Einführung in Modula-3
Funktionale Programme

- **Funktionale Programmierung**
- **Funktionale Programme in Modula-3**
- **Funktionen, Parameter**
- **Einfache Datentypen**
- **Rekursion**

Funktionale Programmierung

- **Konzept**
  - Formulierung von ***Funktionsdefinitionen***
  - Ausführung eines funktionalen Programms besteht in der ***Berechnung*** eines ***Ausdrucks*** mit Hilfe dieser Funktionen
  - Berechnung liefert ein ***Ergebnis*** zurück
  - Es existiert keine Möglichkeit, Zwischenergebnisse zu speichern
- **Was benötigt man dazu**
  - Daten / ***Datentypen***
  - ***elementare*** Funktionen
  - Möglichkeit, Funktionen zu ***definieren***
  - Ausdrucksmittel zur ***Vernetzung*** von Funktionen
- **Anmerkung:**
  - Es gibt rein funktionale Programmiersprachen MIRANDA, LISP
  - Viele sog. Hochsprachen erlauben eine gewisse Art der funktionalen Programmierung (Modula-3).

Funktionen in Modula-3

- **Eine Funktion**
  - hat einen ***Namen***
  - hat keinen oder mehrere ***Eingabeparameter*** (Argumentbereich)
  - hat einen ***Ergebnistyp*** (Ergebnisbereich)
  - hat eine ***Berechnungsvorschrift***
  - ist frei von ***Seiteneffekten***

Signatur

```
PROCEDURE Quadrat ( x : REAL) : REAL =
BEGIN
  RETURN ( x * x );
END Quadrat;
```

Berechnungsvorschrift **Name** **formale Eingabeparameter** **Ergebnistyp**

Aufruf einer Funktion

```
MODULE QuadratM EXPORTS Main;
(* Dieses Programm berechnet das Quadrat einer Zahl
   Autor           : Horst Lichter
   Umgebung        : SRC-Modula-3 rel. 3.6, Windows NT 4.0
   Erstellt        : 20.08.98    Letzte Aenderung: 20.08.98
*)

IMPORT SIO;

PROCEDURE Quadrat ( x : REAL) : REAL =
BEGIN
  RETURN ( x * x );
END Quadrat;

BEGIN
  SIO.PutReal (Quadrat (SIO.GetReal()));
END QuadratM.
```

Deklaration und ***Definition*** der Funktion

Aufruf der Funktion mit einem ***aktuellen*** Parameter;

Das ***Ergebnis*** der Funktion wird an die Prozedur PutReal ***übergeben***

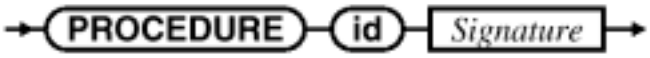
Formale & aktuelle Parameter

- **Parameter**
  - erlauben es, Funktionen mit ***Eingabewerten*** zu versorgen
  - haben eine ***Typ***
  - dadurch werden Funktionen ***flexible einsetzbar***
- **Formale Parameter**
  - werden in der ***Definition*** einer Funktion angegeben
  - dienen als ***Stellvertreter*** im ***Rumpf*** der Funktion für die zur Laufzeit des Programms übergebenen aktuellen Parameter
- **Aktuelle Parameter**
  - Beim ***Aufruf*** einer Funktion müssen ihre formalen Parameter gemäß ihrer Definition an aktuelle Parameter ***gebunden*** werden.
  - Diese werden dann im Rumpf ***verwendet***.

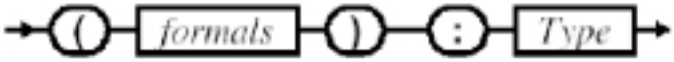
Syntax von M3-Funktionen - 1

Declaration

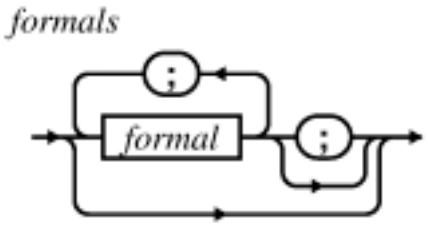
Procedure heading

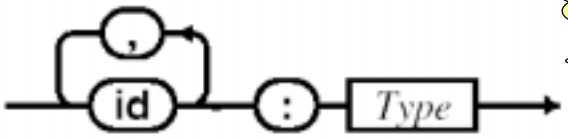


Signature



Syntax von M3-Funktionen - 2





Vereinfachte Darstellung !

