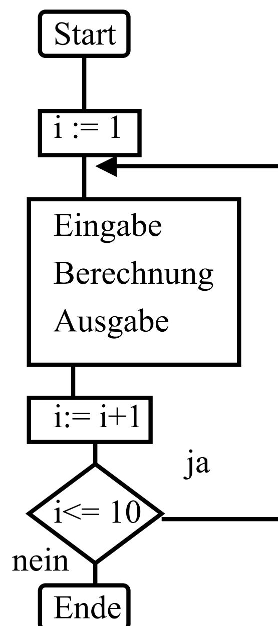
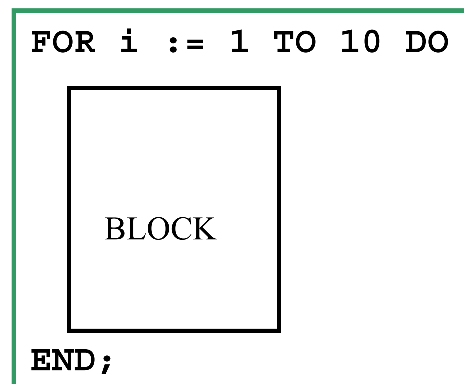
Taschenrechner Version 6.1

Wie kann ich ein Programmteil **10 mal** wiederholen ?



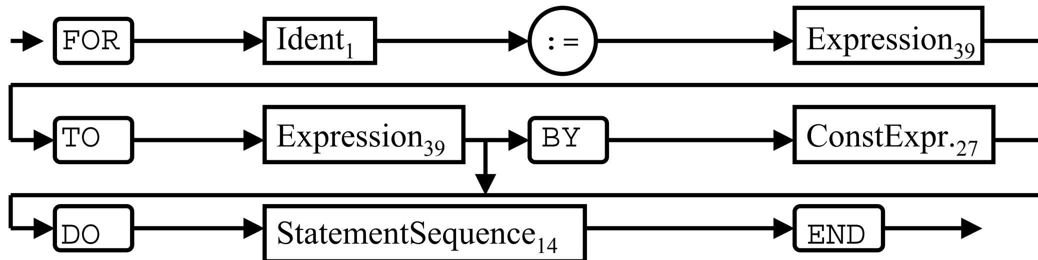


Taschenrechner Version 6.1

```
Module Taschenrechner;
(* Dies ist das Taschenrechnerprogramm Version 6.1 *)
FROM    InOut IMPORT (*PROC*) WriteInt, ReadInt, WriteString,
                              WriteLn, Read;
CONST   max = 10;  (* GENAU 10 Berechnungen *)
VAR     x,y,Erg,i : INTEGER;
        Op        : CHAR;
BEGIN
   WriteString ('Hallo, ich bin ein Taschenrechner.');WriteLn;
   FOR i := 1  TO max DO
        WriteString('Eingabe x:'); ReadInt(x);
        WriteString('Eingabe y:'); ReadInt(y);
        WriteString('Welche Operation (+,-,*) ?'); Read(Op);
        CASE Op OF
                '+' : Erg := x+y
          |     '-' : Erg := x-y
          |     '*' : Erg := x*y
         ELSE WriteString('Die Eingabe war nicht korrekt')
        END;
        WriteString('Ergebnis:'); WriteInt(Erg); WriteLn;
   END;
END Taschenrechner.
```

Schleifen

FOR-Anweisung



- Die Zahl der Durchläufe muss **vorher** schon feststehen.
- Die Laufvariable kann den Typ `INTEGER`, `CARDINAL`, `BOOLEAN`, `CHAR`, Enumeration oder Subrange haben.
- Die Ausdrücke hinter `:=` und `TO` müssen **ausdruckskompatibel** mit der Laufvariablen sein.
- Der Konstantenausdruck (die Schrittweite) muss vom Typ `INTEGER` oder `CARDINAL` (jeweils <> 0) sein.
  Bei Fehlen, wird Schrittweite 1 angenommen.

