3 Imperative Programmiersprachen

In diesem Kapitel betrachten wir wesentliche Aspekte imperativer Programmiersprachen, d.h. die Klasse der ältesten und am weiten verbreitetsten Programmiersprachen.

Imperative Programmiersprachen besitzen typischerweise die folgenden fünf Hauptaspekte, die den *Kern jeder imperativen Programmiersprache* bilden.

- Variablen und Zuweisungen
- Standard-Datentypen
- Kontrollabstraktionen
- Prozeduren
- Ausnahmebehandlung

Wir werden diese Aspekte im Folgenden nacheinander betrachten. Als Programmiersprache für konkrete Beispiele verwenden wir die Sprache Java, sofern dies nicht anders angegeben ist.

3.1 Variablen

Wie wir schon bei unserer einfachen imperative Programmiersprache im vorhergehenden Kapitel gesehen haben, bestehen imperative Programme im Kern aus Zuweisungen, die mittels Kontrollstrukturen in eine festgelegte Abfolge gebracht werden können. Eine **Variable** wird dabei durch einen Bezeichner identifiziert (**Variablenname**) und die Zuweisung verändert den zu dieser Variablen assoziierten Wert. Variablen können somit als Behälter (≈ Speicherzellen) angesehen werden, die Werte aufnehmen können und deren Werte auch **verändert** werden können.

Beispiel 3.1 (Variable)**.** Betrachten wir die folgende Variablendeklaration und Zuweisung in Java:

```
int n = 1;
n = 42;
```

Die erste Zeile führt eine neue Variable mit dem Bezeichner `n` und dem Wert `1` ein. Die zweite Zeile verändert den zu dieser Variablen zugeordneten Wert zu `42`.

In klassischen imperativen Programmiersprachen können also die Werte von Variablen ausgelesen und auch verändert werden. Da dies nicht in allen Programmiersprachen so ist (insbesondere nicht in funktionalen oder logischen Programmiersprachen), spricht man auch von **mutierbaren** Variablen.

3.1.1 Mutierbare und nicht mutierbare Variablen

Die Veränderung von Variablenwerten kann manchmal zu schwer überschaubaren Effekten führen, insbesondere bei der Veränderung globaler Variablen, die an verschiedenen Programmstellen verwendet werden, oder auch bei der Veränderung von Variablen in nebenläufigen Programmen, was leicht zu sog. "race conditions" führen kann, wodurch Programme ein nicht vorhersagbares Verhalten haben können. Aus diesem Grund schränken manche, gerade neuere Programmiersprachen, die Mutierbarkeit von Variablen zu Gunsten einer erhöhten Programmiersicherheit ein. Dies wollen wir hier kurz diskutieren, bevor wir uns den klassischen mutierbaren Variablen zuwenden.

Funktionale und logische Programmiersprachen, die später noch genauer erläutert werden, lassen keine Wertänderungen bei Variablen zu. Bei diesen Sprachen stehen, wie in der Mathematik, Variablen für unbekannte aber feste Werte. Dies vermeidet Seiteneffekte, wodurch flexiblere Auswertungsstrategien mit z.B. nebenläufiger oder paralleler Auswertung leicht unterstützt werden. Außerdem erleichtert dies die Verifikation von Programmen.

Auf Grund der möglichen Probleme mutierbarer Variablen, insbesondere bei nebenläufigen Programmen, unterscheiden manche Programmiersprachen zwischen mutierbaren und nicht mutierbaren Variablen.

