Grundlagen der Programmierung (Vorlesung 15)

Ralf Möller, FH-Wedel

- Vorige Vorlesung
  - Blöcke, Funktionen
  - Auswertestrategien
- Inhalt dieser Vorlesung
  - Prozeduren
  - Rekursion
- Lernziele
  - Grundlagen der systematischen Programmentwicklung

Funktionsauswertung: Beispiel

- even(i : N_0) : B

```
if i = 0
   then true
   else odd( i - 1 )
end if
```

- odd(i : N_0) : B

```
if i = 0
   then false
   else if i = 1
           then true
           else even( i - 1 )
        end if
end if
```

Vereinfachung der Auswertung

- Das Ergebnis des Aufrufs von even(3) ist direkt gegeben aus dem Aufruf von odd(2)
- Mit anderen Worten: Das Ergebnis von odd(2) wird nicht weiter verrechnet, sondern direkt als Wert von even(3) zurückgegeben (Tail-Call)
- Die lokalen Variablen für even(3) können vor dem Auswerten von odd(2) eliminiert werden
- Der auszuwertende Term verkleinert sich dann entsprechend

Prozeduren: Schreibweise (am Beispiel)

- var f : array [0 .. n-1] of N_0
- vertausche(i : N_0 ; j : N_0)

```
begin
   var x : N0 ;
   x := f[j] ;
   f[j] := f[i] ;
   f[i] := x
end
```

Prozeduren

- Auswertung wie bei Funktionen über Blockbildung
- Aber: Keine Werteberechnung
- Sondern: Erzeugung eines Seiteneffektes
  - Meist: Ausgabe von Texten (print)
  - Auch: Referenzierung von globalen Variablen

Prozeduren: Vorbedingung und Nachbedingungen

- Behandlung einfach, da Auswertestrategie und Parameterübergabe über Blockbildung mit Variablenumbenennung und textueller Ersetzung erklärt

Funktionsauswertung: 1. Beispiel

- var a : array [0..n-1] of N_0;
- sum(i : N_0) : N_0

```
  if i > n-1
    then 0
    else a[i] + sum(i+1)
  end if
```

- Was ist das Ergebnis von sum(0) ?
- Wie erfolgt die Auswertung?
  - Beispiel für n = 3 siehe Tafel
- sum ruft sich selbst auf: sum ist rekursiv

Rekursion: Zentrale Idee

(1)	für einige wenige einfache Parameter (Argumente) ist das Resultat direkt (ohne Rekursion) berechenbar
(2)	für die komplexeren Parameterwerte wird die Berechnung auf einfache Werte zurückgeführt
Abbrechen	die Rekursion muß abbrechen, d.h. die rekursiven Aufrufe müssen immer mit einfacheren Werte ausgeführt werden
einfache Werte	was einfache Werte sind, hängt vom Problem ab

kleine Zahlen	große Zahlen
kurze Listen	lange Listen
einfache Ausdrücke	geschachtelte Ausdrücke

Rekursion: Beispiel

- var a : array [0..n-1] of N_0;
- sum(i : N_0) : N_0

```
if i > n-1
  then 0
  else a[i] + sum(i+1)
end if
```

Rekursionsformen: Endrekursion

- var a : array [0..n-1] of N_0;
- sum'(i : N_0, acc: N_0) : N_0
  if i > n-1
    then acc
    else sum'(i+1, a[i] + acc)
  end if
- sum(j : N_0) : N_0
  sum'(j, 0)
- sum(0)

Akkumulator

Endrekursion ist Spezialfall des Tail-Call-Prinzips

Umwandlung von Endrekursion in eine Schleife

```
sum(j : N0) : N0
    sum'(j, 0)

sum'(i, acc: N0) : N0
    if i > n-1
      then acc
      else sum'(i+1, a[i] + acc)
    end if
```

"Rekursives Programm"

```
sum(i : N0) : N0
 begin
   var acc : N0 ;
   acc := 0 ;
   while ¬(i > n-1)
     acc := a[i] + acc ;
     i := i+1
   end while ;
   acc
 end
```

"Iteratives Programm"

Was soll die Schleifenumwandlung?

- Wir haben gesehen, daß die Programmierung mit Schleifen inhärent komplex bezüglich der Vor- und Nachbedingungen ist
- Rekursive Programme sind nahe an der Problemspezifikation, die Überprüfung der Korrekteheit bzgl. Vor- und Nachbedindungen ist wesenlich einfacher
- Strategie: Entwickle rekursives Programm und transformiere dieses in eine While-Schleife
- Das Transformationsschema sichert die Korrektheit des While-Programmes zu
- Die Transformation kann durch Compiler geschehen!

Rekursion ist einfacher zu verstehen!

