1.3 Anweisungen

Einfache Anweisungen

(Elementaranweisungen, simple/primitive statements)

Zusammengesetzte Anweisungen

(composite statements, control structures)

Syntax:

Statement: SimpleStatement

CompositeStatement

Semantik:

Die Menge der Variablen eines Programms bestimmt den

Zustandsraum eines Programms.

Eine bestimmte Belegung der Variablen definiert einen Zustand.

Formal:

Zustand: Variablen → Werte (partielle Abbildung)

Die Ausführung einer Anweisung bewirkt als

Effekt: einen geänderten Zustand,

d.h. manche Variablen erhalten neue Werte.

Es gibt verschiedene Möglichkeiten für die Festlegung der Semantik einer Anweisung:

operationell, d.h. durch Beschreibung der ausgeführten Aktionen, meist umgangssprachlich, z.B.

i++; erhöht i um 1;

axiomatisch, d.h. durch Aussagen/Prädikate über die Zustände vor bzw. nach der Ausführung, z.B.

$$\{P_{i+1}^i\}$$
 i++; $\{P\}$ *;

denotationell, d.h. durch Funktion : Zustand \rightarrow Zustand , z.B.

$$[[i++;]]: w \to w|_{w(i)+1}^{i} **$$
.

1.3.1 Einfache Anweisungen

SimpleStatement: EmptyStatement

ExpressionStatement

••••

EmptyStatement:

ExpressionStatement: StatementExpression;

StatementExpression: AssignmentExpression

IncrementExpression

DecrementExpression

••••

Leeranweisung (empty statement) hat keinen Effekt:

Zuweisung (assignment statement)
ersetzt den Wert der Variablen
durch den Wert des Ausdrucks:

$$\{P_{E}^{v}\}\ v = E; \{P\}^{*}$$

Achtung: In Modula,... ist ; Trennzeichen zwischen Anweisungen;

In Java,... ist ; letztes Zeichen einer Anweisung!

Mehrfachzuweisung (multiple assignment) ist oft praktisch:

$$(V1, V2, ..., Vn) := (E1, E2, ..., En)$$

z.B. für (x,y) := (y,x) (vgl. Musteranpassung in Haskell)

$$(x,x) := (x++,x--)$$

Zulassen? Welcher Effekt?

Verschiedene Präzisierungen der Semantik möglich!

... wird aber in imperativen Sprachen nur selten unterstützt, und auch **nicht in Java**.

Variablenmodifikation *, z.B.

neuer Wert der Variablen ergibt sich durch Anwendung eines dyadischen Operators

auf den alten Wert und den

Wert des Ausdrucks:

Variable += Expression;

 $\{P_{v+E}^{v}\}\ v += E; \{P\}$

Auch mit monadischen Inkrementoder Dekrement-Operatoren, z.B.

-- Variable;

 $\{P_{v-1}^{v}\} --v ; \{P\}$

Achtung: Pre/Post haben den gleichen Effekt; die Werte sind zwar verschieden, werden aber durch das ; "weggeworfen".

1.3.2 Blöcke

CompositeStatement: Block

CaseStatement

RepetitiveStatement

••••

Block: { { Declaration | Statement } }

CaseStatement: ConditionalStatement

SwitchStatement

RepetitiveStatement: WhileStatement

DoStatement

ForStatement

Verbundanweisung, Block (block) ohne Vereinbarungen:

führt die Anweisungen in der angegebenen Reihenfolge nacheinander aus

```
{ Statement1 Statement2 ..... }
```

(verallgemeinerbar für mehr als 2 Anweisungen)

In etlichen Sprachen gibt es nur *solche* Verbundanweisungen (ohne Vereinbarungen, in Modula z.B. mit <u>BEGIN END</u> geklammert).

In Java,... dagegen:

Java: Ein Block kann an beliebigen Stellen

Vereinbarungen (declarations)

von Variablen oder Konstanten enthalten.

In der vollständigen Java-Syntax wird das deutlich dadurch, dass eine solche Vereinbarung - irreführend -

LocalVariableDeclaration<u>Statement</u>

heißt. Dieses "Statement" hat lediglich den Effekt, eine Variable oder Konstante zu *vereinbaren* - evtl. aber auch die Variable zu *initialisieren*.

Variablen/Konstantenvereinbarung:

```
VariableDeclaration:
    [final] TypeIdentifier VariableDeclarators;

VariableDeclarators:
    VariableDeclarator { , VariableDeclarator }

VariableDeclarator:
    VariableIdentifier [ = Expression ]
```

... zuzüglich Kontextbedingung: keine Namenskollisionen

Vereinbarung einer Variablen legt Name und Typ fest, z.B.

Java,...: int x;

Modula,...: VAR x: INTEGER;

Typangabe legt statisch fest,

welche Werte die Variable dynamisch annehmen kann.

Vorteile:

right passende Speicherzuweisung und Codeerzeugung,

* statische Absicherung gegen manche Fehler,

■ Dokumentationseffekt

Achtung: Vor der ersten Zuweisung an eine nicht initialisierte Variable v ist w(v) undefiniert, d.h. v enthält keinen Wert, kann also nicht gelesen werden.

Dies wird statisch, d.h. durch Analyse des Programmtextes durch den Übersetzer gesichert, z.B.

```
double x, y, z = 0;
x = 3.14;
z = x + y + z;

Übersetzungsfehler
```

Blockstruktur (block structure)

Beachte: Die Syntax erlaubt die Schachtelung von Blöcken (nested blocks)

Die in einem Block vereinbarten Namen (und die so benannten Variablen, Konstanten, ...) heißen lokal (local) zu diesem Block und sind außerhalb des Blocks unbekannt.

Die in eventuellen umschließenden Blöcken vereinbarten Namen heißen nichtlokal zu diesem Block.

Der Gültigkeitsbereich (scope) einer Variablen, Konstanten, ...
erstreckt sich vom Ort seiner Vereinbarung ("defining occurrence",
im Gegensatz zu "applied occurrence") bis zum Ende des zugehörigen
Blocks. Der Wert einer Variablen geht verloren, wenn das Programm
den Block verlässt.

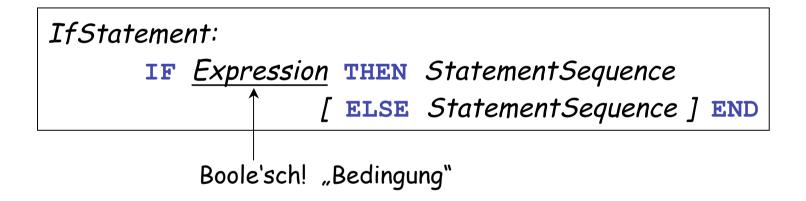
(In manchen Sprachen - nicht in Java:

- Der Sichtbarkeitsbereich einer Variablen/Konstanten ist eventuell kleiner als der Gültigkeitsbereich ihres Namens: durch Wiederverwendung des gleichen Namens in einem inneren Block wird die Variable/Konstante des äußeren Blocks verdeckt.
 - In Java stellt dies aber eine unzulässige Namenskollision dar.)

1.3.3 Fallunterscheidungen

Bedingte Anweisung (Alternative, conditional/if statement)

Wünschenswerte Syntax (Modula, vereinfacht):



führt die erste Anweisungsfolge aus, wenn die angegebene Bedingung erfüllt ist, andernfalls die zweite (sofern vorhanden).

Java:

```
ConditionalStatement:

if ( Expression ) Statement [else Statement ]

Boole'sch! "Bedingung" (condition C)
```

führt die erste Anweisung aus, wenn die angegebene Bedingung erfüllt ist, andernfalls die zweite (sofern vorhanden):

sofern die Auswertung von C erfolgreich ist und keine Nebenwirkungen hat

17

Achtung bei geschachtelten Alternativen:

```
if(b1) { x = y; if(b2) z = 0; } else x = 0;
if(b1) if(b2) z = 0; else x = 0;
```

Zu welchem if gehört das else?

Festlegung auf "zum nächsten if!" löst Problem der unbestimmten Zuordnung ("dangling else").

(Bei der Modula-Syntax ist diese Irritation von vornherein ausgeschlossen.)

```
Guter Programmierstil:
                               if (B1) S1
                               else if (B2) S2
                               else if (B3) S3
                               else if (Bn) Sn
                               else "must not happen"
       mit Bi ∧ Bk ≡ F für alle i,k
       und V_i Bi \equiv T
Prägnantere Schreibweise:
                           if B1 → S1
                               \square B2 \rightarrow S2
(Dijkstra 1976,
                               \blacksquare B3 \rightarrow S3
"", "guarded command")
                                 Bn → Sn
                               fi
```

Auswahlanweisung (case/switch statement)

Wünschenswerte Syntax (Modula, vereinfacht):

```
CaseStatement:

CASE Expression OF Case { | Case }

[ ELSE StatementSequence ] END

Case: CaseLabels : StatementSequence

CaseLabels: ConstantExpression { , ConstantExpression }
```

wertet den angegebenen Ausdruck aus und führt dann diejenige Anweisungsfolge aus, die den erhalten Wert als Marke trägt.

