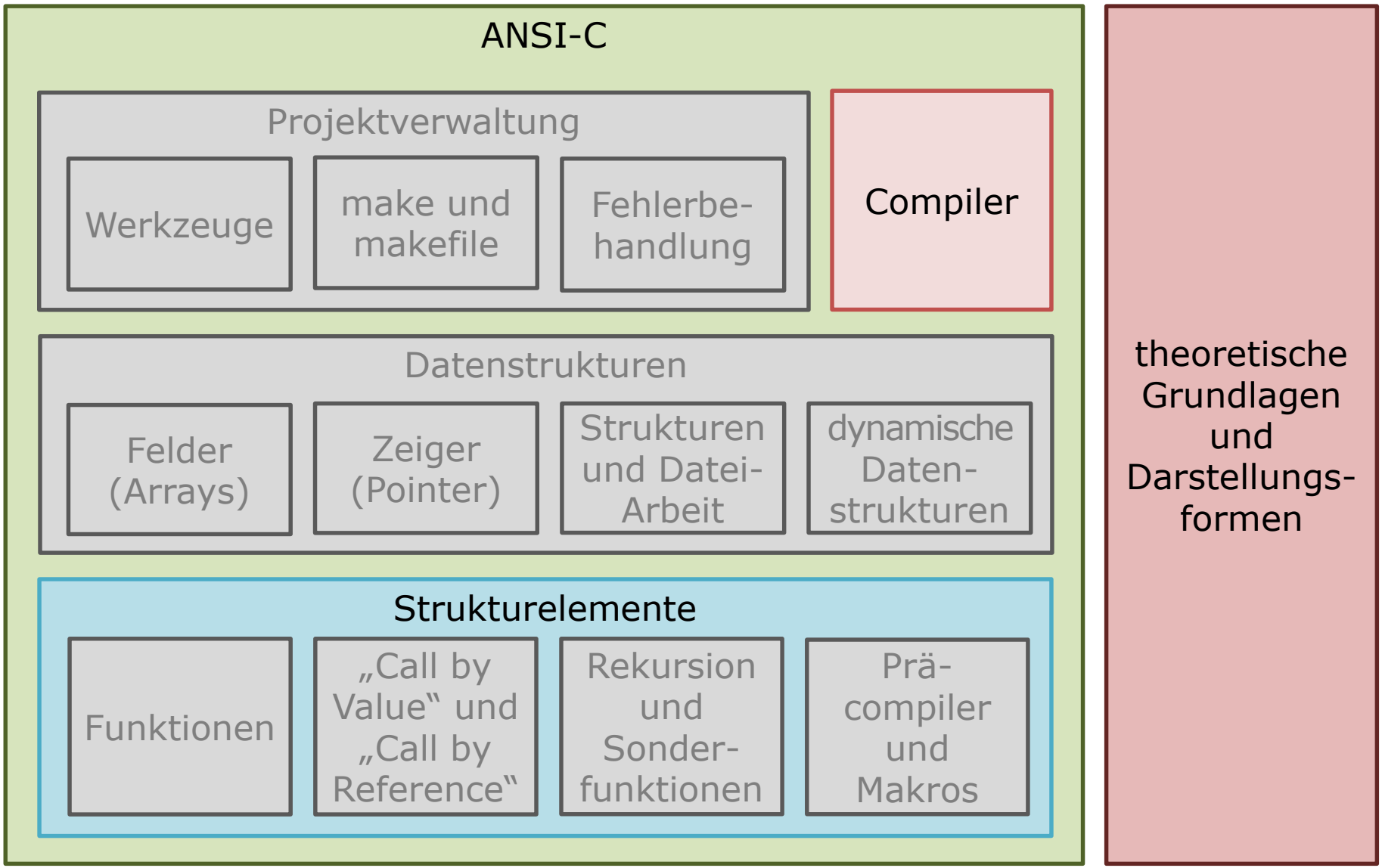
BERUFSAKADEMIE SACHSEN
STAATLICHE STUDIENAKADEMIE
DRESDEN
UNIVERSITY OF COOPERATIVE EDUCATION

Imperative Programmierung

Strukturelemente

Prof. Dr.-Ing. Tenshi Hara
tenshi.hara@ba-sachsen.de

Aufbau der Lehrveranstaltung



ARITHMETISCHE OPERATOREN

für numerische Datentypen (auch Zeichen)

- `+` – Addition
- `-` – Subtraktion
- `*` – Multiplikation
- `/` – ganzzahlige Division ohne Rest

zusätzlich bei ganzzahligen Datentypen

- `%` – Modulo (Ergebnis der ganzzahligen Restdivision)
- `++` – Inkrement (`i++` und `++i` entsprechen beide `i=i+1;`)
- `--` – Dekrement (`i--` und `--i` entsprechen beide `i=i-1;`)

Achtung! `i++` und `++i` (analog `i--` und `--i`) sind nicht äquivalent.

Beispiel: `int i,j,p,q; i=1; j=i++; p=1; q=++p;`

→ Ergebnis: `i` ist 2, `j` ist aber 1; `p` ist 2 und `q` ist auch 2.

