Praktische Informatik für Wirtschaftsmathematiker, Ingenieure und Naturwissenschaftler I (PIWIN I, 3 V + 1 Ü) WS 2002/03

4. Vorlesungswoche

Grundlagen imperativer und objektorientierter Programmierung

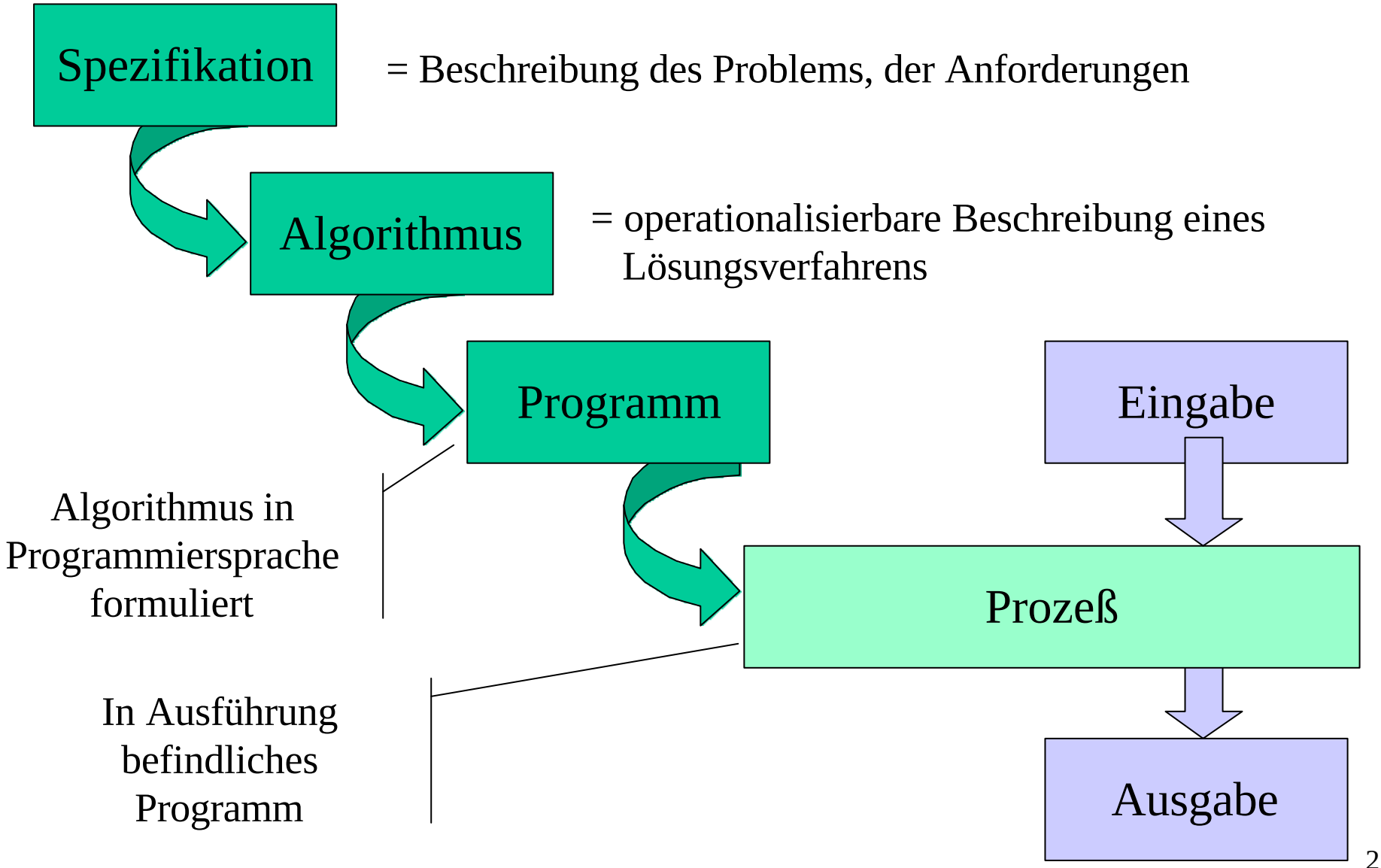
Funktion, Prozedur, Methode und Rekursion

Unterlagen: Gumm/Sommer, Kapitel 2

Echtle/Goedicke, Einführung in die Programmierung mit Java, dpunkt Verlag, Kapitel 4

Doberkat/Dißmann, Einführung in die objektorientierte Programmierung mit Java, Oldenbourg, Kapitel 4

Stationen im Entwurf von Algorithmen und Programmen



Übersicht

- Begriffe
  - Spezifikationen, Algorithmen, formale Sprachen, Grammatik
- Programmiersprachenkonzepte
  - Syntax und Semantik
  - imperative, objektorientierte, funktionale und logische Programmierung
  - formale Sprachen und Grammatik
- Grundlagen der Programmierung
  - imperative Programmierung:
    - Verfeinerung, elementare Operationen, Sequenz, Selektion, Iteration, funktionale Algorithmen und Rekursion, Variablen und Wertzuweisungen, Prozeduren, Funktionen und Modularität, Zuweisung, Sequenz
  - objektorientierte Programmierung
- Algorithmen und Datenstrukturen
- Berechenbarkeit und Entscheidbarkeit von Problemen
- Effizienz und Komplexität von Algorithmen
- Programmentwurf, Softwareentwurf