Formale & aktuelle Parameter

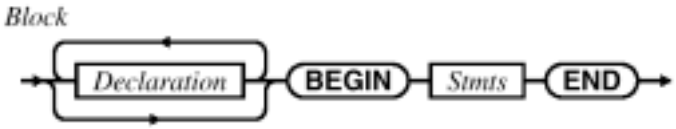
```
PROCEDURE Quadrat ( x : REAL) : REAL =
BEGIN
  RETURN ( x * x );
END Quadrat;

BEGIN
  SIO.PutReal (Quadrat (2.5));
END QuadratM.
```

Binden der aktuellen Werte (Parameter) an die formalen Parameter (Stellvertreter)

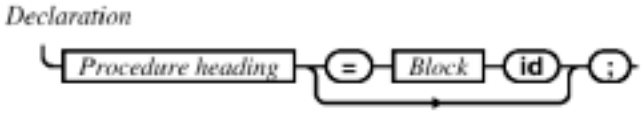
Vorschau: ...

Deklarationen (vorläufig)

- **Wir wissen bereits:**
  - ***Blöcke*** können ***Deklarationen*** enthalten
  - und bestehen aus ***Anweisungen*** (Statements)



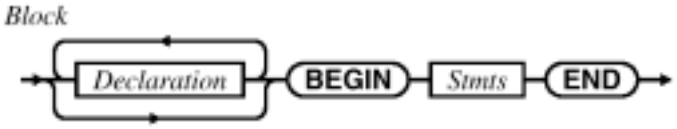
- **Deklarationen:**
  - Idee: Namen (***Bezeichner***) werden vereinbart, damit diese später benutzt werden können
  - Die in einem Block deklarierten Bezeichner sind nur innerhalb des Blockes ***gültig***



Vorschau: ...

Anweisungen (vorläufig)

- **Anweisungen:**
  - sind in Blöcken enthalten und werden durch ";" getrennt
  - atomare Bausteine für Modula-3 Programme



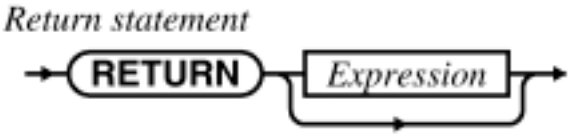
  - Die Folge der Anweisungen eines Blocks wird bei Ausführung in der ***Reihenfolge der Aufschreibung*** abgearbeitet

- **Beispiele für Anweisungen:**
  - RETURN-Anweisung
  - CALL-Anweisung (Funktionsaufruf)

Ausdrücke

Vorschau: ...

- **Ausdrücke**
  - bezeichnen Elemente, die ***ausgewertet*** werden
  - liefern einen Wert (Ergebnis)
- **Beispiele**
  - arithmetische Ausdrücke
    - x * x
  - logische Ausdrücke
    - x AND y
- **Viele Anweisungen können Ausdrücke enthalten**



Datentypen (vorläufig)

Vorschau: ...

- **Typbegriff**
  - Im Zusammenhang mit Programmiersprachen hat der Begriff ***Typ*** oder auch ***Datentyp*** eine zentrale Bedeutung.
- **Definition**
  - Unter einem ***Datentyp*** versteht man die ***Zusammenfassung*** von ***Wertebereich*** und ***Operationen*** zu einer ***Einheit.***
  - Ein Typ hat eine Struktur und Literale (Aggregate).
- **Man unterscheidet:**
  - ***einfache*** (skalare) Datentypen
  - ***zusammengesetzte*** Datentypen
    Konstruktionsmittel: Datentypkonstruktoren
  - ***vordefinierte*** Datentypen
  - ***selbstdefinierte*** Datentypen

Vordefinierte Datentypen

Vordefinierte Datentypen in Modula-3

- **Einfache, skalare Datentypen**
  - **Ganze Zahlen (diskret)**
    - darunter fallen die Typen **`INTEGER`** und **`CARDINAL`**
  - **Zeichen (diskret)**
    - Werte eines bestimmten Zeichenvorrates; z.B. definiert der ASCII-Zeichenvorrat 128 Zeichen
  - **Wahrheitswerte (diskret)**
    - Werte sind {wahr, falsch}, Literale **`TRUE`**, **`FALSE`**
  - **Gleitkommazahlen**
    - reelle Zahlen, **`REAL`**, **`LONGREAL`**, **`EXTENDED`**
- **Zusammengesetzte Datentypen**
  - **Texte**
    - sind eine Folge von Zeichen