Zuweisungen, Bedingungsoperator

`=` – Zuweisung: Variable links vom Gleichheitszeichen wird Ausdruck rechts vom Gleichheitszeichen zugewiesen

`+=`, `-=`, `*=`, `/=` und `%=` sind verkürzte Schreibweisen; es gilt:

`i ⊚= <Statement>` ⇔ `i = i ⊚ <Statement>` ∀ ⊚ ∈ {+,-,*,/,%}

- Bedingungsoperator ist der einzige dreistellige Operator

```
<Boolean Statement> ? <Case TRUE> : <Case FALSE>
```

Beispiel: `i += (j>=0) ? j : (-1 * j);` (entspricht `i=i+abs(j);`)

Operatoren

Vergleichsoperatoren

`<` – kleiner, `<=` – kleiner gleich

`>` – größer, `>=` – größer gleich

`==` – gleich, `!=` – ungleich

logische Operatoren

`&&` – logische Konjunktion (and)

`||` – logische Disjunktion (or)

`!` – logische Negation (not)

bitweise Operatoren

`&` – bitweise Konjunktion

`|` – bitweise inklusive Disjunktion

`^` – bitweise exklusive Disjunktion

`~` – bitweise Negation

`>>` – bitweise Rechtsverschiebung

`<<` – bitweise Linksverschiebung

logische Werte (ab C99):

Datentyp `bool` (in `<stdbool.h>`, `_Bool`) mit Werten `true` und `false`

Ausdruck, Anweisung

Ausdruck (Expression)

- beliebiger arithmetischer oder logischer Ausdruck
- Zuweisung oder Funktionsaufruf (jeweils ohne Semikolon als Abschluss)

Ausdrücke mit Wert !=0, >0 oder <0 werden als „wahr“ (`true`) gewertet. ==0 ergibt „falsch“ (`false`)!

Anweisung (Statement)

- Ausdruck oder Funktionsaufruf mit Semikolon als Abschluss
- Kontrollstruktur
- ein Semikolon allein ist eine gültige (leere) Anweisung

Kontrollstrukturen

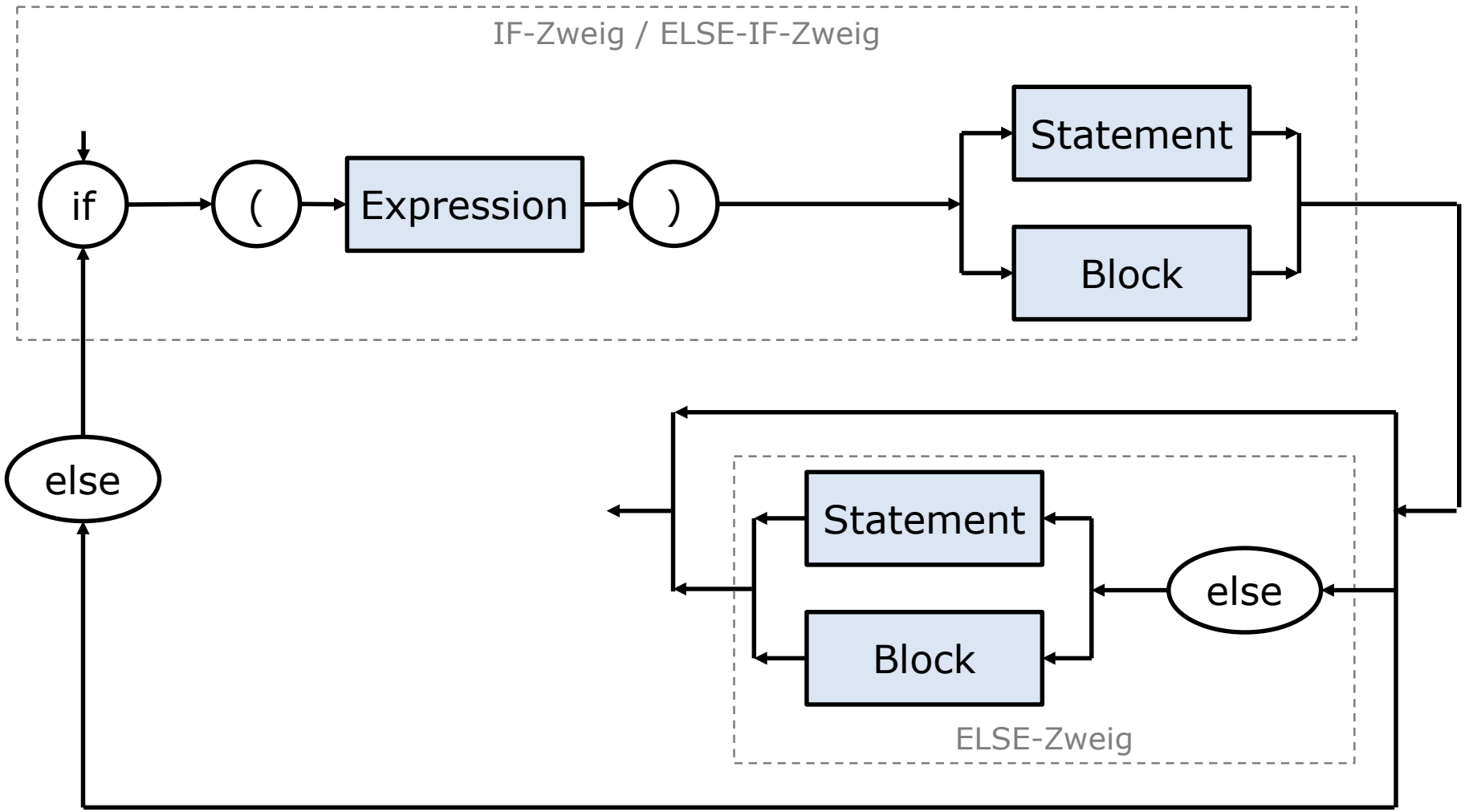
BEDINGTE VERZWEIGUNG (1/3)