Kern imperativer Sprachen

- Zuweisungen
  - Variable speichern / tragen / beinhalten Werte
  - Zuweisungen müssen typverträglich sein
    - Variable haben einen Typ, daher zunächst (einfache) Datentypen und Operationen
- Kontrollstrukturen
  - Sequentielle Komposition, Sequenz
  - Alternative, Fallunterscheidung
  - Schleife, Wiederholung, Iteration
- Verfeinerung
  - Blockstrukturierung
  - Unterprogramme, Prozeduren, Funktionen
    - Rekursion

Unterprogramm

Grundidee:

- Probleme werden in Teilprobleme zerlegt
  - die durch bekannte oder neu zu entwickelnde Algorithmen gelöst werden
- aus den Lösungen der Teilprobleme wird eine Lösung für das Gesamtproblem bestimmt

Dieses Konzept wird in Programmiersprachen durch Unterprogramme unterstützt

- Block mit eigenem Bezeichner mit Eingabeparametern und Ausgabeparametern
- dadurch mehrfache Verwendung im Programm möglich
- Wiederverwendbarkeit / Nützlichkeit hängt vom Problem, aber auch vom Grad der Abstraktion ab

- Varianten:
  - Prozedur: Unterprogramm ohne ausgezeichneten Rückgabeparameter
  - Funktion: Unterprogramm mit ausgezeichnetem Rückgabeparameter (möglicherweise weitere)
  - Methode: Funktion/Prozedur, die für ein Objekt / einen speziellen Datentyp definiert ist.

zur Erinnerung

- Definition des Begriffs „Informatik“

  nach der Akademie Francaise, angelehnt an „informatique“

  „Behandlung von Information mit rationalen Mitteln“

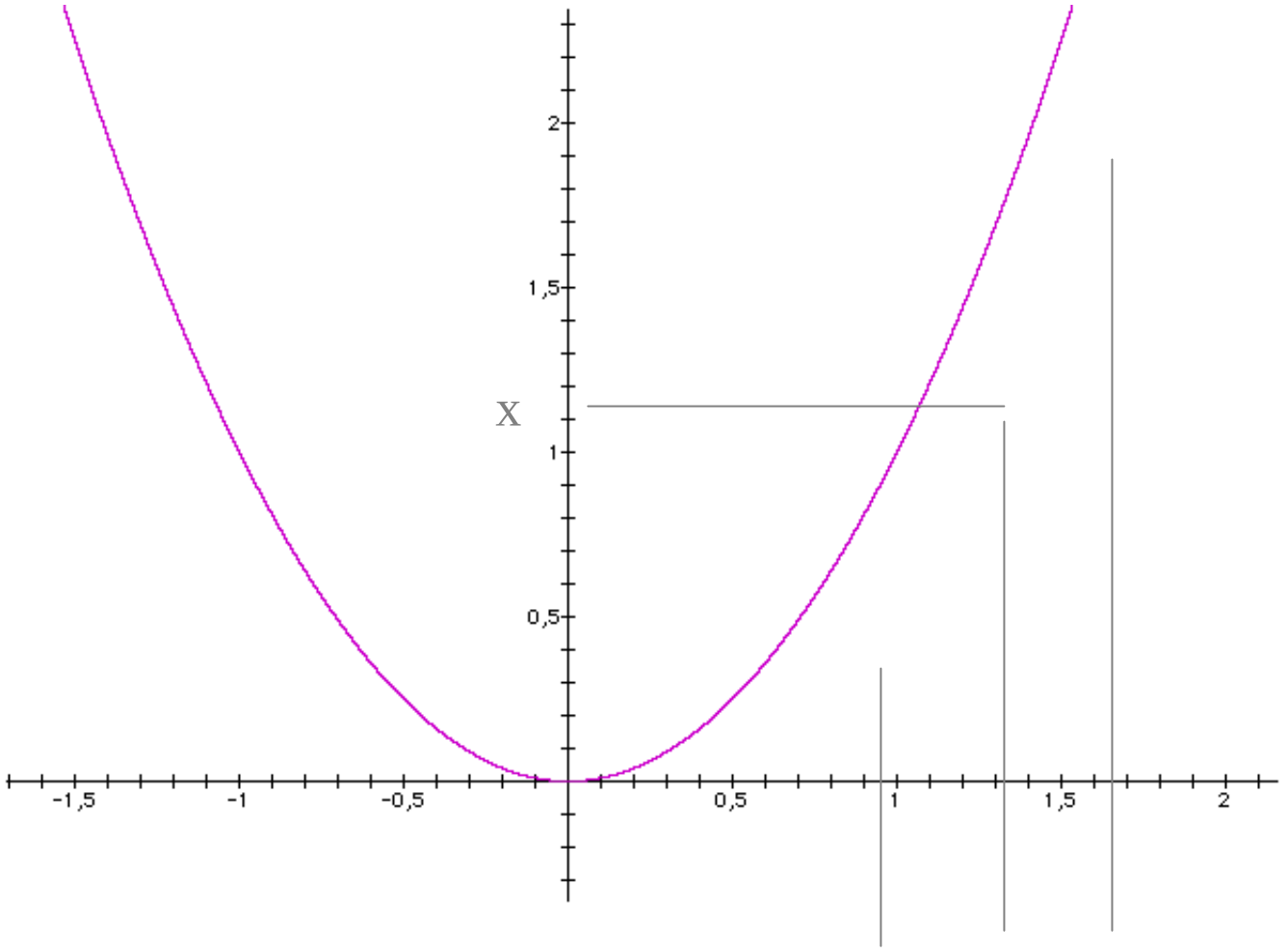
  wobei rationale Mittel nach Descartes auszeichnet:

