Function Call

call-name (< ([*prototype-cast*] [*intermediate-result-definition*]) [*parameter*] [, [*parameter*]] ... >)

This chapter covers the following topics:

- Calling User-Defined Functions
- Restrictions
- Syntax Description

Related Topics: `DEFINE FUNCTION` | `DEFINE PROTOTYPE`

Calling User-Defined Functions

Function calls can be used to call user-defined functions which are defined inside special objects of type function.

There are different ways to call a function:

- Symbolic Function Call
- Function Call Using a Variable

Symbolic Function Call

When using the symbolic function call, the user specifies exactly the name of the function to be executed at runtime.

If only a symbolic function call is specified in the Natural source, the corresponding Natural function definition is retrieved automatically, unless a suitable prototype definition has been specified before. The corresponding name of the object, which contains the Natural function definition, is retrieved according to the symbolic logical function name. This is done using the link records of the *FILEDIR.SAG* file. In this case, the corresponding function definition must have been stowed before the link record can be generated for the first time.

This feature causes that all parameter definitions of a Natural function call are always checked for valid format/length definitions.

Function Call Using a Variable

In a function call using a variable, the name of the desired function definition is stored inside an alphanumeric variable. At runtime, Natural jumps into the corresponding function definition, the name of which has been stored inside the variable.

In order to identify these two kinds of function calls, a corresponding prototype definition must be specified. Additionally, the prototype may contain the whole signature of the function definition. If no signature has been given, the function call must contain a `PT` clause in order to specify the missing parts of the signature. Therefore, the `VARIABLE` keyword of such a prototype specified inside the `PT` clause has no effect. For variable function calls, there must be a valid prototype with the same name as the alphanumeric variable containing the function name.

If no prototype can be assigned to the function call, a special *prototype-cast* is necessary in order to define the return format/length at compilation time. The *prototype-cast* and the parameter list must be enclosed in pointy brackets and parentheses, as displayed in the syntax diagram.

If you want to use the variable method, you must define a prototype with the same name as the *variable-name* using the keyword `VARIABLE`.

Example:

```
DEFINE PROTOTYPE VARIABLE variable-name
```

Note:
You can only use a function call when the operand involved cannot be modified. However, if a function call is used in an `INPUT` statement, the return value will be displayed as an "output only" field (`AD=O`).

Restrictions

Function calls are *not* allowed in the following situations:

- in a `DEFINE DATA` statement;
- in a database access or update statement (`READ`, `FIND`, `SELECT`, `UPDATE`, `STORE`, etc.);
- in an `AT BREAK` or `IF BREAK` statement;
- as an argument of the system functions `AVER`, `COUNT`, `MAX`, `MIN`, `NAVER`, `NCOUNT`, `NMIN`, `OLD`, `SUM`, `TOTAL`;
- as index notation.

Syntax Description

A function call may consist of the following syntax elements:

- call-name
- prototype-cast
- intermediate-result-definition
- Parameter(s)

call-name

```
{ function-name            }
{ prototype-variable-name  }
```

Operand Definition Table:

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
prototype-variable-name		S	A			A	U											yes	no

Syntax Element Description:

function-name	The *function-name* clause is the symbolic function name. The corresponding function definition is defined in a certain function object file.
prototype-variable-name	The *prototype-variable-name* is the name of the variable containing the real name of the function which is to be called. An alphanumeric or Unicode variable with the same name must have already been defined.

prototype-cast

```
PT= { prototype-name           }
    { prototype-variable-name  }
```

The *prototype-cast* must be used for function calls where no signature is specified in the corresponding function prototype (for example, signature clause of prototype definition is defined as `UNKNOWN`).

intermediate-result-definition

```
IR= { format-length [/array-definition]                 }
    { ( { A [/array-definition] } ) DYNAMIC              }
    {   { U                     }                        }
    {   { B                     }                        }
```

This clause enables you to specify the *format-length/array-definition* 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.

If, in addition, a prototype is valid for the function call, it is checked that the *format-length/array-definition* of the return value of the function definition is move-compatible to the intermediate result. If this is not the case, an error will be raised. The intermediate result is taken for the return value.

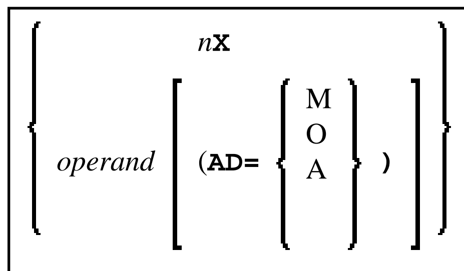
Alternatively, arrays are possible as return values, that is, array definitions may be specified as intermediate results. With an *array-definition*, you define the lower and upper bound of a dimension in an array definition. See *Array Dimension Definition* in the *Statements* documentation.

format-length	The format and length of the field. For information on format/length definition of user-defined variables, see *Format and Length of User-Defined Variables*.
A, B or U	Data format: Alphanumeric, binary or Unicode for dynamic variables.
array-definition	With an *array-definition*, you define the lower and upper bounds of the dimensions in an array definition. See *Array Dimension Definition* in the *Statements* documentation.
DYNAMIC	A field may be defined as DYNAMIC. For further information on processing dynamic variables, see *Introduction to Dynamic Variables and Fields*.

Parameter(s)

Each parameter may be an operand when calling the function. If a parameter is defined with the keyword OPTIONAL in the subprogram's DEFINE DATA PARAMETER statement, the corresponding operand values may be omitted in the function call. In this case, use the *n*X notation (where *n* is a whole integer greater than or equal to 1) or just omit this argument.

You can specify the session parameter AD for each argument.



Operand Definition Table:

Operand	Possible Structure					Possible Formats												Referencing Permitted	Dynamic Definition
operand	C	S	A	G		A	N	P	I	F	B	D	T	L	C	G	O	yes	yes

For an example of the proper usage of this function call, see the example in the description of the DEFINE PROTOTYPE statement.

<table>
<tr><td>nX</td><td colspan="2">Parameters to be Skipped:

With the notation nX you can specify that the next n parameters are to be skipped (for example, 1X to skip the next parameter, or 3X to skip the next three parameters); this means that for the next n parameters no values are passed to the subprogram.

A parameter that is to be skipped must be defined with the keyword OPTIONAL in the subprogram's DEFINE DATA PARAMETER statement. OPTIONAL means that a value can - but need not - be passed from the invoking object to such a parameter.</td></tr>
<tr><td rowspan="5">AD=</td><td colspan="2">Attribute Definition:

If operand is a variable, you can mark it in one of the following ways:</td></tr>
<tr><td>AD=O</td><td>Non-modifiable, see session parameter AD=O.

Note:
Internally, AD=O is processed in the same way as BY VALUE (see the section parameter-data-definition in the description of the DEFINE DATA statement).</td></tr>
<tr><td>AD=M</td><td>Modifiable, see session parameter AD=M.

This is the default setting.</td></tr>
<tr><td>AD=A</td><td>Input only, see session parameter AD=A.</td></tr>
<tr><td colspan="2">If operand is a constant, AD cannot be explicitly specified. For constants, AD=O always applies.</td></tr>
</table>