Problem	Geld wechseln Wieviele verschiedene Möglichkeiten gibt es, 1,- DM zu wechseln mit Fünfzig-, Zehn-, Fünf-, Zwei-, und Einpfennigstücken?
allgemein	Können wir einen Algorithmus schreiben, der die Anzahl der Möglichkeiten zum Wechseln eines beliebigen Geldbetrages berechnet?
Lösung	rekursiv

Beispiel: Geldwechsel (1)

# **gesamt**	den Betrag a mit n verschiedenen Münzen zu wechseln
$=$	
#	Möglichkeiten, den Betrag a mit allen außer der 1. Münzart zu wechseln
$+$	
#	Möglichkeiten, den Betrag $a - d$ mit allen n Münzarten zu wechseln, wobei d der Nennwert der 1. Münzart ist.

Beispiel: Geldwechsel (2)

elementare Fälle	Wann kennen wir das Ergebnis ohne weitere Berechnungen?
$a = 0$	genau eine (1) Wechselmöglichkeit
$a < 0$	keine (0) Wechselmöglichkeiten
$n = 0$	keine (0) Wechselmöglichkeiten

Rekursionsformen: lineare Rekursion

- var a : array [0..n-1] of N_0;
- sum(i : N_0) : N_0

```
if i > n-1
  then 0
  else a[i] + sum(i+1)
end if
```

- Berechnungsergebnis des rekursiven Aufrufs wird noch weiter verrechnet (kein Tail-Call)

Umwandlung von linearer Rekursion in Endrekursion

- Akkumulatortechnik
- Siehe Beispiel auf Folie zur Endrekursion
- Nach Erstellung eines endrekursiven Programmes wird die Umwandlung in ein iteratives Programm offensichtlich

Rekursionsformen: Baumrekursion

- fib(n : N_0) : N_0

```
if n = 0
   then 1
   else if n = 1
            then fib(n-1) + 1
            else fib(n-1) + fib(n-2)
        end if
end if
```

- In diesem Spezialfall:
  Umwandlung in iteratives Programm trivial

Rekursionsformen: Baumrekursion

```
hanoi (n : N0; von, nach, über : N0)
  if n = 0
    then print("Bewege oberste Scheibe von Säule
                        ~D nach Säule ~D.",
                       von, nach)
    else hanoi(n-1, von, über, nach);
         hanoi(0, von, nach, über);
         hanoi(n-1, über, nach, von)
  end if
```

- Umwandlung in iteratives Programm nicht trivial
- Es gibt allerdings eine einfache iterative Lösung

5.3 Parallelität

Konstrukte	Folge, Auswahl und Wiederholung werden sequentiell ausgeführt,
↪	Auswertung von Ausdrucken und aktuellen Parametern werden sequentiell ausgeführt
Vorstellung	ein Prozessor führt genau einen Algorithmus Schritt für Schritt hintereinander aus
	Viele Algorithmen enthalten Anweisungen und Ausdrücke, die unabhängig voneinander ausgeführt werden können, deren Berechnungsreihenfolge beliebig ist
↪	diese Algorithmen sind geeignet, von Mehrprozessormaschinen ausgeführt zu werden
↪	viele Prozessoren ⇒viel gleichzeitig ⇒Zeitgewinn

Parallelität und Rekursion

teile & herrsche	rekursive Algorithmen, die ein Problem in mehrere einfache Probleme dergleichen Art aufteilen, enthalten oft unabhängige Berechnungsschritte, die unabhängig voneinander (gleichzeitig) ausgewertet werden können.
Beispiele	Fibonacci, sortieren und mischen
nicht	Türme von Hanoi

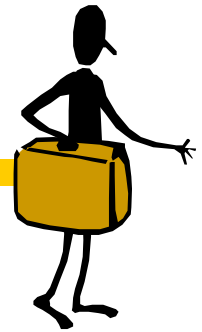
5.4 Zusammenfassung

Algorithmen–Konstruktion

Sequenz

Verzweigung	if ... then ... else ... end if
Wiederholung	while ... do ... end while for ... to ... do ... end for
Teilalgorithmen	Prozeduren und Funktionen mit Parametern
Rekursion	durch Wiederverwendung des gleichen Algorithmus mit *einfacheren* Werten
Parallelität	durch gleichzeitiges Ausführen unabhängiger Teile Beispiel: teilen und herrschen

Zusammenfassung, Kernpunkte



- Funktionen und Prozeduren
- Rekursionsformen
- Automatische Umwandlung von rekursiven Programmen in iterative Programme

Was kommt beim nächsten Mal?

- Algorithmen
- Betriebsmittel, Aufwand
- Asymptotische Komplexität von Algorithmen