Vordefinierte Datentypen

Ganzahlige Datentypen

- **Typen: `INTEGER` und `CARDINAL`**
  - **Integer-Zahlen** sind ***ganzzahlige*** Werte innerhalb der Unter- und Obergrenze des jeweiligen Rechners
  - **Cardinal-Zahlen** sind sind ***nicht-negative*** ganzzahlige Werte, d.h. zwischen 0 und der Obergrenze des jeweiligen Rechners
- **Wertebereich**
  - auf einen 32-Bit-Rechner:
    - **`INTEGER`** [-2147483648 .. 2147483647] oder [-2^{31} .. $2^{31}-1$]
    - 1 Bit für das Vorzeichen, 31Bit für die Zahlendarstellung
  - Zahlen sind geordnet (***Ordinaltyp***)
- **Operationen**
  - arithmetische Operationen (**`+`**, **`-`**, **`*`**, **`DIV`**, **`MOD`**)
  - Vergleichsoperationen (**`=`**, **`#`**, **`<`**, **`>`**, **`<=`**, **`>=`**)
  - vordefinierte Funktionen (**`FIRST(type)`**, **`LAST(type)`**, **`INC(z)`**, **`DEC(z)`**, **`ABS(z)`**)

Vordefinierte Datentypen

Beispiel

```
MODULE Zahlen EXPORTS Main;
(* Dieses Programm zeigt dem Umgang mit ganzen Zahlen *)

IMPORT SIO;

PROCEDURE Modulo (x,y: INTEGER): INTEGER =
(* MOD ist def. : x MOD y = x - y*(x DIV y) *)
BEGIN
  RETURN ( x - y*(x DIV y) );
END Modulo;

BEGIN
  (* Ausgabe der Unter- und Obergrenze von INTEGER *)
  SIO.PutInt (FIRST(INTEGER)); SIO.Nl();
  SIO.PutInt (LAST(INTEGER));  SIO.Nl();

  SIO.PutInt (20 DIV 6); SIO.Nl();     (* = 3 *)
  SIO.PutInt (20 MOD 6); SIO.Nl();     (* = 2 *)
  SIO.PutInt (Modulo(20,6));           (* = 2 *)
END Zahlen.
```


Vordefinierte Datentypen

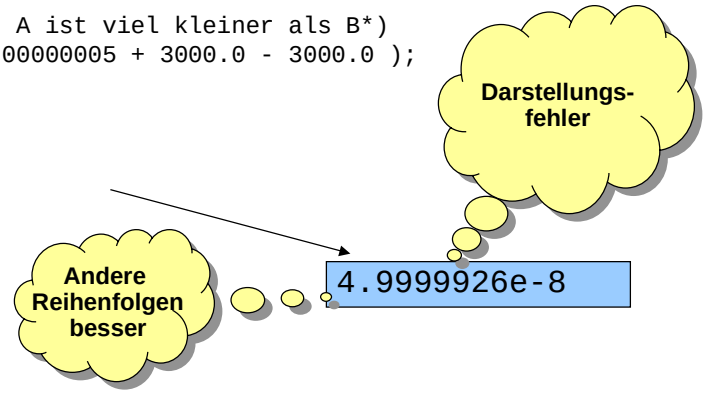
Gleitpunktdatentypen

- **Typen: REAL, LONGREAL, EXTENDED**
  - repräsentieren die darstellbaren reellen Zahlen
- **Darstellung**
  - Wertebereich ist ***beschränkt*** (im Unterschied zur Mathematik)
  - Genauigkeit der Darstellung ist ***beschränkt***
    - Bspl: wir haben 4 Stellen zur Verfügung
      größte Zahl: 9999
      kleinste Zahl: 0.001
    - Probleme: die Zahlen 0.0005 und 0.000089 sind nicht zu unterscheiden (es wird gerundet)
  - Rechnen mit Gleitkommazahlen ist immer **fehlerbehaftet**!
    - "Numerik" liefert Techniken und Algorithmen, um Fehler zu beherrschen

Vordefinierte Datentypen

Beispiel für Gleitkommazahlen

```
MODULE Gleitkomma EXPORTS Main;
(* Dieses Programm zeigt Rundungsprobleme bei Gleitkommazahlen *)

IMPORT SIO;


BEGIN
  (* A + B - B mit A ist viel kleiner als B*)
  SIO.PutReal ( 0.00000005 + 3000.0 - 3000.0 );

END Gleitkomma.
```



