4.6 Bemerkungen zur UP-Technik

• Grundidee bisher: Algorithmus → Programm

– Pr. selbst ist Objekt der Verarbeitung (hat darum einen Namen)
in ≥ 2 Stufen: Compiler übersetzt, danach Abarbeitung;
– das Objekt „Pr." kommuniziert auch mit seiner Umgebung:
mind. gibt es input und output;

• Frage: Ist dies verallgemeinerbar ?
In folg. Richtung ⇒
komplexer Algor. ⟶ Teilalgorithmen geringerer Komplexität:
a) jeder wieder ein Pr.-objekt
b) Gesamtprogramm aus ihnen zus.-gesetzt

⇒ Forderung: 1. Die Teile müssen definierbar sein.
2. Es muß Regeln für ihr Zus.-setzen geben.

• Realisierungsstufen:

1. Stufe: einfachste → Standardfunktionen und Standardprozeduren
einer PS (vordef. Pr.-objekte);
muß nur lernen:
a) welche gibt es;
b) was tun sie;
c) wie anwenden (aufrufen);

2. Stufe: selbstdefinierbare Pr.-objekte;
Klar: dazu entspr. Mechanismus der PS benötigt!

Def. 4.5: Ein Teilalgorithmus, der in einem Pr. definierbar und wiederholt anwendbar ist, heißt **Unterprogramm**.

→ Bem.: Programme werden mittels UP strukturiert.

∃ 2 Arten von UP:

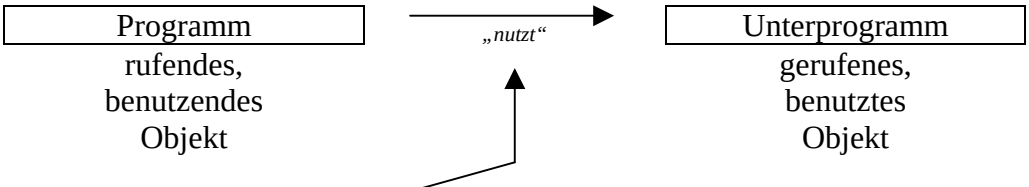
(1) geschlossene UP → Routinen;
(2) offene UP → Makros;

gemein haben beide 2 Ebenen:
Ebene der Def. der UP und Ebene der Benutzung der UP;
Def. der UP:
a) Teilalgor. als Zusf. von Anweisungen und Datenobjekten (⇒ benannte Objekte)
b) Regeln zur Kommunikation mit Umgebung (⇒ formale Parameter)

Benutzung der UP: hier gibt es den Unterschied;
Benutzung = Aufruf des UP (als Anweisung);
bei (1) wird Anweis.-folge des UP zur Laufzeit vollständig abgearbeitet;
bei (2) wird Anweis.-folge des UP zur Übersetzungszeit an entspr. Stelle kopiert; *{führt zur Übersetzungszeit zur Verlängerung des Pr.; Kommunikation mit Umgebung hier schon aufgelöst}*

Bem.: Makros vor allem bei Assembler;
imperative Sprachen nur geschlossene UP-Technik;
ADA - beide;

- Nutzung der Teilalgor. erfordert klare Schnittstellenbeschreibung zw. rufendem und gerufenem Pr.:



Interface (Schnittstelle) dient zur Beschreibung der Ordn.-beziehung in dieser Hierarchie;
in Protokoll legt Schnittstelle die Art und Weise der Benutzerbeziehngen fest;

⇒ Schnittstellenbeschreibung:
Def. des Objektes „UP“;
Nutzung des „UP“;

Routinen (in versch. PS) → Prozeduren, Funktionen

Bem.: Unterscheiden sich in Def. und Nutzung!

am Bsp. Pascal →

A) Prozeduren

1. Def. von Prozeduren:

PROCEDURE *prozname* (*parameterliste*);

vereinbarungsteil; {für lokale Objekte}

anweisungsteil; {Prozedurkörper; enthält algorithm. Elemente}

Schnittstelle: ist der Teil | *prozname* (*parameterliste*); |

– Objektbezeichner; frei wählbar;
– Parameterliste
- definierte Anzahl;
- definierte Reihenfolge;
- definierte Typen;
- definierte Art des Parameteraustausches zw. rufendem und gerufenem Objekt;

Parameter:
- für flexible Anpassung eines UP an versch. Einsatzfälle;
- Prozedur kann Eingabe- und Ausgabegrößen benötigen;
- sind in Parameterliste in Art und Anzahl anzugeben;
- heißen formale Parameter;
- werden im HP nicht vereinbart;
- formal = zur Beschreibung als formale Größen im Anweisungsteil der Prozedur; „Platzhalter“ für einzusetzende Werte, wenn Prozedur ausgeführt wird;

in Mathem. →
$f(x, a, b, c) = ax^2 + bx + c \Rightarrow x, a, b, c$ – formale Parameter;
$f(1, 2, 3, 4) = 2 \cdot 1^2 + 3 \cdot 1 + 4 = 9$

⇒ formale Parameter benötigt, weil zum Zeitpunkt des Aufschreibens des Algor. noch unbekannt auf welche Eingabe- und Ausgabewerte er angewandt werden soll;