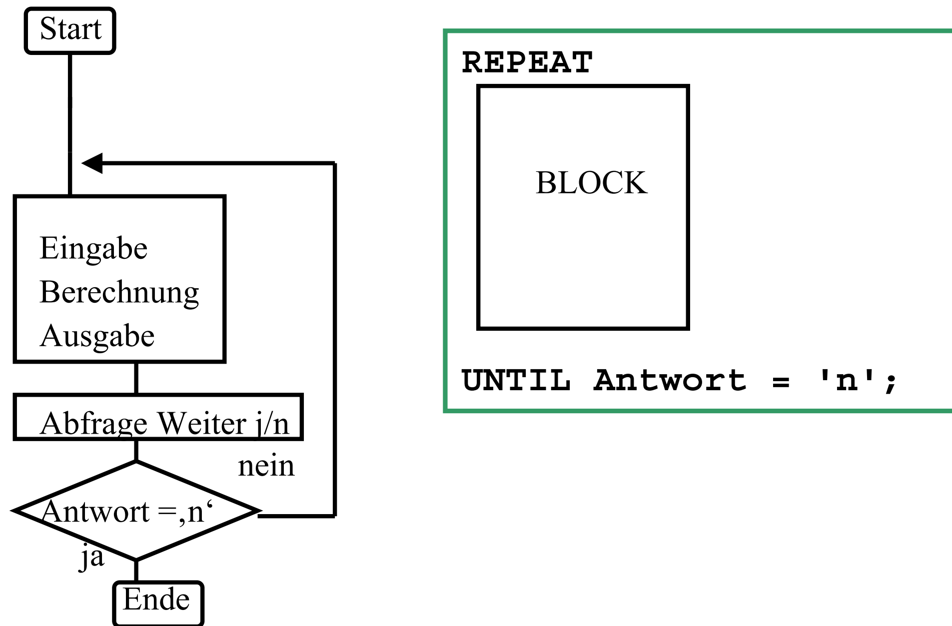
Schleifen

Ausführung einer FOR-Anweisung

- Der **Anfangswert** und der **Endwert** werden zu Begin berechnet (d.h. nur **einmal**!)
- Der Wert der Laufvariablen **sollte** im Schleifenrumpf **nicht** durch den Programmierer **verändert** werden!
- Der Wert der Laufvariablen ist nach Beendigung der Schleife **undefiniert.**
- **Problem:**
  Es ist nur eine **konstante** Anzahl von Aufrufen möglich.

Taschenrechner Version 6.2

Wie kann ich **beliebig oft** wiederholen?

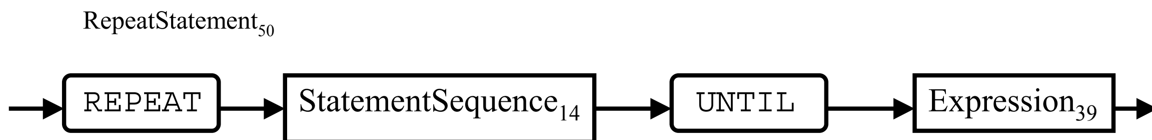


Taschenrechner Version 6.2

```
Module Taschenrechner; (* Taschenrechnerprogramm Version 6.2 *)
FROM     InOut IMPORT (*PROC*) WriteInt, ReadInt, WriteString,
                               WriteLn, Read;
VAR       x, y, Erg    : INTEGER;
         Op, Antwort : CHAR; (* Wiederholung (j)a oder (n)ein *)
BEGIN
   WriteString ('Hallo, ich bin ein Taschenrechner.'); WriteLn;
   REPEAT
     WriteString('Eingabe x:'); ReadInt(x);
     WriteString('Eingabe y:'); ReadInt(y);
     WriteString('Welche Operation (+,-,*) ?'); Read(Op);
     CASE Op OF
         '+' : Erg := x+y
       | '-' : Erg := x-y
       | '*' : Erg := x*y
      ELSE WriteString('Die Eingabe war nicht korrekt')
     END;
     WriteString ('Ergebnis:'); WriteInt(Erg);
     WriteString('Weiter (j)a oder (n)ein ?'); Read(Antwort);
   UNTIL Antwort = 'n'; END;
END Taschenrechner.
```

Schleifen

Repeat-Anweisung



(1) Der **Schleifenrumpf** einmal ausgeführt.