Vordefinierte Datentypen

Zeichentyp CHAR

- **Typ CHAR**
  - **CHAR** (character) bezeichnet eine ***endliche***, ***geordnete Menge*** von Zeichen
  - **CHAR** ist ein ***Ordinaltyp***
- **Wertebereich**
  - Latin-1, viele Rechner benutzen den ASCII-Zeichensatz
  - Zeichenliterale: 'A' 'z' '1'
  - Spezialzeichen: `\n` Zeilenvorschub `\t` Tabulator `\\` Backslash
    `\'` Apostroph `\f` Seitenumbruch
    `\r` Wagenrücklauf `\"` Anführungszeichen
- **Operationen**
  - Vergleichsoperationen (**=**, #, **<**, **>**, **<=**, **>=**)
  - Vordefinierte Funktionen **FIRST(CHAR), LAST(CHAR), INC(z), DEC(z), ORD(z), VAL(i)**

Vordefinierte Datentypen

Beispiel - Zeichen

```
MODULE KleinGross EXPORTS Main;
(* Dieses Programm wandelt einen Klein- in einen Grossbuchstaben um *)

IMPORT SIO;

PROCEDURE Offset (): INTEGER =
BEGIN
  RETURN (ORD('A') - ORD('a'));
END Offset;


BEGIN
  (* Kleinbuchstaben einlesen und umwandeln *)
  SIO.PutChar ( VAL(ORD(SIO.GetChar()) + Offset(), CHAR) );

END KleinGross.
```


Vordefinierte Datentypen

Texte (Zeichenketten)

- **Typ TEXT**
  - repräsentiert eine ***beliebig lange Folge*** von Zeichen (kann auch leer sein)
  - in vielen Sprachen nicht explizit vorhanden
- **Wertebereich**
  - Textliterale werden in " " notiert
  - z.B. "Das ist ein Text mit Zeilenvorschub\n"
"\"Dieser Text beginnt und endet mit einem Hochkomma\""
- **Operationen**
  - Konkatenation : &
    - Bspl: "Heute " & "ist " & "Freitag." ⇨ "Heute ist Freitag."
  - Schnittstelle des Moduls "Text"
    - Equal, Length, Empty, FindChar ….

Vordefinierte Datentypen

Wahrheitswerte

- **Typ BOOLEAN**
  - repräsentiert die beiden vordefinierten Wahrheitswerte
- **Wertebereich**
  - wahr, Literal **TRUE**
  - falsch, Literal **FALSE**
- **Operationen**
  - Komplement (**NOT**), Oder (**OR**), Und (**AND**)

p	q	NOT q	p OR q	p AND q
1	1	0	1	1
1	0	1	1	0
0	1	0	1	0
0	0	1	0	0

1 wahr
0 falsch

Vordefinierte Datentypen

Beispiel für Wahrheitswerte

```
MODULE BooleanM EXPORTS Main;
(* Dieses Programm berechnet die Wahrheitstabelle NOT, OR, AND *)
IMPORT SIO, Fmt;

PROCEDURE NotOrAnd ( a, b : BOOLEAN) : TEXT =
BEGIN
  RETURN (Fmt.Bool(a) & " " & Fmt.Bool(b) & " : " &
          Fmt.Bool( NOT(b))       & " " &
          Fmt.Bool( a OR b)       & " " &
          Fmt.Bool(a AND b) );
END NotOrAnd;

BEGIN
  (* Ausgabe der Wertetabelle *)
  SIO.PutLine(NotOrAnd(FALSE, FALSE));
  SIO.PutText(NotOrAnd(TRUE, FALSE));
  SIO.PutLine(NotOrAnd(FALSE, TRUE));
  SIO.PutLine(NotOrAnd(TRUE, TRUE));

END BooleanM.
```

produzierte Ausgabe

```
FALSE FALSE : TRUE  FALSE FALSE
TRUE  FALSE : TRUE  TRUE  FALSE
FALSE TRUE  : FALSE TRUE  FALSE
TRUE  TRUE  : FALSE TRUE  TRUE
```