Bei der Programmiersprache Scala[1], die Ideen der objektorientierten und funktionalen Programmierung miteinander kombiniert, gibt es unterschiedliche Deklarationen für Variablen, je nachdem, ob diese mutierbar sein sollen oder nicht. Eine nicht mutierbare Variable wird mit dem Schlüsselwort `val` deklariert, wie z.B.

```
val x = 42
```

Eine Zuweisung an diese Variable führt dann zu einem Fehler:

```
x = 99
...error: reassignment to val...
```

Mutierbare Variablen werden dagegen mit dem Schlüsselwort `var` deklariert, sodass der folgende Code fehlerlos abgearbeitet wird:

```
var y = 42
y = y + 1
```

Die Programmiersprache Rust[2], die von Mozilla-Entwicklern als „sicherere" Alternative

[1] `https://www.scala-lang.org`
[2] `https://www.rust-lang.org/`

zu C oder C++ entworfen wurde, unterscheidet ebenfalls zwischen mutierbaren und nicht mutierbaren Variablen. Eine unveränderbare Variable wird ähnlich zu funktionalen Sprachen mittels dem Schlüsselwort `let` eingeführt (ebenso wie bei Scala muss man keinen Typ angeben, da dieser automatisch inferiert wird):

```
let x = 42;
```

Eine Zuweisung, d.h. Veränderung dieser Variablen ist unzulässig:

```
x = x + 1;
...error: cannot assign twice to immutable variable
```

Dagegen ist eine Veränderung möglich, wenn die Variable explizit mit dem Schlüsselwort `mut` bei der Deklaration als mutierbar eingeführt wird:

```
let mut y = 42;
y = y + 1;
```

Die folgenden Anweisungen sind auch zulässig:

```
let z = 42;
let z = z + 1;
```

Hierbei wird allerdings die in der ersten Zeile eingeführte Variable `z` nicht verändert, sondern durch die zweite Zeile wird eine neue Variable deklariert, die als Wert `z+1` hat.

Das Konzept von mutierbaren und nicht mutierbaren Variablen wird bei Rust insbesondere benutzt, um eine sichere Speicherverwaltung ohne "garbage collection" zu unterstützen.

3.1.2 Operationales Modell imperativer Sprachen

Im Folgenden betrachten wir mutierbare Variablen und wollen für diese eine präzise Beschreibung angeben. Eine Variable ist also eine Einheit einer Programmiersprache, der gewisse Attribute zugeordnet sind. In einer getypten Programmiersprache sind dies auf jeden Fall der Typ und der Wert einer Variablen. Wegen der Möglichkeit, Werte zu verändern, ist es außerdem wichtig, *wo* diese Werte gespeichert sind, denn es ist möglich, dass mehrere Variablen auf den gleichen Speicherort verweisen, z.B. durch

- Referenzen auf gleiche Objekte mit Variablenattributen,
- Referenzparameter bei Prozeduraufrufen, oder
- Zeiger auf Variablen.

Diese Möglichkeiten erläutern wir später, aber wir müssen bei der Modellierung von Variablen darauf achten, dass wir zwischen dem Behälter und dem Wert in diesem Behälter unterscheiden, ansonsten könnten wir diese Aspekte nicht korrekt erklären.
Aus diesem Grund müssen wir für ein präzises operationales Modell für imperative Sprachen zwei Arten von Bindungen modellieren:

1. Name $\mapsto$ (Referenz,Typ): Diese Bindungen werden in einer **Umgebung** aufgesammelt.

2. Referenz $\mapsto$ Wert: Diese Bindungen werden in einem **Speicher** aufgesammelt.

Referenzen entsprechen also den Speicheradressen eines Computers (z.B. Zahlen $\mathbb{N}$), aber da wir von einer konkreten Implementierung abstrahieren, lassen wir die genaue Struktur von Referenzen offen (d.h. dies ist eine beliebige Menge). Somit modellieren diese Bindungen wie folgt:

- **Umgebung** E: Dies ist eine Liste von Paaren $x : (\mathit{ref}, \mathit{typ})$, wobei x ein Name ist (später werden auch noch anderen Arten von Bindungen hinzugefügt).

- **Speicher** M: Dies ist eine Abbildung

  $$\text{Referenzen} \to \mathit{Int} \cup \mathit{Bool} \cup \ldots$$

  wobei der Zielbereich die Menge aller möglichen Werte ist.

Die wichtigste Grundoperation auf einer Umgebung ist das *Auffinden von Bindungen*, das wir durch folgendes Inferenzsystem definieren (hierbei bezeichnet ";" die Listenkonkatenation):

Axiom

$$E; x : \beta \vdash^{\mathit{lookup}} x : \beta$$

Regel:

$$\frac{E \vdash^{\mathit{lookup}} x : \beta}{E; y : \gamma \vdash^{\mathit{lookup}} x : \beta} \qquad \text{falls } x \neq y$$

Hierdurch modellieren wir die Suche nach dem letzten Eintrag, d. h. die Umgebung wird als Stack verwaltet.

Imperative Sprachen basieren auf folgendem Verarbeitungsprinzip:

- **Deklarationen verändern die Umgebung**

- **Zuweisungen verändern den Speicher**

Um dieses Prinzip durch ein Inferenzsystem zu beschreiben, wählen wir die folgende **Basissprache zur Beschreibung von Deklarationen und Anweisungen**:

$$\langle E \mid M \rangle \; \alpha \; \langle E' \mid M' \rangle$$

Hierbei sind E, E' Umgebungen, M, M' Speicher und α ist eine Anweisung (Deklaration, Zuweisung, Kontrollstruktur, u.ä.). Falls ein solches Element der Basissprache ableitbar ist, interpretieren wir dies wie folgt:

> Durch Abarbeitung von α wird die Umgebung E in E' und der Speicher M in M' überführt.

Beispiel 3.2 (Deklarationen ganzzahliger Variablen in Java)**.**

$$\langle E \mid M\rangle \texttt{ int x } \langle E; x : (l, int) \mid M[l/0]\rangle \qquad \text{mit } l \in free(M)$$

Dieses Axiom ist intuitiv wie folgt zu interpretieren:

- Erzeuge eine *neue* Bindung für `x` mit Referenz l
- l („location“) ist eine freie Speicherzelle, d. h. die *Menge aller freien Speicherzellen* $free(M)$ ist wie folgt definiert:

$$free(M) = \{l \mid l \text{ ist Referenz} \wedge M(l) \text{ ist undefiniert}\}$$

- Initialisiere Wert von `x` mit dem Wert `0`

Analog funktioniert die Deklaration boolescher Variablen:

$$\langle E \mid M\rangle \texttt{ boolean x } \langle E; x : (l, boolean) \mid M[l/\texttt{false}]\rangle \qquad \text{mit } l \in free(M)$$

In diesen Regeln haben wir festgelegt, dass Variablenwerte initialisiert werden. Manche Programmiersprachen lassen diese Initialisierung offen, was aber problematisch ist, denn dadurch können sich Programme in mehreren Läufen nichtdeterministisch verhalten, fehlerhafte Zeiger existieren, und ähnliches. Um dies zu vermeiden, ist manchmal auch eine *Deklaration mit Initialisierung* möglich, die wir wie folgt beschreiben können:

$$\langle E \mid M\rangle \texttt{ int x = } n \ \langle E; x : (l, int) \mid M[l/n]\rangle$$

wobei n eine Zahl und $l \in free(M)$ ist.

Beim Umgang mit Variablen ist es wichtig zu unterscheiden zwischen

- der Referenz (Behälter, **L-Wert, l-value**), und
- dem aktuellen Wert (**R-Wert, r-value**)

der Variablen. Dies ist auch relevant bei der Parameterübergabe bei Prozeduren. L/R bezieht sich dabei auf die Seiten einer Zuweisung.

Da nicht alle Ausdrücke einen L-Wert haben, wohl aber einen R-Wert, benötigen wir unterschiedliche Inferenzsysteme zum Ausrechnen von L/R-Werten. Die Berechnung von R-Werten kann analog zur Wertberechnung von Ausdrücken (vgl. Kapitel 2.2.3) erfolgen,

wobei wir hier nur noch das Nachschlagen von Variablendeklarationen berücksichtigen:

$$\frac{E \vdash^{lookup} x : (l, \tau)}{\langle E \mid M \rangle \vdash^{R} x : M(l)}$$

$$\frac{}{\langle E \mid M \rangle \vdash^{R} n : n} \quad \text{falls } n \text{ Zahl}$$

$$\frac{\langle E \mid M \rangle \vdash^{R} e_1 : v_1 \quad \langle E \mid M \rangle \vdash^{R} e_2 : v_2}{\langle E \mid M \rangle \vdash^{R} e_1 \texttt{+} e_2 : v_1 + v_2}$$

$$\frac{\langle E \mid M \rangle \vdash^{R} e_1 : v_1 \quad \langle E \mid M \rangle \vdash^{R} e_2 : v_2}{\langle E \mid M \rangle \vdash^{R} e_1 \texttt{*} e_2 : v_1 \cdot v_2}$$

Die Berechnung von L-Werten könnte analog erfolgen, aber diese muss eingeschränkter sein. Z.B. hat `x + 3` *keinen L-Wert*, da dies nicht links bei einer Zuweisung stehen darf. Aus diesem Grund müssen wir die Berechnung von L-Werten anders definieren.

Zunächst einmal hat jede Variable einen L-Wert:

$$\frac{E \vdash^{lookup} x : (l, \tau)}{\langle E \mid M \rangle \vdash^{L} x : l}$$

Die Regeln für L-Werte werden später noch erweitert.

3.1.3 Zuweisung

Die Zuweisung ist die elementare Operation, die mit Variablen in imperativen Sprachen assoziiert ist. Hierbei findet man hauptsächlich die folgenden syntaktischen Darstellungen:

$$e_1 \;=\; e_2 \quad (\text{C, Java})$$
$$e_1 \;:=\; e_2 \quad (\text{Pascal, Modula})$$

Einschränkung (dies wird häufig in der statischen Semantik, also zur Übersetzungszeit, überprüft):

> e_1 muss einen L-Wert haben.

Beispiel: Konstanten haben nur R-Werte (ebenso Wertparameter bei Prozeduren, siehe später). In Java werden Konstanten durch das Schlüsselwort `final` deklariert:

```
static final int mean = 42;
```

Hierdurch wird die Bindung des Namens „`mean`“ an die `int`-Konstante 42 definiert. Auf Grund dieser Deklaration ist die Zuweisung

```
mean = 43;
```

nicht erlaubt. Dieser Fehler wird in Java schon im Übersetzungsvorgang erkannt.

Um diese Einschränkung zu formalisieren, tragen wir Konstanten als Paar (Typ,Wert) in die Umgebung ein:

Deklaration einer Konstanten:

$$\langle E \mid M \rangle \texttt{ static final int } x \texttt{ = } n \; \langle E; x : (int, n) \mid M \rangle$$

Benutzung einer Konstanten (nur als R-Wert!):

$$\frac{E \vdash^{lookup} x : (\tau, v)}{\langle E \mid M \rangle \vdash^{R} x : v}$$

Damit können wir nun die **Semantik der Zuweisung** der Form $e_1 = e_2$ wie folgt festlegen:

- Berechne den L-Wert von e_1: l
- Berechne den R-Wert von e_2: v
- Schreibe v in den Behälter l, d.h. ändere den Speicher so, dass $M(l) = v$

Formal können wir dies durch folgend Inferenzregel spezifizieren:

$$\frac{\langle E \mid M \rangle \vdash^{L} e_1 : l \quad \langle E \mid M \rangle \vdash^{R} e_2 : v}{\langle E \mid M \rangle \; e_1 = e_2 \; \langle E \mid M[l/v] \rangle}$$

Eine weitere Forderung an die Zuweisung ist bei statisch getypten Programmiersprachen die **Typkompatibilität** der linken und rechten Seite. Für eine Zuweisung $e_1 = e_2$ kann das bedeuten:

- e_1 und e_2 haben gleiche Typen (strenge Forderung), oder
- Der Typ von e_2 muss in den Typ von e_1 *konvertierbar* sein (implizite Konvertierung).

Beispiel 3.3 (Typkonvertierung)**.** Ganze Zahlen sind in Gleitkommazahlen konvertierbar:

```
float x;
int i = 99;
x = i; // x hat den Wert 99.0
```

Umgekehrt kann man `float` nach `int` nur mit einem Genauigkeitsverlust konvertieren. Aus diesem Grund ist dies häufig unzulässig (z. B. in Java). Erlaubt ist dies nur mit expliziter Konvertierung („type cast“):

```
double x = 1.5;
int i;
i = x; // unzulässig
```

```
i = (int) x; // type cast, i hat den Wert 1
```

Typkonvertierungen sind von der jeweiligen Programmiersprache abhängig:

- C: Wahrheitswerte $\approx$ ganze Zahlen: $0 \approx$ `false`, $\neq 0 \approx$ `true`
- Java: `boolean` und `int` sind nicht konvertierbar:

  ```
  boolean b;
  int i;
  i = b; // unzulässig
  ```

Wir könnten die Typkonvertierung wie folgt formalisieren:

1. Berechne die Typen der linken und rechten Seite der Zuweisung (z.B. durch ein neues Inferenzsystem mit der Basissprache $E \vdash^T e : \tau$, wobei E eine Umgebung, e ein Ausdruck und τ ein Typ der jeweiligen Programmiersprache ist).
2. Füge damit in die Inferenzregel für die Zuweisung die Bedingung der Typkompatibilität und, falls es notwendig und erlaubt ist, Konvertierungsfunktionen ein.

Die Details überlassen wir als Übung.

In vielen Programmiersprachen gibt es Abkürzungen für häufige Zuweisungsarten:

- Zuweisung

  ```
  x = x + 3;
  ```

  Modula-3: `INC(x,3);`
  C, Java: `x += 3;`

  Als Verallgemeinerung steht die Abkürzung

  ```
  var op= expr;
  ```

  für die Zuweisung

  ```
  var = var op expr
  ```

- „`x++`“ und „`++x`“ entsprechen „`x += 1`“
- „`x--`“ und „`--x`“ entsprechen „`x -= 1`“

3.2 Kontrollabstraktionen

Ein Programmablauf in einer imperativen Programmiersprache ist im Wesentlichen eine Folge von Zuweisungen. Da es wichtig ist, dass Programme lesbar und wartbar sind, müssen auch Zuweisungsfolgen strukturiert werden (keine Verwendung von `gotos`, vgl. (Dijkstra, 1968)). Hierzu bieten imperative Programmiersprachen **Kontrollabstraktionen** an, d.h. eine strukturierte Zusammenfassung von häufigen Ablaufschemata zu programmiersprachlichen Einheiten. Im Folgenden diskutieren wir typische Kontrollabstraktionen imperativer Sprachen.

3.2.1 Sequentielle Komposition

Wenn S_1 und S_2 Anweisungen sind, dann ist auch die *sequentielle Komposition* $S_1; S_2$ eine Anweisung. Hierbei wird erst S_1 und dann S_2 ausgeführt. Formal können wir dies durch folgende Regel ausdrücken:

$$\frac{\langle E \mid M\rangle\ S_1\ \langle E' \mid M'\rangle \qquad \langle E' \mid M'\rangle\ S_2\ \langle E'' \mid M''\rangle}{\langle E \mid M\rangle\ S_1; S_2\ \langle E'' \mid M''\rangle}$$

Anmerkungen:

- In Java können Deklarationen mit Anweisungen gemischt werden
- Häufig ist es auch erlaubt, Anweisungsfolgen zu **Blöcke** zusammenzufassen, wie z.B. in Java:

  `{`S_1`;` S_2`;...;` S_n`;}`

  In diesem Fall sind die Deklarationen innerhalb des Blocks außerhalb unsichtbar, was wir einfach so formalisieren können:

  $$\frac{\langle E \mid M\rangle\ S\ \langle E' \mid M'\rangle}{\langle E \mid M\rangle\ \{S\}\ \langle E \mid M'\rangle}$$

- Benutzung des Semikolons: Hier existieren zwei unterschiedliche Sichtweisen:
  1. „`;`“ bezeichnet die sequentielle Komposition und steht zwischen Anweisungen (≈ Pascal). Dies führt zu folgender Blocksyntax:

     `begin` S_1`;` S_2`;` ... `;` S_n `end`

     Hier darf hinter der letzten Anweisung S_n kein Semikolon stehen!
  2. „`;`“ kennzeichnet das Ende einer Anweisung und steht hinter jeder Anweisung (≈ Java). Dies führt zu folgender Blocksyntax:

     `{` S_1`;` S_2`;` ... `;` S_n`;` `}`

  In manchen Programmiersprachen werden auch beide Sichtweisen syntaktisch unterstützt, wie z.B. in Modula-3.

3.2.2 Bedingte Anweisungen

Eine typische Form bedingter Anweisungen, d.h. Fallunterscheidungen, finden wir z.B. in der Programmiersprache Ada:

```
if bexp then S1 else S2 endif
```

Hierbei ist `bexp` ein boolescher Ausdruck und sind `S1` und `S2` Anweisungsfolgen. Diese Form ist zwar länger als in C-ähnlichen Sprachen, aber die `if-endif`-Klammerung vermeidet hierbei Mehrdeutigkeiten („dangling else“), die bei einem fehlenden `else`-Teil auftreten können. Z.B. verwendet Java die Syntax

```
if (bexp) S1 else S2
```

wobei `S1` und `S2` Anweisungen sind und der `else`-Teil auch fehlen kann. Als Beispiel betrachte man den folgenden Code-Ausschnitt:

```
x = 0;
y = 1;
if (y < 0)
  if (y > -5)
    x = 3;
else x = 5;
```

Wozu gehört nun das „`else`“? Falls dies zum ersten `if` gehört, gilt am Ende `x = 5`. Falls dies zum zweiten `if` gehört, gilt am Ende `x = 0`.

Üblich (auch in Java) ist die folgende Konvention:

> Ein `else` gehört immer zum letztmöglichen `if`, welches keinen `else`-Zweig hatte.

In diesem Beispiel gehört das `else` also zum zweiten `if`, d. h. die Einrückungen sind falsch gewählt.
In Zweifelsfällen kann (und sollte) man immer Klammern setzen:

```
x = 0;
y = 1;
if (y < 0)
  {if (y > -5)
    x = 3;
   else x = 5;}
```

```
x = 0;
y = 1;
  if (y < 0)
   {if (y > -5)
      x = 3; }
  else x = 5;
```

Wir definieren nun die Bedeutung bedingter Zuweisungen durch einige einfache Inferenz-

regeln, die jeweils über den Wahrheitswert des booleschen Ausdrucks unterscheiden:

$$\frac{\langle E \mid M\rangle \vdash^R bexp : true \quad \langle E \mid M\rangle\ S_1\ \langle E' \mid M'\rangle}{\langle E \mid M\rangle\ if\ (bexp)\ S_1\ else\ S_2\ \langle E, M'\rangle}$$

$$\frac{\langle E \mid M\rangle \vdash^R bexp : true \quad \langle E \mid M\rangle\ S_1\ \langle E' \mid M'\rangle}{\langle E \mid M\rangle\ if\ (bexp)\ S_1\ \langle E \mid M'\rangle}$$

$$\frac{\langle E \mid M\rangle \vdash^R bexp : false \quad \langle E \mid M\rangle\ S_2\ \langle E' \mid M'\rangle}{\langle E \mid M\rangle\ if\ (bexp)\ S_1\ else\ S_2\ \langle E \mid M'\rangle}$$

$$\frac{\langle E \mid M\rangle \vdash^R bexp : false}{\langle E \mid M\rangle\ if\ (bexp)\ S_1\ \langle E \mid M\rangle}$$