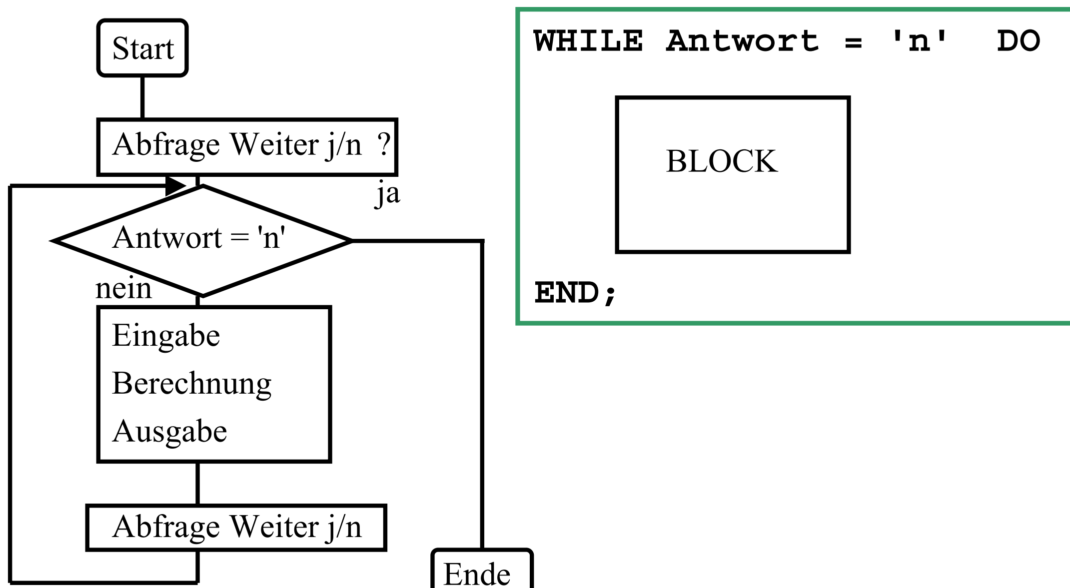
(2) Dann wird die **Schleifenbedingung** ausgewertet:

a) Falls **`TRUE`**, dann ist die **`REPEAT`**-Schleife beendet und die darauf folgende Anweisung wird ausgeführt

b) Falls **`FALSE`** gehe zurück zu (1)

Taschenrechner Version 6.3

Bis jetzt muss eine Berechnung ausgeführt werden. Wie kann man die Schleife so gestalten, dass man auch ohne Berechnung das Programm beenden kann?



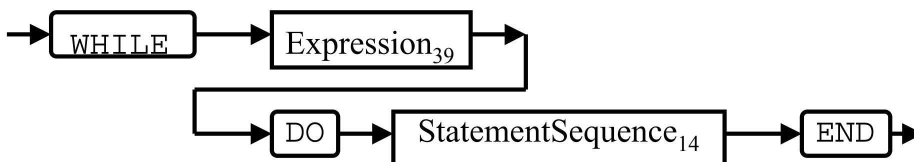
Taschenrechner Version 6.3

```
Module Taschenrechner;  (* Taschenrechnerprogramm Version 6.3 *)
FROM     InOut IMPORT (* PROC *) WriteInt, ReadInt, WriteString,
                                 WriteLn, Read;
VARx, y, Erg: INTEGER;
         Op, Antwort : CHAR; (* Wiederholung (j)a oder (n)ein *)
BEGIN
   WriteString('Hallo, ich bin ein Taschenrechner.'); WriteLn;
   WriteString('Weiter (j)a oder (n)ein ?'); Read(Antwort);
   WHILE Antwort = 'j' DO
         WriteString('Eingabe x:'); ReadInt(x);
         WriteString('Eingabe y:'); ReadInt(y);
         WriteString('Welche Operation (+,-,*) ?'); Read(Op);
         CASE Op OF
                  '+' : Erg := x+y
          |       '-' : Erg := x-y
          |       '*' : Erg := x*y
          ELSE WriteString('Die Eingabe war nicht korrekt')
         END;
         WriteString ('Ergebnis:'); WriteInt(Erg);
         WriteString('Weiter (j)a oder (n)ein ?'); WriteLn;
         Read(Antwort);
   END;
END Taschenrechner.
```