Vernetzung von Funktionen

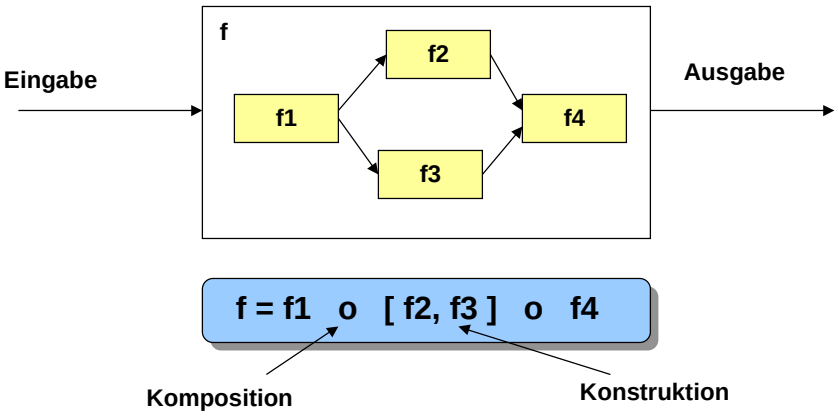
Funktionskomposition

- **Um komplexe Ausdrücke zu berechnen,**
  - werden Funktionen vernetzt.
- **Funktionalformen (oder Funktionale)**
  - beschreiben "Vernetzungsmuster"
- **Beispiele für Funktionalformen**
  - **Komposition**
    - f o g : x = g : (f : x)
  - **Konstruktion**
    - [f, g, h, ...] : x = (f : x, g : x, h : x, ...)
  - **Bedingung**
    - if t then f else g : x =

$$\begin{cases} f : x & \text{, falls } t : x = \text{TRUE} \\ g : x & \text{, falls } t : x = \text{FALSE} \\ ? & \text{, sonst} \end{cases}$$

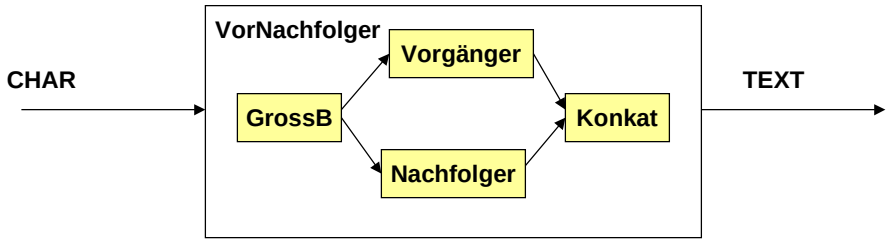
Vernetzung von Funktionen

schematisches Beispiel



Vernetzung von Funktionen

Beispielnetz



VorNachfolger = GrossB o [Vorgänger, Nachfolger] o Konkat

Beispiel:

Eingabe: 'h'

Ausgabe: "GI"

Vernetzung von Funktionen

Beispiel: Detaillierung

```
MODULE VorNachfolger EXPORTS Main;
IMPORT SIO;

PROCEDURE GrossB(c : CHAR) : CHAR =
...
PROCEDURE Vorgaenger (c : CHAR) : CHAR =
...
PROCEDURE Nachfolger (c : CHAR) : CHAR =
...
PROCEDURE AddOrd (c, d : CHAR) : CARDINAL =
. . .
PROCEDURE Konkat (c, d : CHAR) : TEXT =
...

BEGIN
  (* VorNachfolger = GrossB o   [ Vorgänger, Nachfolger ]   o   Konkat *)
  SIO.PutText ( Konkat ( Vorgaenger( GrossB( SIO.GetChar() )) ,
                         Nachfolger( GrossB( SIO.GetChar() ))
                       )
              );

END VorNachfolger.
```

Da **keine Variablen** (Zwischenspeicher) verwendet werden, muß der Wert **zweimal** eingelesen werden !

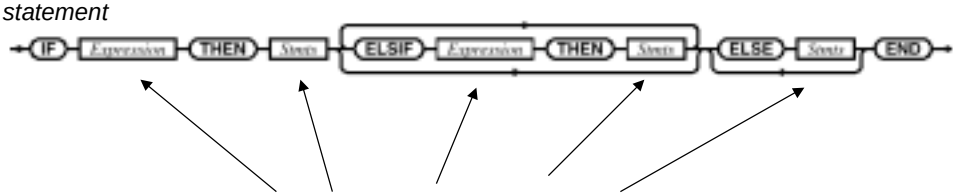
Funktion GrossB muß **zweimal** ausgeführt werden, da Modula-3 die **Konstruktion** von Funktionen nicht unterstützt.

Bedingungen in funkt. Programmen

Funktionale Programme

- **Idee:**
  - In Abhängigkeit von ***Bedingungen*** soll eine ***alternative*** Programmkomponente ausgeführt werden
  - Bedingungen müssen einen Wert vom Typ ***BOOLEAN*** liefern
  - Modula-3 bietet dazu u.a. die ***IF-Anweisung*** an

If statement



Bei funktionalen Programmen stehen hier ***ausschließlich*** Funktionsaufrufe!

