

# Einführung in die Programmierung (PRG1x) & Elementare Algorithmen und Datenstrukturen (ADE1x)

**Heinz Dobler**

Version 2.2, 2020



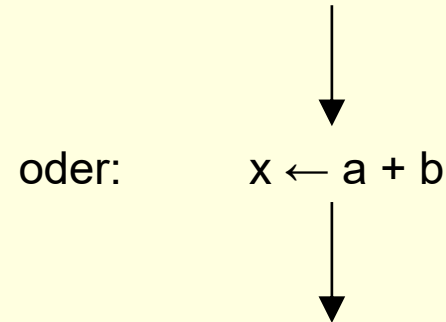
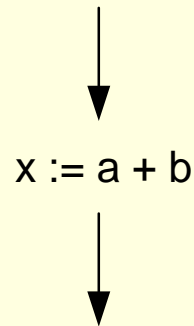
**Blaise Pascal**

\* 19. Juni 1623, † 19. August 1662

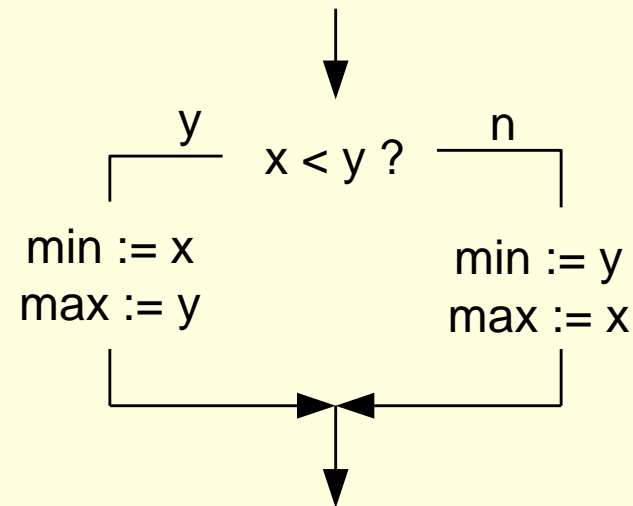
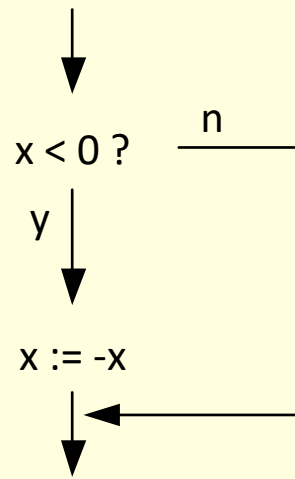
Bildquelle: [content.answers.com](https://content.answers.com)

- Darstellungsformen (inkl. Beispiel)
- Erstes Pascal-Programm
- Programmiersprache Pascal
- Gültigkeitsbereiche
- Ausdrücke
- Standardfunktionen
- Benutzerdefiniertes *Heap*-Management
- Einfügen eines Elements in eine ...

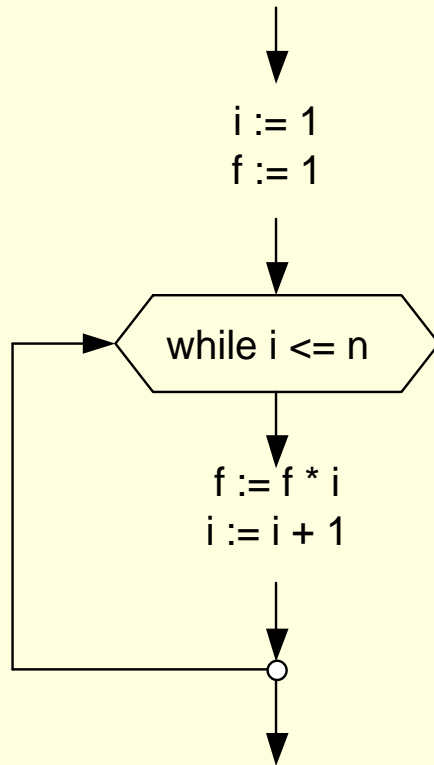
- Zuweisungsanweisung (Bsp.: Summe)



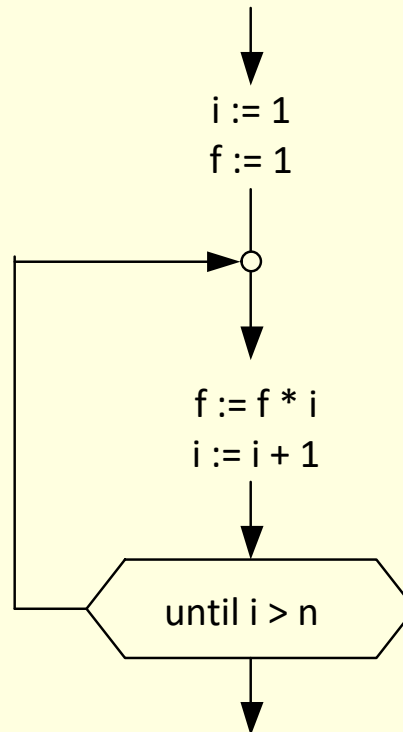
- Binäre (einseitig u. zweis.) Verzweigung (Bsp.: Absolutbetr. u. Min./Max.)



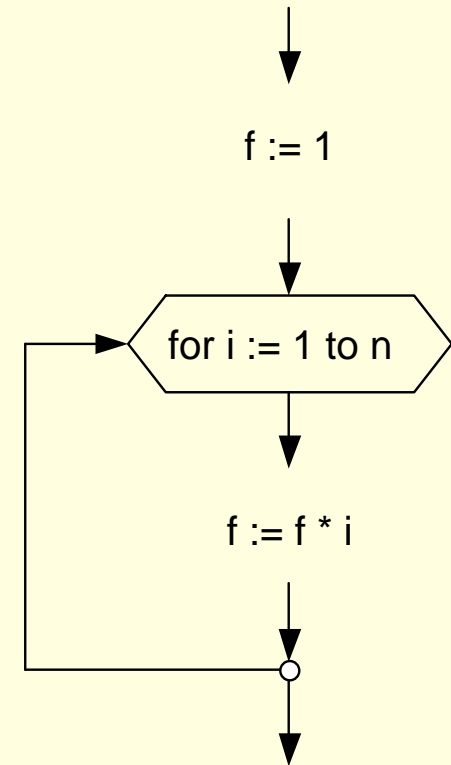
## ■ Schleifen (Bsp.: Fakultätsberechnung)



Abweisschleife  
(*while*-Schleife)

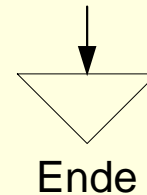


Durchlaufschleife  
(*repeat-until*-Schleife)

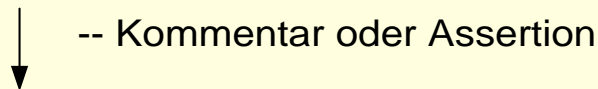


Zählschleife  
(*for*-Schleife)

- Anfang und Ende eines Algorithmus

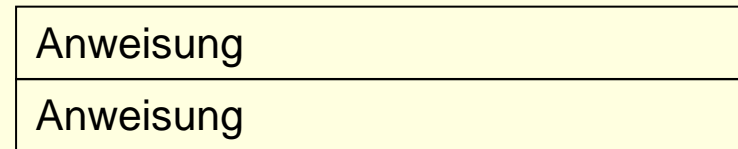


- Kommentar oder Assertion (Zusicherung)

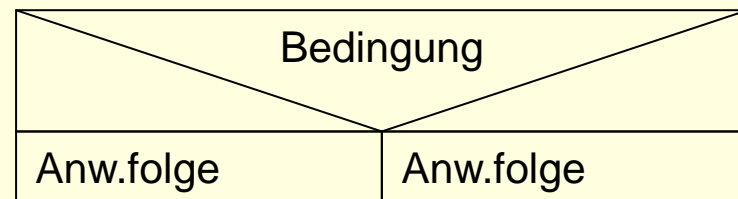


... auch Nassi-Shneidermann-Diagramm genannt

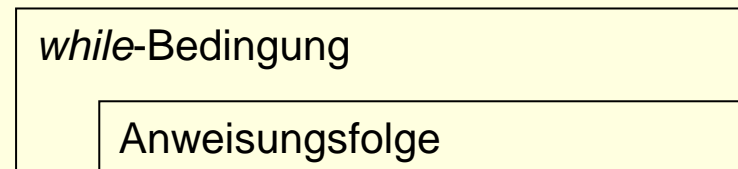
- Anweisungsfolge



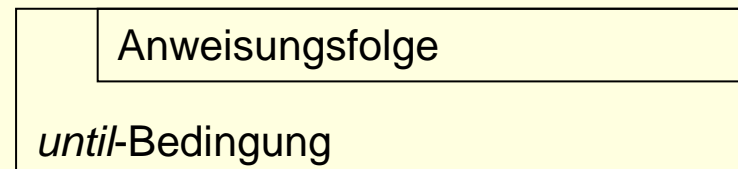
- Binäre Verzweigung



- Abweisschleife  
(*while*-Schleife)



- Durchlaufschleife  
(*repeat-until*-Schleife)



## Aufgabenstellung für Mittelwert-Berechnung

- *Gegeben:* Folge von ganzen Zahlen größer 0
- *Gesucht:* Arithmetisches Mittel
- *Beispiel:* 17, 4, 21, 0  $\Rightarrow$  14 ( $= 42 / 3$ )

## Algorithmus *Mean*

### Schritt 1: Initialisierung

Setze die beiden Variablen *total* und *numbers* auf 0.

### Schritt 2: Einlesen

Lies einen Wert für die Variable *value* ein.

### Schritt 3: Abbruch?

Wenn *value* = 0 gehe zu Schritt 5,  
sonst mache bei Schritt 4 weiter.

### Schritt 4: Berechnung und Schleife

Erhöhe den Wert der Variable *total* um *value* und  
erhöhe den Wert der Variable *numbers* um 1.  
Gehe zu Schritt 2.

### Schritt 5: Durchschnittsberechnung und Ausgabe

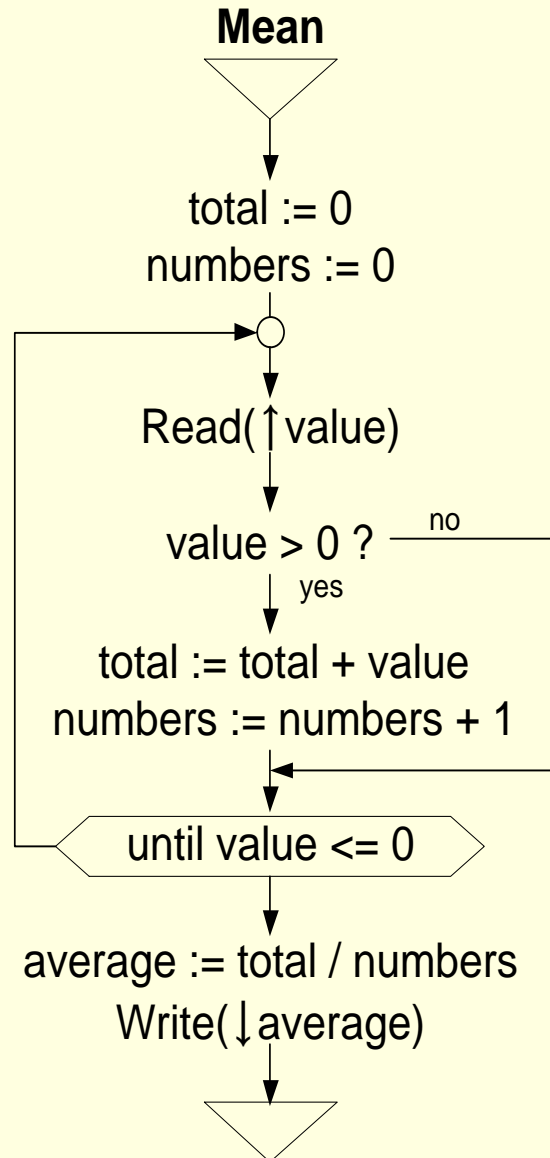
Wenn *total* > 0 dann:

Berechne den Durchschnitt in der Variable *average*  
durch Division von *total* durch *numbers*.  
Gib den Durchschnittswert in *average* aus.

Sonst:

Gib 0 aus.





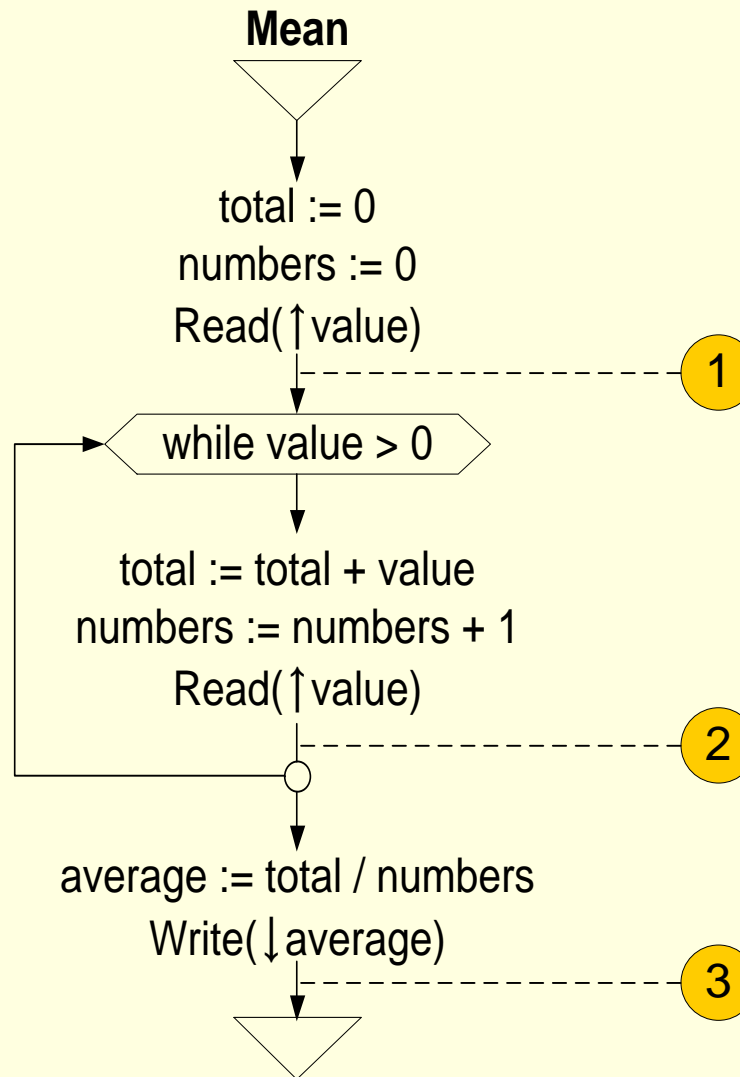
```
Mean()  
begin  
  total := 0  
  numbers := 0  
  repeat  
    Read(↑value)  
    if value > 0 then  
      total := total + value  
      numbers := numbers + 1  
    end -- if  
  until value = 0  
  if numbers > 0 then  
    average := total / numbers  
    write(↓average)  
  else  
    write(↓0)  
  end -- if  
end Mean
```

```
(* Mean:                                     HDO, 2007-09-07
-----
  Computation of mean value by reading a series
  of integer values
=====*)
PROGRAM Mean;
  VAR
    total, numbers, value: INTEGER;
BEGIN
  total := 0;
  numbers := 0;
  REPEAT
    Read(value);
    IF value > 0 THEN BEGIN
      total := total + value;
      numbers := numbers + 1
    END (*IF*)
  UNTIL value <= 0;
  IF numbers > 0 THEN
    write(total / numbers)
  ELSE
    write('no values')
  END. (*Mean*)
```

Kommentar zum  
Programm

Deklarationsteil

Programmrumpf  
(Anweisungsteil)



Durchschnittsberechnung für folgende Eingabe (*values*):

1, 9, 6, 8, 0

Stelle	<i>value</i>	<i>total</i>	<i>numbers</i>	<i>average</i>
1	1	0	0	?
2	9	1	1	?
2	6	10	2	?
2	8	16	3	?
2	0	24	4	?
3	0	24	4	6

# Erstes Pascal-Programm

```
(* MeanPgm:                                HDO, 2007-09-07 *)
(* -----                                *)
(* Calculation of mean for sequence of numbers. *)
(*=====*)
PROGRAM MeanPgm;
  VAR
    total, numbers, value: INTEGER;
    mean: REAL;
BEGIN

  WriteLn('MeanPgm: Calculation of Mean');
  WriteLn;
  total := 0;
  numbers := 0;
  write('value > ');
  ReadLn(value);
  WHILE value > 0 DO BEGIN
    total := total + value;
    numbers := numbers + 1
    ReadLn(value);
  END; (*WHILE*)
  WriteLn;
  IF numbers > 0 THEN BEGIN
    mean := total / numbers;
    WriteLn('mean = ', mean:5:2); (* 5 chars, 2 decimals *)
  ELSE
    WriteLn('Zero numbers, no mean')

END. (*MeanPgm*)
```

Kommentar zum  
Programm

Deklarationsteil

Programmrumpf  
(Anweisungsteil)

## ■ Symbole mit besonderer Bedeutung

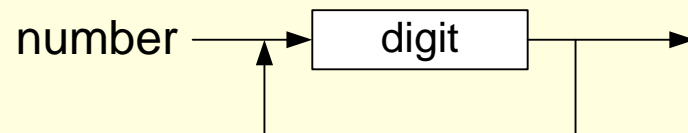
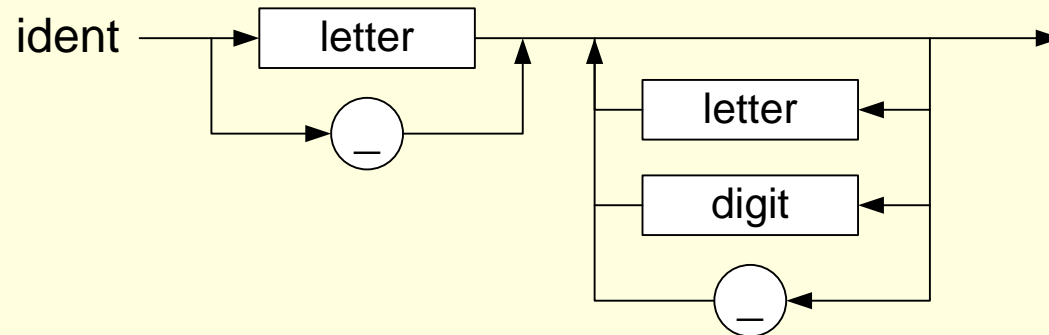
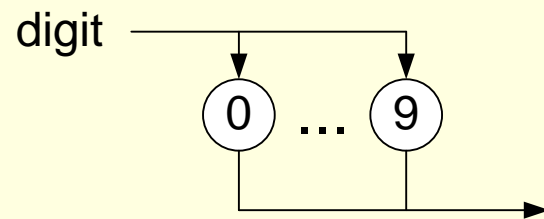
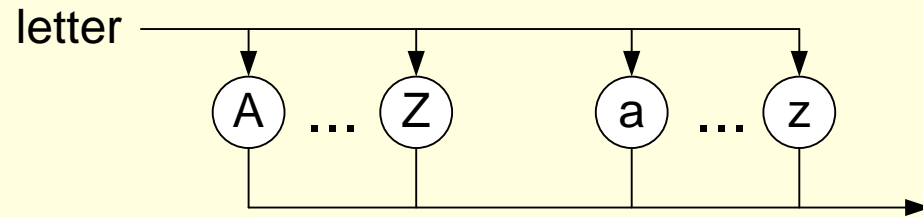
+ - \* / = < > [ ] . , ( ) : ; ^ @ \$ # { }

## ■ Verbundsymbole

<> <= >= := .. (\* \*)

## ■ Schlüsselwörter (insges. 51)

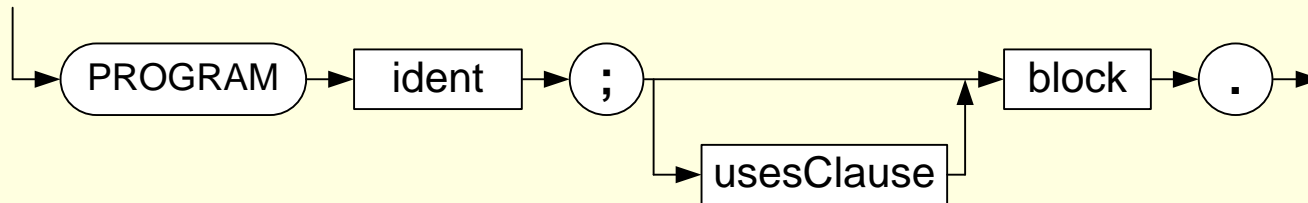
AND	ELSE	INLINE	PROCEDURE	UNIT
ARRAY	END	INTERFACE	PROGRAM	UNTIL
ASM	EXPORTS	LABEL	RECORD	USES
BEGIN	FILE	LIBRARY	REPEAT	VAR
CASE	FOR	MOD	SET	WHILE
CONST	FUNCTION	NIL	SHL	WITH
CONSTRUCTOR	GOTO	NOT	SHR	XOR
DESTRUCTOR	IF	OBJECT	STRING	
DIV	IMPLEMENTATION	OF	THEN	
DO	IN	OR	TO	
DOWNTO	INHERITED	PACKED	TYPE	





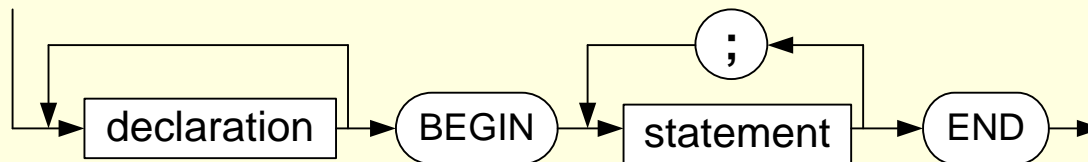
# Pascal: Syntaxdiagramme (1)

## Pascal Program



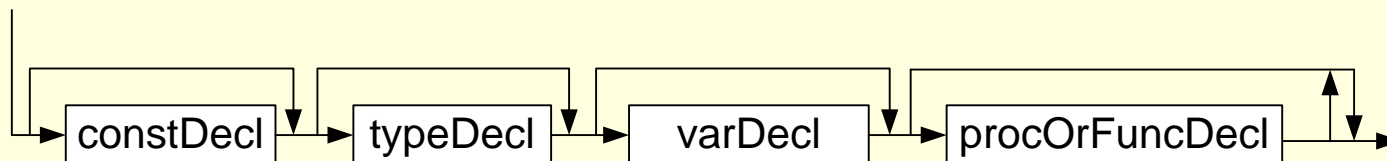
## block

Semikolon trennt Statements



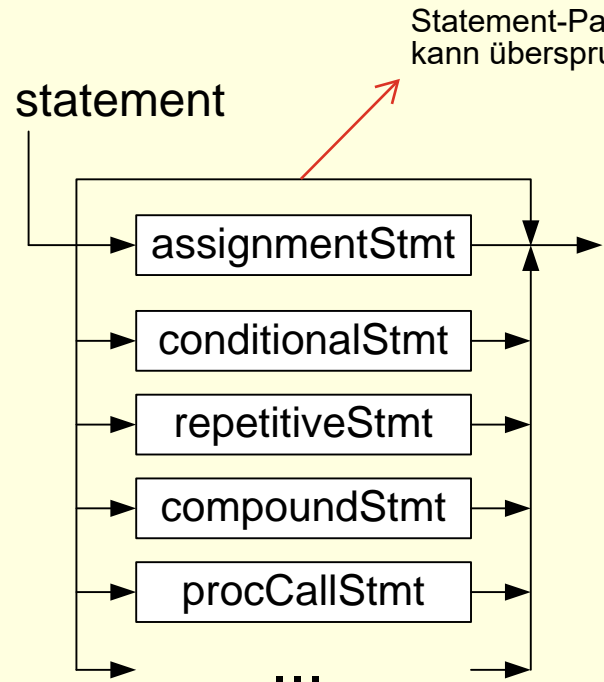
--> bei nur einem Statement braucht man keinen ;

## declaration wird formal benötigt, kann aber übersprungen werden

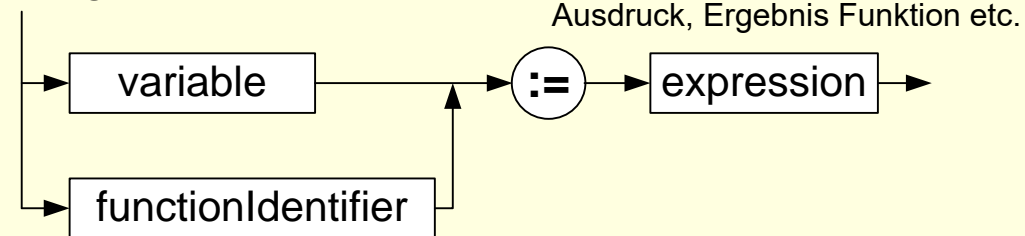


Diese Reihenfolge ist empfohlen, die Grammatik lässt aber eine beliebige Reihenfolge zu.

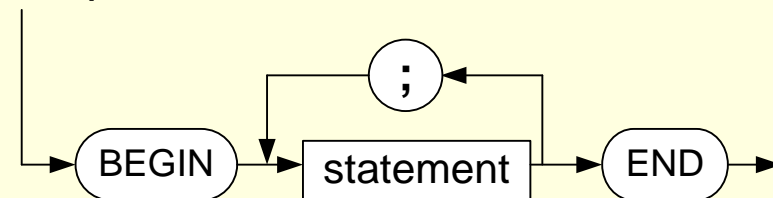
# Pascal: Syntaxdiagramme (2)



## assignmentStmt



## compoundStmt



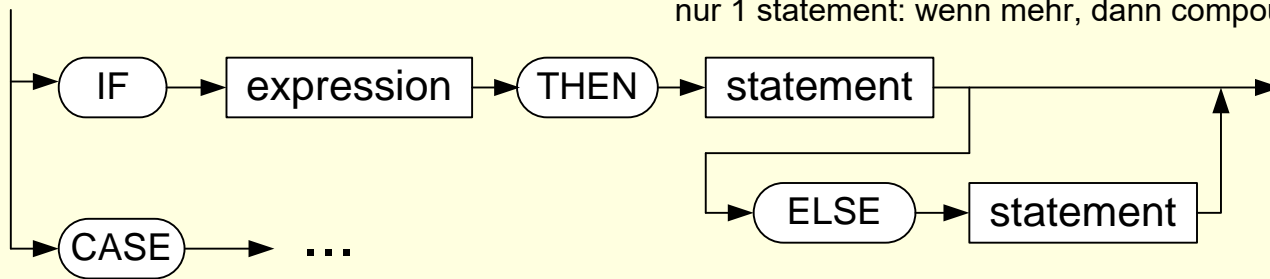
--> Es folgt:  
kürzest mögliches Pascal-Diagramm:

```
PROGRAM _;  
BEGIN  
END.
```

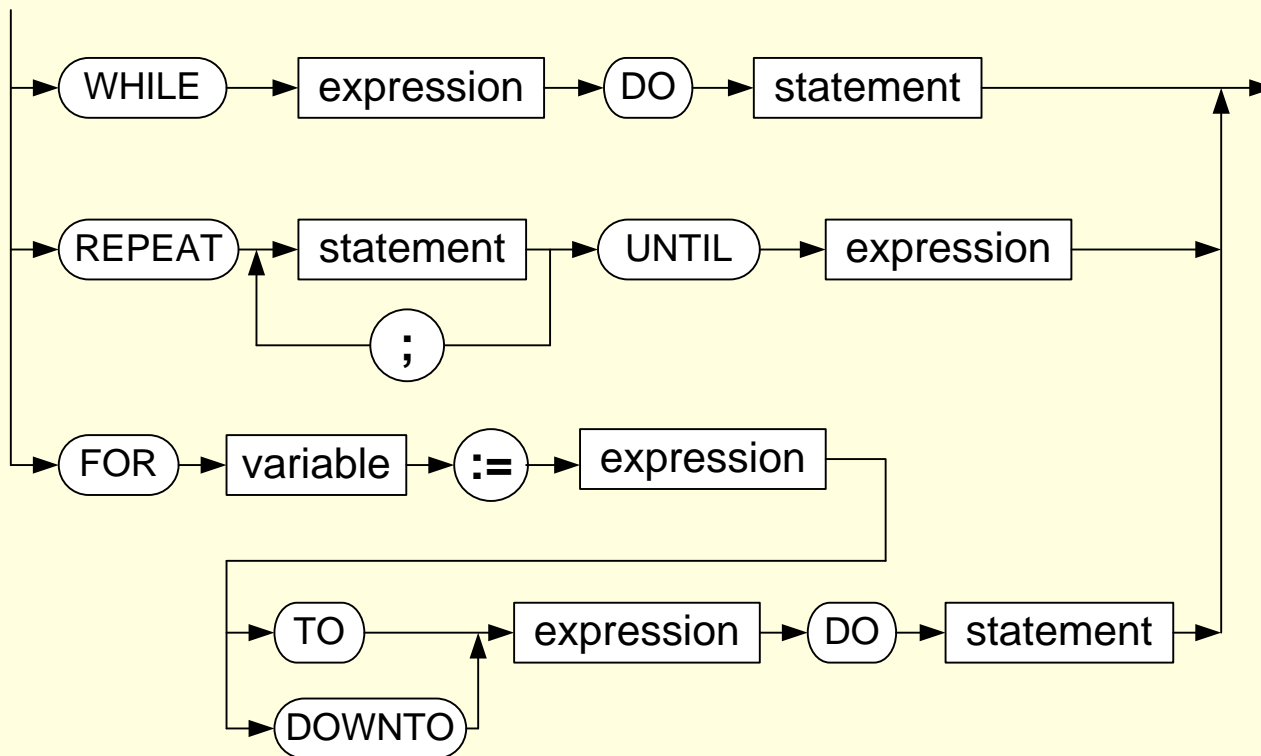
# Pascal: Syntaxdiagramme (3)

## conditionalStmt

nur 1 statement: wenn mehr, dann compound statement

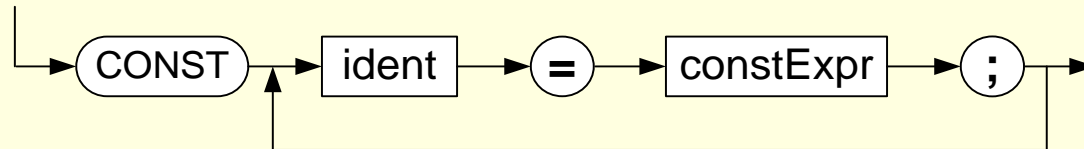


## repetitiveStmt

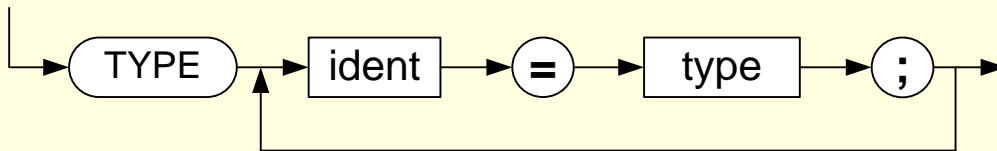


# Pascal: Syntaxdiagramme (4)

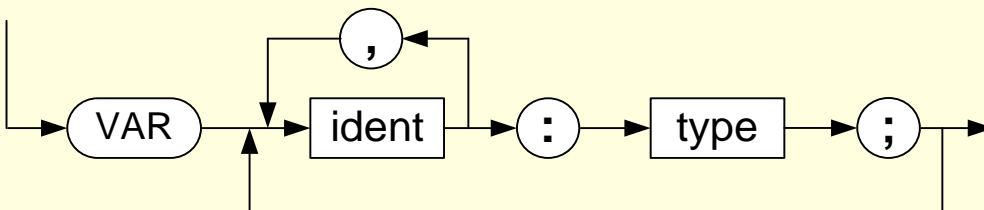
constDecl



typeDecl

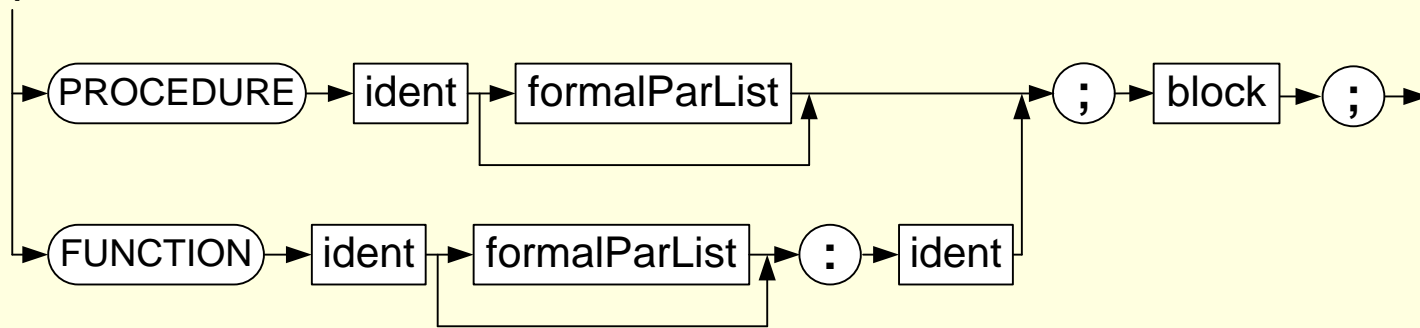


varDecl

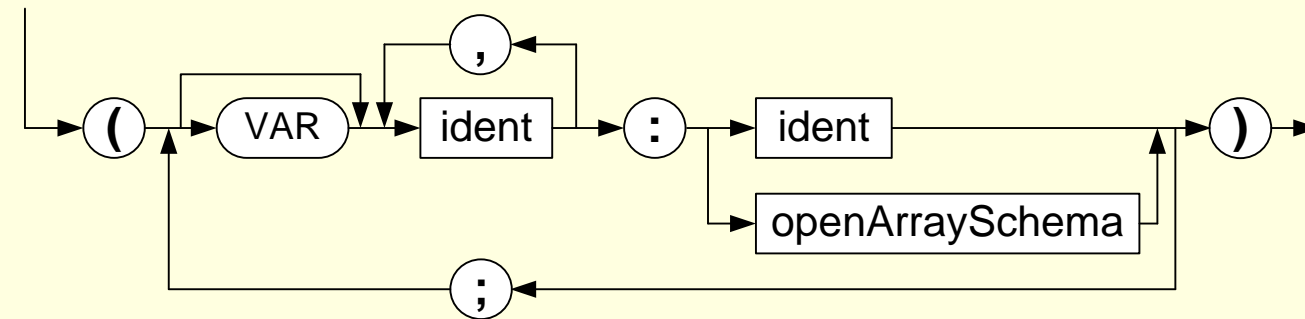


# Pascal: Syntaxdiagramme (5)

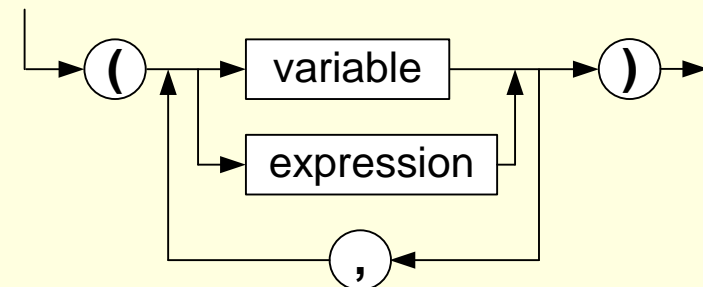
## procOrFuncDecl

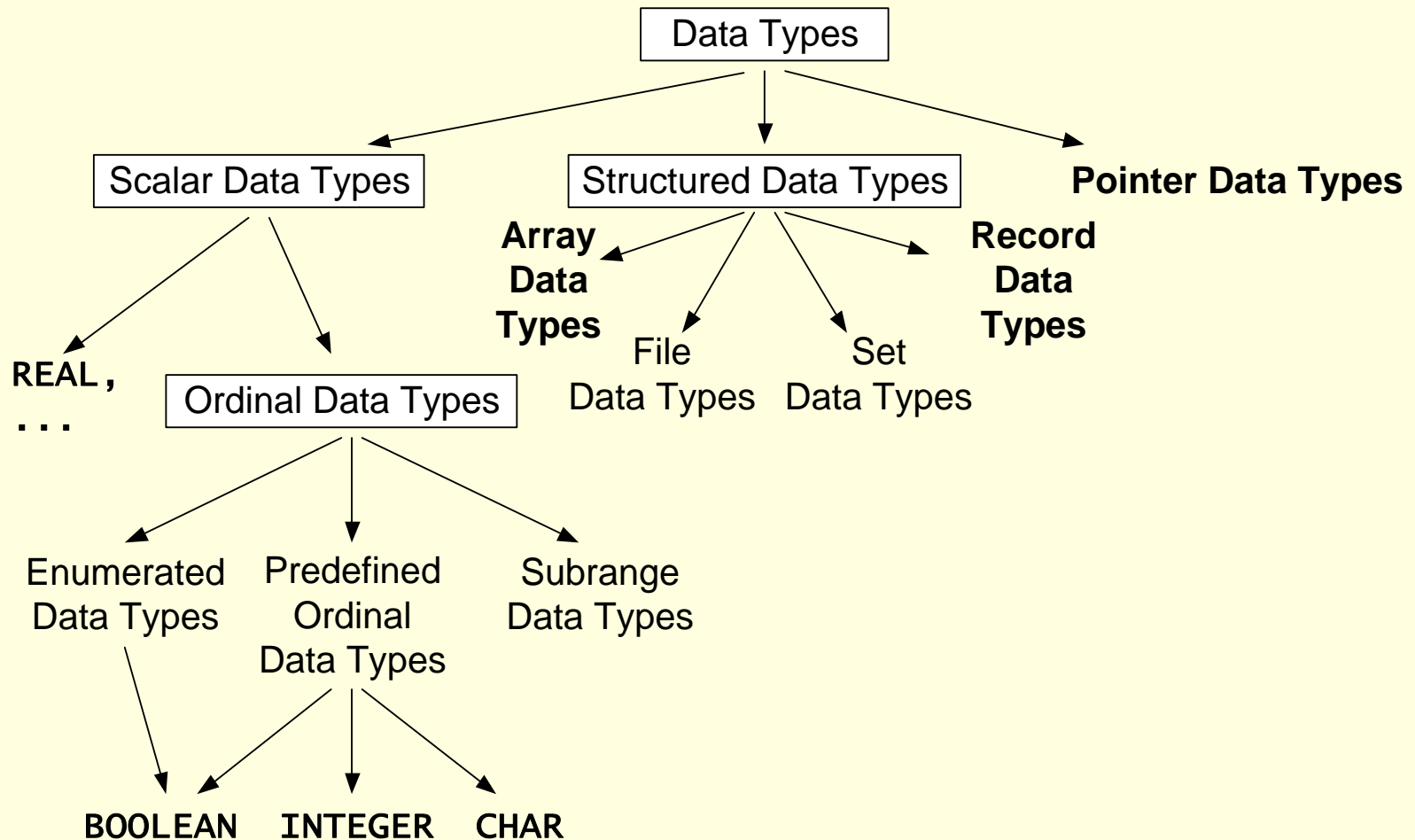


## formalParList

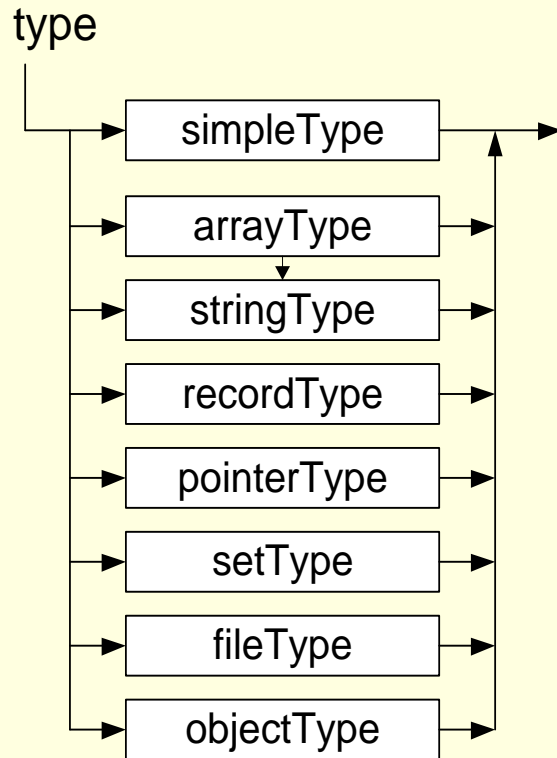


## actualParList



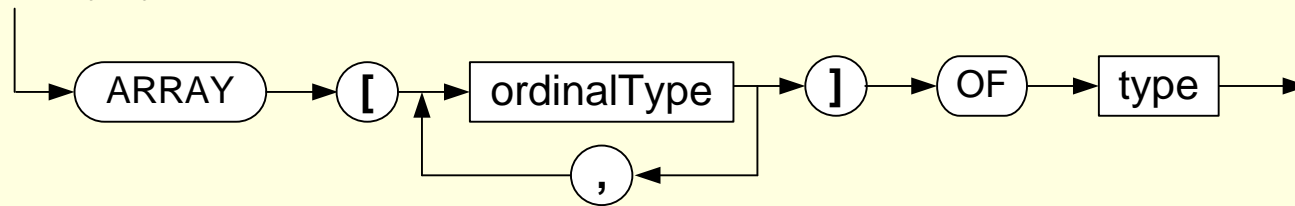


# Datentypen in Pascal (2)

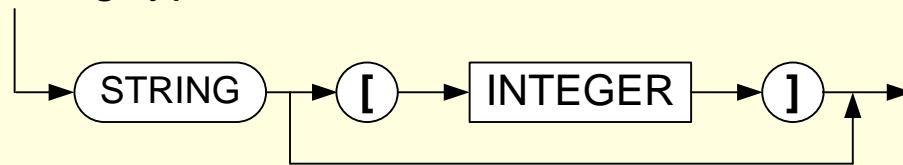


# Datentypen in Pascal (3)

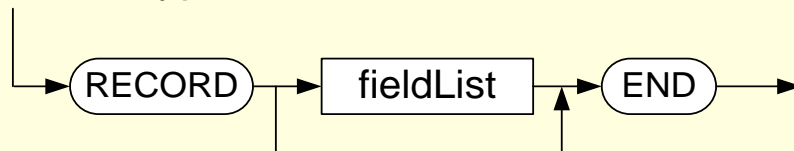
arrayType



stringType



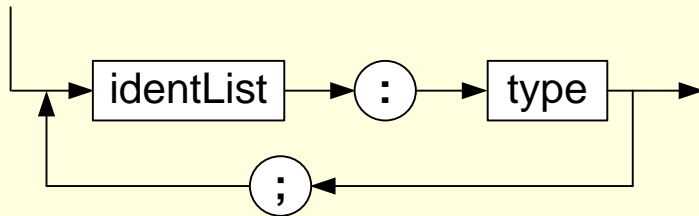
recordType



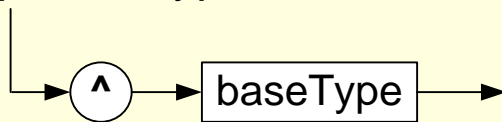


# Datentypen in Pascal (4)

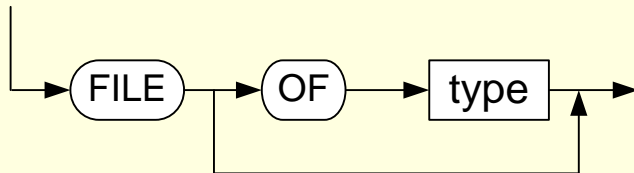
fieldList

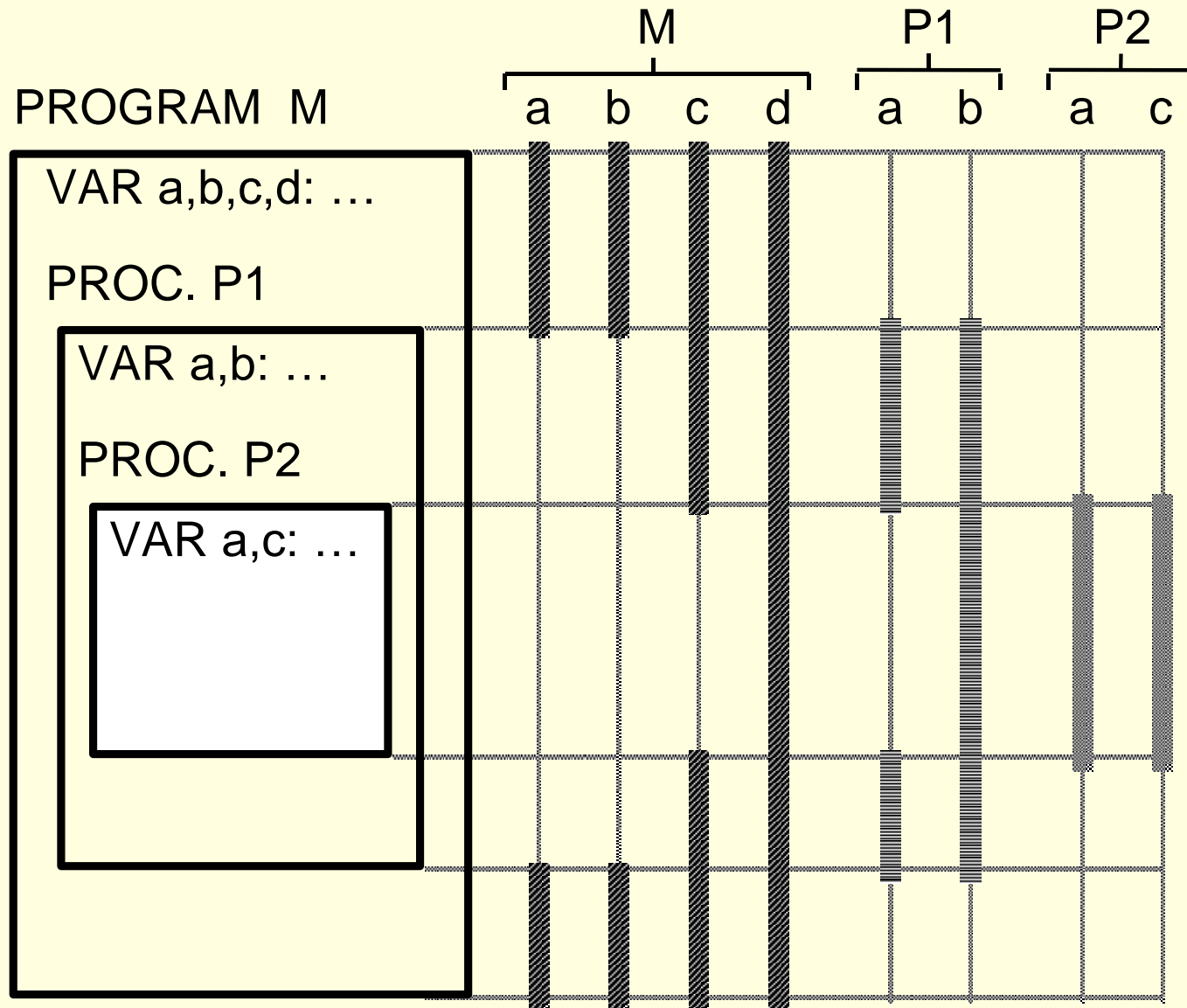


pointerType



fileType





## ■ Grammatik

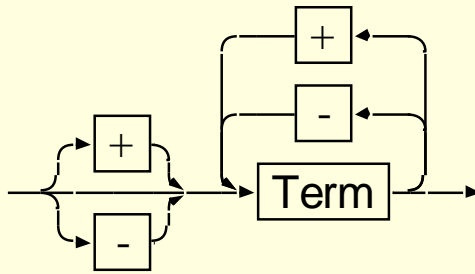
SimpleExpr = [ „+“ | „-“ ] Term { ( „+“ | „-“ ) Term } .

Term = Fact { ( „\*“ | „/“ ) Fact } .

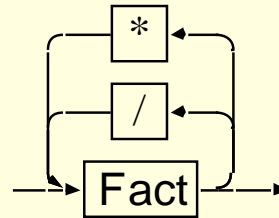