`if`-Verzweigung – Syntax

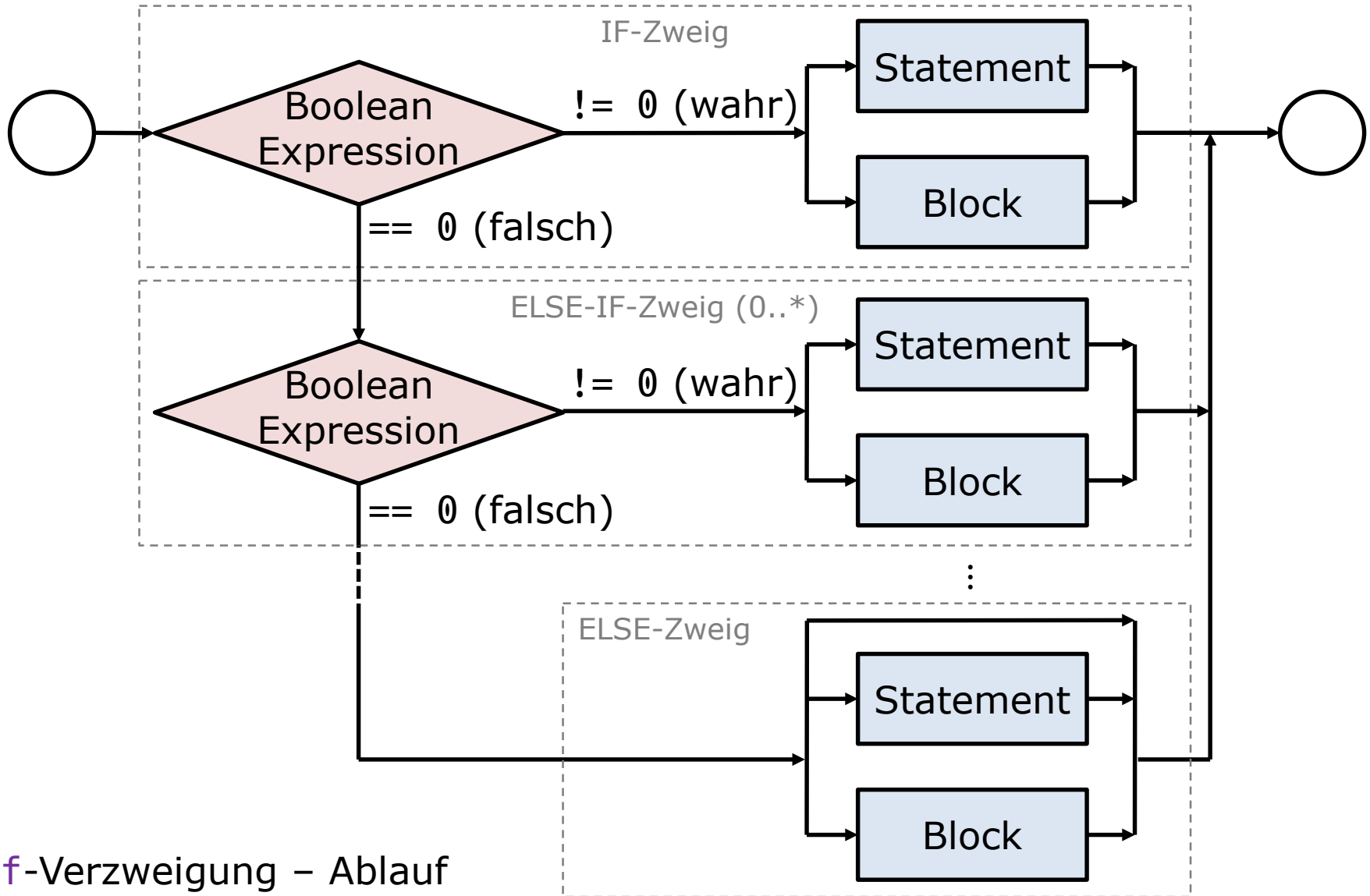
```
<if Statement> ::= "if" "(" <Expression> ")"
      <Statement> | <Block>
   {
      "else" "if" "(" <Expression> ")"
         <Statement> | <Block>
   }*
   [
      "else"
         <Statement> | <Block>
   ]
```

BEDINGTE VERZWEIGUNG (2/3)

if-Verzweigung – Syntax



BEDINGTE VERZWEIGUNG (3/3)



`if`-Verzweigung – Ablauf

MEHRFACHVERZWEIGUNG (1/3)

