

Grundlagen der Programmierung (Vorlesung 14)

Ralf Möller, FH-Wedel

■ Vorige Vorlesung

- Verifikation von Anweisungen und Anweisungsfolgen
- Schleifen

■ Inhalt dieser Vorlesung

- Funktionen und Prozeduren

■ Lernziele

- Grundlagen der systematischen Programmentwicklung

Schleifen

Beweisregel

für while –Schleifen

falls

(1)

$\{I \wedge B\} \ S \ \{I\}$

dann gilt

$\{I\} \ \text{while } B \text{ do } S \text{ end while} \ \{I \wedge \neg B\}$

Schleifen (2)

Terminierung zusätzliche Bedingungen

falls

Invariante $\{I \wedge B\} \ S \ \{I\}$

Fortschritt $\{I \wedge B \wedge t > T\} \ S \ \{t = T\}$

Beschränkung $I \wedge t \leq 0 \Rightarrow \neg B$

dann gilt die Nachbedingung $I \wedge \neg B$ und die Schleife terminiert:

$\{I\} \ \text{while } B \text{ do } S \text{ end while} \ \{I \wedge \neg B\}$

Variante t ist eine ganzzahlige Funktion
 T ist eine Konstante

Ganzzahlige Division mit Rest mittels While

■ Spezifikation

■ $\text{var } x, y, q, r : \mathbb{N}_0$

■ $\{y > 0\} \ S \ \{q * y + r = x \wedge r < y\}$

■ $V \qquad P$

■ $P: \underbrace{q * y + r = x}_I \wedge \underbrace{r < y}_{\neg B}$

■ $I \qquad \neg B$

■ Gut: Spezifikation schon in der richtigen Form

■ $\{V\} S_0 \{I\};$

■ $\{I\} \text{ while } B \text{ do } \{I \wedge B\} S_1 \{I\} \text{ end while } \{P\}$

Ganzzahlige Division mit Rest mittels While

```
var x, y, q, r : N0
```

■ $\{y > 0\} \quad q, r := 0, x \{q * y + r = x\}$
V So I

B

- while $r \geq y$ do

■ $\{q * y + r = x \wedge r \geq y\} \quad q, r := q+1, r-y \quad \{q * y + r = x\}$

```

end while

```

- $\{q^* y + r = x \quad \wedge \quad r < y\}$

P

$$I \wedge \neg B$$

Terminierungsbeweis

- Für die Variante t wähle: r
- Zeige Fortschritt:
 - $\{I \wedge B \wedge t > T\} S \{t = T\}$ muß korrekt sein, d.h.
 $\{I \wedge B \wedge r > T\} q, r := q+1, r-y \{r = T\}$ muß korrekt sein
 - | Also $r \sqcap r-y$ in Nachbedingung einsetzen
 - | Damit ergibt sich $r-y = T$
 - | Es ergibt sich: $r > T$, wenn $y > 0$
 - $(I \wedge t \sqcap 0) \rightarrow \neg B$ muß gültig sein, d.h.
 $(I \wedge r \sqcap 0) \rightarrow r < y$ muß gültig sein
 - | Unter der Voraussetzung, daß $r \sqcap 0$ und $y > 0$ ist $r < y$ immer erfüllt

repeat –Schleife

aus einer Bedingung und einer Anweisung kann
eine repeat –Schleife aufgebaut werden

Syntax





repeat
 Anweisung
until *Bedingung*

Semantik

mit Hilfe der while –Schleife
 Anweisung;
while \neg *Bedingung* do
 Anweisung
end while

Semantik

informell

- (1) die Anweisung (der Schleifenrumpf) wird ausgeführt
- (2) die Bedingung wird ausgewertet
- (3a) ist das Resultat `true`, so wird die Ausführung des Programms hinter der Schleife fortgesetzt
- (3b) ist das Resultat `false`, so wird die Ausführung der Schleife wiederholt
-  der Schleifenrumpf wird 1, 2, ... mal ausgeführt.
-  nicht für Schleifen zu gebrauchen, deren Rumpf möglicherweise nicht ausgeführt wird.
-  `while` –Schleifen allgemeiner
-  `repeat` –Schleife **nicht** effizienter als `while` –Schleife

for –Schleife

Zählschleife

Syntax

for $Variable := Ausdruck_1$ to $Ausdruck_2$ do
 $Anweisung$
end for

$Variable$

ist die Laufvariable

$Ausdruck_1$

ist der Startwert

$Ausdruck_2$

ist der Endwert

$Anweisung$

ist der Schleifenrumpf

Semantik

Bedeutung: erklärt mit einer while –Schleife

$Variable := Ausdruck_1;$
 $Endwert := Ausdruck_2;$
while $Variable \leq Endwert$ do
 $Anweisung;$
 $Variable := Variable + 1$
end while



while –Schleifen allgemeiner

5 Funktionen und Prozeduren

5.1 Modularität

Einzelteile	der Algorithmen unabhängig vom Algorithmus selbst
\hookrightarrow	unabhängig entwickelbar
\hookrightarrow	an verschiedenen Stellen einsetzbar im selben Algorithmus in verschiedenen Algorithmen
\leftrightarrow	Teile betrachten wie neue Elementaroperationen

