beginner

COLLABORATORS

	TITLE : beginner		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		February 24, 2025	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

1 beginner **1**

1.1 Procedures and Functions 1

1.2 Functions 1

1.3 One-Line Functions 3

1.4 Default Arguments 3

1.5 Multiple Return Values 5

Chapter 1

beginner

1.1 Procedures and Functions

```
Procedures and Functions
************************

   A function is a procedure which returns a value.  This value can be
formed from any expression so it may depend on the parameters with which
the function was called.  For instance, the addition operator + can be
thought of as a function which returns the sum of its two parameters.
```

```
  Functions
  One-Line Functions
  Default Arguments
  Multiple Return Values
```

1.2 Functions

```
Functions
=========

   We can define our own addition function, add, in a very similar way to
the definition of a procedure.

     PROC main()
       DEF sum
       sum:=12+79
       WriteF('Using +, sum is \d\n', sum)
       sum:=add(12,79)
       WriteF('Using add, sum is \d\n', sum)
     ENDPROC

     PROC add(x, y)
       DEF s
       s:=x+y
     ENDPROC s
```

This should generate the following output:

```
Using +, sum is 91
Using add, sum is 91
```

In the procedure add the value s is returned using the ENDPROC label. The value returned from add can be used in expressions, just like any other value. You do this by writing the procedure call where you want the value to be. In the above example we wanted the value to be assigned to sum so we wrote the call to add on the right-hand side of the assignment. Notice the similarities between the uses of + and add. In general, add(a,b) can be used in exactly the same places that a+b can (more precisely, it can be used anywhere (a+b) can be used).

The RETURN keyword can also be used to return values from a procedure. If the ENDPROC method is used then the value is returned when the procedure reaches the end of its code. However, if the RETURN method is used the value is returned immediately at that point and no more of the procedure's code is executed. Here's the same example using RETURN:

```
PROC add(x, y)
  DEF s
  s:=x+y
  RETURN s
ENDPROC
```

The only difference is that you can write RETURN anywhere in the code part of a procedure and it finishes the execution of the procedure at that point (rather than execution finishing when it reaches the end of the code). In fact, you can use RETURN in the main procedure to prematurely finish the execution of a program.

Here's a slightly more complicated use of RETURN:

```
PROC limitedadd(x,y)
  IF x>10000
    RETURN 10000
  ELSEIF x<-10000
    RETURN -10000
  ELSE
    RETURN x+y
  ENDIF
  /* The following code is redundant */
  x:=1
  IF x=1 THEN RETURN 9999 ELSE RETURN -9999
ENDPROC
```

This function checks to see if x is greater than 10,000 or less than -10,000, and if it is a limited value is returned (which is generally not the correct sum!). If x is between -10,000 and 10,000 the correct answer is returned. The lines after the first IF block will never get executed because execution will have finished at one of the RETURN lines. Those lines are therefore just a waste of compiler time and can safely be omitted (as the comment suggests).

If no value is given with the ENDPROC or RETURN keyword then zero is returned. Therefore, all procedures are actually functions (and the terms procedure and function will tend to be used interchangeably). So, what happens to the value when you write a procedure call on a line by itself, not in an expression? Well, as we will see, the value is simply discarded (see Turning an Expression into a Statement). This is what happened in the previous examples when we called the procedures fred and WriteF.

1.3 One-Line Functions

One-Line Functions
==================

Just as the IF block and FOR loop have horizontal, single line forms, so does a procedure definition. The general form is:

```
PROC name (arg1, arg2, ...) IS expression
```

Alternatively, the RETURN keyword can be used:

```
PROC name (arg1, arg2, ...) RETURN expression
```

At first sight this might seem pretty unusable, but it is useful for very simple functions and our add function in the previous section is a good example. If you look closely at the original definition you’ll see that the local variable s wasn’t really needed. Here’s the one-line definition of add:

```
PROC add(x,y) IS x+y
```

1.4 Default Arguments

Default Arguments
=================

Sometimes a procedure (or function) will quite often be called with a particular (constant) value for one of its parameters, and it might be nice if you didn’t have to fill this value in all the time. Luckily, E allows you to define default values for a procedure’s parameters when you define the procedure. You can then just leave out that parameter when you call the procedure and it will default to the value you defined for it. Here’s a simple example:

```
PROC play(track=1)
  WriteF('Starting to play track \d\n', track)
  /* Rest of the code... */
ENDPROC

PROC main()
  play(1)  -> Start playing from track 1
  play(6)  -> Start playing from track 6
```

```
       play()   -> Start playing from track 1
     ENDPROC
```

This is an outline of a program to control something like a CD player. The play procedure has one parameter, track, which represents the first track that should be played. Often, though, you just tell the CD player to play, and don’t specify a particular track. In this case, play starts from the first track. This is exactly what happens in the example above: the track parameter has a default value of 1 defined for it (the =1 in the definition of the play procedure), and the third call to play in main does not specify a value for track, so the default value is used.