Schleifen

WHILE-Anweisung

WhileStatement$_{49}$



(1) Die **Schleifenbedingung** wird ausgewertet:

a) Falls **`TRUE`**, wird der **Schleifenrumpf** einmal ausgeführt und bei (1) die **`WHILE`**-Bedingung erneut abgeprüft.

b) Falls **`FALSE`,** ist die **`WHILE`**-Schleife beendet.

(2) Die darauf folgende Anweisung wird ausgeführt.

Problem: Doppelte Programmierung der Abbruchbedingung nötig

Taschenrechner Version 6.4

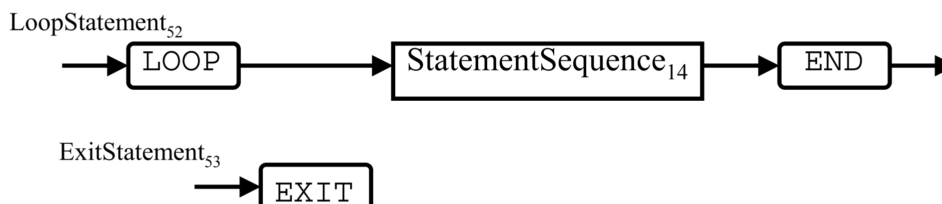
Wie kann ich eine Schleife an einer **beliebigen** Stelle verlassen?

```
Module Taschenrechner; (* Taschenrechnerprogramm Version 6.4 *)
FROM     InOut IMPORT (* PROC *) WriteInt, ReadInt, WriteString,
                                 WriteLn, Read;
VAR x, y, Erg : INTEGER;
         Op   : CHAR;
BEGIN
   WriteString ('Hallo, ich bin ein Taschenrechner.'); WriteLn;
   WriteString('Weiter (j)a oder (n)ein ?'); Read (Antwort);
   LOOP
         WriteString('Eingabe x:'); ReadInt(x);
         WriteString('Eingabe y:'); ReadInt(y);
         WriteString('Welche Operation (+,-,*) ?'); Read(Op);
         CASE Op OF
                   '+' : Erg := x+y
          |        '-' : Erg := x-y
          |        '*' : Erg := x*y
          ELSE WriteString('Die Eingabe war nicht korrekt'); EXIT
         END;
         WriteString('Ergebnis:'); WriteInt(Erg);
         WriteString('Weiter (j)a oder (n)ein ?'); Read(Antwort);
         IF Antwort = 'n' THEN EXIT END
   END
END Taschenrechner.
```


Schleifen

LOOP-Anweisung mit EXIT-Anweisung

- allgemeinste und flexibelste Form der Schleife
- `WHILE`, `REPEAT`, `FOR` sind wegen ihrer größeren Sicherheit der `LOOP` vorzuziehen



- Der Rumpf wird solange ausgeführt, bis man auf die **`EXIT`**-Anweisung stößt.
- Die **`EXIT`**-Anweisung bewirkt das **sofortige** Verlassen der Schleife. Danach wird mit der darauf folgenden Anweisung fortgefahren.
- In einer **`LOOP`**–Schleife dürfen mehrere **`EXIT`**s vorkommen.