  – „nur dasjenige gilt als wahr, was so klar ist, dass kein Zweifel bleibt“
  – „größere Probleme sind in kleinere aufzuspalten“
  – „es ist immer vom Einfachen zum Zusammengesetzten hin zu argumentieren“
  – „das Werk muss am Ende einer abschließenden Prüfung unterworfen werden“

Beispiel: einfache Numerik Funktionen

- Berechnung der Quadratwurzel float sqrt(float n) für n > 0 durch eine Funktion (Unterprogramm)
- Nützlichkeit klar,
  - in vielen Programmen unabhängig vom Kontext verwendbar
  - daher auch in Bibliotheken / Libraries stets verfügbar
- eine Berechnungsidee: Intervallschachtelung
  - finde eine untere Schranke
  - finde eine obere Schranke
  - verringere obere und untere Schranke bis der Abstand hinreichend gering geworden ist
  - etwas konkreter: halbiere Intervall, fahre mit demjenigen Teilintervall fort, das das Resultat enthält

Quadrat-Wurzel Berechnung mittels Intervallschachtelung



m_2 m_i m_1

(©M. Goedicke, UGH Essen)

Beispiel: float sqrt(float n)

Quadrat-Wurzel Berechnung mittels Intervallschachtelung

Pseudocode:

Start: Intervall [0,x+1], Mitte m = 0,5*(uG+oG)

Algorithmus:

Berechne neue Mitte m = 0,5*(uG+oG)

Falls ($m^2 > x$) dann oG = m

sonst uG = m

Abbruch: falls oG - uG < ε

Beispiel float sqrt(float x)

```
float sqrt( float x)
{


float uG = 0,  oG = x + 1,  m, epsilon = 0.001f;

do { m = (uG + oG)/2;
    if (m*m > x)
        oG = m;
    else
        uG = m;
  }
while (oG - uG > epsilon);
return (m) ;
}
```

float: Deklaration des Datentyps für den Rückgabewert

x: Eingabeparameter

sqrt: Funktionsbezeichner

uG, oG, m, epsilon: lokale Variable

m: Rückgabewert

Verwendung von Prozeduren, Funktionen

Aufruf einer Prozedur gilt als Anweisung,

Aufruf einer Funktion gilt als Ausdruck

daher entsprechend verwendbar:

z.B.

...

```
float m = sqrt( x ) ;
system.out.println("Wurzel " + x + " beträgt ungefähr " + m);
```

also

- Funktionen auf der rechten Seite einer Zuweisung und in Ausdrücken
- Prozeduren anstelle einer Zuweisung
- Funktionen anstelle einer Zuweisung, falls kein Rückgabewert definiert ist (void) oder der Rückgabewert ignoriert werden soll

Die Lösung von Teilproblemen durch Prozeduren (Funktionen) nennt man prozedurale (funktionale) Abstraktion.

Top-Down Entwurf

- Top-Down Strategie
  - Zerlege Problem in Teilprobleme
  - Löse Teilprobleme
  - Kombiniere Lösung der Teilprobleme zur Lösung des Gesamtproblems
- Im Entwurf
  - Zerlege Problem in Teilprobleme,
  - Deklariere für jedes Teilproblem eine Prozedur / Funktion, die im Entwurf zunächst unausgefüllt bleibt (stubs = Stummel)
  - Löse Gesamtproblem mit Hilfe der Stubs für Teilprobleme
  - Entwerfe / entwickle Lösung für die Teilprobleme, fülle Stubs
- Hinweis: definiere Teilprobleme so, dass sie als bekannte allgemeine Probleme aus der Informatik erkennbar werden, für die eine bekannte Lösung genutzt werden kann (aus SW-Bibliotheken)
- Vorgehen liefert wg funktionaler Abstraktion eine Zerlegung nach Funktionen, nach Aufgaben
  - typisch für imperative Programmierung
  - es existieren Alternativen: Zerlegung nach Daten
  - Sichtweise für objektorientierte Programmierung etwas anders

Beispiel: einfaches Spiel

```
int main()
{ int spieler =1 ; boolean fertig = false ;
   init() ;
   while (!fertig)
   {
     visualisiereSpiel() ;
     macheZug() ;
      if ( Spielende() )
         fertig = true ;
     else SpielerWechsel() ;
   }
GratuiereSieger() ;
}
```

- int init()
- int macheZug()
- int SpielerWechsel()
- int visualisiereSpiel()
- boolean Spielende()
- int GratuliereSieger()

Kommunikation zwischen Haupt- und Unterprogramm

- Haupt- und Unterprogramme eines Programms teilen sich bei der Bearbeitung (als Prozeß) einen gemeinsamen Adreßraum
- Kommunikation über globale Variable
  - Variable, die als global deklariert sind, können in Unterprogrammen ebenfalls gelesen & verändert werden
  - globale Variable führen zu Funktionen, deren genaue Auswirkungen schwer überblickt werden -> sogenannte Seiteneffekte
  - daher nur sehr begrenzt sinnvoll einsetzbar
- Kommunikation erfolgt über Parameter
  - Eingabeparameter: liefern Informationen, die innerhalb des Unterprogramms nur gelesen werden
  - Rückgabeparameter einer Funktion: liefert Wert der von der aufrufenden Funktion / Prozedur gelesen werden kann
    - begrenzte Möglichkeiten
    - häufig für einfache Rückgaben, z.B. boolesche Resultate, Auftreten von Fehlern genutzt (sofern nicht Exceptions in der Sprache vorgesehen sind)
  - Ausweg: Aufrufparameter, die Rückgabe erlauben, sog. Variablenparameter (bei Gumm/Sommer durch bezug zur Sprache Pascal)