2. Nutzung von Prozeduren: (im rufenden Progr.)

durch (Prozedur-) Anweisung: *prozname* (*aktuelle parameterliste*);

akt. Parameter:
ersetzen bei Aufruf die formalen Parameter;
mit ihren akt. Werten wird die Prozedur ausgeführt;
⇒ müssen im Vereinbarungsteil des rufenden Programms def. sein;
⇒ müssen zur Zeit des Aufrufes einen Wert besitzen;
Beziehung zw. akt. und form. Param.:
- gleiche Anzahl; *{strenge Forderung}*
- übereinstimmende Reihenfolge; *{wegen der Semantik der Param.}*
- übereinstimmende Typen;
- Sicherung der Regeln des Parameteraustausches; *{später}*

Bsp.:

```
PROCEDURE  maximum (a, b : real; VAR max : real);
BEGIN
        IF a > b THEN  max := a  ELSE max := b
END;
```

nun HP (ruf. Pr.), wo *x, t* zwei dort def. Var.;
wäre folg. Aufruf möglich:

```
maximum (19.3*2.7, sqr(x), t );
```

!! *x, t* als real-Var. vereinbart !!

3. Regeln für den Parameteraustausch:

- Parameter sind für die Kommunikation zw. rufendem und gerufenem Objekt; analog der Kommunik. zw. Pr. und Umwelt;

  rufend → gerufen: Input-Information
  gerufen → rufend: Output-Information
  dazu noch: InOut-Parameter;
  } Für Prozedur

- mit Proz.-aufruf ist ein Mechanismus verbunden, wie die aktuellen Werte an die Prozedur übergeben werden:
  z.B. die Prozedur *maximum* ⇒
  die Par. *a, b, max* im Kopf der Prozedur heißen formale Par.;
  die Par. im Proz.-aufruf *19.3*2.7*, sqr(*x*), *t* dagegen aktuelle Par.;

- gewöhnlich (auch in Pascal) 2 Möglichkeiten der Wertübergabe:

(i) Wertsubstitution (call by value)

! Muß klar sein, welcher Wert →
- Bestimmung des Wertes des entspr. Parameters;
- Mitteilung des Wertes an die Prozedur;

! Nur bei Input-Information!
! In Pascal: für alle Par. ohne VAR-Spezifikation!
Bsp. Prozedur *maximum* → *a, b,* Wertparameter;

– Ablauf/Mechanismus:
- Prozedur richtet „Hilfs-“ Variable mit (*name, wert, typ*) als Platzhalter ein;
- dabei *name* nur der Prozedur bekannt;
- zu verarbeitende Wert bei Aufruf auf entspr. Speicherzelle, so daß er den formalen Parameter ersetzen kann;
  ! Rufende Pr. kennt Verwendung des Wertes nicht!
  ! Wert kann auch aus Ausdruck erzeugt werden!

– Bez.:
- Art des Aufrufes: Wertaufruf (call by value);
- formal Param.: Wertparameter;

– Bsp.: *maximum* →
wenn Wert der Var. *x* gleich 7, so bewirkt der Aufruf
maximum (*19.3*2.7*, sqr(*x*), *t*);
folgendes:
- im UP wird Speicherplatz entspr. des Parametertyps geschaffen;
- Werte der akt. Par. berechnet →
  19.3 * 2.7 liefert 52.11; sqr(*x*) liefert 49;
  und den entspr. Speicherplätzen zugeordnet;
- damit wird UP ausgeführt;
- die im HP def. reelle Var. *t* erhält Output-Inf. von *max* = 52.11 zugewiesen;

(ii) Variablensubstitution (call by reference)

! Alle in Liste der formal. Par. mit VAR gekennzeichneten Parameter sind Variablenparameter.
! Entspr. akt. Par. muß Variable des verlangten Typs sein.
! Parameter für Output-Inf. müssen notw. Variablenparameter sein.
Bsp. Prozedur *maximum* → *max* ist Variablenparameter;

– Ablauf/Mechanismus/Wirkung:
- zum Zeitpunkt des UP-Aufrufs werden die Adressen (-nummern), d.h. der Speicherplatz derjenigen akt. Param., die den mit VAR gekennzeichneten formalen Param. entsprechen, an das UP übergeben;
{Speicherplatz für *t* wird *max* zugeordnet}
- werden im UP diese Param. verarbeitet, wird der entspr. Speicherplatz (des ruf. Pr.) (durch geruf. Pr.) referiert; und die dort befindliche Inf. verändert;
→ Referenzaufruf (call by reference)
- nach Ende des UP stehen die Par.-werte für weitere Arbeit im ruf. Pr. zur Verfügung;

! Rückgabe bzw. Output an ruf. Pr. nur vorgetäuscht!