Bedingungen in funkt. Programmen

Bedingung - IF-THEN-ELSE

- **Typische Formen von Bedingungen:**

```
IF bf THEN
  f1;
ELSE
  f2;
END;
```

Alternative "Entweder-Oder" (zweiseitig bedingte Anweisung)

```
IF bf THEN
  f1;
END;
```

einseitig bedingte Anweisung

```
IF    bf1 THEN
  f1;
ELSIF bf2 THEN
  f2;
ELSIF bf3 THEN
  f3;
...
ELSE
  fn;
END;
```

mehrseitig bedingte Anweisung

Die ***Bedingungen*** (bfi) werden der Reihe nach ausgewertet, bis eine ***WAHR*** ist. Dann wird die entsprechende Anweisung (Funktionsaufruf) ausgeführt.

Ist ***keine*** Bedingung wahr, dann wird der ***ELSE-Zweig*** ausgeführt

Bedingungen in funkt. Programmen

Beispiel - 1

- **Aufgabe:**
  - Eingabe ist eine ganze Zahl
  - Stelle fest, ob diese 1-, 2-, 3-, mindestens 4-stellig oder negativ ist!

```
PROCEDURE ErmittleStelligkeit(i : INTEGER) : TEXT =
BEGIN
  IF IstEinstellig(i)     THEN RETURN ("einstellig") END;
  IF IstZweistellig(i)    THEN RETURN ("zweistellig") END;
  IF IstDreistellig(i)    THEN RETURN ("dreistellig") END;

  IF IstMinVierstellig(i) THEN RETURN ("min. vierstellig")
  ELSE RETURN ("negativ")
  END;
END ErmittleStelligkeit;

BEGIN
  SIO.PutText(ErmittleStelligkeit(SIO.GetInt()));
END Stelligkeit.
```


Bedingungen in funkt. Programmen

Beispiel - 2

```
PROCEDURE ErmittleStelligkeit(i : INTEGER) : TEXT =
BEGIN
  IF    IstEinstellig(i)     THEN RETURN ("einstellig")
  ELSIF IstZweistellig(i)    THEN RETURN ("zweistellig")
  ELSIF IstDreistellig(i)    THEN RETURN ("dreistellig")
  ELSIF IstMinVierstellig(i) THEN RETURN ("min. vierstellig")
  ELSE                            RETURN ("negativ")
  END;
END ErmittleStelligkeit;

BEGIN
  (* Aufruf der Hauptfunktion *)
  SIO.PutText(ErmittleStelligkeit(SIO.GetInt()));
END Stelligkeit.
```

ELSIF-Konstruktionen machen Bedingungen semantisch klarer

Bedingungen in funkt. Programmen

Beispiel - 3

```
MODULE Stelligkeit EXPORTS Main;
(* Dieses Programm berechnet die Stelligkeit von Zahlen *)
IMPORT IO;

PROCEDURE IstEinstellig ( i : INTEGER) : BOOLEAN =
BEGIN
  RETURN ( (i>=0) AND (i<10) );
END IstEinstellig;

PROCEDURE IstZweistellig ( i : INTEGER) : BOOLEAN =
BEGIN
  RETURN ( (i>=10) AND (i<100) );
END IstZweistellig;

PROCEDURE IstDreistellig ( i : INTEGER) : BOOLEAN =
BEGIN
  RETURN ( (i>=100) AND (i<1000) );
END IstDreistellig;

PROCEDURE IstMinVierstellig ( i : INTEGER) : BOOLEAN =
BEGIN
  RETURN ( (i>=1000) );
END IstMinVierstellig;
```


Rekursion

Rekursion

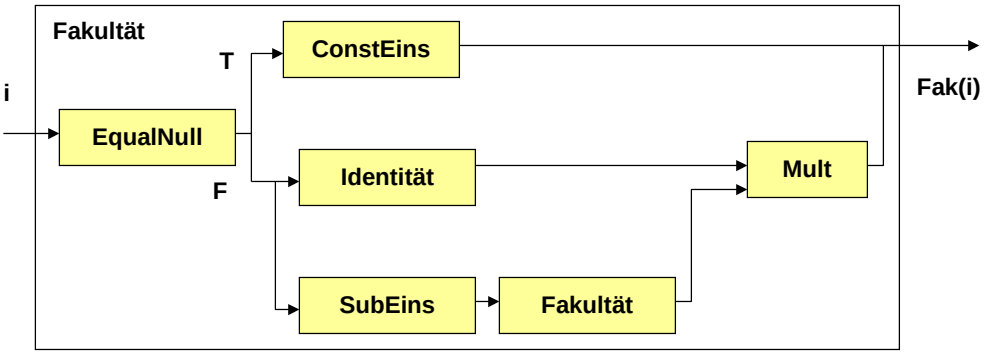
- **Idee:**
  - Allgemein bezeichnet man mit ***Rekursion*** die Definition eines Problems, einer Funktion oder ganz allgemein eines Verfahrens "***durch Rückführung auf sich selbst***".
- **Rekursive Funktion**
  - Darunter verstehen wir Funktionen, die sich ***selbst wieder aufrufen.***
- **Beispiel: Matrioschka-Puppen**