Java (vereinfacht):

```
SwitchStatement:
       switch (Expression ) SwitchBlock
SwitchBlock:
       { { CaseGroup } }
CaseGroup:
       SwitchLabel { SwitchLabel } StatementSequence
SwitchLabel:
      case ConstantExpression:
      default:
```

mit folgenden Kontextbedingungen:

Kontextbedingungen:

X Der Expression muss vom Typ int, short, byte oder char sein.

X Die Typen der ConstantExpressions, die die einzelnen Fälle markieren, müssen mit diesem Typ verträglich sein;

X ihre Werte müssen paarweise verschieden sein.

X default darf höchsten einmal vorkommen.

Semantik:

- ① Nachdem der Wert des Expression ermittelt wurde, wird mit der Ausführung bei derjenigen Anweisung fortgefahren, die mit diesem Wert markiert ist.
- ② Wenn der Wert nicht als Marke vorkommt, wird bei default fortgefahren.
- 3 Wenn auch default nicht als Marke vorkommt, passiert nichts.

! Achtung

Achtung: Genau genommen handelt es sich gar nicht

um eine Fallunterscheidung, sondern um eine

Sprungleiste: (computed "go to") die Ausführung macht einen

Sprung zur jeweils gewählten Marke. *

Den Effekt einer echten Fallunterscheidung erreicht man, indem man jede Anweisungsfolge mit einer Anweisung abschließt, die die gesamte Auswahlanweisung abbricht:

break;

(d.i. eine weitere Elementaranweisung, die später behandelt wird)

[Variante des SwitchStatement in C#:

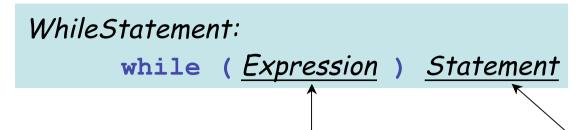
break; (oder eine ähnliche Anweisung)
muss in jeder Anweisungsfolge vorkommen.]

1.3.4 Schleifen

Schleife (Wiederholungsanweisung, repetitive statement, loop) veranlasst die wiederholte Ausführung einer Anweisung(sfolge)

Merke:

- ► Variable und Schleifen sind die beiden typischen Charakteristika imperativer Sprachen!
- Viele Probleme, die in der applikativen Programmierung rekursiv gelöst werden, löst man in der imperativen Programmierung iterativ, d.h. mit einer Schleife.
- → Die Termination ist bei Iteration genauso wenig gesichert wie bei der Rekursion: es drohen "nichtabbrechende Schleifen".



"pre-checked loop"

Boole'sch! "Bedingung" Schleifenrumpf (loop body)

Semantik:

- 1. Wenn die Bedingung nicht erfüllt ist, tue nichts;
- 2. sonst führe den Rumpf aus und fahre bei 1. fort.

I heißt Schleifeninvariante (loop invariant)

[In Modula mit schönerer Syntax

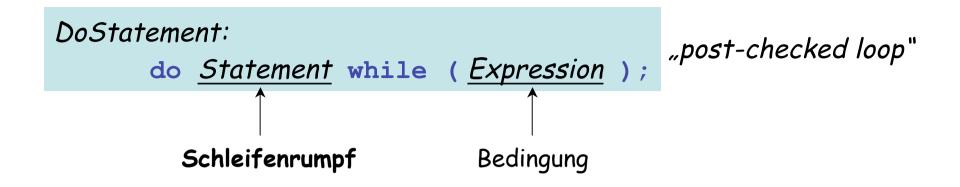
WHILE Expression DO StatementSequence END]

Beispiel: Bestimme kleinste Quadratzahl q oberhalb von n $(q, n \in N)$

```
int i = 1, q = 1;
while (q <= n) {
   i++;
   q = i*i; }</pre>
```

! Von der Korrektheit einer Schleife kann man sich gut mit Hilfe einer passenden Schleifeninvariante überzeugen, hier z.B. mit

$$(i-1)^2 \le n \land q = i^2 *$$



Semantik: wie bei S while (C) S

Achtung: Die Bedingung wird hier erst nach einmaliger

Ausführung des Rumpfes S geprüft - was Ursache

vieler Programmierfehler ist. Die while-Schleife

ist daher sicherer als die do-Schleife.