Fact = c | v | „(“ SimpleExpr „)“ .

## ■ Syntaxdiagramme

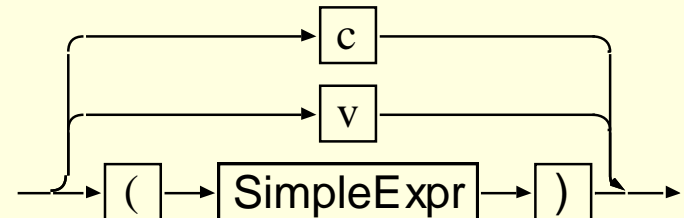
SimpleExpr



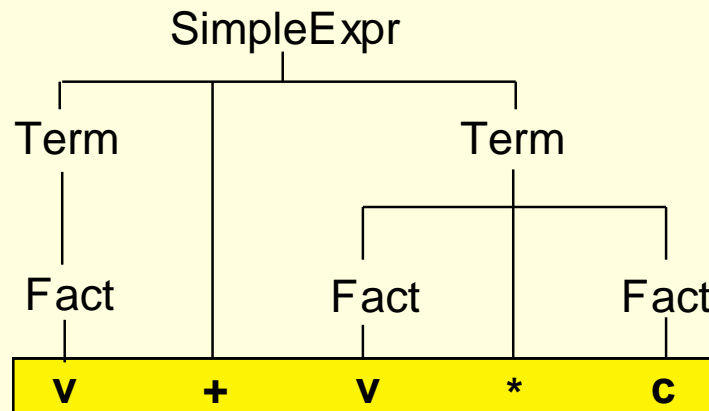
Term



Fact



## ■ Syntaxbaum



Operator	Operation	Operanden- typen	Ergebnistyp
+	Addition	Integer	Integer
		Real	Real
-	Subtraktion	Integer	Integer
		Real	Real
*	Multiplikation	Integer	Integer
		Real	Real
/	Division	Integer	Real
		Real	Real
DIV	ganzzahlige Division	Integer	Integer
MOD	Restbild.	Integer	Integer

## Wichtige Regeln

- Wenn beide Operanden bei +, −, \*, DIV, MOD von einem Integer-Typ sind, dann ist das Ergebnis vom „gemeinsamen“ (*common*) Integer-Typ.
- Wenn einer oder beide Operanden bei +, − oder \* von einem Real-Typ sind, dann ist das Ergebnis ebenfalls vom Typ REAL.
- Der Typ von  $x / y$  ist immer REAL.

## ■ Erweiterung um logische Operatoren

SimpleExpr = [ „+“ | „-“ ] Term  
                  { ( „+“ | „-“ | **OR** | **XOR** ) Term } .  
Term = Fact { ( „\*“ | „/“ | **AND** ) Fact } .  
Fact = c | v | „(“ SimpleExpr „)“ | **NOT** Fact.

- *Vorrang:*  
NOT vor AND vor OR, XOR
- *Beispiel:*  
 $a \text{ AND NOT } b \text{ OR } c = (a \text{ AND } (\text{NOT } b)) \text{ OR } (c)$
- *Tipp:* Immer Klammerung verwenden