Anmerkungen

- **Ziel:**
  - Es darf keine ***unkontrollierte*** (unendliche) Rekursion entstehen.
  - Dies führt immer zu einem ***Laufzeitfehler.***
- **Konsequenz**
  - Jeder rekursive Funktionsaufruf gehört in eine ***bedingte Anweisung.***
    - Rekursionsabbruch: in definierten Fällen wird der rekursive Aufruf nicht ausgeführt
  - Muster:
    -
```
IF … THEN
  rekursiver Aufruf
ELSE
  Rekusionsabbruch
```
- **Bemerkung**
  - Rekursion führt zu ***prägnaten, knappen*** und ***eleganten*** Algorithmen für bestimmte Problemstellungen.

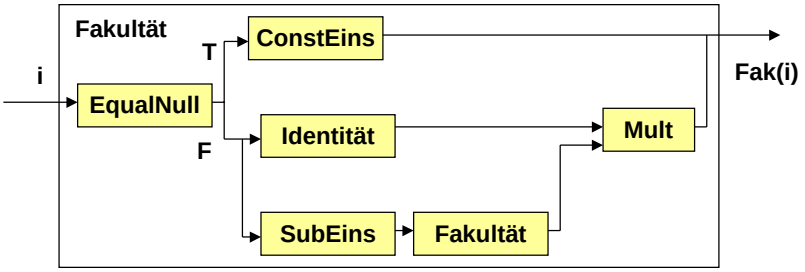
Beispiel Fakultät - 1

- **Fakultät is definiert:**
  - fak : n 1 , falls n = 0
    n * fak(n-1) n ∈ N
- **Funktionsnetz für Fakultät**



Rekursion

Beispiel Fakultät - 2




```
PROCEDURE Fakultaet (i : INTEGER) : INTEGER =
BEGIN
  IF EqualNull(i)
  THEN RETURN(ConstEins(i))
  ELSE RETURN (Mult(Identitaet(i), Fakultaet(SubEins(i))));
  END;
END Fakultaet ;
...
  SIO.PutInt(Fakultaet(SIO.GetInt()));
```


Rekursion

Beispiel Fakultät - 3

```
PROCEDURE EqualNull ( i : INTEGER) : BOOLEAN =
BEGIN
  RETURN ( i = 0 );
END EqualNull;

PROCEDURE ConstEins ( i : INTEGER) : INTEGER =
BEGIN
  RETURN ( 1 );
END ConstEins;

PROCEDURE Mult ( i, j : INTEGER) : INTEGER =
BEGIN
  RETURN ( i * j );
END Mult;

PROCEDURE Identitaet ( i : INTEGER) : INTEGER =
BEGIN
  RETURN ( i );
END Identitaet;

PROCEDURE SubEins ( i : INTEGER) : INTEGER =
BEGIN
  RETURN ( i - 1 );
END SubEins ;
```

Diese Funktionen sind nur im Sinne der **reinen funktionalen** Programmierung notwendig.

Sie können in der Definition der Funktion "Fakultaet" durch entsprechende **Ausdrücke** ersetzt werden!

Rekursion

Beispiel Fakultät - 4

```
MODULE FakultaetM1 EXPORTS Main;
(* Dieses Programm berechnet die Fakultaetsfunktion *)

IMPORT SIO;


PROCEDURE Fakultaet ( i : INTEGER) : INTEGER =
BEGIN
  IF i = 0
  THEN RETURN 1
  ELSE RETURN (i * Fakultaet(i-1));
  END;
END Fakultaet ;


BEGIN
  SIO.PutInt(Fakultaet(SIO.GetInt()));
END FakultaetM1.
```


Rekursion

Fibonacci-Funktion (rekursiv)

