Introduction to Systematic Programming
Unit 4 - Doing Arithmetic; Producing Better Output

4.1 Arithmetic expressions

When an algorithm involves some calculation, we show it by writing down an appropriate **arithmetic expression**, eg. on the right hand side of the assignment step:

```
TaxablePay := 52 * WeeklyPay - 10 * TaxCode
```

So far we have tacitly assumed that provided such expressions are formed according to the normal rules of arithmetic, they can be transferred directly into Ada programs (after writing `*` for multiplication for example). This is indeed generally correct, but as with all facets of programming, it is important to have an exact understanding, so let us now be more precise.

Expressions

Expressions are composed of **operands** (items from which we can obtain a value, such as variables, literal constants, and other expressions enclosed in brackets) and **operators** (eg. `+`,`-`,`*`, which act on values to yield new values).

Operators

`+`,`-` represent addition and subtraction in the usual way.
`-` is also used in representing negative quantities, eg. `-35`

`*` stands for multiplication, but note that (unlike in normal arithmetic), it can never be omitted; ie. implied multiplication is not allowed, eg:

wrong:	`a(b+c)`
right:	`a*(b+c)`

`**` stands for exponentiation, ie. raising a number to a power, eg: `a**3` represents `a` cubed (which could also be written as `a*a*a`)

Operators are applied in the usual order, ie. first `**`, next `*` , then `+` and `-`. The operator `**` is said to have a higher **priority** than `*`, and `*` a higher priority than + and `-`. Operators are thus applied in order of *descending* priority. Within groups of operators of equal priority, evaluation proceeds from left to right, eg:

`3+4*2`	gives	`11` *not* `14`	(`*` before +, descending priority)
`10-7+2`	gives	`5` *not* `1`	(left to right with equal priority)

Where normal priority does not give the desired order, (round) brackets can be used to group operands and operators as required. As is usual, such (sub-)expressions enclosed in brackets are worked out first. Brackets can be nested to any depth always provided that they are properly matched, but care is required; incorrect bracketing is a frequent source of error. Square and curly brackets should *not* be used to group terms in expressions.

Examples

```
(3+4)*2          gives    14

2+(3+1)**2       gives    18

10-(7+2)         gives    1

8*[5+3]          --  !? illegal in Ada

2-{8-3}          --  !? illegal in Ada

-- Assuming that x, y and z are variables, is the
-- following a correct expression?
(((3*y+2)*y)+4*y+21+(x+1)*(x+2))*z      -- Check it and see
```

4.2 Working with integer and real values

We may use the arithmetic operators +, - and * with either integer or real values but we *cannot mix* integer and real values in the same arithmetic expression. Also, the type of an expression is the same as the type of the operands making up that expression. Thus:

for	+, - or *	if both operands are `Integer`, these give an `Integer` result
for	+, - or *	if both operands are `Float`, these give a `Float` result
for	+, - or *	one `Float` operand and one `Integer` operand is illegal in Ada

Thus:	`3 * 4`	gives	`12`	(of type `Integer`)
	`3.0 * 4.0`	gives	`12.0`	(of type `Float`)
	`3.0 + 4`	`-- !? illegal in Ada`		

Exponentiation

An exception to the above properties is the exponentiation operation, where the power must always be integer, no matter whether the number being raised to the power is integer or real. Thus:

for	`a**b`	if `a` is `Integer` and `b` is `Integer`, this gives an `Integer` result
for	`a**b`	if `a` is `Float` and `b` is `Integer`, this gives an `Float` result

Thus:	`2 ** 3`	gives	`8`	(of type `Integer`)
	`2.5 ** 2`	gives	`6.25`	(of type `Float`)
	`2.0 ** -3`	gives	`0.125`	(of type `Float`)
	`2.5 ** 0.5`	`-- !? illegal in Ada`		
	`2 ** -3`	`-- !? illegal in Ada`		(produces a non-integer value)