Variablenparameter, call by reference

- Beobachtung: call by value
  - bei einem Funktionsaufruf werden die Parameter mit konkreten Werten belegt
  - Sichtweise: Parameter sind funktionslokale Variable, die bei Aufruf der Funktion mit den Werten beim Aufruf initialisiert werden
  - float sqrt(float x) Aufruf: sqrt(4.0) impliziert x = 4.0
  - call by value
- Variablenparameter sind Parameter, die als Referenz / Adresse eines Datentyps deklariert sind
  - z.B. in „C“: void sqrt (float x, float* wurzel) { ... }
  - Aufruf bei Variable float out: sqrt(4.0, &out)
  - bewirkt, dass wurzel die Speicheradresse von out enthält
  - innerhalb von sqrt erfolgt dann z.B. Zuweisung: *wurzel = 2.0
  - call by reference ist in Wirklichkeit auch call by value, Wert ist bei einer Referenz jedoch der Speicherort
  - Idee stets gleich, wird in Programmiersprachen unterschiedlich beschrieben!!!
  - in Java: stets call by value, nur Objekte werden per Referenz übergeben

Parameterübergabe in Java

```
class BspKlasse1 {
  void plusEins (int x) {
     x = x+1 ;
  }
}
```

Verwendung:

```
BspKlasse1 b = new BspKlasse1() ;
int y = 3 ;
b.plusEins(y) ;
```

```
class BspKlasse2 {
  void plusEins (Waehrung w) {
     Waehrung eins = new Waehrung(1,0) ;
     w.erhoehe(eins) ;
  }
}
```

Verwendung:

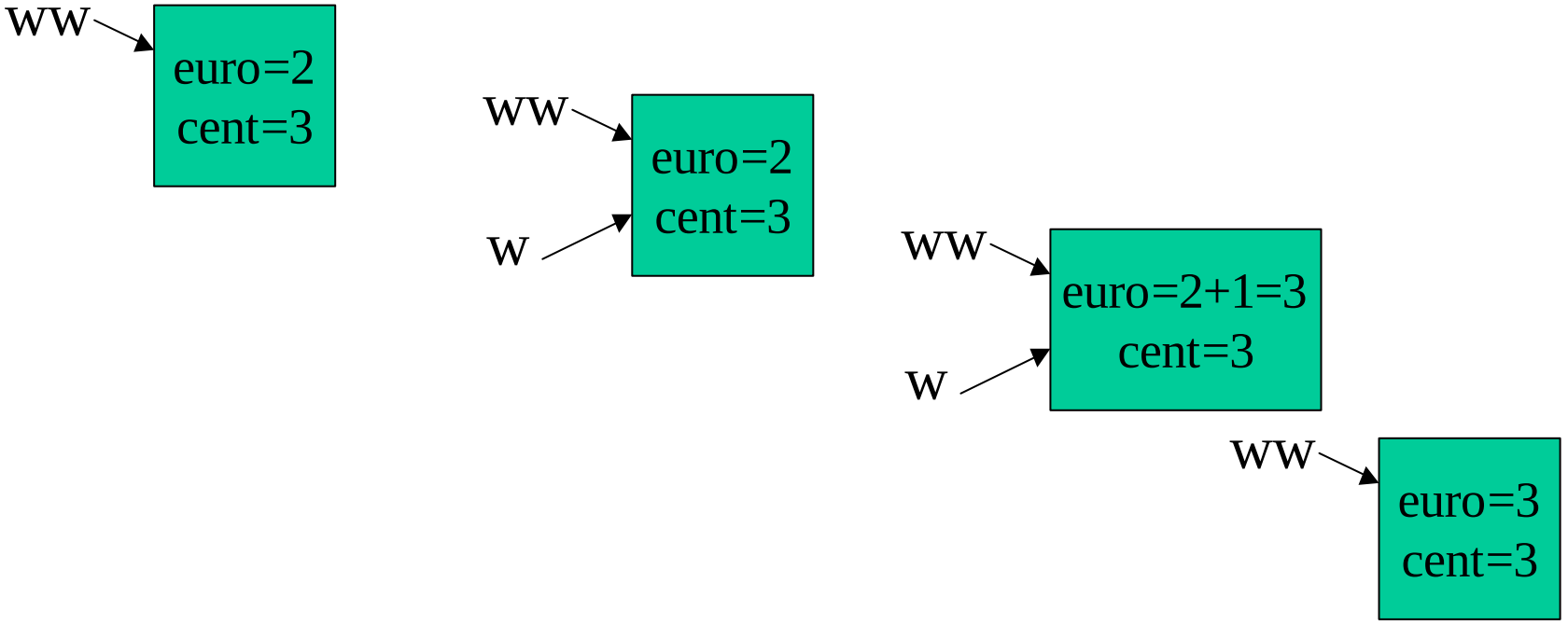
```
BspKlasse2 b = new BspKlasse2() ;
Waehrung ww = new Waehrung(2,3);
b.plusEins(ww) ;
```

Welche Werte hat y vor / nach Aufruf von plusEins ?

Welche Werte hat ww vor / nach Aufruf von plusEins ?