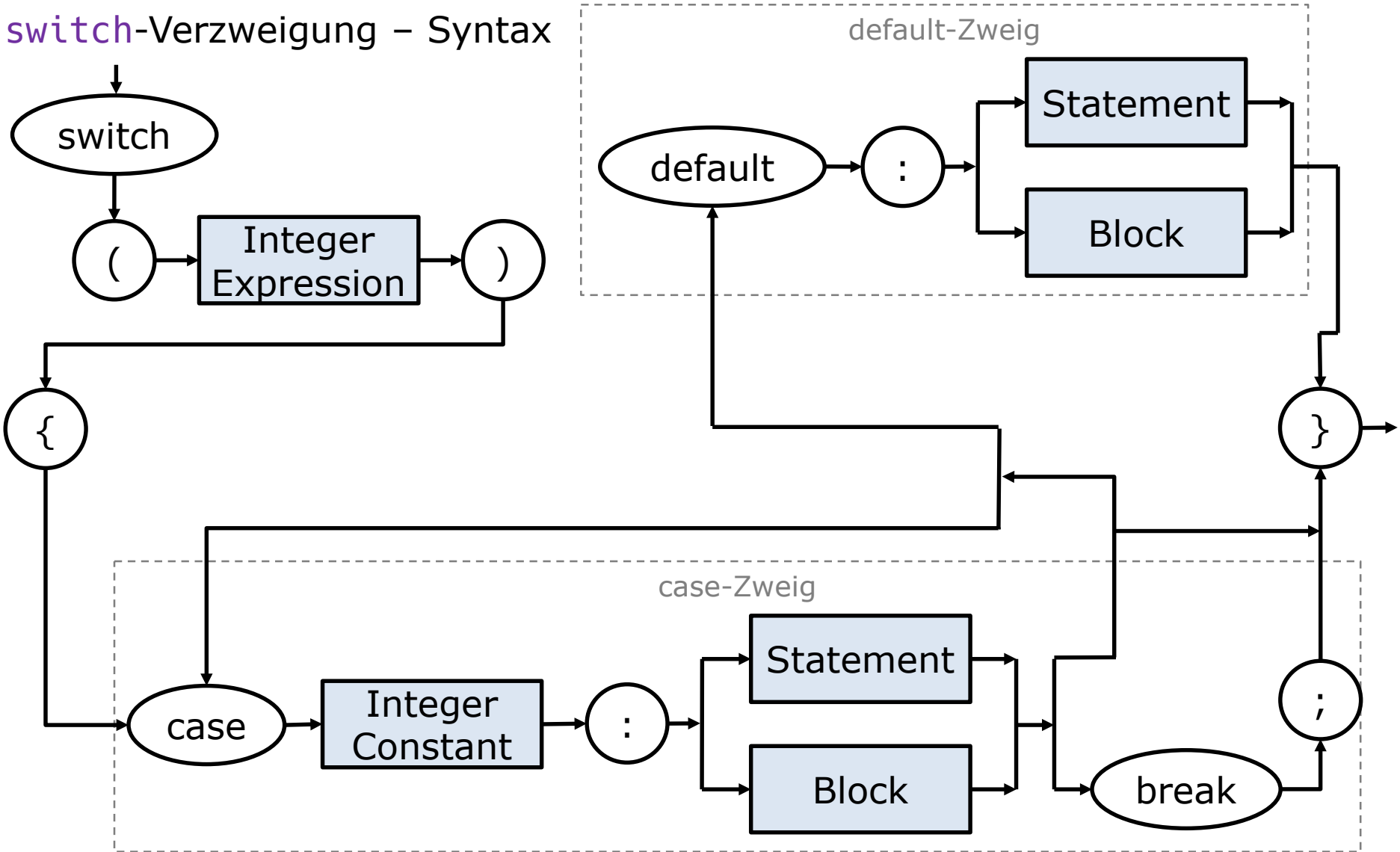
`switch`-Verzweigung – Syntax

```
<switch Statement> ::= "switch" "(" <Integer Expression> ")"
                       "{"
                          {
                             "case" <Integer Constant> ":"
                                <Statement> | <Block>
                                ["break" ";"]
                          }+
                          [
                             "default" ":"
                                <Statement> | <Block>
                          ]
                       "}"
```

Achtung! `case`-Zweige sind Einsprungstellen! Ohne `break` wird auch der direkt folgende `case-` bzw. am Ende der `default`-Zweig abgearbeitet!