Hierbei ist zu beachten, dass wir die Veränderungen der Umgebung bei Abarbeitung des `then`- oder `else`-Teils ignorieren. Dies bedeutet, dass Deklarationen, die in diesen Teilen gemacht werden, außerhalb der bedingten Anweisung nicht sichtbar sind. Während diese Sichtweise bei Programmiersprachen mit lexikalischer Bindung üblich ist, kann dies z.B. bei Skriptsprachen anders sein. Aus diesem Grund ist es wichtig, diesen Aspekt genau zu spezifizieren, was mit unserem Inferenzsystem recht einfach ist.

Neben dieser Hauptform bedingter Anweisungen existieren in verschiedenen Programmiersprachen unterschiedliche Varianten hiervon:

- `elseif` (Ada):

  ```
  if b1 then S1
  elseif b2 then S2
  elseif b3 then S3
  else S4
  endif
  ```

  Dies ist äquivalent zu

  ```
  if b1 then S1
  else
    if b2 then S2
    else
      if b3 then S3 else S4 endif
    endif
  endif
  ```

- `case`/`switch`: Fallunterscheidung über den Wert eines Ausdrucks (in Java-Syntax):

  ```
  switch (exp) {
    case v1: S1; break;
    case v2: S2; break;
  ```

```
  ...
  case v_n: S_n; break;
  default: S_{n+1};
}
```

Dies ist äquivalent zu

```
vc = exp; // vc ist eine neue Variable
if (vc == v_1) S_1
else if (vc == v_2) S_2
     else ...
     else if (vc == v_n) S_n
          else S_{n+1};
```

Allerdings ist es üblich, dass der Compiler eine effizientere Übersetzung wählt. Zum Beispiel müssen die Werte v_i konstant sein, so dass man dann eine Sprungtabelle generieren kann, die das Auffinden der Alternative in konstanter Zeit ermöglicht.

Falls ein `break` am Ende eines `case`-Zweiges fehlt, wird mit dem nächsten Zweig weitergemacht. Dies führt allerdings leicht zu Programmierfehlern, wenn man ein `break` versehentlich vergisst. Daher wird bei anderen Programmiersprachen eine Syntax verwendet, bei denen man keine `break`-Anweisungen schreiben muss. Z.B. wird die obige `switch`-Anweisung in der Programmiersprach Go[3] wie folgt geschrieben:

```
switch exp {
  case v_1: S_1;
  case v_2: S_2;
  ...
  case v_n: S_n;
  default: S_{n+1};
}
```

3.2.3 Schleifen

Schleifen ermöglichen ein wiederholtes Ausführen von Anweisungen. Semantisch entsprechen Schleifen einer Rekursion. Dies haben wir auch bei der denotationellen Semantik gesehen, wo wir für die Interpretation von Schleifen eine rekursive Funktion benötigt haben und die Semantik dann als kleinsten Fixpunkt dieser rekursiven Funktion interpretieren.
Prinzipiell ist eine einzige `while`-Schleife (und Fallunterscheidungen) für ein Programm ausreichend. Allerdings bieten imperative Programmiersprachen verschiedene Arten von Schleifen an, die wir nachfolgend kurz diskutieren.

[3] `https://www.go-lang.org/`

Java-Syntax für `while`-Schleifen:

```
while (B) S
```

Umgangssprachlich kann man die Bedeutung so beschreiben:

> Solange `B` wahr ist, führe `S` aus.

