User-Defined Functions

This chapter covers the following topics:

- Introduction to User-Defined Functions
- Difference between Function Call and Subprogram Call
- Function Definition (DEFINE FUNCTION)
- Prototype Definition (DEFINE PROTOTYPE)
- Symbolic and Variable Function Call
- Automatic/Implicit Prototype Definition (APT)
- Prototype Cast (PT Clause)
- Intermediate Result for Return Value (IR Clause)
- Combinations of Possible Prototype Definitions
- Recursive Function Call
- Behavior of Functions in Statements and Expressions
- Usage of Functions as Statements

Introduction to User-Defined Functions

Functions, as do subprograms, give you the possibility to receive data, to change it and to give the results to the calling module. The advantage of using functions over subprograms is that function calls can be used directly in statements and expressions without the need for additional temporary variables.

Normally, depending on the parameters that are given to the function, the result is produced in the function and is returned to the calling object. If other values are to be returned to the calling module, this can be done by using the parameters; see *Subprogram*.

Once the function code has been completely executed, control is given back to the calling object and the program continues with the statement that comes after the function call.

For further information, see also:

- Natural object type Function
- Function Call
- Natural statements `DEFINE FUNCTION`, `DEFINE PROTOTYPE`

Difference between Function Call and Subprogram Call

The following two examples show the difference between using function calls and subprogram calls.

Example of Using a Function Call:

The following example comprises a program object that uses a function call, a function object containing a function definition created with a `DEFINE FUNCTION` statement, and a copycode object created with a `DEFINE PROTOTYPE` statement.

Program Object:

```
/* Excerpt from a Natural program using a function call
INCLUDE C#ADD
WRITE #ADD(< 2,3 >) /* function call; no temporary variable necessary
END
```

Function Object:

```
/* Natural function definition
DEFINE FUNCTION #ADD
  RETURNS (I4) BY VALUE
  DEFINE DATA PARAMETER
  1 #SUMMAND1 (I4) BY VALUE
  1 #SUMMAND2 (I4) BY VALUE
  END-DEFINE

  #ADD := #SUMMAND1 + #SUMMAND2
END-FUNCTION
END
```

Copycode Object (for example, `C#ADD`):

```
/* Natural copycode containing prototype
DEFINE PROTOTYPE #ADD
  RETURNS (I4)
  DEFINE DATA PARAMETER
  1 #SUMMAND1 (I4) BY VALUE
  1 #SUMMAND2 (I4) BY VALUE
  END-DEFINE
END-PROTOTYPE
```

If you want to achieve the same functionality by using a subprogram, you must use temporary variables.

Example of Using a Subprogram:

The following example comprises a program object that calls a subprogram object, involving the use of a temporary variable.

Program Object:

```
/* Natural program using a subprogram
DEFINE DATA LOCAL
1 #RESULT (I4) INIT <0>          /* temporary variable
END-DEFINE

CALLNAT 'N#ADD' USING #RESULT 2 3 /* result is stored into #RESULT
WRITE #RESULT                     /* print out the result of the subprogram
END
```

Subprogram Object (for example, N#ADD):

```
/* Natural program using a subprogram
DEFINE DATA PARAMETER
1 #RETURN   (I4) BY VALUE RESULT
1 #SUMMAND1 (I4) BY VALUE
1 #SUMMAND2 (I4) BY VALUE
END-DEFINE

#RETURN := #SUMMAND1 + #SUMMAND2
END
```

Function Definition (DEFINE FUNCTION)

The function definition contains the Natural code to be executed when the function is called. As with subprograms, you need to create a Natural object, in this case, of type "Function" which contains the function definition. The function definition is created using the Natural statement `DEFINE FUNCTION`.

The function call itself can be in any object type which contains executable code.

Prototype Definition (DEFINE PROTOTYPE)

To be able to compile function calls, Natural needs information about the *format-length/array-definition* of the return value. This information is then made available to the compiler in the prototype definition. This definition is created using the Natural statement `DEFINE PROTOTYPE`. You can also include the definition of the parameter to be passed back, which is then checked at compile time.

Since Natural makes the connection between "calling" and "called" objects at runtime, and not before, the computer does not know with which type of a function return value it is dealing at compile time. This is due to the fact that the object containing the function does not necessarily have to exist (at compile time). It is for this reason that the prototype definition is created, so that the *format-length/array-definition* can be generated into the generated program at compile time.