[In Modula mit schönerer Syntax

REPEAT StatementSequence UNTIL Expression]

Beispiel: größter gemeinsamer Teiler von $a,b \in N$:

```
int r; // Rest
do { r = a%b;
    a = b;
    b = r; }
while(r!=0);
// a == ggT
```

sieht gut aus - aber scheitert bei b=0! Besser ist

```
while (b!=0) {
    int r = a%b;
    a = b;
    b = r; }
// a == ggT
```

(liefert allerdings 0 falls a=b=0)

Laufanweisung (for statement):

```
ForStatement:
    for ( [Initialization]; [Expression]; [Update])
        Statement

Initialization:
    StatementExpression { , StatementExpression }
    VariableDeclaration

Update:
    StatementExpression { , StatementExpression }
```

mit Boole'schem Expression (Bedingung C)

Semantik:

wie bei I while (C) { SU}

(wobei man sich ; statt der Kommas denken muss)

Fehlendes C wirkt wie true.

Der Gültigkeitsbereich der vereinbarten Variablen beschränkt sich auf die Laufanweisung.

Bemerkungen:

- X Die Variablenvereinbarungen werden üblicherweise mit *Initialisierungen* verwendet.
- X Die Laufanweisung wird typischerweise dann eingesetzt, wenn beim Eintritt die maximale Anzahl der Durchläufe bereits feststeht.

Typische Formulierungen für unbegrenzte Wiederholung der selben Anweisung:

```
while(true) statement

do Statement while(true);

for(;;) statement z.B. for(;;);
```

Typische Anwendung der Laufanweisung:

Schleifenrumpf für alle Elemente einer Menge, Folge o.ä. ausführen Abstrakte Formulierung z.B. für eine Zahlenmenge myIntSet:

FOR ALL $x \in myIntSet$ DO ... statements using $x \dots END$

Dafür alternative Syntax in Java 1.5:

```
for(int x : myIntSet) {....}
```

for (Type Identifier: Expression) Statement

(Details später)

1.3.5 Sprünge

(auch "Sprunganweisung", jump statement, go to statement)

erlauben die willkürliche Fortsetzung der Programmausführung an einer anderen Stelle

- "Guter Sprung" (good go to)
 beendet vorzeitig eine zusammengesetzte Anweisung,
 d.h. setzt die Ausführung bei der nächsten Anweisung fort
- "Schlechter Sprung" (bad go to)
 setzt die Ausführung an einer explizit angegebenen
 Stelle des Programms fort

Statement: **SimpleStatement CompositeStatement** Label: Statement SimpleStatement: **EmptyStatement** ExpressionStatement **JumpStatement** \rightarrow Label: Identifier

JumpStatement: BreakStatement

ContinueStatement

GotoStatement

ReturnStatement

BreakStatement: break [Label];

ContinueStatement: continue [Label];

GotoStatement: goto Label;

ReturnStatement: return [Expression];

Label: Identifier

1.3.5.1 Abbruchanweisung

BreakStatement: break [Label];

break; beendet die direkt umschließende

Schleife bzw. Fallunterscheidung

(falls vorhanden, sonst Syntaxfehler

wegen nicht erfüllter Kontextbedingung)

Eine Anweisung kann mit einer Marke (label) versehen werden (oder auch mit mehreren - siehe obige Syntax); Markennamen kollidieren nicht mit gleichen Namen anderer Art, z.B. Variablennamen.

break label; beendet die nächste umschließende,
mit label markierte
Schleife bzw. Fallunterscheidung bzw. Block
(falls vorhanden, sonst Syntaxfehler
wegen nicht erfüllter Kontextbedingung)

1.3.5.2 Abbruch eines Schleifenrumpfs

ContinueStatement: continue[Label];

- beendet den direkt umschließenden Schleifenrumpf bzw.
- den Rumpf der nächsten umschließenden,
 mit Label versehenen Schleife

(jeweils falls vorhanden, sonst Syntaxfehler)

und beginnt (gegebenenfalls) mit der nächsten Iteration.

1.3.5.3 Wilde Sprünge mit Sprungbefehl goto

GotoStatement: goto Label;

- ◆ setzt Programmausführung bei der markierten Anweisung fort,
- erlaubt unkontrollierte Sprünge und damit "Spaghetti-Code", ist als bad go to verpönt und aus vielen Sprachen verbannt auch aus Java (nicht aber aus C#).

1.3.5.4 Rücksprung

aus einem Unterprogramm (→1.4) mittels

ReturnStatement: return [Expression];

... beendet das Unterprogramm und fährt an der Aufrufstelle fort.

Der Wert des angegebenen Ausdrucks wird als Ergebniswert geliefert.