Real division

For real values division is straightforward: if `x` and `y` are values of type `Float` then `x/y` denotes the division of `x` by `y` and produces a `Float` result. Thus:

`11.0 / 5.0`	gives	`2.2`	(of type `Float`)
`10.0 / 5.0`	gives	`2.0`	(of type `Float`) *not* `2` (of type `Integer`)

The operator / has the same priority as `*`. Thus:

`11.0 + 4.0/2.0`	gives	`13.0`	*not*	`7.5`	(/ before +, descending priority)
`11.0/5.0/2.0`	gives	`1.1`	*not*	`4.4`	(left to right with equal priority)

Note that as real arithmetic is only approximate, the value `11.0/5.0` might actually be 'stored' as `2.200001` or `2.199999` rather than the 'true' value `2.200000`. The precise form will vary from computer to computer according to the precision with which real values are stored internally within that computer.

Integer division

Sometimes we want a form of division that gives whole number, rather than fractional, results. For example, if we share 14 apples amongst 5 people, asking how many they will get each, we want the answer `2`, not `2.8`. This form of division, known as **integer division**, is expressed using the same operator as for real division, but when applied to two integer operands it produces an integer result by discarding any remainder, ie:

`14 / 5`	gives	`2`
`16 / 3`	gives	`5`
`18 / 6`	gives	`3`

In this sort of calculation, we often want to know how much is "left over", ie. what the remainder is. This is given by another operator, `REM`, which is represented by a keyword rather than an operator symbol. Thus:

`14 REM 5`	gives	`4`
`15 REM 5`	gives	`0`
`16 REM 5`	gives	`1`

Notes: In the above examples, all operands are shown (for simplicity) as literal constants, but they could (of course) be replaced by any other sort of operands producing the values shown, eg. by variables or arithmetic expressions (in brackets).

The operator `REM` has the same priority as `*` (multiplication) and `/` (division).

In Ada we cannot use `REM` with `Float` operands or with operands of mixed type.

4.3 Relational operators

We have already used some of these in conditions to express comparison between numbers. The full list of relational operators is:

>	meaning	greater than
<	meaning	less than
>=	meaning	greater than or equal to (≥)
<=	meaning	less than or equal to (≤)
=	meaning	equal to
/=	meaning	not equal to (≠)

When placed between numeric operands (both `Integer` or both `Float`, but *not* mixed), they yield the truth value `True` or `False` according to whether the specified relation does, or does not, hold (respectively); eg:

`3 >= 2`	gives	`True`
`3 >= (2 + 1)`	gives	`True`
`3.0 >= 3.5`	gives	`False`
`7 = (14 / 2)`	gives	`True`
`7 /= 7`	gives	`False`
`7 = 7.0`	`-- !? illegal in Ada`	

Relational operators have lower priority than arithmetic operators, so that comparands which involve arithmetic operators need not be bracketed; eg: `(x*3) = (y+k*2)` can also validly be written as: `x*3 = y+k*2.` Note that using brackets where they are not strictly necessary is *not* an error and, indeed, can often help make the structure of a complicated expression easier to understand.

4.4 Converting between integer and real values

We stated above that all the operands in a algebraic expression must be of the same type (with the exception of the exponentiation operator). This also applies in an assignment step, where the variable which appears on the left hand side of the `:=` must be of the same type as the expression which appears on the right hand side. However we often need to combine `Integer` and `Float` values in an expression, or to assign the value held in an `Integer` variable to a variable of type `Float`, or vice versa.

For example, we may wish to calculate the total cost of `Quantity` items each costing £3.43 where `Quantity` is a `Integer` variable, and assign the result to the `Float` variable `TotalCost`. We could not simply write:

```
TotalCost :=  Quantity * 3.43;   -- !? mixed Integer and Float operands
```

as the left hand operand of the `*` operator, ie. `Quantity`, has an `Integer` value (`20` say) and the right hand operand `3.43` is a `Float` value. To overcome such problems Ada provides a special 'built-in' function `Float` which converts an `Integer` value into its corresponding `Float` value. Thus we should write:

```
TotalCost :=  Float(Quantity) * 3.43;
```

Hence if the value of `Quantity` is `20`, `Float(Quantity)` produces the `Float` result `20.0` which can then be multiplied by `3.43` to produce the required total as a `Float` value. This value can then be assigned to the `Float` variable `TotalCost`.