Prozeduren

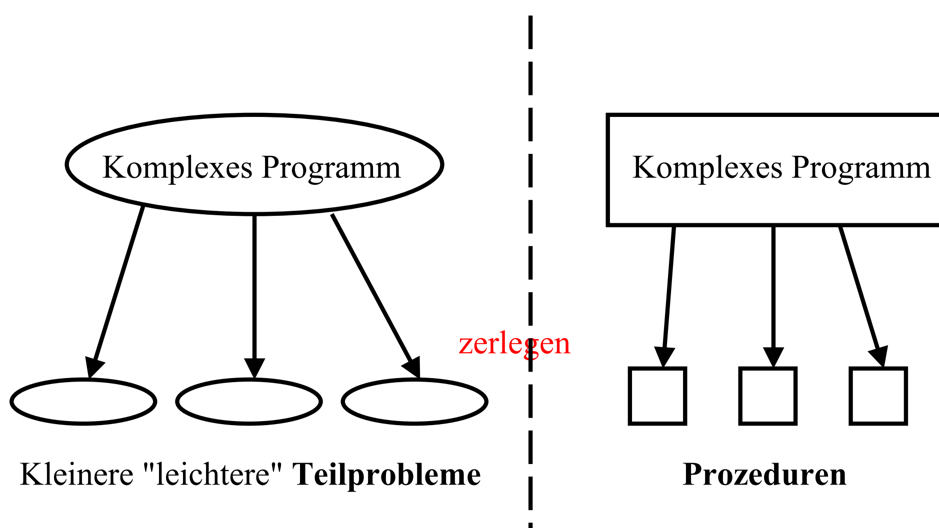
Wie kann man ein Programm in **Teilbereiche** strukturieren ?

Idee: Programmteile die **mehrmals** im Programm vorkommen, können als "**Prozeduren**" programmiert werden. Diese stellen **Unterprogramme** dar, die von dem Hauptprogramm oder anderen Prozeduren aufgerufen werden können.

Realisierung: Modularisierung durch Prozeduren

Prozeduren

Wesentlicher Vorteil:



Ein großes Programm (Problem) kann in kleinere Prozeduren (Teilprobleme) zerlegt werden.

Prozeduren

Beispiel:

Unser Taschenrechner soll mit einer Funktion zur **Potenzberechnung** erweitert werden.

→ X^Y für X,Y aus den natürlichen Zahlen

Prozeduren

Option 1: Direktes Einfügen der Potenzfunktion in unser bestehendes Taschenrechnerprogramm

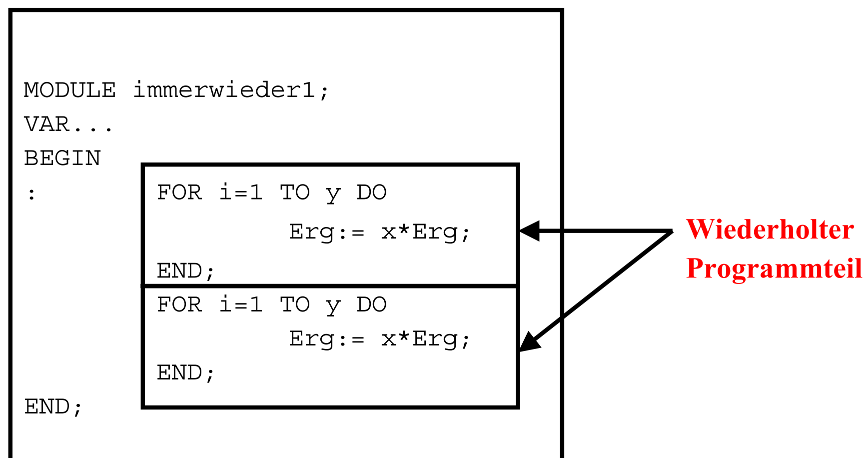
```
MODULE Taschenrechner; (* Programmfragment *)
VAR x,y,Erg,i : INTEGER;
:
BEGIN
:
   WriteString ('Welche Operation (+,-,*,^) ?'); Read(Op);
   CASE Op OF
        '+' : Erg := x+y;
     |  '-' : Erg := x-y;
     |  '*' : Erg := x*y;
     |  '^' : Erg := 1;
              For i := 1 TO y DO
                Erg := x * Erg;
              END
    ELSE WriteString('Die Eingabe war nicht korrekt');
   END;
End Taschenrechner.
```

Prozeduren

Nachteile des bisherigen Ansatzes:

1) Wiederverwendung in anderen Programmteilen nicht möglich !
(nur durch Duplizierung des entsprechenden Programmteils)

Beispiel: Wiederholte Berechnung von X^Y



Prozeduren

2) Schlechtere Lesbarkeit (bei wachsender Programmgröße)

- Wie finde ich den Teil, der XY implementiert ?
- Wie finde ich alle Programmzeilen, die für die Implementierung von XY benötigt werden ?
- Wo ist Erg definiert ?

Prozeduren

3) **Änderungen** bzw. **Modifikationen** im Programm werden erschwert

- potentielle Inkonsistenz wenn der Programmteil kopiert wurde !
- potentielle Fehlerquelle, da es schwer wird die "richtigen" Zeilen zu finden und zu ändern.

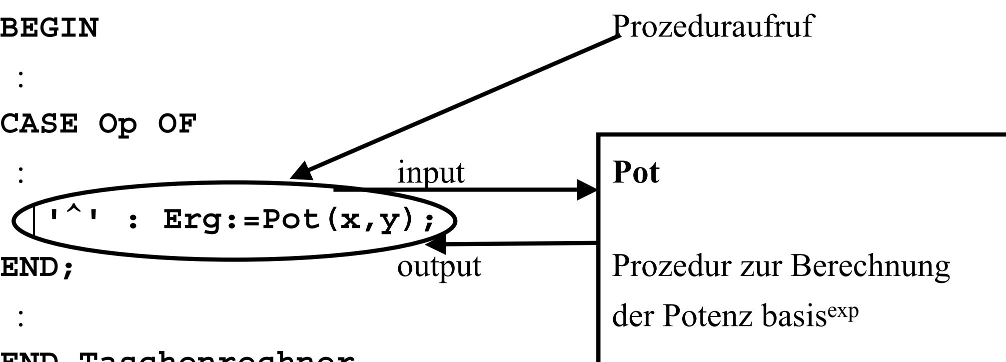
4) **Fehlersuche** (Debugging) wird erschwert

- Das Programm kann für gewöhnlich nur als Ganzes getestet werden, obwohl z.B. nur eine einzelne Teilfunktion geändert wurde.

Prozeduraufruf

Option 2: Aufruf eines entsprechenden **Unterprogramms** zur Berechnung der Potenzfunktion

```
MODULE Taschenrechner
:
   BEGIN
    :
   CASE Op OF
    :
     '^' : Erg:=Pot(x,y);
   END;
    :
   END Taschenrechner.
```

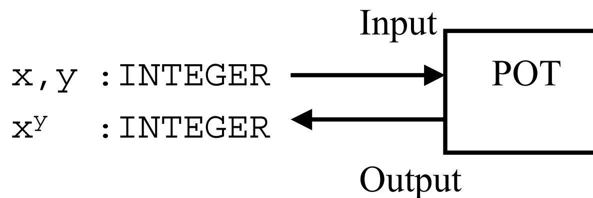


Das Konzept ist vergleichbar mit dem Einsatz **mathematischer Funktionen**.

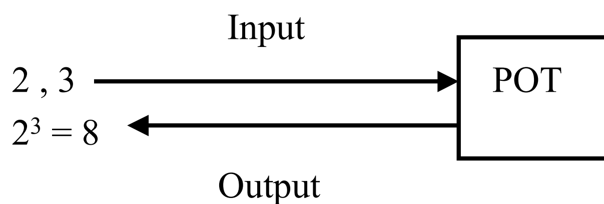
Prozeduraufruf

Ein- bzw. Ausgabeparameter von Prozeduren

Eine Prozedur benötigt 0 bis n **Eingabeparameter** (Input) und kann 0 bis 1 **Ausgabeparameter** (Output) liefern.



Beispiel:



Prozeduraufruf

Jeder Prozedur ist eine **Schnittstelle (Signatur)** zugeordnet.

Die Signatur enthält

- die **Anzahl** und den **Typ** der Parameter, die von der entsprechenden Prozedur benötigt werden, sowie
- den **Ausgabetyp** (falls vorhanden).

Typ-In_1, ...,Typ-In_n → **Typ-Out**

Beim Prozeduraufruf müssen die aktuellen Parameter in Anzahl und Typ immer zur Signatur der Prozedur passen.

Prozeduraufruf

Beispiel:

Die Signatur von **`Pot(basis,exp:INTEGER):INTEGER`** ist gegeben durch **`INTEGER,INTEGER→INTEGER`**.

Folgende **Fehler** sind an der Signatur von Pot erkennbar:

- falscher **Parametertyp:** `Erg := Pot('zwei','drei');`
- falsche **Anzahl** von Parametern: `Erg := Pot(1);`
- falscher **Typ** beim **"RETURN"** der Prozedur:

```
PROCEDURE Pot(basis,exp : INTEGER ): INTEGER;
 :
BEGIN
        :
        RETURN TRUE;
END Pot;
```


Prozeduraufruf

Überprüfung der Signatur

Der Compiler prüft bereits beim **Übersetzen des Programms** (**compile time**), ob alle Prozeduraufrufe mit der Signatur der Prozedur übereinstimmen:

➔ Liegt ein Fehler vor, so wird dieser vom Compiler gemeldet, und der Programmierer kann den Fehler beheben.

Vorteil:

Somit werden Fehler, die aufgrund von nicht kompatiblen Typen zur **Laufzeit** (**run time**) auftreten könnten, schon zur Übersetzungszeit abgefangen (static type checking).

Typisierung von Programmiersprachen (1)

1) Eine Sprache wird als **statisch typisiert** (statically typed) bezeichnet, wenn der Typ jedes Ausdrucks zur Übersetzungszeit bekannt ist.

   Beispiele: MODULA-2, PASCAL, C

2) Ansonsten ist sie **dynamisch typisiert** (dynamically typed)
   Beispiele: C++, andere objekt-orientierte Sprachen

Typisierung von Programmiersprachen (2)

3) Untypisiert (non-typed) heißen alle Sprachen, bei denen kein Typ zur Übersetzungszeit bekannt ist. In solchen Fällen kann keine Typüberprüfung durchgeführt werden.

   Beispiele: LISP, SMALLTALK

4) Eine Sprache ist **streng typisiert** (strongly typed), wenn zugesichert ist, dass alle Programme, die der Programmierer als korrekt ansieht, zur Laufzeit keine Typfehler erzeugen.
   Der Typ jedes Ausdrucks muss jedoch nicht bekannt sein (→ statisch oder dynamisch typisiert).
   Solche Programme heißen **typsicher** (type safe).

Typüberprüfung

Beispiel für eine typsichere Programmiersprache: MODULA-2

```
MODULE Typfehler;
 VAR  x: REAL;
      y: INTEGER;
 BEGIN
    x := 2.3;
    y := x; (* compile time TYPE ERROR *)
 END Typfehler.
```

Eine Zuweisung **INTEGER** → **REAL** ist z.B. in C/C++ möglich. In solchen Fällen wird hierbei intern eine **Typumwandlung** (type coercion) durchgeführt.

Typüberprüfung

Hinsichtlich der Typkompatibilität von INTEGER und REAL ergeben sich folgende mögliche bzw. nicht mögliche Zuweisungen:

Sprache	Zuweisung: Integer → Real	Zuweisung: Real → Integer
MODULA-2	nicht möglich	nicht möglich
PASCAL	nicht möglich	nicht möglich
C	möglich (interne Konvertierung)	möglich (automatisches TRUNC)
C++	möglich (interne Konvertierung)	möglich mit Warnung (automatisches TRUNC)
ADA	nicht möglich	nicht möglich
ALGOL	möglich (interne Konvertierung)	nicht möglich
FORTRAN	möglich (interne Konvertierung)	nicht möglich