(iii) Namenssubstitution (call by name)

– analog Makro-Technik: textuelle Substitution, wenn Par. eine Fkt. ist; falls Par. eine Var., so wie Variablensubst.;
– in Pascal nicht;

4. Zusammenfassung und Bemerkungen:

- akt. Param. müssen im HP, formale Param. brauchen nicht vereinbart zu werden;
- Obj., die nur Eingabegrößen für UP → Wertsubstitution ⇒ ohne VAR;
→ „Schreibverbot“;
- Obj., die nur Ausgabegrößen für UP → Variablensubstitution ⇒ mit VAR;
→ „Leseverbot“;
- Typ-Namen für formale Param. bzw. Datentyp der akt. Param.:
  – einfache (CHAR, BYTE, BOOLEAN, INTEGER, REAL);
  – STRING (= STRING[255]);
  – strukturierte (ARRAY, RECORD, STRING[*x*]) müssen global mit Typdeklaration umbenannt werden;
- Par. gleichen Typs und gleicher Übergabeart durch Komma, sonst durch Semikolon getrennt;
- Prozedur im Vereinbarungsteil des HP vereinbart, Aufruf im Anweisungsteil;
- Prozedur wie HP in mehreren Schritten entwickeln;
- READ(-LN), WRITE(-LN) sind Standard-Prozeduren;

B) Funktionen

- oft eine Prozedur zu benutzen, die
  a) genau 1 Wert produziert, der
  b) innerhalb eines Ausdruckes aufzurufen ist;

in Mathem. heißt derartige Prozedur „Funktion“;
die Eingabegrößen sind die Argumente;

1. Def. von Funktionen: →Funktionsvereinbarung

FUNCTION *funktionsname* (*parameterliste*) : ***typname***;

vereinbarungsteil; *Schnittstelle* *Typbezeichn.*
anweisungsteil; ⟶ *Funktionskörper*

- Parameter: nur Input-Parameter;
- Funktion liefert Wert vom Typ *typname*;
- zw. *funktionsname* und *typname* existiert folg. Beziehung:
  Im Anweisungsteil muß *funktionsname* ein Wert zugewiesen werden, d. h. mindestens 1x dort *funktionsname* := *ausdruck*;

⇒ Interpretation des *funktionsname* wie eine Variable.
{Proz.-name kommt innerhalb einer Prozedur nicht vor.}

2. Nutzung von Funktionen: →Funktionsaufruf

- rufen in einem Ausdruck durch
  funktionsname (*akt. parameterliste*);
  ! Parameterlisten können auch leer sein!

- Fkt. nur für nichtstrukturierte Datentypen, d. h.
  typname kann einfacher Typ, STRING oder Zeigertyp sein;

Bem.: Andere Auffass. → auch strukt. DT als Ergebnis.

- Zusf. der Unterschiede zur Prozedur:
  (1) Fkt.-namen mind. 1x im Anweisungsteil ein Wert zugewiesen.
  (2) Keine strukt. Datentypen.
  (3) Fkt. innerhalb eines Ausdrucks aufrufen.

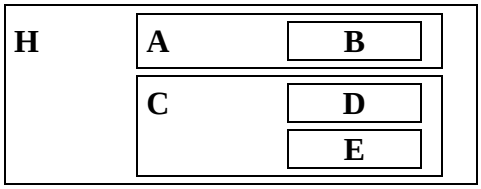
3. Funktionen mit Seiteneffekten: folg. Bsp.

```
PROGRAM  bsp;
    VAR a,b : integer;
    FUNCTION    d ( x : integer) : integer;
        BEGIN
            d := a + x;
            a := a + 1;          → Seiteneffekt !
        END;                        !!! Schlechter Progr.-stil !

BEGIN {HP}
    a := 1;
    b := d (1)  + d (a);
    WRITELN(Output, b);
    a := 1;                      unterschiedliche Erg.
    b := d (a)  + d (1);            { 6 bzw. 5 }
    WRITELN(Output, b);
END.
```

C) Gültigkeitsbereiche von Objekten

- prozedurale Abstraktion → hierarch. Strukturierung der Programme
  ⇒ entspr. Gültigkeitsbereich der Namen (Objekte);

- z.B.:



- Grundprinzip: Jeder vereinbarte Name gilt nur für den Block, in dem die Vereinbarung steht.

man sagt: Der Name ist für diesen Block lokal.
Namen, die in umfass. Block def. sind, heißen global.

- allg.:
  ① Hierarchiebez. zw. den Blöcken – Über-, Unterord.-, Gleichstellung;
  ② im übergeordneten Block vereinb. Objekte gelten in allen untergeordneten; sie sind globale Obj.; alle Standardnamen, z. B. alle „oben“ def. Obj. sind „unten“ aufrufbar, also auch UP;
  ③ im untergeordn. Block vereinbarte Objekte sind dem übergeordneten verborgen → lokale Objekte;
  ④ Regelung im Kollisionsfall: bei Namensgleichheit hat im UP das lokale den Vorrang;

! Warnung:
Verwendung globaler Größen in UP ist gefährlich
⇒ Seiteneffekte
⇒ besser entspr. Parametervermittlung zw. ruf. + gerufenem Pr.

für obiges Bsp. →

Objekt def. in	verfügbar in
H	H und A..E
A	A, B
B	B
C	C, D, E
D	D
E	E

D) Beispiele

1. Bestimmung von Nullstellen einer Funktion

geg.: $y = f(x)$; $[a, b]$; Genauigkeitsschranke $\varepsilon > 0$;
ges.: alle Nullstellen von $f(x)$ in $[a, b]$ mit Genauigkeit ε;

Lösung:

Idee → Taste Intervall mit Schrittweite Δx ab. Ist ein Intervall $[x, x + \Delta x]$ mit Nullstelle gefunden, so Approximation derselben bis auf ε-Genauigkeit durch Halbieren des Intervalls.