Similarly, suppose we need to perform the calculation:

```
TaxDue := (52 * WeeklyPay - TaxThreshold) * TaxRate / 52;
                -- !? illegal in Ada
```

Assuming `WeeklyPay` and `TaxThreshold` are `Integer` variables and `TaxDue` and `TaxRate` are of type `Float`, we must write:

```
TaxDue := Float(52 * WeeklyPay - TaxThreshold) * TaxRate / Float(52);
```

or possibly:

```
TaxDue := Float(52 * WeeklyPay - TaxThreshold) * TaxRate / 52.0;
```

It is also possible to perform conversions the other way round, ie. to convert from a value of type `Float` to a value of type `Integer`. In this case the 'built-in' function `Integer` can be used which converts a real value into an integer by rounding the real value to the nearest whole number. Thus, for example, to output the value held in the variable `TotalCost` to the nearest £ we could write:

```
Put(Item => "Total cost to the nearest pound is ");
Put(Item => Integer(TotalCost));
```

4.5 More on output

In previous units we have used the procedure `Put` to output results. We now consider the use of this procedure in more detail for outputting integer and real values according to different layouts or **formats**.

4.5.1 Formatted integer output

Assuming that `Number` and `Size` are integer values the step:

```
Put(Item => Number, Width => Size);
```

will cause `Number` to be printed using at least `Size` print positions. If `Number` has exactly `Size` digits, then only these `Size` digits are printed (no blank spaces are left either before or after the value). If `Number` has fewer than `Size` digits, enough blank spaces are 'printed' before the value of `Number` so that exactly `Size` print positions are used. If `Number` is negative, one of these print positions will be occupied by a minus sign. We will see later what happens if `Number` has more than `Size` digits.

For example, assuming that the value of `Number` is `123` then the following output is produced (where a Δ denotes a blank space):

```
Put(Item => Number, Width => 3);       outputs:        123
Put(Item => Number, Width => 6);       outputs:        ΔΔΔ123
Put(Item => Number, Width => 8);       outputs:        ΔΔΔΔΔ123
```

After a `Put` step the 'current print position' is left immediately after the last digit of the number just printed. Thus:

```
Put(Item => Number, Width => 3);
Put(Item => Number, Width => 3);
```

produces the output : `123123` while:

```
Put(Item => Number, Width => 5);
Put(Item => Number, Width => 5);
```

produces the output : `ΔΔ123ΔΔ123`

Since the output produced in the first case might be confused with the output of a single six digit value `123123` it is usually better to avoid this confusion by specifying a value for `Width` greater than the number of digits in the value expected to be output.

4.5.2 Formatted real output

In addition to the `Float` value to be 'printed' the procedure `Put` may have two additional arguments: the second argument specifies the minimum number of print positions to be used before the decimal place; the third argument specifies how many decimal places (ie. the number of digits appearing after the decimal point) are to be printed. If the value to be output has fewer than the number of digits specified to be printed before the decimal place, the leading print positions are padded-out with spaces. If the value to be output is negative, one of these print positions will be occupied by a minus sign. For example assuming that the `Float` variable `RealNum` has the value `-123.456`, then:

```
Put(Item => RealNum, Fore => 4, Aft => 3);     outputs:    –123.456
Put(Item => RealNum, Fore => 6, Aft => 1);     outputs:    ΔΔ-123.5
Put(Item => RealNum, Fore => 8, Aft => 2);     outputs:    ΔΔΔΔ-123.46
```

Note: The value printed is rounded so that it is correct to the specified number of decimal places. If the number is positive, no plus sign is output.