Erklärung

Es findet stets call by value statt:
Bei primitiven Datentypen wie int wird Wert übergeben,
es gibt keine Rückwirkung von x+1 auf y.
Bei Klassen / Objekten wird Referenz übergeben,
die Änderung von w wirkt daher auf ww:



Abarbeiten von Funktionsaufrufen

- 1. Idee
  - textuelle Ersetzung
  - bei der Übersetzung des Quelltextes in Maschinensprache wird an jeder Stelle des Funktionsaufrufes der Text eingefügt
- Nachteile
  - keine rekursiven Funktionen
  - bei Prozeduren ok, bei Funktionen wegen Rückgabeparametern umständlich
  - erzeugt unnötig umfangreichen Code in Maschinensprache
- daher nur in speziellen Kontexten unterstützt
- sogenannte Makroexpansion
  - Skript Sprachen
  - C, C- Präprozessor lässt sich hierfür einsetzen, da er parametrisierte textuelle Ersetzungen am Quellcode durchführt, bevor der Compiler eingesetzt wird.

Abarbeiten von Funktionsaufrufen

2. Idee, und wesentlich bedeutsamer:

- unterliegende Basismaschine unterstützt Funktionsaufrufe
- Prozess besteht aus Speicherbereichen für
  - Verwaltungsinformation für das Betriebssystem (Prozessorstatuswort, Programmzähler, ...)
  - Programmcode
  - Heap (= Haufen)
    - Menge aller Variablen / Objekte, die zur Laufzeit dynamisch allokiert wurden (new, malloc, ...) und noch nicht freigegeben wurden
      - Freigabe in Sprachen entweder explizit (delete, free) oder automatisch als Garbage Collection nicht mehr referenzierbarer Objekte
    - gesonderter Speicherbereich nötig, weil Objekte auch nach Terminierung der sie erzeugenden Funktion / Methode (new, malloc, ...) noch existieren
  - Stack (= Stapel)
    - bei jedem Funktionsaufruf wird ein neues Element auf dem Stapel erzeugt, das u.a. die Parameter und lokalen Variable der Funktion enthält
    - bei Terminierung einer Funktion wird das zugehörige (oberste) Element vom Stapel entfernt (Speicherplatz wird freigeben, d.h. beim nächsten Funktionsaufruf für andere Inhalte genutzt)

Rekursive Funktionen und Prozeduren

- Rekursion ist ein wichtiges Hilfsmittel zur Strukturierung des Kontrollflusses von Algorithmen und zur Beschreibung von Datenstrukturen
- Eine Funktion f ist rekursiv, wenn der Funktionsrumpf einen Aufruf der Funktion f selbst enthält oder einer Funktion g, die wiederum f aufruft.
- Beispiel: Fakultätsfunktion

math. Def:

$$n! = \begin{cases} 1, & \text{falls } n = 0 \\ n \cdot (n-1)!, & \text{falls } n > 0 \end{cases}$$

rekursive Funktion:

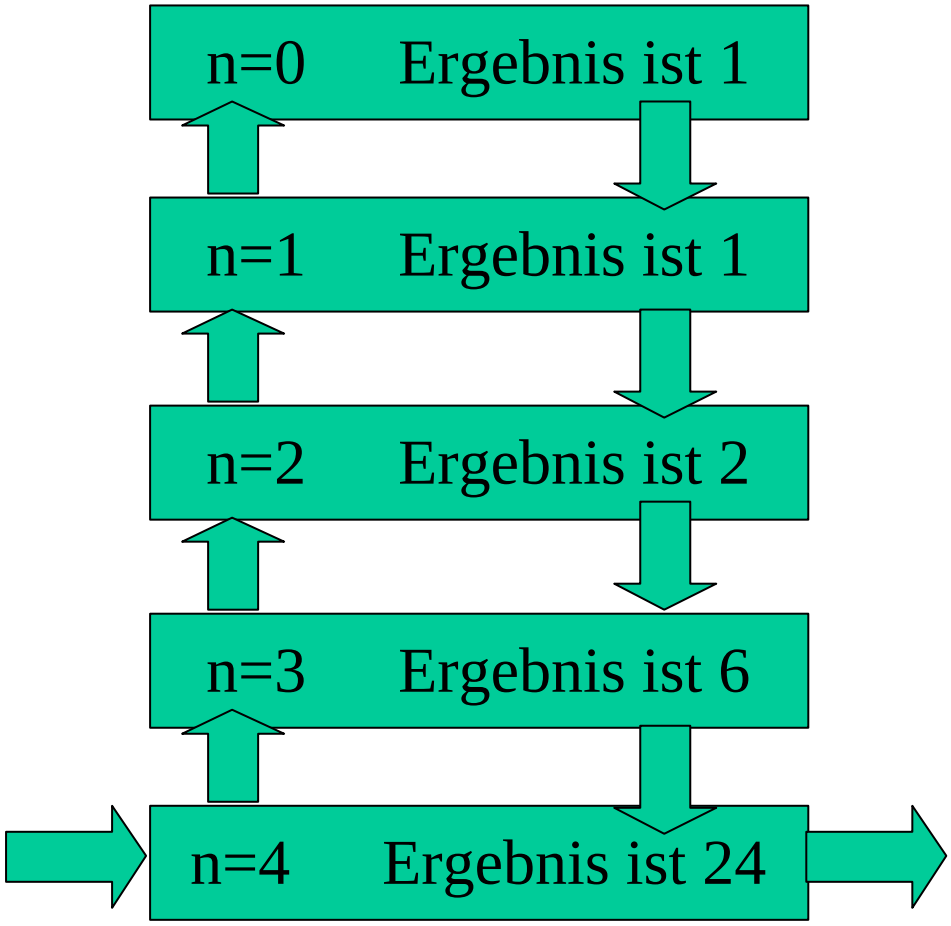
```
int fakultaet(int n)
{
  1: if (n = = 0)   return(1) ;
  2: if (n > 0)     return( n * fakultaet(n-1) ) ;
  3: return(-1) ;
}
```