There are two constraints on the use of default arguments:

1. Any number of the parameters of a procedure may have default values defined for them, although they may only be the right-most parameters. This means that for a three parameter procedure, the second parameter can have a default value only if the last parameter does as well, and the first can have one only if both the others do. This should not be a big problem because you can always reorder the parameters in the procedure definition (and in all the places it has been called!).

   The following examples show legal definitions of procedures with default arguments:

   ```
   PROC fred(x, y, z) IS x+y+z         -> No defaults

   PROC fred(x, y, z=1) IS x+y+z       -> z defaults to 1

   PROC fred(x, y=23, z=1) IS x+y+z    -> y and z have defaults

   PROC fred(x=9, y=23, z=1) IS x+y+z  -> All have defaults
   ```

   On the other hand, these definitions are all illegal:

   ```
   PROC fred(x, y=23, z) IS x+y+z   -> Illegal: no z default

   PROC fred(x=9, y, z=1) IS x+y+z  -> Illegal: no y default
   ```

2. When you call a procedure which has default arguments you can only leave out the right-most parameters. This means that for a three parameter procedure with all three parameters having default values, you can leave out the second parameter in a call to this procedure only if you also leave out the third parameter. The first parameter may be left out only if both the others are, too.

   The following example shows which parameters are considered defaults:

   ```
   PROC fred(x, y=23, z=1)
     WriteF('x is \d, y is \d, z is \d\n', x, y, z)
   ENDPROC

   PROC main()
     fred(2, 3, 4) -> No defaults used
     fred(2, 3)    -> z defaults to 1
     fred(2)       -> y and z default
     fred()        -> Illegal: x has no default
   ```

```
          ENDPROC

     In this example, you cannot leave out the y parameter in a call to
     fred without leaving out the z parameter as well.  To make y have its
     default value and z some value other than its default you need to
     supply the y value explicitly in the call:

           fred(2, 23, 9) -> Need to supply 23 for y
```

These constraints are necessary in order to make procedure calls unambiguous. Consider a three-parameter procedure with default values for two of the parameters. If it is called with only two parameters then, without these constraints, it would not be clear which two parameters had been supplied and which had not. If, however, the procedure were defined and called according to these constraints, then it must be the third parameter that needs to be defaulted (and the two parameters with default values must be the last two).

1.5 Multiple Return Values

Multiple Return Values
======================

So far we've only seen functions which return only one value, since this is something common to most programming languages. However, E allows you to return up to three values from a function. To do this you list the values separated by commas after the ENDPROC, RETURN or IS keyword, where you would normally have specified only one value. A good example is a function which manipulates a screen coordinate, which is a pair of values: the x- and y-coordinates.

```
     PROC movediag(x, y) IS x+8, y+4
```

All this function does is add 8 to the x-coordinate and 4 to the y-coordinate. To get to the return values other than the first one you must use a multiple-assignment statement:

```
     PROC main()
       DEF a, b
       a, b:=movediag(10, 3)
       /* Now a should be 10+8, and b should be 3+4 */
       WriteF('a is \d, b is \d\n', a, b)
     ENDPROC
```

a is assigned the first return value and b is assigned the second. You don't need to use all the return values from a function, so the assignment in the example above could have assigned only to a (in which case it would not be a multiple-assignment anymore). A multiple-assignment makes sense only if the right-hand side is a function call, so don't expect things like the following example to set b properly:

```
       a,b:=6+movediag(10,3)  -> No obvious value for b
```

If you use a function with more than one return value in any other expression (i.e., something which is not the right-hand side of an assignment), then only the first return value is used. For this reason the return values of a function have special names: the first return value is called the regular value of the function, and the other values are the optional values.

```
PROC main()
  DEF a, b
  /* The next two lines ignore the second return value */
  a:=movediag(10, 3)
  WriteF('x-coord of movediag(21, 4) is \d\n', movediag(21,4))
ENDPROC
```