4.5.3 Formats which don't 'fit' the value to be output

What happens if the number to be output is too large to fit into the specified format?

- In the case of integer output, if the number has more than `Width` digits (including a minus sign if it is negative), the value is printed *in full* (ie. it is not truncated in any way) and so uses *more* than the specified number of print positions.

- A similar situation occurs when outputting real values, where if the value to be printed has more digits before the decimal point (including a minus sign if it is negative) than the value of `Fore`, `Put` uses the minimum number of print positions needed to output the number to the required precision, and so again uses *more* than the specified number of print positions.

Thus for the example values of the `Integer` variable `Number`, and the `Float` variable `RealNum` given above, we would have:

```
Put(Item => Number, Width => 1);               outputs:    123
Put(Item => Number, Width => 2);               outputs:    123
Put(Item => RealNum, Fore => 1, Aft => 2);     outputs:    –123.46
Put(Item => RealNum, Fore => 2, Aft => 3);     outputs:    -123.456
```

4.5.4 Default format settings

In fact the above explains approximately what happens if the extra arguments in a `Put` step are missed out altogether. More precisely, the **default values** of `Width` = `1` (when outputting integer values), and `Fore` = `1` and `Aft` = `2` (when outputting real values) are assumed if their specification is omitted in a `Put` step. This choice for the values of `Width` and `Fore` means that `Put` uses the minimum number of print positions needed to output the number. Thus, for example, if `IntNum` is an `Integer` variable whose value is `8`, and `FltNum` is a `Float` variable whose value is `6.35`, then:

```
Put(Item => "The cost of ");
Put(Item => IntNum);
Put(Item => " items, at £");
Put(Item => FltNum);
Put(Item => " each, is the total cost £");
Put(Item => Float(IntNum) * FltNum);
```

produces the output:

```
The cost of 8 items, at £6.35 each, is the total cost £50.80
```

Notice the extra space at the end of the string `"The cost of "`, and the extra space at the start of the string `" items, at £"`. These are present so that the value of `IntNum`, which is output as the single digit `8`, is surrounded by a single space on either side. Also, the value of `FltNum` (output using only the four print positions required by `6.35`) is preceded directly by a `£`, so the previously output string `" items, at £"` does not have a space at the end, but rather finishes with a `£`. Note that on some printers a `£` will be displayed as a # character.

We see that using the defaulted form is better when outputting mixed text and numeric values, whereas we shall see in the following example that using `Put` with specified formatting is better for outputting tables of results.

4.6 Example

Problem:

The program is to ask the user to input an integer data value and if this is 2 or more, list out all possible factorisations of that value into two integers, except for the number itself times 1, which is to be disregarded. The program is then to output a message stating how many factorisations are found, unless none are found, in which case a more suitable message is to be produced. The program then asks the user to input another integer value and repeats the calculation. If the user types a 0 or a 1 the program terminates.

Solution:

i) **First thoughts:**

What output is to be produced? Perhaps something like this:

```
Type in a number greater than 1 (0 or 1 to quit):  30

Factorisations of 30 are:
ΔΔΔΔΔΔΔ2ΔΔΔΔΔΔΔΔ15
       3        10
       5         6
       6         5
      10         3
      15         2
There were 6 divisors found.

Type another number greater than 1 (0 or 1 to quit):  997

Factorisations of 997 are:
No divisors found -  997 is prime.

Type another number greater than 1 (0 or 1 to quit):  0
```

Notice that we have chosen a particular layout for the table of factors, where the first divisor occupies the first 8 print positions, and the second divisor the next 10 print positions.

ii) **Outline a scheme:**

How to get this output? In briefest outline:
Ask for user's number. While this number is bigger than 1, deal with this number (ie. find and output the factorisations and number of divisors), then ask for the next user's number. Thus in outline we have:

```
Get(Item => Number)
WHILE Number > 1 LOOP
     Deal with this number
          {ie. find and output the factors of this number}
     Get(Item => Number)
END LOOP
```