## ■ Erweiterung um relationale Operatoren

Expr = SimpleExpr [ ( „<“ | „<=“ | „>“ | „>=“ | „=“ | „<>“ )  
                                SimpleExpr ] .

## ■ Konvertierungsfunktionen

- *Ord* liefert Ordinalzahl für Wert eines ord. Typs  
Ord(x): LONGINT für x mit einem ordinalen Datentyp
- *Chr* liefert Zeichen mit entsprechender Ordinalzahl  
Chr(x: BYTE): CHAR
- *High* liefert größten Wert im Argumentbereich  
(bzw. größten Index eines Open Arrays)  
High(x): Datentyp von x oder Indexdatentyp von x
- *Low* liefert kleinsten Wert im Argumentbereich  
Low(x): Datentyp von x oder Indexdatentyp von x
- *Round* rundet Real-Wert nach LONGINT  
Round(x: REAL): LONGINT
- *Trunc* schneidet Real-Wert ab und liefert LONGINT  
Trunc(x: REAL): LONGINT

## ■ Arithmetische Funktionen (1)

- *Abs* liefert den Absolutbetrag des Arguments  
 $Abs(x)$ : Datentyp von  $x$
- *Int* liefert Ganzzahlteil des Arguments  
 $Int(x: REAL): REAL$
- *Frac* liefert Nachkommateil des Arguments  
 $Frac(x: REAL): REAL$
- *Exp* liefert  $e$  hoch  $x$  ( $e^x$ )  
 $Exp(x: REAL): REAL$
- *Ln* natürlicher Logarithmus  
 $Ln(x: REAL): REAL$
- *Pi* Zahl  $\pi$   
 $Pi: REAL$
- *Sqr* Quadrat des Arguments  
 $Sqr(x)$ : Datentyp von  $x$
- *Sqrt* Quadrat**wurzel** des Arguments  
 $Sqrt(x: REAL): REAL$