It is important to remember that a prototype definition never contains executable code. A prototype definition simply contains the following information about the function call: the *format/length/array-definition* of the return value or the parameter being passed back.

Symbolic and Variable Function Call

To define a variable function call, it is always necessary to use a `DEFINE PROTOTYPE VARIABLE` statement. Otherwise, the function call is assumed to be an implicit symbolic function call.

See the section *Function Call* for more details about this topic.

Automatic/Implicit Prototype Definition (APT)

If neither an explicit prototype definition (EPT) nor a `PT` clause exists, a search for the prototype definition takes place in the generated program. For further information, see *Combinations of Possible Prototype Definitions* below.

Prototype Cast (PT Clause)

In order to find the corresponding prototype of a specific function, Natural searches for a prototype which bears the name of the function. If this is not the case, it is assumed that the function call is symbolic. In this case, the function "signature" must be defined by using the keyword `PT=` in the function call.

Intermediate Result for Return Value (IR Clause)

This clause enables you to specify the format/length of the return value for a function call without using an explicit or implicit prototype definition, that is, it enables the explicit specification of an intermediate result. For further information, see *Function Call*, *intermediate-result-definition*.

Combinations of Possible Prototype Definitions

The following table explains the effects on the prototype definition according to various syntax combinations that are possible when using the `DEFINE PROTOTYPE` statement and/or the clauses available in the function call. The following possibilities are available in order to define parts of a function prototype taking effect only on the function call to which they belong:

- **Explicit DEFINE PROTOTYPE Definition (EPT)**
  Can decide on symbolic/variable function call; parameter definition; return value definition.
- **Prototype Cast (PT Clause)**
  Can decide on parameter definition; return value definition.
- **Intermediate Result for Return Value (IR Clause)**
  Can decide on return value definition.

Case	Explicit prototype definition in `DEFINE PROTOTYPE` (EPT)	`PT` clause in function call (PT)	`IR` clause in function call (IR)		Automatic reading-in of prototype definition from GP (APT)	Prototype behavior
1	x	x	x		-	SV(EPT), PS(PT), R(IR)
2	-	x	x		-	S, PS(PT), R(IR)
3	x	-	x		-	SV(EPT), PS(EPT), R(IR)
4	-	-	x		x	S, PS(APT), R(IR)
5	x	x	-		-	SV (EPT), PS(PT), R(PT)
6	-	x	-		-	S, PS(PT), R(IR)
7	x	-	-		-	SV(EPT), PS(EPT), R(EPT)
8	-	-	-		x	S, PS(APT), R(APT)

Where:

EPT	Explicit `DEFINE PROTOTYPE` statement.
PT	Prototype Cast (`PT` clause).
IR	Intermediate Result for Return Value (`IR` clause).
APT	Automatic Prototype Definition via external generated program (GP).
S	Symbolic function call.
V	Variable function call.
SV(EPT)	Explicit prototype definition decides whether a symbolic or variable function call is performed.
R(IR)	The return variable (R) is defined by the `IR` clause in the function call.
R(PT)	The return variable (R) is defined by the `PT` clause in the function call.
R(EPT)	The return variable (R) is defined by the explicit `DEFINE PROTOTYPE` statement.
PS(PT)	The parameter signature (PS) (that is, the parameter definition, without return value definition) is defined by the `PT` clause in the function call.
PS(EPT)	The parameter signature (PS) (that is, the parameter definition, without return value definition) is defined by the explicit `DEFINE PROTOTYPE` statement.
PS(APT)	The parameter signature (PS) is defined automatically by reading in the prototype definition from the generated program (GP).
R(APT)	The return variable (R) is defined by the automatic prototype definition via external generated program (GP).

For example, the behavior of Case1 shown in the table above:

What is the behavior if an explicit `DEFINE PROTOTYPE` statement (EPT) is used, and in the function call, the `PT` and `IR` clauses are defined?

The EPT definition decides whether a symbolic or variable function call is performed. The variable function call is assumed when `DEFINE PROTOTYPE VARIABLE` has been defined previously. The parameter signature (that is, the format/length definition of all parameters without return value definition) is defined by the `PT` clause, and the format/length of the return value is defined by the `IR` clause in the function call. In this case, no automatic prototype definition (APT) will be started.

In conclusion, the following general rules can be derived from the above cases:

- In case of variable function calls, there must always be an explicit prototype definition (EPT) for the call.
- The `PT` clause does not decide whether it is a symbolic or variable function call.
- The definitions in the `PT` clause overwrite the EPT definitions for parameters and return value.
- The definitions in the `IR` clause overwrite the return value definition.
- If neither an EPT nor a `PT` clause exists, a search for the prototype definition takes place in the generated program (automatic prototype definition).

Recursive Function Call

If a function is to be called recursively, the function prototype must be contained in the function definition, or be inserted by means of an INCLUDE file.

Example:

Function Object:

```
/* Function definition for calculation of the math. factorial
DEFINE FUNCTION #FACT
  RETURNS (I4) BY VALUE
  DEFINE DATA PARAMETER
  1 #PARA (I4) BY VALUE
  LOCAL
  1 #TEMP (I4)
  END-DEFINE

  /* Prototype definition is necessary
  INCLUDE C#FACT

  /* Program code
  IF #PARA=0
    #FACT := 1
  ELSE
    #TEMP := #PARA - 1
    #FACT := #PARA * #FACT(< #TEMP >)
  END-IF

END-FUNCTION
END
```

Copycode Object (for example, named C#FACT):

```
/* Prototype definition is necessary
DEFINE PROTOTYPE #FACT
  RETURNS (I4)
  DEFINE DATA PARAMETER
  1 #PARA (I4) BY VALUE
  END-DEFINE
END-PROTOTYPE
```

Program Object:

```
/* Prototype definition
INCLUDE C#FACT

/* function call
WRITE #FACT(<12>)
END
```

Behavior of Functions in Statements and Expressions

Instead of operands, function calls can be used directly in statements or expressions. However, this is only allowed in places where operands cannot be modified.

All function calls are executed according to their syntactical sequence which is analyzed at compile time. The results of the function calls are saved in internal temporary variables and passed to the statement or expression.

This fixed sequence makes it possible to allow and execute standard output in functions, without, for example, unwillingly influencing the output of a statement.

Example:

Program:

```
/* Natural program using a function call
INCLUDE CPRINT
PRINT 'before' #PRINT(<>) 'after'
END
```

Function Object:

```
/* Natural function definition
/* function returns integer value 10
DEFINE FUNCTION #PRINT
  RETURNS (I4)
  WRITE '#PRINT'
  #PRINT := 10
END-FUNCTION
END
```

Copycode (for example, CPRINT):

```
DEFINE PROTOTYPE #PRINT END-PROTOTYPE
```

The following is the result which is then sent to the standard output:

```
#PRINT
before       10 after
```

Usage of Functions as Statements

Functions can also be called as statements independently from statements and expressions. In this case, the return value - assuming it has been defined - is not taken into account.

If, however, an independent function is declared after an optional operand list, the operand list must be followed by a semicolon to make it clear that the function call is not a part of the operand list.

Example:

Program Object:

```
/* Natural program using a function call
DEFINE DATA LOCAL
1 #A (I4) INIT <1>
1 #B (I4) INIT <2>
END-DEFINE

INCLUDE CPROTO

WRITE #A #B
```

```
#PRINT_ADD(< 2,3 >) /* function call belongs to operand list just in front of it

WRITE '******'

WRITE #A #B;        /* semicolon separates operand list and function call
#PRINT_ADD(< 2,3 >) /* function call doesn't belong to the operand list
END
```

Function Object:

```
/* Natural function definition
DEFINE FUNCTION #PRINT_ADD
  RETURNS (I4) BY VALUE
  DEFINE DATA PARAMETER
  1 #SUMMAND1 (I4) BY VALUE
  1 #SUMMAND2 (I4) BY VALUE
  END-DEFINE

  #PRINT_ADD := #SUMMAND1 + #SUMMAND2
  PRINT '#PRINT_ADD =' #PRINT_ADD
END-FUNCTION
END
```

Copycode Object (for example, named CPROTO):

```
/* Natural copycode containing prototype
DEFINE PROTOTYPE #PRINT_ADD
  RETURNS (I4)
  DEFINE DATA PARAMETER
  1 #SUMMAND1 (I4) BY VALUE
  1 #SUMMAND2 (I4) BY VALUE
  END-DEFINE
END-PROTOTYPE
```