Notice how we have re-used the subalgorithm introduced in [2.5]. For the main processing step, we can choose a possible divisor and see if it "goes" exactly into `Number` (ie. leaves remainder zero - can use the operator `REM` to check this), and if it does, then output the factorisation found and increment a count of the number of divisors found. Must do this for each integer from `2` up to the largest possible exact divisor (ie. the value of `Number` divided by 2) => a further repetition involved.

iii) **Algorithm** (some detail omitted)

```
Get(Item => Number)

WHILE Number > 1 LOOP

     -- Deal with this number, ie. determine and output the factors

     NumOfDivisors := 0
     TrialDivisor := 2
     Output a heading for the table

     WHILE TrialDivisor <= Number / 2 LOOP
          IF Number REM TrialDivisor = 0 THEN
               -- ie. if an exact divisor is found
               Put(Item => TrialDivisor)
               Put(Item => Number / TrialDivisor)
               NumOfDivisors := NumOfDivisors + 1
          END IF
          TrialDivisor := TrialDivisor + 1
     END LOOP

     IF NumOfDivisors > 0 THEN
          Put(Item => NumOfDivisors)
     ELSE
          Put(Item => "No divisors found")
     END IF

     Get(Item => Number)

END LOOP
```

Note: We can now see that an algorithm is, in effect, an informal, incomplete program - it's a step in the process of producing a complete correct program.

iv) **Program:**

After filling in obligatory items such as declarations, semi-colons and field width specifiers in `Put` steps, and after adding further suitable output statements to obtain the required format, eg. explanatory text and new lines, we obtain the complete Ada program:

```
WITH CS_Int_IO;   USE CS_Int_IO;      -- For Get and Put for Integer I/O
WITH Ada.Text_IO; USE Ada.Text_IO;    -- For Put for text, and New_Line

PROCEDURE Factors IS

     Number        : Integer; -- To hold each number specified by the user
     TrialDivisor  : Integer; -- To hold each possible (ie. trial) divisor
     NumOfDivisors : Integer; -- To count the number of divisors

BEGIN

     Put(Item => "Type in a number greater than 1 (0 or 1 to quit): ");
     Get(Item => Number);

     WHILE Number > 1 LOOP

     -- Deal with this number, ie. determine and output the factors

          NumOfDivisors := 0;
          TrialDivisor := 2;

          New_Line;
          Put(Item => "Factorisations of ");
          Put(Item => Number);
          Put(Item => " are:");
          New_Line;
          -- Note the use of blanks in the strings "Factorisations of "
          -- and " are:" and the unformatted output of the value of
          -- Number to ensure that only one space appears on either
          -- side of the value of Number

          WHILE TrialDivisor <= Number / 2 LOOP
               IF Number REM TrialDivisor = 0 THEN
                    -- ie. if an exact divisor is found
                    Put(Item => TrialDivisor, Width => 8);
                    Put(Item => Number / TrialDivisor, Width => 10);
                    New_Line;
                    -- Note how formatted output steps are used here to
                    -- ensure that the two columns in the table of results
                    -- line-up, right justified, at the 8th and 18th
                    -- (ie. 8 + 10) print positions on the page
                    NumOfDivisors := NumOfDivisors + 1;
               END IF;
               TrialDivisor := TrialDivisor + 1;
          END LOOP;

          IF NumOfDivisors > 0 THEN
               Put(Item => "There were ");
               Put(Item => NumOfDivisors);
               Put(Item => " divisors found.");
          ELSE
               Put(Item => "No divisors found - ");
               Put(Item => Number);
               Put(Item => " is prime.");
          END IF;

          New_Line;
          New_Line;
          Put(Item =>
                "Type another number greater than 1 (0 or 1 to quit): ");
          Get(Item => Number);

     END LOOP;

  END Factors;
```