I) Pseudokode

```
BEGIN
        Anf.-werte lesen;              {a, b, ε , Δ x}
        x auf Intervallanfang setzen;
        WHILE Intervallende nicht erreicht DO
                BEGIN          {Suche}
                        einen Schritt nach rechts;
                        IF Nullstelle gefunden
                          THEN Nullstelle drucken
                          ELSE IF Nullstellenintervall gefunden
                                  THEN   Nullstelle bestimmen
                END            {Suche}
END.
```

II) Verfeinerung

- Anf.-werte lesen: *a, b, epsilon, deltax* einlesen;
- *x* auf Int.-anfang: *xl* := *a* ; *yl* := *f* (*xl*) ; *xr* := *xl* + *deltax* ;
  {1. Teilint. untersuchen} *yr* := *f* (*xr*) ;
  IF ABS (*yl*) < *epsilon* THEN Nullst. *xl* drucken;
  IF ABS (*yr*) < *epsilon* THEN Nullst. *xr* drucken;
  IF Nullst.int. gefunden THEN Nullst. bestimmen;
- 1 Schritt nach rechts:
  xl := *xr* ; *xr* := *xr* + *deltax* ;
  yl := *yr* ; *yr* := *f* (*xr*) ;
- Nullstellenint. gefunden: *yr* * *yl* <= 0;
  {damit auch Fall erfaßt, daß Nullstelle auf Int.-rand}
- Nullstelle bestimmen:

```
REPEAT
        mitte := (xl + xr) / 2;
        IF f (mitte) * yl  > 0
                THEN  yl := f (mitte);   xl := mitte;
                ELSE  yr := f (mitte);   xr := mitte;
UNTIL  ABS( f (mitte)) < epsilon;
Nullstelle drucken;
```

```
PROGRAM    Nullstellen;

  USES CRT;
  VAR a, b, epsilon, deltax, x1, x2, y1, y2: REAL;

  FUNCTION  f(x: REAL): REAL;
    BEGIN
        f := (x – 1.12) * (x – 2.13) * (x – 3.14)          {Nullstellen bei 1.12,
    END;                                                              2.13 und 3.14 }
```

```
PROCEDURE anfwertelesen(VAR ug, og, eps, delta:REAL);
        BEGIN
          WRITE(´Untere Intervallgrenze gleich :  ´); READLN(ug);
          WRITE(´Obere Intervallgrenze gleich :  ´); READLN(og);
          WRITE(´Genauigkeit gleich :  ´); READLN(eps);
          WRITE(´Schrittweite gleich :  ´); READLN(delta)
        END;  {Ende anfwertelesen}
PROCEDURE nullstdrucken(x: REAL);
        BEGIN
          WRITELN(´Nullstelle bei x = ´, x:8:5, ´     y = ´, f(x):7:6)
        END;
PROCEDURE nullstbestimmen(xl, xr, yl, yr:REAL);
        VAR mitte: REAL;
        BEGIN
          REPEAT
                mitte := (xl + xr) / 2;
                IF  f(mitte) * yl > 0
                  THEN  BEGIN   yl:=f(mitte); xl:=mitte  END
                  ELSE   BEGIN   yr:=f(mitte); xr:=mitte  END
          UNTIL        ABS(f(mitte)) < epsilon;
          Nullstdrucken(mitte)
        END;            {Nullstelle bestimmen}
PROCEDURE   schrittnachrechts(VAR xl, xr, yl, yr:REAL);
  BEGIN
        xl := xr;         xr := xr + deltax;
        yl := yr;         yr := f(xr)
  END;
PROCEDURE    anfsetzen(VAR xl, xr, yl, yr: REAL);
   BEGIN
        xl := a; yl := f(xl);          xr := xl + deltax;       yr := f(xr);
        IF   ABS(yl) < epsilon  THEN  nullstdrucken(xl);
        IF   ABS(yr) < epsilon  THEN  nullstdrucken(xr);
        IF yl * yr < = 0  THEN  nullstbestimmen(xl, xr, yl, yr)
   END;
BEGIN         {HP}
  CLRSCR;
  anfwertelesen(a, b, epsilon, deltax);
  anfsetzen(x1, x2, y1, y2);
  WHILE  (x1 < b)   DO
        BEGIN
                schrittnachrechts(x1, x2, y1, y2);
                IF  ABS(y2) < epsilon
                        THEN  BEGIN
                                        nullstdrucken(x2);
                                        schrittnachrechts(x1, x2, y1, y2)
                                END
                        ELSE  IF       y1 * y2 <= 0
                                          THEN   nullstbestimmen(x1, x2, y1, y2)
        END;  {Ende suchen}
    READLN
END.
```

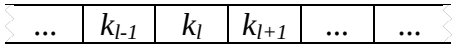
2. Mischen zweier Files

Geg.: 2 Files *i* und *j* mit Integer–Werten, die aufsteigend sortiert sind.
Ges.: Mischung von *i* und *j* zu einem aufsteigend sortierten File *k*.