Rekursionsanker

Rekursion

Fehlerfall

Aufruf der Fakultätsfunktion



Situation im Stack
Aufbauphase endet mit
4: fakultaet: n=0, Zeile1
3: fakultaet: n=1, Zeile 2
2: fakultaet: n=2, Zeile 2
1: fakultaet: n=3, Zeile 2
0: fakultaet: n=4, Zeile 2

Abbauphase nach 5. Aufruf
3: fakultaet: n=1*1, Zeile 2
2: fakultaet: n=2, Zeile 2
1: fakultaet: n=3, Zeile 2
0: fakultaet: n=4, Zeile 2

2: fakultaet: n=2*1, Zeile 2
1: fakultaet: n=3, Zeile 2
0: fakultaet: n=4, Zeile 2

1: fakultaet: n=3*2, Zeile 2
0: fakultaet: n=4, Zeile 2

Rekursion ist ein mächtiges Konzept

Viele Probleme lassen sich mit rekursiven Funktionen elegant beschreiben und lösen:

Beispiele:

- Türme von Hanoi
- Backtracking für Suchalgorithmen z.B. in Spielstrategien

Rekursion ist aber nicht-trivial:

z.B. für die ULAM Funktion ist Terminierung für nat. Zahlen bis heute unbewiesen

$$f(n) = \begin{cases} 1, & \text{falls } n = 1 \\ f(n/2), & \text{falls n gerade} \\ f(3n+1), & \text{sonst} \end{cases}$$

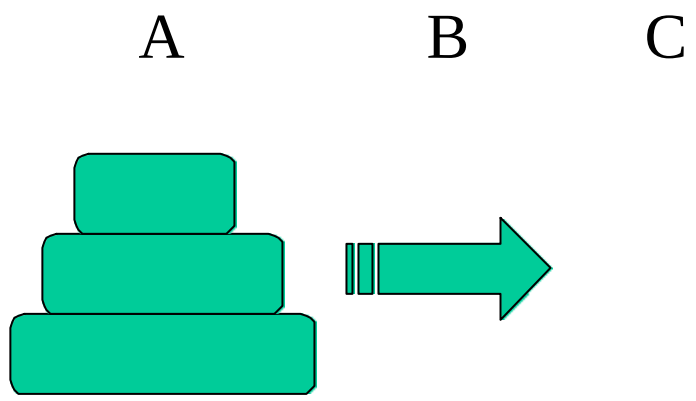
z.B. Erkennen der Berechnung: McCarthys „91-Funktion“:

$$f(n) = \begin{cases} n-10, & \text{falls } n > 100 \\ f(f(n+11)), & \text{sonst} \end{cases}$$

Beispiel: Türme von Hanoi

- Ein Stapel von N Scheiben verschiedener Durchmesser sei als Turm aufgeschichtet. Der Durchmesser nimmt nach oben ab. Der Turm steht auf Platz A, soll nach Platz C verlagert werden, wobei Platz B als Zwischenlager benutzt werden darf.
  - Randbedingung: jeweils nur 1 Scheibe darf bewegt werden
  - Randbedingung: es darf nie größere auf kleinerer Scheibe liegen
- Lösungsidee
  - Falls Turm mit N-1 Scheiben auf B, größte Scheibe auf A, dann kann einfach die Scheibe von A nach C verschoben werden, und äquivalentes Problem mit N-1 Scheiben für Start B, Ziel C und Zwischenlager A tritt auf.

```
int hanoi( int n, platz start, zwischen, ziel) {
    if n=1 verschiebeScheibe(start,ziel) ;
    else {
    hanoi(n-1,start,ziel,zwischen) ;
    verschiebeScheibe(start,ziel);
    hanoi(n-1,zwischen,start,ziel);
    }
 }
```



Zur Klassifikation rekursiver Methoden

Definition:

Eine rekursive Methode heißt primitiv rekursiv, wenn sie folgendes Schema hat, das auf den natürlichen Zahlen für n beruht .
Schema:

$$f(n) = \begin{cases} s_0, & \text{falls } n = 0 \\ h(\mathrm{n}, \mathrm{f}(\ \mathrm{pred}(\mathrm{n})\ )), & \text{sonst} \end{cases}$$

- basiert auf Peano Axiomen für nat. Zahlen:
  - 0 ist nat. Zahl,
  - succ(n) ist nat. Zahl, wenn n nat. Zahl ist

Eine wechselseite Rekursion liegt vor, wenn eine Funktion f eine Funktion g aufruft, die ihrerseits wiederum f aufruft.
Wechselseitige Rekursion ist natürlich auch mit mehr als 2 Funktionen möglich.

Zur Klassifikation rekursiver Methoden

Definition:

Eine rekursive Methode heißt endrekursiv (tail recursive), wenn sie folgendes Schema hat.

Schema:

$$f(x) = \begin{cases} g(x), & \text{falls } P(x) \\ f(\,r(x)\,), & \text{sonst} \end{cases}$$

- endrekursiv, weil f() die letzte Aktion im „else“ Zweig ist.
- Beobachtung:

f(x) = f(r(x))

= f(r(r(x))) = f(r^2(x))

= ...

= f(r^k(x))= g(r^k(x))

wobei k die kleinste natürliche Zahl ist, für die P(r^k(x))=true

Daher einfaches iteratives Programm

```
f_iterativ(float x){ while (!P(x)) { x = r(x) ; } return(g(x)) ; }
```

Zur Klassifikation rekursiver Methoden

Definition:

Eine rekursive Methode heißt linear, wenn in den einzelnen Zweigen der bedingten Anweisung, die die Rekursion steuert, jeweils höchstens ein Aufruf der Methode vorkommt.

Schema:

$$f(x) = \begin{cases} g(x), & \text{falls } P(x) \\ h(x, f(\,r(x)\,)), & \text{sonst} \end{cases}$$

- Verallgemeinerung des primitiv rekursiven Schemas (r = pred(x)) und der Endrekursion h(x,y)=y
- Beobachtung:

f(x) = h(x,f(r(x)))

= h(x, h(r(x), f(r(r(x))))) = h(x, h(r(x), f(r^2(x))))

= ...

= h(x, h(r(x), h(r^2(x), h(..., h(r^{k-1}(x),g(r^k(x)))...)

wobei k die kleinste natürliche Zahl für die P(r^k(x))=true.

Transformation linear rekursiver Funktionen in iterative nicht-rekursive Funktionen

iteratives Programmfragment zur Implementierung einer linear rekursiven Funktion f(x)

```
stack s ;
while ( !P(x) )
{
      s.push(x)  ;
      x = r(x) ;
}
f = g(x) ;
while (!s.empty())
{
    x = s.top() ;
    s.pop() ;
    f = h(x,f) ;
}
```

stack s
Stapel mit Operationen
push(x): legt x oben auf den Stapel
pop(): entfernt oberstes Element
top(): liefert oberstes Element (ohne es zu entfernen)

Manche linear rekursiven Funktionen erlauben Transformation in endrekursive Funktion

Theorem :

Sei f linear rekursive Funktion mit einer Funktion h,

die assoziative Operation mit Linkseinheit ist, d.h.

i) $\forall (x, y, z).\ h(x, h(y, z)) = h(h(x, y), z)$

ii) $\exists e, \forall x\ .\ h(e, x) = x$

sei $f\_endrec(x, a) = \begin{cases} h(a, g(x)) & \text{falls } P(x) \\ f\_endrec(r(x), h(a, x)), & \text{sonst} \end{cases}$

für alle a, x, für die f(x) terminiert, ist $f\_endrec(x, a) = h(a, (f(x))$

und für $a = e$ gilt dann $f(x) = f\_endrec(x, e)$.

- Beweis durch vollst. Induktion
- Bsp: fakultaet mit h(x,y)=x*y, e=1
- praktisch, weil endrekursive Funktion ohne Stack iterativ implementiert werden können

$$fakt(n) = \begin{cases} 1, & \text{falls } n = 0 \\ n \cdot fakt(n-1), & \text{sonst} \end{cases}$$

Beweis durch vollst. Induktion

Hilfsbeh.: Für alle a,x für die f(x) terminiert gilt
f_endrec(x,a)=h(a,f(x))

Beweis: Wenn f(x) terminiert, existiert n mit $P(r^n(x))$

Induktion über kleinstes k für das $P(r^k(x))$ gilt.

- Falls k=0

  gilt P(x), Behauptung trivial

- Sei k=n+1, dann gilt !P(x) und $P(r^k(x))$ also $P(r^n(r(x)))$

  nach Induktionsvoraussetzung mit Ersetzung von x durch r(x) und a durch h(a,x) gilt f_endrec(r(x),h(a,x))=h(h(a,x),f(r(x)))

  Damit

  f_endrec(x,a)=f_endrec(r(x),h(a,x)) wg. not P(x)

  = h(h(a,x),f(r(x))) wg Ind.-voraussetzung

  = h(a,h(x,f(r(x)))) wg Assoziativität von h

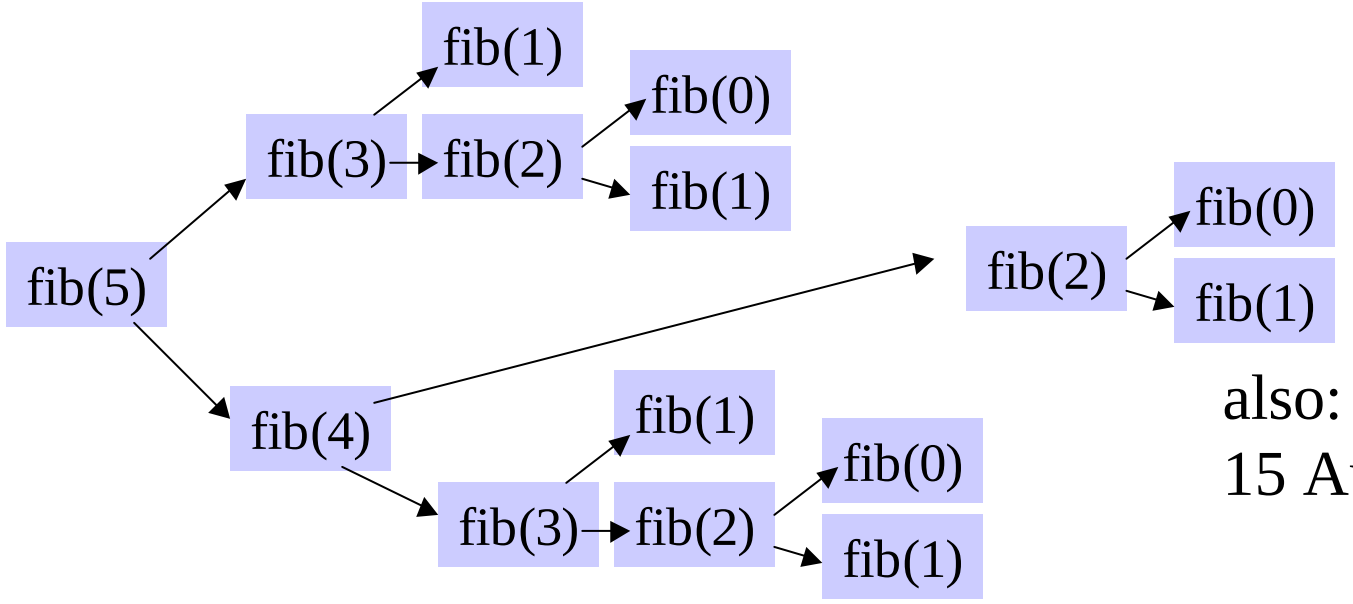
  = h(a,f(x)) nach Def f(x)