Algorithmus

quadratsumme(x,y) $\text{quadrat}(x) + \text{quadrat}(y)$

Benutzung `quadratsumme(3,4)`

Name des Algorithmus: **quadratsumme**

formale

Parameter **x** und **y**

Aufruf `quadratsumme(3,4)`

aktuelle

Parameter **3** und **4**

Funktionen sind Algorithmen, die einen Wert berechnen

Prozeduren sind Algorithmen, die einen Zustand (globale Variablen) verändern

Vorteile	von Algorithmen mit Parametern (Module, Prozeduren und Funktionen)
Schrittweise Verfeinerung	mit top–down Vorgehen wird in natürlicher Weise unterstützt
Abgeschlossenheit	Ein Teilalgorithmus kann unabhängig von dem Kontext, in dem er verwendet wird, entwickelt werden
information hiding	ein Modul braucht nur die Schnittstelle und eine Beschreibung was der Modul macht, nach außen bekannt zu machen. Wie ein Modul eine Funktion berechnet (mit welchem Verfahren), wird versteckt

Urbildbereich (Parameter) und Bildbereich (Ergebnistyp)

■ Beispiel:

- Funktion \sin

- Urbildbereich: $\mathbb{R} \rightarrow \text{Parameter}$

- Bildbereich \mathbb{R} , genauer: Intervall $[-1, 1] \rightarrow \text{Typ}$

■ Mehr als ein Parameter möglich

■ Stelligkeit einer Funktion

- Anzahl der Parameter

Funktionen: Schreibweise

- ## ■ Erklärung anhand von Beispielen

■ `quadrat(x : N0) : N0`

- `x * x` — Ausdruck
- `:` — (formaler) Parameter
- `:` — Parametertrenner
- `N0` — Parametertyp
- `N0` — Ergebnistyp

■ `quadratsumme(a : N0 ; b : N0) : N0`
`quadrat(a) + quadrat(b)`
 Funktionsaufruf
 mit Aktualparameter

Vereinbarung: abkürzende Schreibweise

- $\text{quadratsumme}(a, b : N_0) : N_0$
 $\text{quadrat}(a) + \text{quadrat}(b)$
- steht für
- $\text{quadratsumme}(a : N_0 ; b : N_0) : N_0$
 $\text{quadrat}(a) + \text{quadrat}(b)$

Vorbedingungen und Nachbedingungen

- Zum Teil Vorbedingungen als Typen der Parameter formulierbar
- Parametertypen legen die Vorbedingungen meist nur partiell fest
- Problembeispiel: Division durch 0
- $\text{div}(x : N_0 ; y : N_0) : N_0$
- Gleiches gilt für die Ergebnistypen

Funktionsrumpf

- Folge von Anweisungen
- mit "abschließendem" Ausdruck
- $\text{min}(x : N_0 ; y : N_0) : N_0$
 if $x > y$
 then y
 else x
 end if

Funktionen: Motivation

Auswechselbarkeit von Teil-Algorithmen ohne Veränderung der Funktionalität des Gesamtalgorithmus



Wartbarkeit Veränderbarkeit, Erweiterbarkeit, Anpassung

Wiederverwendung Teilalgorithmen können in einer *Bibliothek (library)* von Algorithmen gespeichert und wiederverwendet werden

Information Hiding (Kapselung) (1)

Information verstecken ist eine wesentliche Aufgabe der Zerlegung in Teilalgorithmen



es wird nach außen nur die Schnittstelle des Algorithmus (die formalen Parameter und das Resultat) bekanntgemacht



einschließlich der Vorbedingungen

Welche Annahmen werden über die Parameter gemacht?

Information Hiding (Kapselung) (2)



einschließlich der Invarianten

Welche globalen Variablen werden nicht verändert?



einschließlich der Nachbedingungen

Welche Eigenschaften haben die veränderten globalen Variablen?



die Realisierung (Implementierung) bleibt nach außen hin unbekannt (versteckt) und kann so verändert oder ausgewechselt werden

■ Noch zu klären: Semantik des Funktionsaufrufs

Zuweisung und Arrays: Ergänzung

- Arraydenotation über Tupel
 - $\text{var } f : \text{array } [0..2] \text{ of } N_0; f := (42, 17, 9)$
- Zuweisung von Arrays: Kopiersemantik
 - $\text{var } f, g : \text{array } [0..2] \text{ of } N_0;$
 - $f := (42, 17, 9);$
 - $g := f; \dots$
- Arrays als Werte von Funktionen (am Beispiel)
 - $f(i : N_0) : \text{array } [0..1] \text{ of } N_0$
 $(i+1, i-1)$
- Erweiterung der Zuweisung für Arrays
 - $\text{var } a : \text{array } [0..1] \text{ of } N_0; \quad a := f(3)$
 - $\text{var } x, y : N_0; \quad (x, y) := f(3)$

Auswertestrategien

nicht strikte Auswertung

Boolescher Ausdrücke :

Konjunktionen und Disjunktionen werden nur so weit ausgewertet, bis das Resultat bekannt ist.