I. Analyse und Problemdarstellung

Geg.:

Ziel:



wobei: a) $k_l \le k_{l+1}$ für alle *l*
b) alle Elemente aus *i* und *j* in *k* vorhanden

II. Entwurf einer Lösung

(1) Initialisierung der File-Durchläufe für *i* und *j* ;
Vereinbarung / Anlegen eines leeren Files *k* ;

(2) Solange noch nicht *i* und *j* ganz durchlaufen:
wenn Pufferinhalt *i*^ < *j*^, dann kopiere Inhalt von *i*^ nach *k* ;
anderenfalls kopiere Inhalt von *j*^ nach *k* ;

(3) Solange *i* noch nicht ganz durchlaufen: kopiere restl. Komponenten nach *k* ;

(4) Solange *j* noch nicht ganz durchlaufen: kopiere restl. Komponenten nach *k* ;

(5) Abschluß aller Dateiarbeiten;

III. Programm (-teile)

```
TYPE  basistyp = integer;
      dateityp = FILE OF basistyp;

PROCEDURE mischen (VAR i, j, k : dateityp);

(* In Turbo Pascal muessen Files VAR-Parameter sein,  sonst Compiler-Fehler 126. *)

VAR x, y : basistyp;

BEGIN     RESET (i);
          RESET (j);
          REWRITE (k);
```

```
        WHILE NOT EOF(i) AND NOT EOF(j) DO
                BEGIN       READ(i, x);
                            READ(j, y);
                            IF x < y THEN BEGIN      WRITE(k, x);
                                                     SEEK(j, filepos(j)-1 )
                                                END
                                         ELSE BEGIN  WRITE(k, y);
                                                     SEEK(i, filepos(i)-1 )
                                                END
                END;
        WHILE NOT EOF(i) DO   BEGIN READ(i, x);
                                    WRITE(k, x)
                                END;
        WHILE NOT EOF(j) DO   BEGIN READ(j, y);
                                    WRITE(k, y)
                                END;
        CLOSE(i); CLOSE(j); CLOSE(k)
END;
```

E) Rekursive Routinen

→ eine Prozedur / Funktion ruft sich selbst auf;
nicht alle PS unterstützen rekursive Aufrufe (Fortran);

- z.B.: *n!* durch folg. rekursive Darstellung bestimmbar:
  $n! = n \cdot (n-1)!$ *mit* $0! = 1$;

- Rechner zur Ausführung derartiger Routinen besonders gut geeignet;

- Selbstaufruf ⇒ durch geeignete akt. Parameter muß man für Abbruch der Rekursion sorgen
  → z. B. indem sich ein Parameter ändert und bei Überschreiten best. Grenzen die Rekursion beendet wird;

  !! Abbruchbedingung !!

- Bsp. 1) Berechnung von *n!*

```
FUNCTION  fak (n : Integer) : Integer;
        BEGIN
                IF  n  = 0
                  THEN   fak := 1
                  ELSE   fak := n * fak( n - 1)
        END;
```

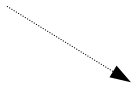
⇒ ! Bei $n < 0$ erfolgt kein Abbruch bei Eingabe eines falschen Eingabeparameters !
→ besser folg. Anweisungsteil:

```
BEGIN
  IF  n < 0
      THEN  WRITELN('Eingabepar. muß nichtneg. sein! Abbruch!')
      ELSE IF ...
END;
```

◆ Bsp. 2) Berechnung der Binomialkoeffizienten $\binom{n}{k} = \frac{n!}{(n-k)!\,k!}$

für gegebene *n* und *k*

Lösung: $\binom{n}{k} = \binom{n}{k-1} \cdot \frac{n-k+1}{k}$



→ FUNCTION *binom* (*n, k* : Integer) : Real;

```
BEGIN
      IF  k > 0
        THEN binom := binom (n, k – 1) * (n – k + 1) / k
        ELSE binom := 1
END;
```

◆ Rekursion ≠ Programmiertrick. Ist Methode, Prinzip!

Wdh. Ausführung auf zweierlei Art:

(a) durch Ineinanderschachteln (Rekursion)
(b) durch Aneinanderreihung (Iteration)

Bsp.: Berechnung des Binomialkoeffzienten durch Iteration:

→ FUNCTION *binom2* (*n, k* : Integer) : Real;

```
VAR   i : Integer;
      s : Real;
BEGIN
   s := 1;
   IF  k > 0  THEN  FOR  i := 1 TO  k  DO  s :=  s *(n – i +1) / i ;
   binom2 := s ;
END;
```

Ob der Programmierer für ein Problem die Lösung mittels Iteration oder Rekursion sucht → seine Sache!?

Rekursion: eleganter, oft aber Laufzeitineffezienz;
Iteration: wenn möglich, so nehmen;

! Rekursion vor allem bei Verabeitung rekursiver Datenstrukturen (⇒später).

◆ Arten der Rekursion:

direkte (unmittelbare) R. – eine Routine ruft sich im Anw.-teil selbst auf;
indirekte (mittelbare) R. – gegenseitiger Aufruf von Routinen; z. B. ⇒

```
PROCEDURE   A (x : Real);
 BEGIN
        ...
        B(x);                    {Bezeichner B hier noch unbekannt}
        ...
 END;                                Compiler mit Fehlermeldung:
                                        „Error: 3 Unknown Identifier“

PROCEDURE   B ( y : Real);
 BEGIN
        ...
        A( y);
        ...
 END;
```

⇒ kritische Situation wegen § 1 des Pascal-Gesetzbuches: Nur verwenden, was vorher definiert.

Umgehung der Situation durch **Vorwärtsbezug**
mittels FORWARD-Deklaration ⇒

```
PROCEDURE   B ( y : Real);  FORWARD;

PROCEDURE   A (x : Real);
 BEGIN
        ...
        B(x);
        ...
 END;

PROCEDURE   B;     {! Ohne Parameterliste !!!}
 BEGIN
        ...
        A( y);
        ...
 END;
```

Bezeichner FORWARD zeigt an, daß Körper der Prozedur *B* erst später ausgearbeitet wird. Da Bezeichner und Parameter für *B* def. sind, kann Aufruf von *B* innerhalb *A* korrekt übersetzt werden.