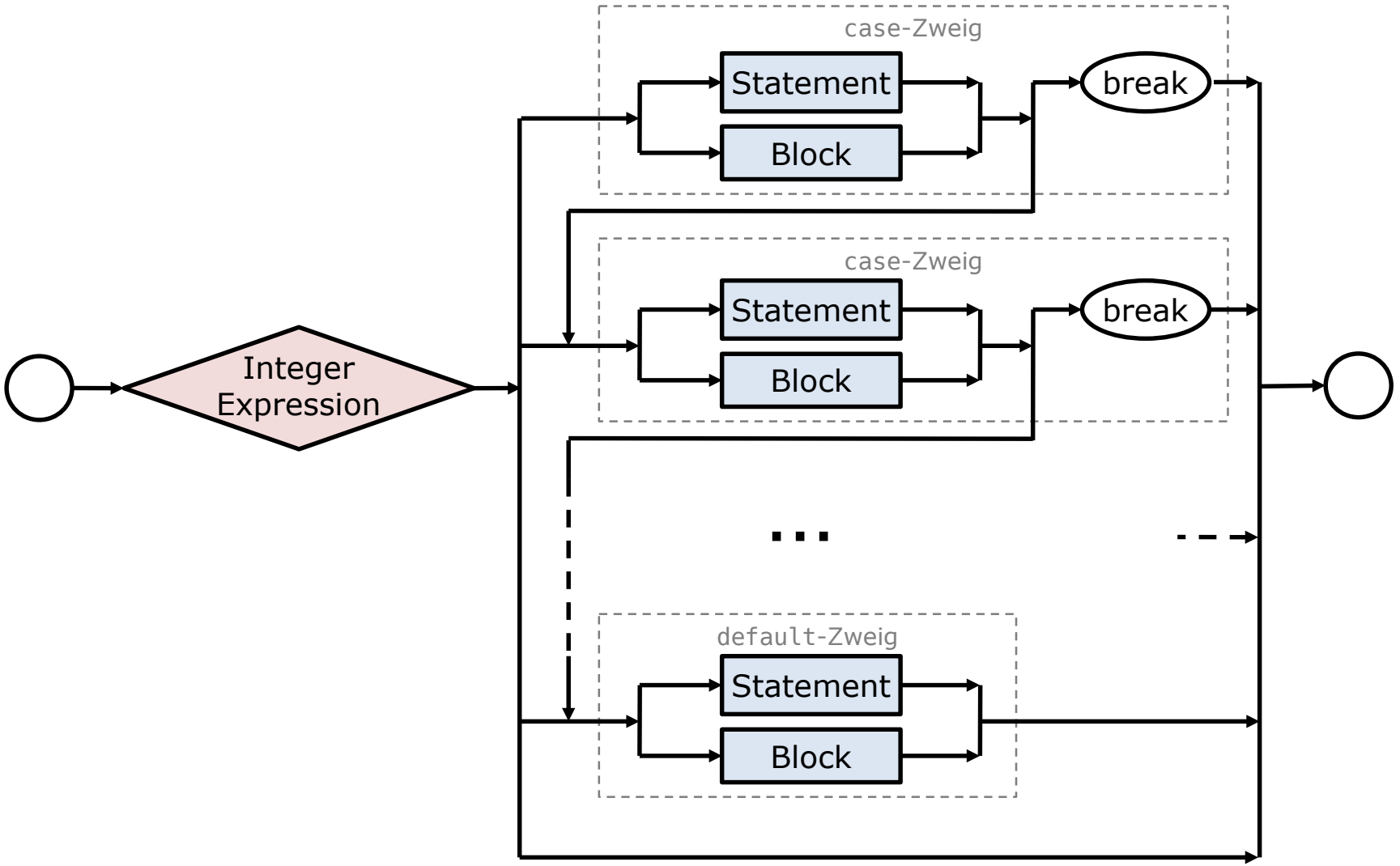
MEHRFACHVERZWEIGUNG (2/3)

switch-Verzweigung – Syntax



MEHRFACHVERZWEIGUNG (3/3)

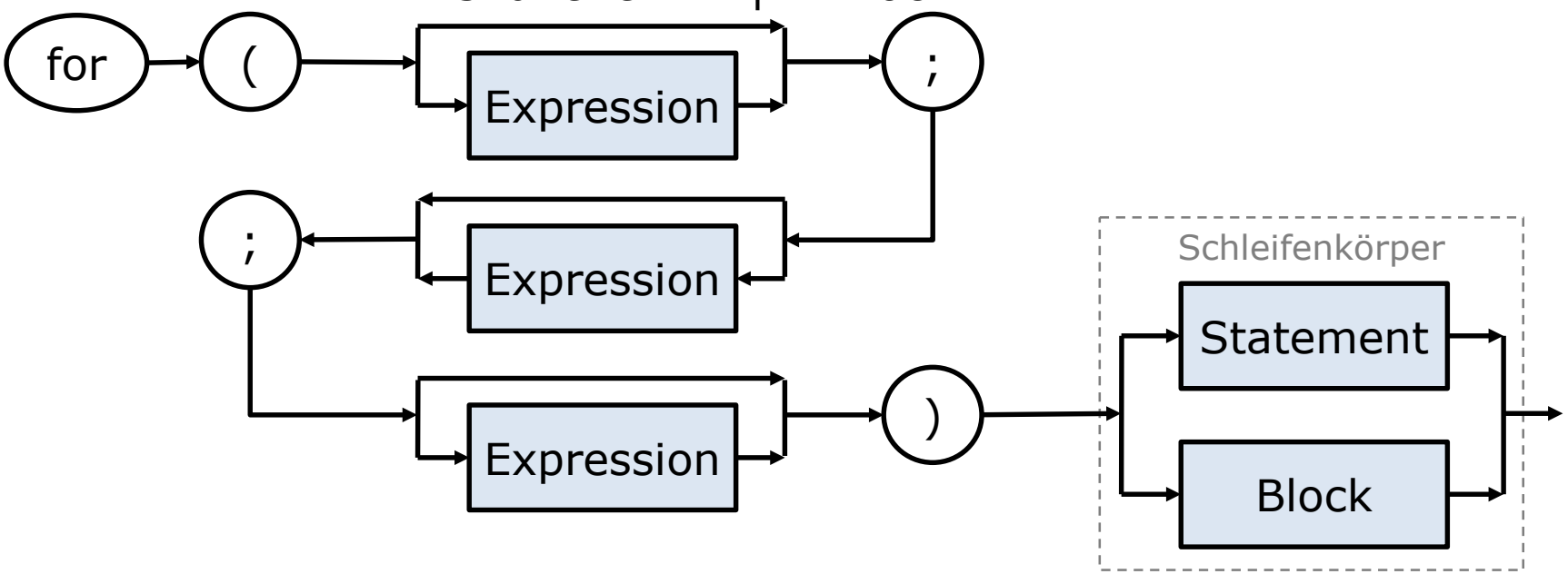
`switch`-Verzweigung – Ablauf



ZÄHLSCHLEIFE (1/2)