- **Wachstum einer Kaninchen-Population**
  - Wieviele Kaninchen-Pärchen gibt es nach n Jahren
    - Jahr 1 : 1 Pärchen
    - Jedes Pärchen hat ab dem zweiten Jahr je ein Pärchen Nachwuchs
  -

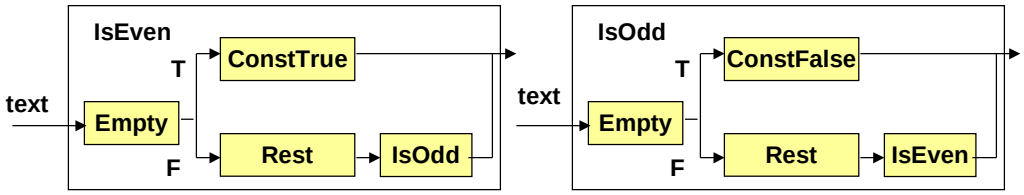
Jahr	1	2	3	4	5	6	7	8	9
Anzahl	1	1	2	3	5	8	13	21	34

```
PROCEDURE Fibonacci (arg: INTEGER): INTEGER =
BEGIN
  IF arg <= 2 THEN
    RETURN 1
  ELSE
    RETURN (Fibonacci (arg -1) + Fibonacci (arg - 2))
  END;
END Fibonacci ;
```

Rekursion

Indirekte Rekursion

- **Definition:**
  - ***Indirekte*** Rekursion kann in einem System von Funktionen definiert werden, die Seite an Seite vereinbart werden und sich ***gegenseitig stützen***.
- **Beispiel:**
  - Die beiden Funktionen (IsEven, IsOdd) sind ***indirekt*** rekursiv.
  - Sie stellen fest, ob eine Zeichenfolge eine gerade oder ungerade Anzahl von Zeichen enthält.



Rekursion

Beispiel - Indirekte Rekursion

```
MODULE EvenOdd EXPORTS Main;
(* Beispiel fuer die indirekte Rekursion *)
IMPORT SIO, Text;

PROCEDURE IsEven (str: TEXT) : BOOLEAN  =
BEGIN
  IF Text.Empty (str) THEN RETURN TRUE;
  ELSE RETURN (IsOdd (Text.Sub(str,1)));
  END;
END IsEven ;

PROCEDURE IsOdd (str: TEXT) : BOOLEAN  =
BEGIN
  IF Text.Empty (str) THEN RETURN FALSE;
  ELSE RETURN (IsEven (Text.Sub(str, 1)));
  END;
END IsOdd ;

BEGIN
  SIO.PutBool ( IsEven (SIO.GetWord()));     SIO.Nl();
  SIO.PutBool ( IsOdd  (SIO.GetWord()));
END EvenOdd.
```

Rekursion Funktional, Rekursion und Iteration

- **Funktionale Algorithmen**
  - kennen keine ***Variablen*** zur Zwischenspeicherung von Ergebnissen.
- **Nichtfunktionale Algorithmen**
  - speichern Zwischenergebnisse in ***Variablen*** ab.
- **Rekursive Algorithmen**
  - lösen ein Problem, in dem sie sich ***selbst wieder aufrufen*** (direkt oder indirekt).
- **Iterative Algorithmen**
  - besitzen Abschnitte, die bei der Ausführung ***mehrmals*** durchlaufen werden
  - Jeder rekursive Algorithmus kann in einen iterativen umgewandelt werden
    - allgemein (siehe Compiler)
    - problemspezifisch

Was haben wir gelernt!

- **Funktionale Programmierung**
  - Funktion, Parameter
  - Vernetzung von Funktionen
- **Anweisungen und Ausdrücke**
- **Datentyp**
  - einfache, zusammengesetzte, vordefinierte, selbstdefinierte
  - einfache (skalare) vordefinierte Datentypen: INTEGER, CARDINAL, REAL, LONGREAL, CHAR, BOOLEAN
- **Fallunterscheidungen: bedingte Anweisungen**
- **Konzept der Rekursion**
  - rekursive Funktionen
  - indirekt rekursive Funktionen
  - Realisierung durch Laufzeitkeller

Glossar

- **Funktionale Programmierung**
- **Funktion, Funktionskopf, -rumpf**
- **Funktionalform**
- **Parameter**
  - Formal- , Aktual-
- **Seiteneffektfreiheit**
- **Rekursion**
  - direkt, indirekt
- **Datentyp**
  - einfacher, elementarer Typ
  - Ordinaltyp
- **Speicherverwaltung nach Kellerprinzip (Laufzeitkeller), statische Speicherverwaltung**
- **Aktivierungsblock**