  mit a=e folgt f(x)=h(e,f(x))=f_endrec(x,e)

Fibonacci-Zahlen

- Fibonacci Zahlen
  - angeblich Kaninchenpopulation nach n Jahren, u.d. Annahme dass ein- und zweijährige genau 1 Nachkommen pro Jahr zeugen
  - allgemein rekursive Funktion dank 2 Aufrufen von fib()

$$fib(n) = \begin{cases} 1, & \text{falls } n <= 1 \\ fib(n-2) + fib(n-1), & \text{sonst} \end{cases}$$



also:
15 Aufrufe von fib

Fibonacci-Zahlen und akkumulierende Parameter

- noch ein Trick zur Programmtransformation:
  akkumulierende Parameter bewirken endrekursive Form
- Beobachtung:
  in fib() werden viele Teilergebnisse mehrfach berechnet.
- Generelle Technik: mehrfach verwendete Resultate sollten gespeichert werden, in Variablen, hier spezielle Parameter.
- acc1, acc2 speichern Werte für fib(k-2), fib(k-1)

```
int fibo_acc(int n, int acc1, int acc2)
{
    if  (n=0) return(acc1) ;
   else return(fibo_acc(n-1,acc2,acc1+acc2)) ;
}
int fib2(int n) { return(fibo_acc(n,1,1)) ; }
```

Bsp: fib2(5) = fibo_acc(5,1,1) = fibo_acc(4,1,2) = fibo_acc(3,2,3) = fibo_acc(2,3,5) = fibo_acc(1,5,8) = fibo_acc(0,8,13)=8

also 6 statt 15 rekursiver Funktionsaufrufe !!!!

Stimmt das auch ?

$$fib(n) = \begin{cases} 1, & \text{falls } n <= 1 \\ fib(n-2) + fib(n-1), & \text{sonst} \end{cases}$$

$$fibacc(n, a_1, a_2) = \begin{cases} a_1, & \text{falls } n = 0 \\ fibacc(n-1, a_2, a_1 + a_2), & \text{sonst} \end{cases}$$

Behauptung: fib(n)=fibacc(n,1,1)

Behauptung2: $\forall k, n \in N_0, k \leq n$ gilt $fibacc(k, fib(n-k), fib(n-k+1)) = fib(n)$

Beweis mit vollst. Induktion, Behauptung2 für n=k => Behauptung

Ind.anfang: k=0, fibacc(0,fib(n),fib(n+1))=fib(n) nach Def.

Ind.schritt: k=r+1:

fibacc(r+1, fib(n-r-1), fib(n-r))	eingesetzt
= fibacc(r,fib(n-r), fib(n-r-1)+fib(n-r))	nach Def fibacc
= fibacc(r, fib(n-r), fib(n-r+1))	nach Def fib
= fib(n)	nach Ind.annahme

Für n=k: fibacc(n,fib(n-n),fib(n-n+1))=fibacc(n,1,1)=fib(n)

Aspekte rekursiver Funktionen

- Berechnungsaufwand
  - entsteht durch die Stackverwaltung bei Aufruf / Terminierung einer Funktion
  - durch die Berechnung innerhalb der Funktion
- Speicherplatzbedarf
  - durch den einzelnen Funktionsaufruf auf dem Stack:
    - je Aufruf alle lokalen Variable und Aufrufparameter
  - Rekursionstiefe: Anzahl der Funktionsaufrufe bis Rekursion endet
- Terminierung:
  - erfordert Argumentation, warum Rekursionanker tatsächlich erreicht wird
- Korrektheit:
  - erfordert Argumentation, warum Resultat stimmt
  - bei rekursiven Problemstellungen oft formal über vollst. Induktion zu erbringen
- Rekursion erlaubt oft sehr abstrakte und elegante Formulierung einer Lösung

  => Denken Sie an Descartes rationale Mittel

Zusammenfassung

- Unterprogramme
  - Prozeduren, Funktionen, Methoden
  - Kommunikation zwischen Haupt- und Unterprogramm
    - call by value, call by reference
  - Top Down Entwurf: funktionale Zerlegung von Problemen
- Rekursion
  - primitiv rekursive Funktionen
  - endrekursive Funktionen
  - linear rekursive Funktionen
  - Ausführung rekursiver Funktionen: Aufrufstack
  - Transformation von rekursiven in iterative Funktionen
    - explizite Programmierung des Stacks für Parameter- und Zwischenwerte
    - Transformation endrekursiver Funktionen in iterative Funktionen
    - Transformation von rekursiven Funktionen in endrekursive Funktionen mittels akkumulierenden Variablen