Benutzer-definierte Funktionen

This document covers the following topics:

- Einführung in benutzer-definierte Funktionen
- Unterschied zwischen Function Call und Subprogram Call
- Definition einer Function (DEFINE FUNCTION)
- Definition eines Prototype (DEFINE PROTOTYPE)
- Symbolischer und variabler Function Call
- Automatische/Implizite Prototype-Definition (APT)
- Prototype Cast (PT-Klausel)
- Zwischenergebnis für den Rückgabewert (IR-Klausel)
- Kombinationen möglicher Prototype-Definitionen
- Rekursiver Function Call
- Behavior of Functions in Statements and Expressions
- Usage of Functions as Statements

Einführung in benutzer-definierte Funktionen

Funktionen (Functions) bieten Ihnen ähnlich wie Subprogramme die Möglichkeit, Daten zu empfangen, diese zu verändern und die Ergebnisse an das rufende Modul zurückzugeben. Der Vorteil, den Funktionen im Vergleich zu Subprogrammen bieten, liegt darin, dass Funktionsaufrufe (Function Calls) direkt innerhalb von Statements und Ausdrücken verwendet werden können, ohne dass dazu zusätzliche temporäre Variablen nötig sind.

Normalerweise wird je nach den Parametern, die der Function beigegeben werden, das Ergebnis in der Function erzeugt und wird dann an das rufende Objekt zurückgegeben. Falls andere Werte an das rufende Modul zurückgegeben werden sollen, kann dies mit Hilfe der Parameter erfolgen; siehe *Subprogramm*.

Sobald der Code der Function vollständig ausgeführt worden ist, wird die Kontrolle an das rufende Objekt zurückgegeben und das Programm fährt mit dem nach dem Function Call vorhandenen Statement fort.

Weitere Informationen:

- Natural-Objekttyp Function
- Function Call

- Natural-Statements `DEFINE FUNCTION`, `DEFINE PROTOTYPE`

Unterschied zwischen Function Call und Subprogram Call

Das folgende Beispiel veranschaulicht den Unterschied zwischen der Verwendung eines Function Call und eines Subprogram Call.

Beispiel für die Verwendung eines 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.

Programm-Objekt:

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

Function-Objekt:

```
/* 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-Objekt (zum Beispiel "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
```

Um die gleiche Funktionalität mit Hilfe eines Subprogramms zu erzielen, müssen Sie temporäre Variablen verwenden.

Beispiel für die Verwendung eines Subprogram Call:

Das folgende Beispiel enthält ein Objekt vom Typ Programm, das ein Objekt des Typs Subprogramm unter Verwendung einer temporären Variablen aufruft.

Programm-Objekt:

```
/* 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
```