Zusammenfassung:

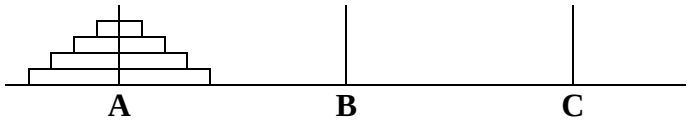
UP-Technik vielleicht wichtigste Technik der ganzen Programmierung bisher ⇒ 3 Gründe vor allem:

1° Kodeeinsparung
2° Wiederverwendbarkeit
3° Routine als Abstraktionseinheit

F) Weitere Beispiele

Bsp.: Turm von Hanoi

folg. Legende: buddhistischen Mönche in Hanoi folg. Aufgabe →
3 senkr. Stäbe; 64 runde Scheiben;



Turm von A nach C, wobei nur kleinere Scheiben oben aufgelegt werden dürfen; Mönche → Lösung = Ende der Welt;

n = 3 Scheiben →

1											
2			2							1	
3	0	0	3	0	1	3	2	1	3	2	0
											1
	1							2			2
0	2	3	1	2	3	1	0	3	0	0	3

bei $n > 0$ Scheiben exakt $2^n - 1$ Bewegungen
$2^{64} - 1 \approx 1.8446744 \cdot 10^{19} \approx$ 584 942 410 000 Jahre, wenn pro Sekunde eine Bewegung;

iterative Lösung → ?
rekursive Lösung → z. B. folg. Teile:

```
...
TYPE stab = (A, B, C);
...
PROCEDURE  turm (n : Integer; quelle, senke, hilf : stab);
 BEGIN
        IF n = 1
          THEN  bewege Scheibe von quelle nach senke
          ELSE  BEGIN
                    turm (n – 1; quelle, hilf, senke);
                    bewege n-te Scheibe von quelle nach senke
                    turm (n – 1; hilf, senke, quelle);
                END;
 END;
```

Bsp.: Fibonacci-Zahlen

* Leonardo Fibonacci → 1202 Liber abaci ⇒
folg. Frage:
Wieviele Kaninchen-Pärchen gibt es nach n Jahren, wenn im Jahr 1 Start mit 1 Pärchen und jedes Pärchen vom 2. Jahr an ein weiteres Pärchen Nachwuchs hat und die Kaninchen unendlich lange leben.
→

Jahr	1	2	3	4	5	6	7	8	9	10	...
Anzahl	1	1	2	3	5	8	13	21	34	55	...

Diese Fibonacci-Zahlen haben allg. Bedeutung
⇒ folg. rekursive Def.:

$f(n) = f(n-1) + f(n-2)$ für $n > 2$

→ brauchen für Abbruch 2 Werte:

$f(1) = 1$ und $f(2) = 1$;

* ⇒
```
FUNCTION  fibonacci (arg : Integer) : Integer;
  BEGIN
    IF  arg <= 2
     THEN fibonacci := 1
     ELSE fibonacci := fibonacci (arg –1)  +  fibonacci (arg –2 )
  END;
```

* Anzahl Aufrufe? Sei c_n – Anzahl Aufrufe bei $arg = n$.

$c_1 = c_2 = 1$

$n > 2$: $c_n = 1 + c_{n-1} + c_{n-2}$

$n > 3$: $c_{n-1} = 1 + c_{n-2} + c_{n-3}$

$c_n = 2 + 2c_{n-2} + c_{n-3} > 2 \cdot c_{n-2} > ... > 2^{n \text{ DIV } 2 - 1} \cdot c_2 > 2^{n \text{ DIV } 2 - 1}$

oder $c_n = 1 + c_{n-1} + c_{n-2} < 2 \cdot c_{n-1} - c_{n-3} < 2 \cdot c_{n-1} < 2^2 \cdot c_{n-2} < ... < 2^n$

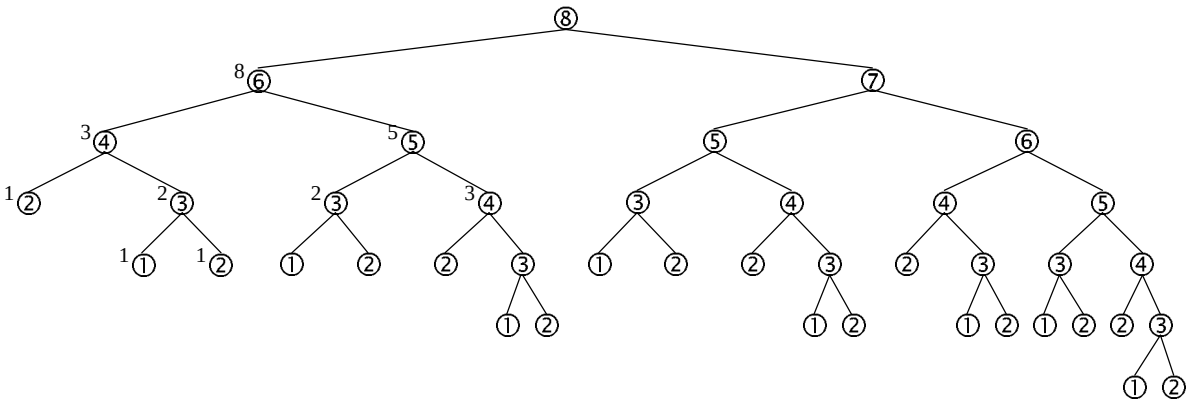
⇒ Funktion *fibonacci* erfordert mit n expon. wachsenden Aufwand
⇒ für $n = 40$ schon sehr große Laufzeiten
⇒ nicht effizient für große n

