Integer range -273 .. 10000--; Not Ada!
type Headings is Integer range 0 .. 359;           -- Not Ada!
```

This will solve the problem of checking errors caused by values outside the range of the type, but the question remains: are these two different types or not? If they are, then:

```
Temperature * Compass_Heading
```

is a valid arithmetic expression on an integer type; if not, type conversion must be used.

In practice, both of these interpretations are useful. Computation concerning the physical world tends to involve calculations among values of many ranges. On the other hand, indices into tables or serial numbers do not require computation between types: it makes sense to ask for the next index into a table, but not to add a table index to a serial number. Ada provides two separate facilities for these two classes of type definition.

A *subtype* is a restriction on an existing type. Discrete (integer and enumeration) types can have a *range constraint*:

```
subtype Temperatures is Integer range -273 .. 10000;
Temperature: Temperatures;

subtype Headings is Integer range 0 .. 359;
Compass_Heading: Headings;
```

The type of a value of a subtype S is still that of the underlying *base* type T; here, the base type of both Temperatures and Headings is Integer. The type is determined at *compile-time*. The value of a subtype has the same representation as that of the base type and is acceptable wherever a value of the base type is needed:

```
Temperature * Compass_Heading
```

is a valid expression, but the statements:

```
Temperature := -280;
Compass_Heading := 365;
```

cause errors because the values are not within the ranges of the subtypes. Violations of the range of a subtype are checked at *run-time*.

Subtypes can be defined on any type that can have its set of values restricted in a meaningful way:

```
subtype Upper_Case is Character range 'A' .. 'Z';
U: Upper_Case;
C: Character;

U := 'a';                  -- Error, out of range
C := U;                    -- Always OK
U := C;                    -- May cause an error
```

Subtypes are essential in defining arrays as will be discussed in Section 5.5. In addition, a named subtype can be used to simplify many statements:

```
if C in Upper_Case then ...        -- Range check
for C1 in Upper_Case loop ...      -- Loop bounds
```

## 4.6 * Derived types

The second interpretation of the relation between two similar types is that they represent separate types that cannot be used together. In Ada, such types are called *derived* types and are indicated by the word new in the definition:

```
type Derived_Character is new Character;
C: Character;
D: Derived_Character;

C := D;                    -- Error, types are different
```

When one type is derived from another type (called its *parent* type), it inherits a *copy* of the set of values and a *copy* of the set of operations, but the types remain distinct. However, it is always possible to explicitly convert between types that are derived from one another:

```
D := Derived_Character(C);         -- Type conversion
C := Character(D);                 -- Type conversion
```

It is even possible to specify a different representation for the derived types; type conversion will then convert between the two representations (Section 5.9).

A derived type may include a restriction on the range of values of the parent type:

```
type Upper_Case is new Character range 'A' .. 'Z';
U: Upper_Case;
```

```
C: Character;

C := Character(U);         -- Always valid
U := Upper_Case(C);        -- May cause an error
```

Ada 83 derived types implement a weak version of inheritance, which is a central concept of object-oriented languages (see Chapter 14). The revised language Ada 95 implements true inheritance by extending the concept of derived types; we will return to study them in great detail.

### Integer types

Suppose that we have defined the following type:

```
type Altitudes is new Integer range 0 .. 60000;
```

This definition works correctly when we program a flight simulation on a 32-bit workstation. What happens when we transfer the program to a 16-bit controller in our aircraft avionics? Sixteen bits can represent signed integers only up to the value 32,767. Thus the derived type would be in error (as would a subtype or a direct use of Integer), violating the portability of programs which is a central goal of the Ada language.

To solve this problem, it is possible to derive an *integer* type without explicitly specifying the underlying parent type:

```
type Altitudes is range 0 .. 60000;
```

The compiler is required to choose a representation that is appropriate for the requested range—Integer on a 32-bit computer and Long_Integer on a 16-bit computer. This unique feature makes it easy to write numeric programs in Ada that are portable between computers with different word lengths.

The drawback of integer types is that each definition creates a new type and we can't write calculations that use different types without type conversions:

```
I: Integer;
A: Altitude;

A := I;                    -- Error, different types
A := Altitude(I);          -- OK, type conversion
```

Thus there is an unavoidable tension between:

- Subtypes are potentially unsafe because mixed expressions may be written and portability is hard to achieve.

- Derived types are safe and portable, but may make a program difficult to read because of extensive type conversions.

## 4.7 Expressions

An expression may be as simple as a literal (24, 'x', True) or a variable, or it may be a complex composition involving operations (including calls to system- or user-defined functions). When an expression is *evaluated* it produces a *value*.

Expressions occur in many places in a program: assignment statements, Boolean expressions in if-statements, limits of for-loops, parameters of procedures, etc. We first discuss the expression by itself and then the assignment statement.

The value of a literal is what it denotes; for example, the value of 24 is the integer represented by the bit string 0001 1000. The value of a variable V is the contents of the memory cell it denotes. Note the potential for confusion in the statement:

```
V1 := V2;
```

V2 is an *expression* whose value is the contents of a certain memory cell. V1 is the *address* of a memory cell into which the value of V2 will be placed.

More complex expressions are created from a function or an operator and a set of parameters or operands. The distinction is mostly syntactical: a function with parameters is written in prefix notation sin(x), while an operator with operands is written in infix notation a+b. Since the operands themselves can be expressions, expressions of arbitrary complexity can be created:

```
a + sin(b) * ((c – d) / (e + 34))
```

When written in prefix notation, the order of evaluation is clearly defined except for the order of evaluation of the parameters of a single function:

```
max(sin(cos(x)), cos(sin(y)))
```

It is possible to write programs whose result depends on the order of evaluation of the parameters of a function (see Section 7.3), but such order dependencies should be avoided at all costs, since they are a source of obscure bugs when porting or even when modifying a program.

Infix notation brings its own problems, namely those of precedence and associativity. Almost all programming languages adhere to the mathematical standard of giving higher precedence to the multiplying operators ("*", "/") over the adding operators ("+", "–"), and other operators will have arbitrary precedences defined by the language. Two extremes are APL which does not define *any* precedences (not even for arithmetic operators), and C which defines 15 levels of precedence! Part of the difficulty of learning a programming language is to get used to the style that results from the precedence rules.

An example of a non-intuitive precedence assignment occurs in Pascal. The Boolean operator and is considered to be a multiplying operator with high precedence, whereas in most other languages such as C it is given a precedence below the relational operators. The following statement:

Pascal

```
if a > b and b > c then ...
```

is in error, because the expression is interpreted to be:

Pascal

```
if a > (b and b) > c then ...
```

and the syntax is not correct.

The meaning of an infix expression also depends upon the *associativity* of the operators, namely how operators of identical precedence are grouped: from left to right, or from right to left. In most cases, but not all, it does not matter (except for the possibility of overflow as discussed in Section 4.1). However, the value of an expression involving integer division can depend on associativity because of truncation:

C

```
int i = 6, j = 7, k = 3;
i = i * j / k;              /* Is the result 12 or 14 ? */
```

In general, binary operators associate to the left so the above example is compiled as:

C

```
i = (i * j) / k;
```

while unary operators associate to the right: in C !++i is computed as if written !(++i).

All precedence and associativity problems can be easily solved using parentheses; they cost nothing, so use them whenever there is the slightest possibility of confusion.

While precedence and associativity are specified by the language, evaluation order is usually left to the implementation to enable optimizations to be done. For example, in the following expression:

```
(a + b) + c + (d + e)
```

it is not defined if a+b is computed before or after d+e, though c will be added to the *result* of a+b before the *result* of d+e. The order can be significant if an expression uses *side-effects*, that is, if the evaluation of a sub-expression calls a function that modifies a global variable.

## Implementation

The implementation of an expression depends of course on the implementation of the operators used in the expression. Nevertheless, certain general principles are worth discussing.

Expressions are evaluated from the inside out; for example a*(b+c) is evaluated:[3]

[3]Note this order is not the only possible one; the optimizer might move the load of a to an earlier position or exchange the loads of b and c, without affecting the result of the computation.

```
load    R1,b
load    R2,c
add     R1,R2       Add b and c, result to R1
load    R2,a
mult    R1,R2       Multiply a and b+c, result to R1
```

It is possible to write an expression in a form which makes this evaluation order explicit:

```
b c + a *
```

Reading from left to right: the name of an operand means load the operand, and the symbol for an operator means apply the operator to the two most recent operands and replace all three (two operands and the operator) with the result. In this case, addition is applied to b and c; then multiplication is applied to the result and to a.

This form, called *reverse polish notation*[4] (*RPN*), may be used by the compiler: the expression is translated into RPN, and then the compiler emits instructions for each operand or operator as the RPN is read from left to right.

If the expression were more complex, say:

```
(a + b) * (c + d) * (e + f)
```

more registers would be needed to store the intermediate results: a+b, c+d, and so on. As the complexity increases, the number of registers will not suffice, and the compiler must allocate anonymous temporary variables in order to store the intermediate results. In terms of efficiency: up to a certain point, it is more efficient to increase the complexity of an expression rather than to use a sequence of assignment statements, so that unnecessary storing of intermediate results to memory can be avoided. However, the improvement quickly ceases because of the need to allocate temporary variables, and at some point the compiler may not even be able to handle the complex expression.

An optimizing compiler will be able to identify that the subexpression a+b in the following expression:

```
(a + b) * c + d * (a + b)
```

need only be evaluated once, but it is doubtful if a common subexpression would be identified in:

```
(a + b) * c + d * (b + a)
```

If the common subexpression is complex, it may be helpful to explicitly assign it to a variable rather than trust the optimizer.

*Constant folding* is another optimization. In the expression:

[4]Named for the Polish logician Lukasiewicz who was interested in a parenthesis-free logical calculus. The notation is *reversed* because he placed the operators before the operands.

```
2.0 * 3.14159 * Radius
```

the compiler will do the multiplication once at compile-time and store the result. There is no reason to lower the readability of a program by doing the constant folding yourself, though you may wish to assign a name to the computed value:

Ada

```
PI: constant := 3.14159;
Two_PI: constant := 2.0 * PI;
Circumference: Float := Two_PI * Radius;
```

## 4.8 Assignment statements

The meaning of the assignment statement:

```
variable := expression;
```

is that the value of expression should be stored in the memory address denoted by variable. Note that the left-hand side of the statement may also be an expression, as long as that expression evaluates to an address:

Ada

```
a(i*(j+1)) := a(i*j);
```

An expression that can appear on the left-hand side of an assignment statement is called an *l-value*; a constant is of course not an l-value. All expressions produce a value and so can appear on the right-hand side of an assignment statement; they are called *r-values*. The language will usually not specify the order of evaluation of the expressions on the left- and right-hand sides of an assignment. If the order affects the result, the program will not be portable.

C defines the assignment statement itself as an expression. The value of:

```
variable = expression;
```

is the same as the value of the expression on the right-hand side:

C

```
int v1, v2, v3;
v1 = v2 = v3 = e;
```

means assign (the value of) e to v3, then assign the result to v2, then assign the result to v1 and ignore the final result.

In Ada assignments are statements, not expressions, and there is no multiple assignment. The multiple declaration:

Ada

```
V1, V2, V3: Integer := E;
```

is considered to be an abbreviation for:

Ada

```
V1: Integer := E;
V2: Integer := E;
V3: Integer := E;
```

and not an instance of multiple assignment.

While C programming style makes use of the fact that assignment is an expression, it probably should be avoided because it can be a source of obscure programming errors. A notorious class of bugs is due to confusion between the assignment ("=") and the equality operator ("=="). In the following statement:

C

```
if (i = j) ...
```

the programmer probably intended just to compare i and j, but i is inadvertently modified by the assignment. Some C compilers regard this as such bad style that they issue a warning message.

A useful feature in C is the combination of an operator with an assignment:

Ada

```
v += e;                 /* This is short for ... */
v = v + e;              /*      this. */
```

Assignment operators are particularly important when the variable is complex, including array indexing, etc. The combination operator not only saves typing, but avoids the possibility of a bug if v is not written identically on both sides. Assignment operators are a stylistic device only, since an optimizing compiler can remove the second evaluation of v.

It is possible to prevent the assignment of a value to an object by declaring the object as a *constant*:

```
const int N = 8;            /* Constant in C */
N: constant Integer := 8;   -- Constant in Ada
```

Obviously a constant must be given an initial value.

There is a difference between a constant and a *static value*[5] which is known at compile-time:

Ada

```
procedure P(C: Character) is
   C1: constant Character := C;
   C2: constant Character := 'x';
begin
   ...
   case C is
      when C1 =>              -- Error, not static
      when C2 =>              -- OK, static
      ...
```

[5]Called a *constant expression* in C++.

```
        end case;
        ...
    end P;
```

The local variable C1 is a constant object, meaning that its value cannot be changed *within* the procedure, even though its value will be different each time the procedure is called. On the other hand, the case statement selections must be known at compile time.

Unlike C, C++ considers constants to be static:

C++

```
    const int N = 8;
    int a[N];                    // OK in C++, not in C
```

### Implementation

Once the expression on the right-hand side of an assignment statement has been evaluated, one instruction is needed to store the value of the expression in a memory location. If the left-hand side is complex (array indexing, etc.), additional instructions will be needed to compute the memory address that it specifies.

If the right-hand side is more than one word long, several instructions will be needed to store the value, unless the computer has a *block copy* operation which can copy a sequence of memory words given: the source starting address, the target starting address and the number of words to copy.

## 4.9 Exercises

1. Read your compiler documentation and list the precisions used for the different integer types.
2. Write $200+55=255$ and $100-150=-50$ in two's complement notation.
3. Let a take on all values in the ranges 50..56 and $-56..-50$, and let b be either 7 or $-7$. What are the possible quotients q and remainders r when a is divided by b? Use both definitions of remainder (denoted rem and mod in Ada), and display the results in graphical form. Hint: if rem is used, r will have the sign of a; if mod is used, r will have the sign of b.
4. What happens if you execute the following C program on a computer which stores short int values in 8 bits and int values in 16 bits?

   C

   ```
   short int i;
   int j = 280;
   for (i = 0; i ¡ j; i++) printf("Hello world");
   ```
5. If a non-standard representation of an enumeration type is used, how would you implement the Ada attribute T'Succ(V)?

6. What will the following program print? Why?

   $\boxed{\text{C}}$

   ```
   int i = 2;
   int j = 5;
   if (i & j) printf("Hello world");
   if (i && j) printf("Goodbye world");
   ```

7. What is the value of i after executing the following statements?

   $\boxed{\text{C}}$

   ```
   int i = 0;
   int a[2] = {10,11};
   i = a[i++];
   ```

8. C and C++ do not have an exponentiation operator; why?

9. Show how modular types in Ada 95 and unsigned integer types in C can be used to represent sets. How portable is your solution? Compare with the set type in Pascal.

10. Given an arithmetic expression such as:

    ```
    (a + b) * (c + d)
    ```

    Java specifies that it be evaluated from left to right, while C++ and Ada allow the compiler the evaluate the subexpression in any order. Why is Java more strict in its specification?

11. Compare the final construct in Java with constants in C++ and Ada.