\leftrightarrow

das Resultat ist auch dann definiert, wenn die Auswertung eines Teilausdrucks nicht definiert ist

$x \wedge y$

\Leftrightarrow if x then y else false

\Leftrightarrow if x then y else x

$x \vee y$

\Leftrightarrow if x then true else y

\Leftrightarrow if x then x else y

Lokale Variablen, Blöcke

- Bisher: Variablen "global" am Anfang eines Algorithmus vereinbart
- Nun: Variablen "lokal" nur für den Teilalgorithmus einer deklarierten Prozedur
- Notation:
 - ```
begin
```

      /      Kopf des Blockes mit lokalen Variablen  

```
var ;
```

```
....
```

      —      Rumpf des Blockes (Sequenz von Anweisungen)  

```
end
```
- Blöcke können Anweisung oder Ausdruck sein

# Blöcke: Freie und gebundene Variablen

---

- Variablen, die im Kopf eines Blockes aufgeführt sind, heißen gebunden (bzgl. eines Blockes)
- Variablen, die im Rumpf vorkommen, aber nicht gebunden sind, heißen frei (bzgl. eines Blockes)



# Auswertestrategie für Blöcke

---

- Vor der Auswertung wird ein Block transformiert
- Für die lokalen Variablen werden neue, in der Auswertereihenfolge bisher nicht benutzte Namen vergeben
- Anschließend werden, die in den Anweisungen im Rumpf des Blockes verwendeten lokalen Variablen durch die entsprechenden neuen Variablen ersetzt
- Ist die bei dieser Ersetzung betrachtete Anweisung wieder ein Block, so werden die freien Variablen ersetzt

# Funktionen: Auswertestrategie (1)

- Sei  $f$  definiert als  $f(x_1: T_1, \dots, x_n: T_n) : T_{\text{result}}$
- $\langle \text{Rumpf von } f \rangle$
- Ein Funktionsaufruf  $f(a_1, \dots, a_n)$  einer  $n$ -stelligen Funktion  $f$  mit Aktualparametern  $a_1, \dots, a_n$  wird ausgewertet, indem aus den (formalen) Parametern und dem Rumpf der Funktion  $f$  folgender Block gebildet wird
  - begin
    - var  $x_1: T_1; x_2: T_2; \dots; x_n: T_n;$
    - $\langle \text{Rumpf von } f \rangle$
  - end
- Anschließend wird der Block wie oben beschrieben zur Auswertung transformiert

# Funktionen: Auswertestrategie (2)

---

- Es entsteht ein Block

- begin  
var  $x_1' : T_1; x_2' : T_2; \dots; x_n' : T_n;$   
    <Rumpf von  $f$  mit Ersetzungen>  
end

- Der Block wird wie folgt umgeformt und ausgewertet

- begin  
var  $x_1' : T_1; x_2' : T_2; \dots; x_n' : T_n;$   
     $x_1', x_2', \dots, x_n' := a_1', \dots, a_n';$   
    <Rumpf von  $f$  mit Ersetzungen>  
end

- wobei  $a_1', \dots, a_n'$  die Ergebnisse der entsprechenden Auswertung von  $a_1, \dots, a_n$  sind

# Parameter, Sichtbarkeit, Lebensdauer

---

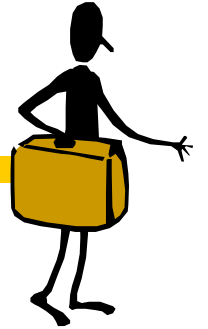
- Der oben skizzierte Übergabemechanismus heißt Call-by-value
- Wertezuweisungen an Parameter sind möglich, da Parameter wie Blockvariablen mit Initialisierung zu behandeln sind
- Transformation garantiert Sichtbarkeitsregeln (Variablenskopus)
- Was passiert bei "Neueintritt" in den Block?
  - "Alte" Namen werden nicht neu vergeben
  - "Alte" Werte gehen also verloren

## Ausdrücke vs. Anweisungen (Beispiel 2)

---

```
■ amittel(g : array [0..n-1] of N0) : N0
begin
 var j, am : N0 ;
 j, am := 0, 0;
 while j < n do
 j, am := j + 1, ((j * am) + g[j]) / (j+1)
 end while;
 am
end
```

# Zusammenfassung, Kernpunkte



## ■ Blöcke

- Auswertungsstrategie,
- Sichtbarkeit (Skopus)
- Lebensdauer

## ■ Funktionen

- Aktual- vs. Formalparameter
- Auswertungsstrategie

# Was kommt beim nächsten Mal?



- Prozeduren
- Rekursion