Formal ist „`while (B) S`“ äquivalent zu „`if (B) {S; while (B) S}`“

Aus dieser Überlegung können wir die Semantik der `while`-Schleife durch folgende abgeleitete Inferenzregeln definieren:

$$\frac{\langle E \mid M\rangle \vdash^R B : false}{\langle E \mid M\rangle\ while\ (B)\ S\ \langle E \mid M\rangle}$$

$$\frac{\langle E \mid M\rangle \vdash^R B : true \qquad \langle E \mid M\rangle\ S\ \langle E' \mid M'\rangle \qquad \langle E \mid M'\rangle\ while\ (B)\ S\ \langle E \mid M''\rangle}{\langle E \mid M\rangle\ while\ (B)\ S\ \langle E \mid M''\rangle}$$

Ähnlich wie bei bedingten Anweisungen ignorieren wir nach Abarbeitung der Schleife alle Veränderungen der Umgebung innerhalb einer Schleife. Dies ist sinnvoll, weil es statisch schwer zu überschauen ist, ob die Schleife abgearbeitet wird.

Die letzte Regel könnte man durch Rückführung auf die Regel für sequentielle Komposition auch kompakter schreiben:

$$\frac{\langle E \mid M\rangle \vdash^R B : true \qquad \langle E \mid M\rangle\ S\ ;\ while\ (B)\ S\ \langle E' \mid M'\rangle}{\langle E \mid M\rangle\ while\ (B)\ S\ \langle E \mid M'\rangle}$$

Varianten von Schleifen

- Repeat-Schleifen: Führe die Anweisung mindestens einmal aus:
  - Pascal: `repeat S until B`
  - Java: `do S while (B)`

  Die letzte Form ist äquivalent zu „`S; while (B) S`“, während in der `repeat`-Form die Bedingung negiert ist.

- Zählschleifen (meistens in Verbindung mit Feldern):
  Führe bei der Schleifenanweisung einen Zähler mit. Üblicherweise werden Zählschleifen mit dem Schlüsselwort „`for`“ eingeleitet, aber es gibt sehr viele Varianten, z. B. in Java:

  ```
  for (initstat; boolexpr; incrstat) stat
  ```

  Dies ist äquivalent zu:

```
{ initstat;
  while (boolexpr) {
     stat
     incrstat;
  }
}
```

Eine typische Anwendung ist das Aufsummieren von Feldelementen:

```
s = 0;
for (int i = 0; i < fieldlen; i++) {
   s = s + a[i];
}
```

- Endlosschleifen: Diese können einfach durch eine `while`-Schleife mit der Bedingung `true` realisiert werden. Z.B. wird bei

  `while (true)` S

  die Anweisung S unendlich oft ausgeführt. Da solche Schleifen z.B. bei Serveranwendungen sinnvoll vorkommen, gibt es in manchen Programmiersprachen auch spezielle Formen, wie z.B. in Modula-3

  `loop` S `end`

  oder in Rust

  `loop { ` S ` }`

Manchmal gibt es auch die Möglichkeit, Schleifen zu verlassen, obwohl die Schleifenbedingung noch erfüllt ist, womit man z.B. auch Endlosschleifen verlassen kann.

- Modula-3 bietet die Anweisung `exit`: verlasse innerste Schleife und mache danach weiter.
- Java bietet die Anweisung `break`: analog zu `exit`, aber man kann auch Marken angeben, um mehrere Schleifen zu verlassen, wie in diesem Beispiel:

```
search:   // Sprungmarke
for (...) {
  while (...) {
    ...
    break search;
    /* verlasse damit die while- und for-Schleife auf einmal. */
    ...
```

```
  }
}
```

- In Rust können Kontrollstrukturen auch in Ausdrücken verwendet werden. In diesem Fall kann man z.B. einem `break` auch einen Rückgabewert mitgeben, der dann als Ergebnis der Schleife verwendet wird:

```
let mut prod = 1;

let result = loop {
    prod = prod*2;

    if prod > 1000000 {
        break prod;
    }
};
```

  Durch die Schleifenberechnung erhält `result` den Wert `1048576`.