⇒ als iterative Lösung:

```
FUNCTION  fibonacci2 (arg : Integer) : Integer;
    VAR  f, f1, f2, i : Integer;
  BEGIN
    f1 := 1;  f2 := 1;
    IF  arg <= 2
          THEN  f  :=  f1
           ELSE FOR  i := 3 TO  arg  DO
                    BEGIN
                       f  := f1 + f2;
                       f1 := f2;
                       f2 := f;
                    END;
     fibonacci2  :=  f;
  END;
```

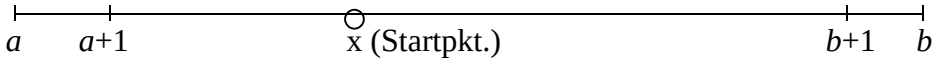
⇒ Aufwand = n * (1 Add + 3 Zuweisungen) ≈ n * Konstante

* Aufrufbaum: Inorder – Reihenfolge



Schachtelungstiefe =7

Bsp.: 1-dimensionale zufällige Irrfahrt



Geg.: $P(x \to x+1) = P(x \to x-1) = \frac{1}{2}$, $x = a + 1(1)\, b - 1$;

$P(a \to a) = P(b \to b) = 1$.

Ges.: Zu erwartende Anzahl Takte bis Abbruch (d. h., bis entweder a oder b erreicht).

· Analyse:

Sei $T(x) = \mathsf{E}$[Anzahl Takte von x bis a oder b].

$\Rightarrow$ (*) $\begin{cases} T(x) = 1 + \frac{1}{2} \cdot T(x+1) + \frac{1}{2} \cdot T(X-1), & a < x < b; \\ T(a) = 0;\ T(b) = 0; \end{cases}$

Scheint analog Fibonacci, wenn dargestellt als
$T(x + 1) = 2 \cdot T(x) - T(x - 1) - 2$, $a < x < b$; (?)

Aber: Für $x = a + 1 \to$
$T(a + 2) = 2 \cdot T(a + 1) - T(a) - 2$, d. h.
$T(a)$ ist bekannt (= 0), jedoch $T(a + 1)$ unbekannt!
Wenn bekannt, so wäre Rekursion anwendbar.

Was tun?

$\Rightarrow$ Schreiben (*) etwas ausführlicher $\to$

$$(**)\begin{cases} T(a) = 0 \\ T(a+1) = 1 + \frac{1}{2}\cdot T(a+2) \\ T(a+2) = 1 + \frac{1}{2}\cdot T(a+3) + \frac{1}{2}\cdot T(a+1) \\ \quad\vdots \\ T(b-2) = 1 + \frac{1}{2}\cdot T(b-1) + \frac{1}{2}\cdot T(b-3) \\ T(b-1) = 1 + \frac{1}{2}\cdot (b-2) \\ T(b) = 0 \end{cases}$$

$\Rightarrow (b - a - 1)$ Gleichungen und ebenso viele Unbekannte

Ausweg: Lösen des Gleichungssystems.

Wie? Iteratives Einsetzen $\Rightarrow$

$$T(a+2) = 2 + \frac{2}{3}\cdot T(a+3);$$

$$T(a+3) = 3 + \frac{3}{4}\cdot T(a+4) \; usw;$$

$$n > 1: \; T(a+n) = n + \frac{n}{n+1}\cdot T(a+n+1) \quad (?)$$

Wie beweisen?

$\rightarrow$ einfaches Verfahren, welches in math. Beweisführung und im Alg.-entwurf große Rolle spielt:

• Vollständige Induktion:

Beweisführung nach folg. allg. Schema:

Sei A eine Aussage, die zu beweisen ist. Möge weiter A einen Parameter n enthalten, dessen Wert eine beliebige natürliche Zahl sein kann. Zum Beweis, daß A für alle $n \in \mathbb{N} = \{1, 2, ...\}$ gilt, genügt es, die folg. Beweise zu führen:

1. Induktionsbeginn (~verankerung):

A gilt für $n = 1$.

2. Induktionsübergang (~schluß) von n zu $n + 1$:

Aus der Ann., daß A für ein $n \in \mathbb{N}$ gilt (Induktionsannahme), wird abgeleitet, daß A auch für $n + 1$ gilt.

Bsp.: *A1*: Für $\forall\, x, n \in \mathbf{N}$ gilt: $x^n - 1$ ist teilbar durch $x - 1$.

Beweis:
1. $n = 1$: Trivial, da $(x - 1) : (x - 1) = 1$.
2. $n \to n + 1$: Sei $(x^n - 1) : (x - 1) = k$.
$x^{n+1} - 1 = x(x^n - 1) + x - 1 = (x - 1) \cdot (x \cdot k + 1)$

■

A2: Für $n \in \mathbf{N}$ und $1 + x > 0$ gilt: $(1 + x)^n \geq 1 + n \cdot x$.