Subprogramm-Objekt (zum Beispiel "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
```

Definition einer Function (DEFINE FUNCTION)

Die Funktionsdefinition besteht aus dem Natural-Code, der ausgeführt werden soll, wenn die Function aufgerufen wird. Ähnlich wie bei Subprogrammen müssen Sie zunächst ein Natural-Objekt anlegen, in diesem Fall vom Typ "Function", das die Funktionsdefinition enthält. Die Funktionsdefinition wird mit Hilfe des Natural-Statements `DEFINE FUNCTION` erstellt.

Bei dem Funktionsaufruf (Function Call) kann es sich um einen beliebigen Objekttyp handeln, der ausführbaren Code enthält.

Definition eines Prototype (DEFINE PROTOTYPE)

Um Function Calls kompilieren zu können, benötigt Natural Informationen über Format/Länge- und ggf. die Array-Definition des Rückgabewerts. Diese Informationen werden dann dem Compiler in der Prototype-Definition zur Verfügung gestellt. Die Prototype-Definition wird mit dem Natural-Statement `DEFINE PROTOTYPE` angelegt. Dort können Sie auch die Definition des zurückzugebenden Parameters einfügen, die dann bei der Kompilierung geprüft wird.

Da Natural die Verbindung zwischen rufenden und gerufenen Objekten zur Laufzeit und nicht vorher herstellt, weiss der Rechner nicht, mit welcher Art von Function-Rückgabewert er es bei der Kompilierung zu tun hat. Das liegt daran, dass das Objekt, das die Function enthält, nicht notwendigerweise (während des Kompilierens) existieren muss. Darum wird die Prototype-Definition angelegt, so dass die Format/Länge- und ggf. die Array-Definition beim Kompilieren in das generierte Programm (GP) hinein generiert werden kann.

Es ist wichtig zu wissen, dass eine Prototype-Definition niemals ausführbaren Code enthält. Eine Prototype-Definition enthält lediglich die folgenden Informationen über einen Function Call: die Format/Länge- und ggf. die Array-Definition des Rückgabewerts bzw. des Parameters, der zurückgereicht wird.

Symbolischer und variabler Function Call

Um einen variablen Function Call zu definieren müssen Sie immer ein `DEFINE PROTOTYPE VARIABLE`-Statement benutzen. Andernfalls wird angenommen, dass es sich bei dem Function Call um einen impliziten symbolischen Function Call handelt.

Weitere Informationen zu diesem Thema finden Sie im Abschnitt *Function Call*.

Automatische/Implizite Prototype-Definition (APT)

Wenn weder eine explizite Prototype-Definition (EPT) noch eine `PT`-Klausel (Prototype Cast) existiert, findet im generierten Programm die Suche nach der Prototype-Definition statt. Weitere Informationen, siehe *Kombinationen möglicher Prototype-Definitionen* weiter unten.

Prototype Cast (PT-Klausel)

Um den entsprechenden Prototype einer bestimmten Function zu finden, sucht Natural nach einem Prototype, der den Namen der Function trägt. Falls ein solcher Prototype nicht gefunden wird, geht Natural davon aus, dass es ein symbolischer Function Call ist. In diesem Fall muss die "Signatur" der Function unter Verwendung des Schlüsselworts `PT=` in dem Function Call definiert werden.

Zwischenergebnis für den Rückgabewert (IR-Klausel)

Mit dieser Klausel können Sie den Rückgabewert für einen Function Call angeben, ohne eine explizite oder implizite Prototype-Definition zu verwenden, das heisst, sie ermöglicht die explizite Angabe eines Zwischenergebnisses. Weitere Informationen siehe *Function Call*, *intermediate-result-definition*

Kombinationen möglicher Prototype-Definitionen

Die folgende Tabelle verdeutlicht die Auswirkungen auf eine Prototype-Definition in Abhängigkeit von verschiedenen Syntax-Kombinationen, die bei der Benutzung des `DEFINE PROTOTYPE`-Statements und/oder der im Function Call vorhandenen Klauseln möglich sind. Folgende Möglichkeiten gibt es, um Teile eines Function Prototype zu definieren, die sich nur auf den Function Call auswirken, zu dem sie gehören:

- **Explizite DEFINE PROTOTYPE-Definition (EPT)**
  Kann symbolischen/variablen Function Call, Parameter-Definition, Rückgabewert-Definition festlegen.

- **Prototype Cast (PT-Klausel)**
  Kann Parameter-Definition, Rückgabewert-Definition festlegen.

- **Zwischenergebnis für den Rückgabewert (IR-Klausel)**
  Kann Rückgabewert-Definition festlegen.

Fall	Explizite Prototype-Definition in `DEFINE PROTOTYPE` (EPT)	`PT`-Klausel im Function Call (PT)	`IR`-Klausel im Function Call (IR)		Automatisches Einlesen der Prototype-Definition aus dem GP (APT)	Prototype-Verhalten
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)

Dabei ist:

EPT	Explizites `DEFINE PROTOTYPE`-Statement.
PT	Prototype Cast (`PT`-Klausel).
IR	Zwischenergebnis für den Rückgabewert(`IR`-Klausel).
APT	Automatische Prototype-Definition über externes GP.
S	Symbolischer Function Call.
V	Variabler Function Call.
SV(EPT)	Explizite Prototype-Definition entscheidet, ob ein symbolischer oder ein variabler Function Call durchgeführt wird.
R(IR)	Die Rückgabevariable (R) wird durch die `IR`-Klausel im Function Call definiert.
R(PT)	Die Rückgabevariable (R) wird durch die `PT`-Klausel im Function Call definiert.
R(EPT)	Die Rückgabevariable (R) wird durch das explizite `DEFINE PROTOTYPE`-Statement definiert.
PS(PT)	Die Parameter-Signatur (PS) (d.h. die Parameter-Definition ohne Rückgabewert-Definition) wird durch die `PT`-Klausel im Function Call definiert.
PS(EPT)	Die Parameter-Signatur (PS) (d.h. die Parameter-Definition ohne Rückgabewert-Definition) wird durch das explizite `DEFINE PROTOTYPE`-Statement definiert.
PS(APT)	Die Parameter-Signatur (PS) wird automatisch durch das Einlesen in die Prototype-Definition aus dem generierten Programm (GP) definiert.
R(APT)	Die Rückgabevariable (R) wird durch automatische Prototype-Definition über externes GP definiert.

Als Beispiel das Verhalten gemäß Fall 1 aus der obigen Tabelle:

Wie ist das Verhalten, wenn ein explizites DEFINE PROTOTYPE-Statement (EPT) verwendet wird und wenn im Function Call die PT- und IR-Klauseln definiert sind?

Die EPT-Definition entscheidet, ob ein symbolischer oder ein variabler Function Call durchgeführt wird. Von einem variablen Function Call wird ausgegangen, wenn zuvor ein DEFINE PROTOTYPE VARIABLE-Statement definiert worden ist. Die Parameter-Signatur (d.h. die Format/Länge-Definition aller Parameter ohne Rückgabewert-Definition) wird durch die PT-Klausel definiert, und die Format/Länge-Angabe des Rückgabewerts wird durch die IR-Klausel im Function Call festgelegt. In diesem Fall wird keine automatische Prototype-Definition (APT) gestartet.

Abschließend können aus den oben aufgeführten Fällen die folgenden allgemeinen Regeln abgeleitet werden:

- Im Falle von variablen Function Calls muss für den Aufruf immer eine explizite Prototype-Definition (ETP) vorhanden sein.
- Die PT-Klausel entscheidet nicht, ob es sich um einen symbolischen oder einen variablen Function Call handelt.
- Die Definitionen in der PT-Klausel überschreiben die ETP-Definitionen für Parameter und Rückgabewert.
- Die Definitionen in der IR-Klausel überschreiben die Rückgabewert-Defintion.
- Wenn weder eine ETP-Definition noch eine PT-Klausel existiert, erfolgt im generierten Program die Suche nach der Prototyp-Definition (automatische Prototyp-Definition).

Rekursiver Function Call

Wenn die Funktion rekursiv aufgerufen werden soll, muss der Prototyp der Function in der Function-Definition enthalten sein oder er muss mittels einer INCLUDE-Datei eingefügt werden.

Beispiel:

Function-Objekt:

```
/* 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-Objekt (zum Beispiel mit dem Namen C#FACT):

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

Programm-Objekt:

```
/* 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
```