- Arithmetische Funktionen (2)
  - *Sin, Cos* Sinus- und Cosinus des Arguments  
 $\text{Sin}(x: \text{REAL}): \text{REAL}$   
 $\text{Cos}(x: \text{REAL}): \text{REAL}$
  - *ArcTan* Arcus Tangens des Arguments  
 $\text{ArcTan}(x: \text{REAL}): \text{REAL}$
- Ordinale Funktionen
  - *Pred* liefert Vorgänger des Arguments  
 $\text{Pred}(x): \text{Datentyp von } x$
  - *Succ* liefert Nachfolger des Arguments  
 $\text{Succ}(x): \text{Datentyp von } x$
  - *Odd* ist das Argument ungerade?  
 $\text{Odd}(x: \text{LONGINT}): \text{BOOLEAN}$



- Zeichenketten-Funktionen (1)
  - *Concat* liefert Konkatenation von Zeichenketten  
(analog zum Operator +)  
Concat(s1[, s2, ... sn]: STRING): STRING
  - *Copy* liefert Teilkette einer Zeichenkette  
Copy(s: STRING; index: INTEGER;  
count: INTEGER): STRING
  - *Length* liefert aktuelle Länge einer Zeichenkette  
Length(s: STRING): INTEGER
  - *Pos* liefert Startposition einer Teilkette in einer Zeichenkette  
Pos(substr: STRING; s: STRING): BYTE
  - *Str* liefert Zeichenkette für numerischen Wert,  
z. B. *Str*(123) = '123'  
Str(x[:width[:decimals]]); VAR s: STRING)

- Zeichenketten-Funktionen (2)
  - *Val* liefert numerischen Wert einer Zeichenkette, z. B. *Val*('123') = 123  
*Val*(s: STRING; VAR v: INTEGER oder REAL;  
VAR code: INTEGER)
- Sonstige Funktionen
  - *UpCase* liefert Großbuchstaben zum Kleinbuchstaben  
*UpCase*(ch: CHAR): CHAR
  - *Random* liefert eine Zufallszahl  
*Random* [(range: WORD)]: WORD, wenn range verwendet wird, sonst REAL

# Benutzerdef. Heap-Management in Borland Pascal

```
PROGRAM HeapMan;
CONST
  KB = 1024;
  memResSize = 64*KB - 8;
VAR
  memRes: POINTER;

FUNCTION HeapFunc(size: WORD): INTEGER; FAR;
BEGIN
  IF size > 0 THEN BEGIN
    IF memRes = NIL THEN BEGIN
      (*HeapFunc := 0;*)      (*size > 0 => runtime error*)
      HeapFunc := 1;          (*causes NIL value*)
    END (*THEN*)
    ELSE BEGIN (*there is a reserved block of memory*)
      FreeMem(memRes, memResSize);
      memRes := NIL;
      HeapFunc := 2;          (*try allocation again*)
    END; (*ELSE*)
  END (*THEN*)
  ELSE BEGIN (*size = 0*)
    HeapFunc := 0;            (*size = 0 => OK*)
  END; (*ELSE*)
END; (*HeapFunc*)

BEGIN (*main program*)
  heapError := @HeapFunc;
  GetMem(memRes, memResSize);
  ...
  New(p);
  IF p = NIL THEN BEGIN
    WriteLn('*** heap overflow');
    HALT;
  END; (*IF*)
END. (*HeapMan*)
```

# Benutzerdef. Heap-Management in Free Pascal

```
UNIT HeapMan;
INTERFACE

  PROCEDURE GetMem(VAR p: POINTER; size: LONGINT);

IMPLEMENTATION

  VAR
    memResSize: LONGINT;
    memRes: POINTER;

  PROCEDURE GetMem(VAR p: POINTER; size: LONGINT);
  BEGIN
    System.GetMem(p, size);
    IF p = NIL THEN BEGIN
      IF memRes <> NIL THEN BEGIN
        System.FreeMem(memRes, memResSize); memRes := NIL;
        WriteLn('ATTENTION: memory reserve freed');
        GetMem(p, size); (*let's try it again*)
      END (*THEN*)
      ELSE BEGIN (*memRes = NIL*)
        WriteLn('ERROR 203: heap overflow'); Halt;
      END; (*ELSE*)
    END; (*IF*)
  END; (*GetMem*)

  BEGIN (*HeapMan*)
    ReturnNilIfGrowHeapFails := TRUE;
    memResSize := 1024;
    memResSize := 1024 * memResSize;
    System.GetMem(memRes, memResSize);
    IF memRes <> NIL THEN
      WriteLn('INFO: memory reserve allocated, size = ', memResSize)
    ELSE BEGIN
      WriteLn('ERROR: memory reserve can not be allocated'); Halt;
    END; (*IF*)
  END. (*HeapMan*)
```

```
PROGRAM HMTest;

  USES
    HeapMan;

  VAR
    ip: ^INTEGER;

  BEGIN (*HMTest*)
    WriteLn('Test for Unit HeapMan');
    WHILE TRUE DO BEGIN
      GetMem(ip, SizeOf(INTEGER));
    END; (*WHILE*)
  END. (*HMTest*)
```

# Einfügen eines Elements in eine ...

## ... doppelt-verkettet Liste

```
PROCEDURE Insert(VAR l: List; n: NodePtr);
  VAR
    succ: NodePtr; (*successor of new node n*)
  BEGIN
    Assert(Sorted(l), 'before Insert: list not sorted');
    IF l.first = NIL THEN BEGIN (*l.last = NIL, too*)
      l.first := n;
      l.last := n;
    END (*THEN*)
  ELSE BEGIN
    succ := l.first;
    WHILE (succ <> NIL) AND (n^.val > succ^.val) D.B.
      succ := succ^.next;
    END; (*WHILE*)
    IF succ = l.first THEN BEGIN (*prepend n*)
      n^.next := l.first;
      l.first^.prev := n;
      l.first := n;
    END (*THEN*)
    ELSE IF succ = NIL THEN BEGIN (*append n*)
      n^.prev := l.last;
      l.last^.next := n;
      l.last := n;
    END (*ELSE*)
    ELSE BEGIN (*insert n in the middle, before succ*)
      n^.prev := succ^.prev;
      n^.next := succ;
      succ^.prev^.next := n;
      succ^.prev := n;
    END; (*ELSE*)
  END; (*ELSE*)
  Assert(Sorted(l), 'after Insert: list not sorted');
END; (*Insert*)
```

# Einfügen eines Elements in eine ...

## ... doppelt-verkettet Liste

```
PROCEDURE Insert(VAR l: List
VAR
    succ: NodePtr; (*succe
BEGIN
    Assert(Sorted(l), 'befor
    IF l.first = NIL THEN BE
        l.first := n;
        l.last := n;
    END (*THEN*)
    ELSE BEGIN
        succ := l.first;
        WHILE (succ <> NIL)
            succ := succ^.next
        END; (*WHILE*)
        IF succ = l.first TH
            n^.next := l.fir
            l.first^.prev :=
            l.first := n;
        END (*THEN*)
        ELSE IF succ = NIL T
            n^.prev := l.las
            l.last^.next :=
            l.last := n;
        END (*ELSE*)
        ELSE BEGIN (*insert
            n^.prev := succ^
            n^.next := succ;
            succ^.prev^.next
            succ^.prev := n;
        END; (*ELSE*)
    END; (*ELSE*)
    Assert(Sorted(l), 'after
END; (*Insert*)
```

## ... doppelt-verkettet Liste mit Anker

```
PROCEDURE Insert(l: ListPtr; n: NodePtr);
VAR
    succ: NodePtr; (*successor of new node n*)
BEGIN
    Assert(Sorted(l), 'before Insert: list not sorted');

    succ := l^.next;
    WHILE (succ <> l) AND (n^.val > succ^.val) D. B.
        succ := succ^.next;
    END; (*WHILE*)

    n^.prev := succ^.prev;
    n^.next := succ;
    succ^.prev^.next := n;
    succ^.prev := n;

    Assert(Sorted(l), 'after Insert: list not sorted');
END; (*Insert*)
```