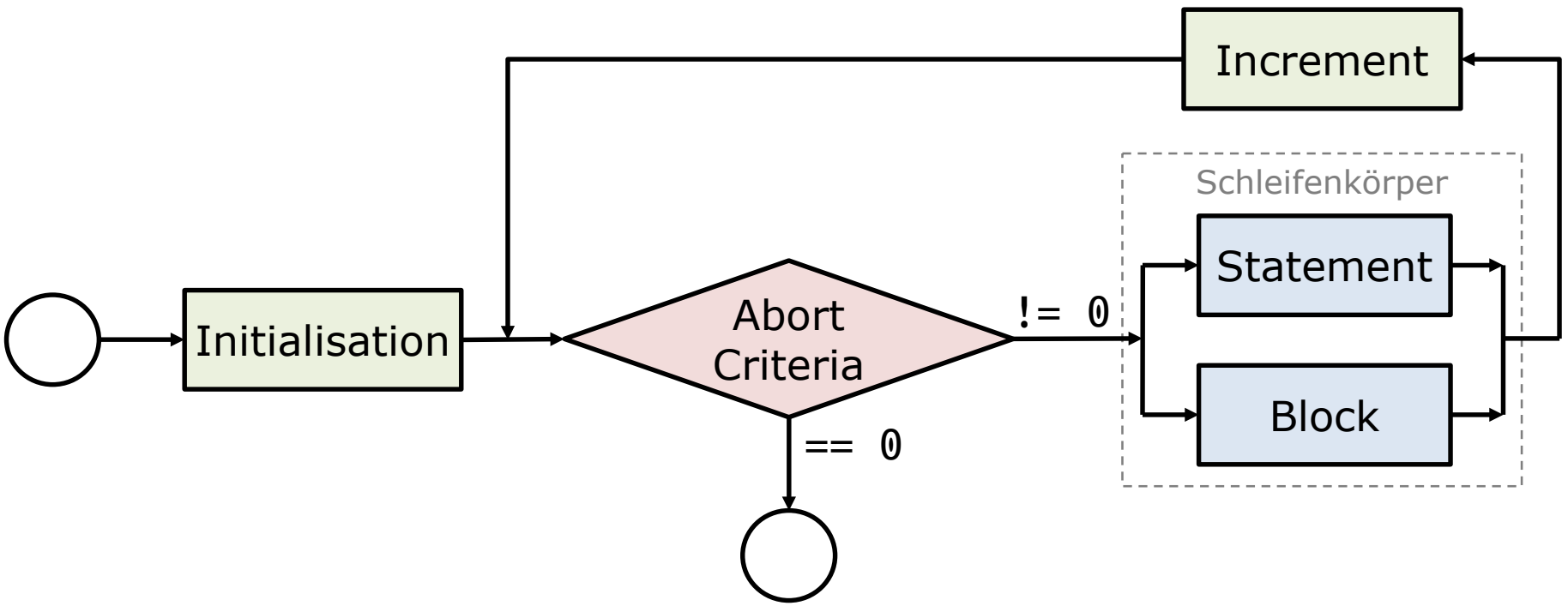
`for`-Schleife – Syntax

```
<for Loop> ::= "for" "("
                    [<Expression>] ";"   (* Intialisierung *)
                    [<Expression>] ";"   (* Abbruchbedingung *)
                    [<Expression>]       (* Inkrement *)
                ")"
                    <Statement> | <Block>
```



ZÄHLSCHLEIFE (2/2)

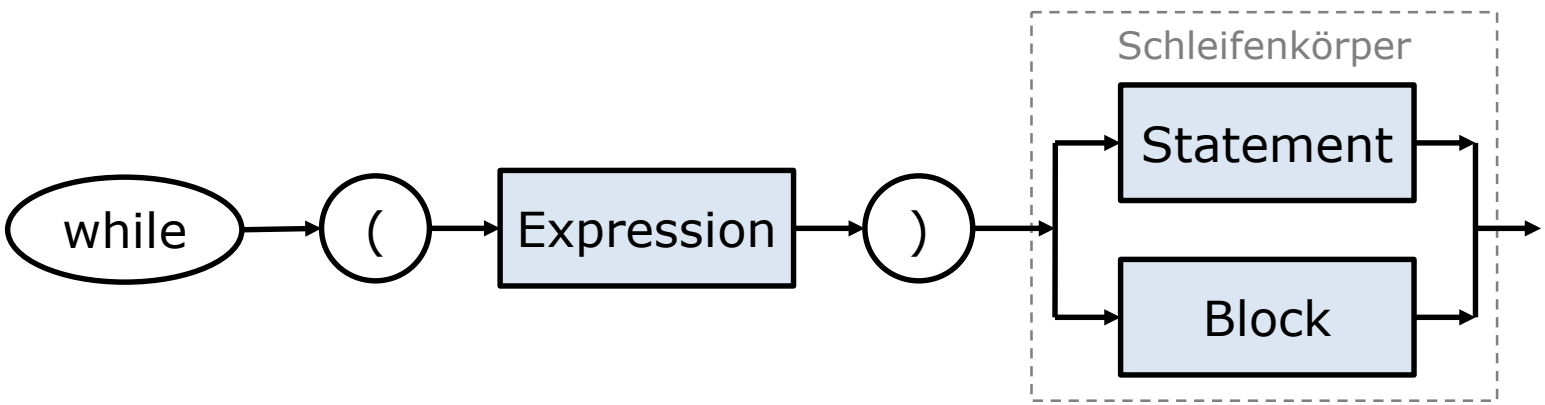
`for`-Schleife – Ablauf



KOPFGESTEUERTE SCHLEIFE (1/2)

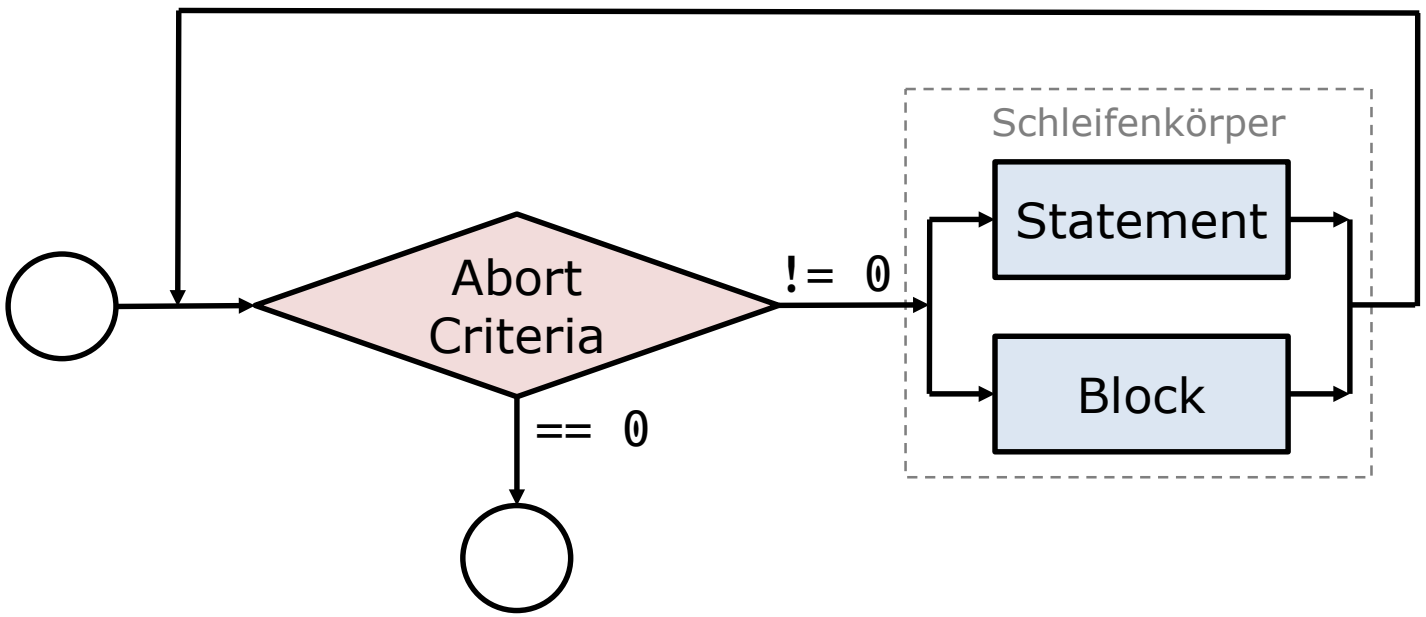
`while`-Schleife (kopfgesteuert) – Syntax

```
<while Loop> ::= "while" "(" <Expression> ")"
                        <Statement> | <Block>
```



KOPFGESTEUERTE SCHLEIFE (2/2)

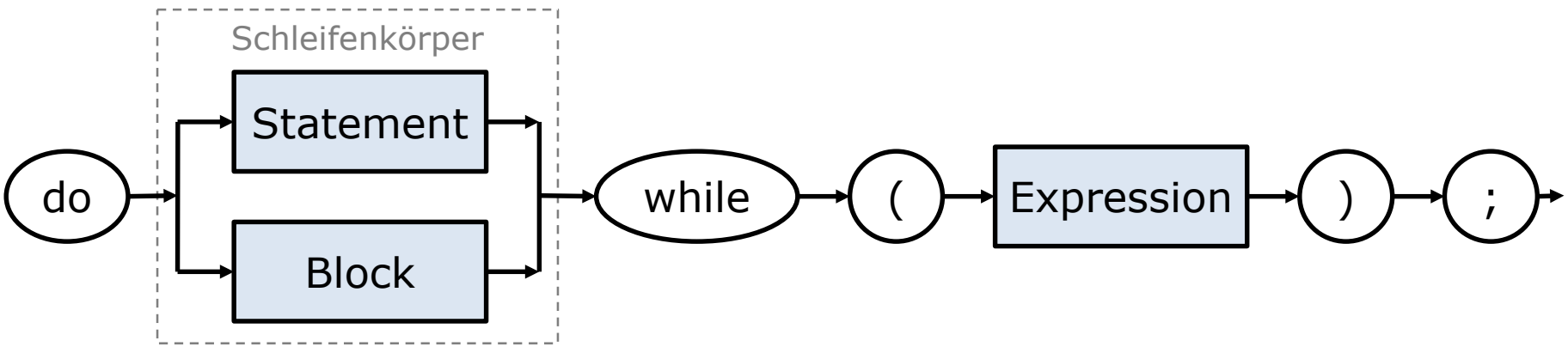
`while`-Schleife – Ablauf



FUßGESTEUERTE SCHLEIFE (1/2)

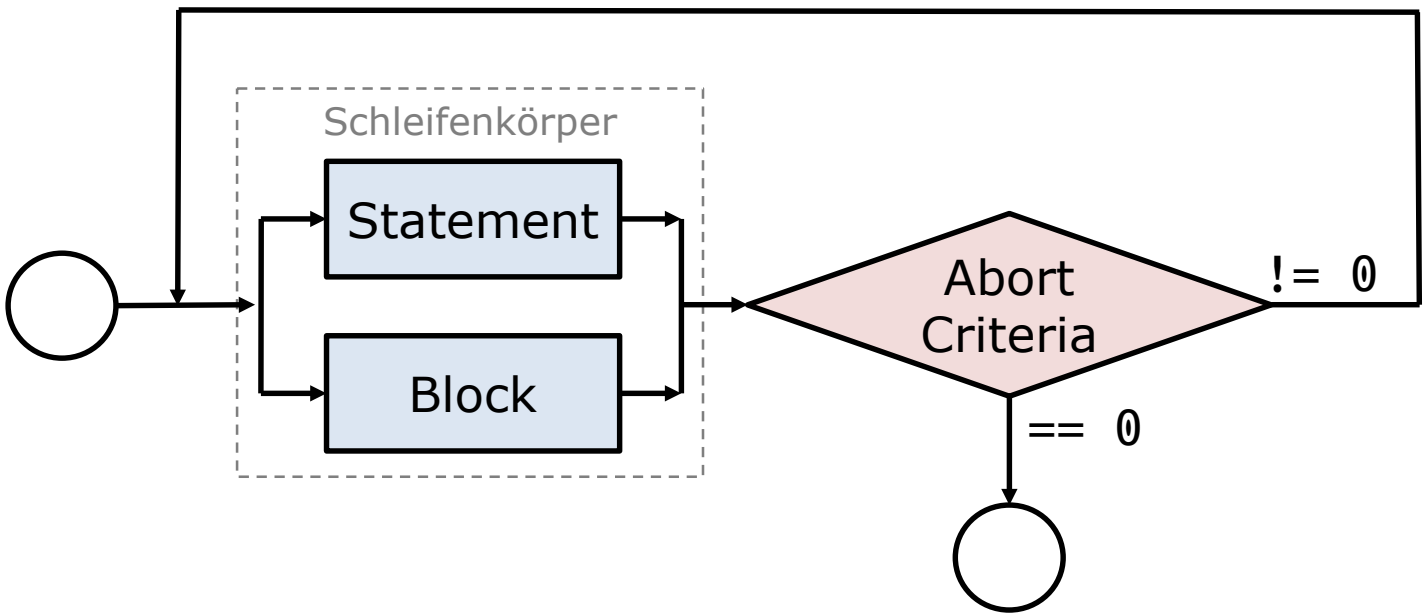
`do…while`-Schleife (fußgesteuert) – Syntax

```
<do…while Loop> ::= "do"
                        <Statement> | <Block>
                    "while" "(" <Expression> ")" ";"
```



FUßGESTEUERTE SCHLEIFE (2/2)

`do…while`-Schleife – Ablauf



Iterations-, Schleifen- und Verzweigungsabbruch

continue

- nur in `for`-, `while`- und do…while-Schleifen gültig
- aktueller Schleifendurchlauf wird abgebrochen
- falls es ein Inkrement gibt, wird die Schleife mit dem neuen Wert reinitiiert (nur bei `for`-Schleife)
- Abarbeitung setzt bei Prüfung der Abbruchbedingung fort

break

- nur in `for`-, `while`- und do…while-Schleifen sowie in `switch`-Verzweigungen gültig
- Abarbeitung setzt direkt nach dem aktuellen Schleifenkörper bzw. dem aktuellen `case`-Zweig fort

Achtung! Bei Nutzung von `break` ist kein Sprung aus mehreren ineinander geschachtelten Schleifen möglich. Es kann nur die jeweils aktuelle Schleife abgebrochen werden. (Vergleiche dazu bei PHP `break 1;`, `break 2;`, etc.)

UNBEDINGTER SPRUNG, RÜCKSPRUNG

goto

- freier Sprungbefehl (Unconditional Jump)
- Ausführung setzt direkt nach angegebener Sprungmarke (Label) fort
- äußerst sparsam verwenden, da Kontrollfluss nicht nachvollziehbar wird
- Syntax: `<goto Statement> ::= "goto" <Label> ";"`
- irgendwo im Quelltext *muss* auftauchen: `<Label><Statement>`

return

- Rückkehr zur unmittelbar aufrufenden Funktion (Sprungbefehl)
- Syntax: "return" [<Expression>] ";"

PROGRAMMABBRUCH

`exit`

- terminiert *sofort* den aufrufenden Prozess
- alle offenen Dateideskriptoren werden geschlossen
- alle Unterprozesse werden an den Hauptprozess (`init`) übergeben
- der direkte Elternprozess wird mittels `"SIGCHLD" <Integer>` informiert
- Syntax: `"exit" "(" <Integer> ")" ";"`