Beweis:
1. $n = 1$: Trivial, da $1 + x \geq 1 + 1 \cdot x$.
2. $n \to n + 1$:

$$(1 + x)^{n+1} = (1 + x)^n + (1 + x) \underset{\text{weil } (1+x) > 0}{\geq} (1 + n \cdot x) \cdot (1 + x) =$$

$$= 1 + n \cdot x + x + n \cdot x^2 = 1 + (n + 1) \cdot x + n \cdot x^2 \geq 1 + (n + 1) \cdot x$$

■

• Beweis von $T(a+n) = n + \frac{n}{n+1} \cdot T(a+n+1), \quad n = 1(1)(b-a-1).$

1. $n = 1$: Trivial, aus (**) .
2. $n \to n + 1$:

$$T(a+n+1) = 1 + \frac{1}{2} \cdot T(a+n+2) + \frac{1}{2} \cdot T(a+n)$$

$$= 1 + \frac{1}{2} \cdot T(a+n+2) + \frac{1}{2} \cdot \left[n + \frac{n}{n+1} T(a+n+1) \right]$$

$$= \frac{2+n}{2} + \frac{1}{2} \cdot T(a+n+2) + \frac{n}{2(n+1)} \cdot T(a+n+1)$$

$$\to T(a+n+1) \cdot \frac{2n+2-n}{2(n+1)} = \frac{2+n}{2} + \frac{1}{2} \cdot T(a+n+2)$$

$$\to T(a+n+1) = (n+1) + \frac{n+1}{n+2} \cdot T(a+n+2)$$

■

• Lösung von (*):

Setzen in (?) $n = b - a - 1 \Rightarrow T(b - 1) = (b - a - 1)$

Nun induktiv: $T(b - \mathrm{n}) = n(b - a - \mathrm{n}), \quad n = 1(1)(b - a - 1)$;

Für $n = b - x \Rightarrow$

$$\boxed{T(x) = (b-x) \cdot (x-a)} \qquad \{\text{geschlossene Lösung}\}$$

• Rekursion:

Mit $T(a) = 0$ und
$T(a + 1) = (b - a - 1)$ wäre
Rekursion der Form

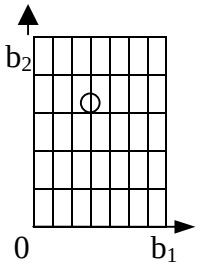
$$\begin{cases} T(a+n) = 2 \cdot T(a+n-1) - 2 - T(a+n-2) \\ T(a+1) = (b-a-1) \\ T(a) = 0 \end{cases}$$

anwendbar.
Bessere Rekursion ist natürlich über die
Formel $T(a+n) = n + \frac{n}{n+1} \cdot T(a+n+1), \quad n \geq 1;$

```
PROGRAM zufaellige_Irrfahrt;

USES CRT;

VAR a, b, x: Integer; z: Real;

FUNCTION  irr(c, d, y: Integer): Real;

  BEGIN
   IF (y = c) OR (y = d)       THEN irr := 0
                               ELSE  irr := (y - c) + (y - c) / (y - c + 1) * irr(c, d, y+1)
                {auch möglich irr := (d - y) + (d - y)/(d - y + 1) * irr(c, d, y-1);
                 bei irr := 1 + 0.5 * ( irr(c, d, y-1) + irr(c, d, y+1)) erfolgt
                 zur Laufzeit eine Fehlermeldung wegen Stack - Überlaufes,
                 sofern d – c > 2 }
  END;

BEGIN {HP}
        CLRSCR;
        WRITE('a =');          READLN(a);
        WRITE('b =');          READLN(b);
        WRITE('x = ');         READLN(x);
        z := irr(a, b, x);
        WRITE('Für x = ', x, ' gilt T( ', x, ') = ', z);
        READLN
END.
```

• Erweiterte Problemstellungen:

(1) Bestimme x^* : $T(x^*) \geq T(x)$.
(2) Je Takt ein Ertrag e. Wie erwartete Gesamtertrag bis zum Abbruch?
(3) Bestimme W. für Abbruch in a bzw. b ?
(4) Wie bei nichtsymm. Irrfahrt, d.h. wenn
$P(x \to x + 1) = p$, $P(x \to x - 1) = q$, $p + q = 1$?
(5) Wie wenn noch $P(x \to x) = r$, $p + q + r = 1$?
(6) Wie im mehrdim. Fall?



• Ein (Weihnachts-) Baum – Problem:

Geg.: 1 Karton mit Weihnachtskugeln, die von 1 bis 12 numeriert sind;
1 Weihnachtsbaum;
Ges.: Die in zufälliger Reihenfolge entnommenen Kugeln sollen in den Baum gehängt werden, so daß eine bestimmte Ordnung erkennbar ist.

z. B. folg. Idee ⇒

- 1. Kugel wird in Spitze gehängt;
- hat 2. Kugel niedrigere Nr., so links unterhalb der 1. Kugel;
- hat 2. Kugel höhere Nr., so rechts unterhalb;
- usw., bis alle Kugeln angebracht;

für Reihenfolge 6, 7, 11, 2, 5, 4, 10, 1, 3, 8, 12, 9 ergibt sich nach dieser Vorschrift: