Algorithmisches Denken und imperative Programmierung

Janis Voigtländer

Universität Bonn

Wintersemester 2011/12

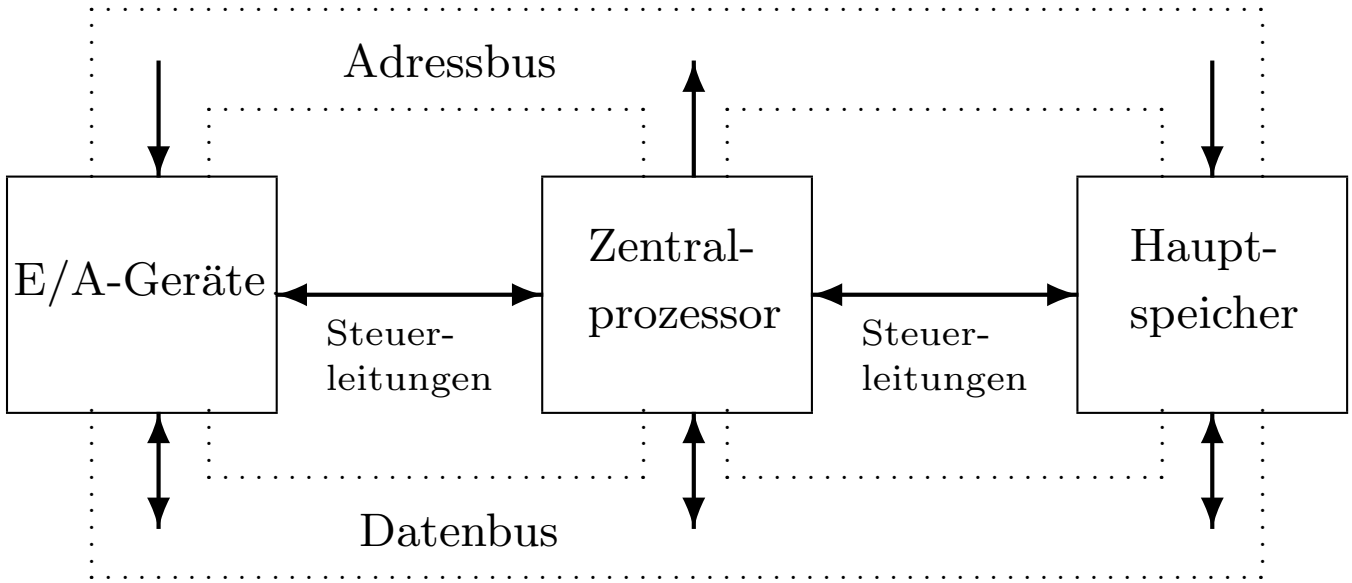
Zu den morgigen (Nicht-)Übungen

Wenn Sie morgen Übung hätten, können Sie ersatzweise genau einen der folgenden Termine aufsuchen:

- Mi, 02.11., 08:15–09:45, A6b
- Mi, 02.11., 10:15–11:45, A301
- Mi, 02.11., 14:15–15:45, A7b
- Mi, 02.11., 16:15–17:45, A121
- Do, 03.11., 12:30–14:00, A6b
- Do, 03.11., 14:15–15:45, A121
- Fr, 04.11., 08:15–09:45, A121
- Fr, 04.11., 12:30–14:00, A6b
- Mo, 07.11., 14:15–15:45, A301
- Mo, 07.11., 16:15–17:45, A121

Imperativ — orientiert am Maschinenmodell

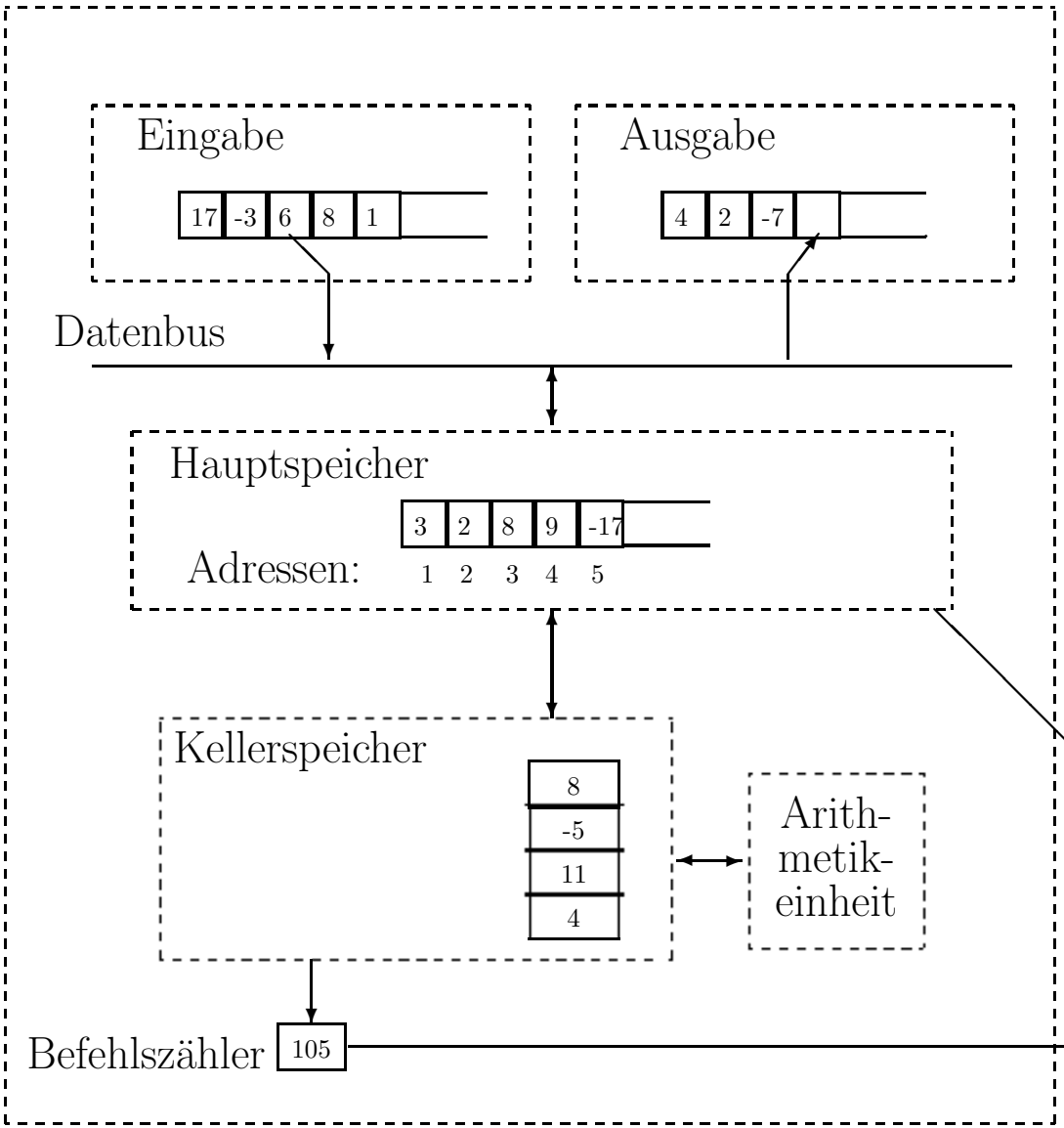
Von-Neumann-Rechner:



Aber wir wollen Programme nicht auf diesem Niveau:

```
 fib: STA count      BMI end      end: RTS
      LDA #0         TAX        count: S 1
      STA temp       ADC temp    temp: S 1
      LDA #1         STX temp
loop: DEC count      JMP loop
```

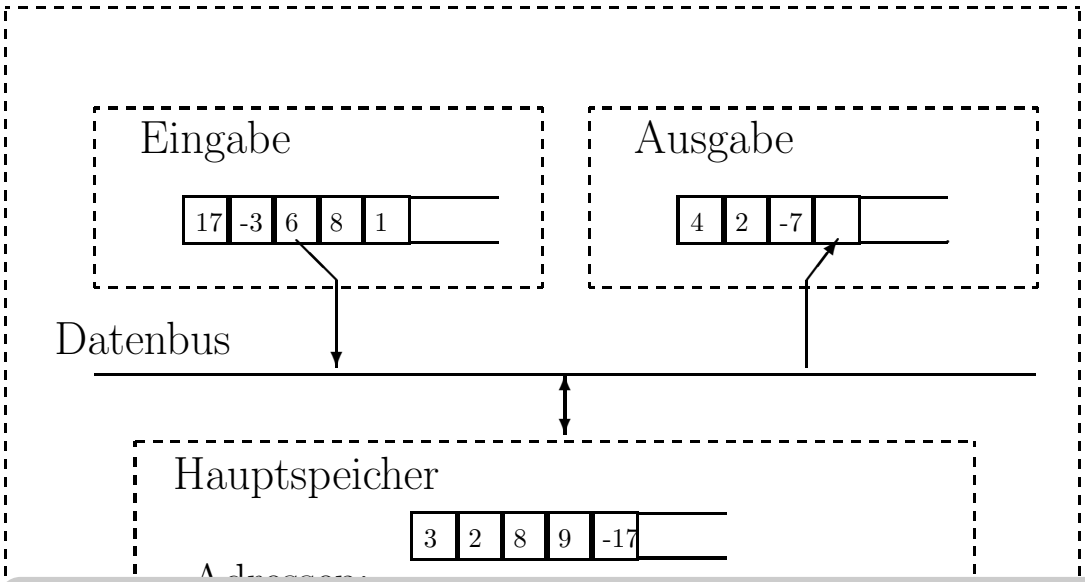
Eine immer noch recht abstrakte Sicht



Essentiell:

- Anweisungsfolge
- Sprünge im Programm
- Zugriff auf Ein-/Ausgabe
- Zugriff auf Speicher (über Adressen)
- Rechenoperationen

Eine immer noch recht abstrakte Sicht



Essentiell:

- Anweisungsfolge
- Sprünge im Programm
- Zugriff auf Ein-/Ausgabe
- Zugriff auf Speicher (über Adressen)
- Rechenoperationen

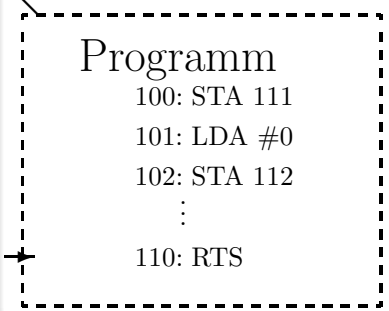
Für C:

Weg:

- explizite Adressen
- beliebige Sprünge
- Zugriff auf Programmspeicher
- Kellerspeicher

Hinzu:

- Variablennamen
- Kontrollstrukturen
- Ausdrücke



Also, abzudeckende operationelle Konzepte:

- aktueller „Zustand“
  - . . . der Ein-/Ausgabe
  - . . . des Datenspeichers
- Aktionen, zur Veränderung des Zustands:
  - Zugriff auf Ein-/Ausgabe
  - Zugriff auf Speicher (über Variablennamen)
  - Rechenoperationen (im weitesten Sinne)
- Tests, abhängig vom aktuellen Zustand
  (aber diesen nicht verändernd)
- Abarbeitungs(reihen)folge, steuerbar

. . . abzudecken durch syntaktische Konstrukte:

- Deklarationen, zur Zuordnung Variablennamen $\leftrightarrow$ Speicherzellen
- einfache Anweisungen:
  - Ein-/Ausgabeoperationen
  - Zuweisungen
- Ausdrücke (enthalten Rechenoperationen)
  - als Teil von Zuweisungen
  - für Tests
- komplexe Anweisungen, Kontrollstrukturen:
  - Sequenz
  - Verzweigungen
  - Wiederholungen („Schleifen“)
  - Schachtelung

- aktueller „Zustand“
  - . . . der Ein-/Ausgabe
  - . . . des Datenspeichers
- Aktionen, zur Veränderung des Zustands:
  - Zugriff auf Ein-/Ausgabe
  - Zugriff auf Speicher (über Variablennamen)
  - Rechenoperationen (im weitesten Sinne)
- Tests, abhängig vom aktuellen Zustand (aber diesen nicht verändernd)
- Abarbeitungs(reihen)folge, steuerbar

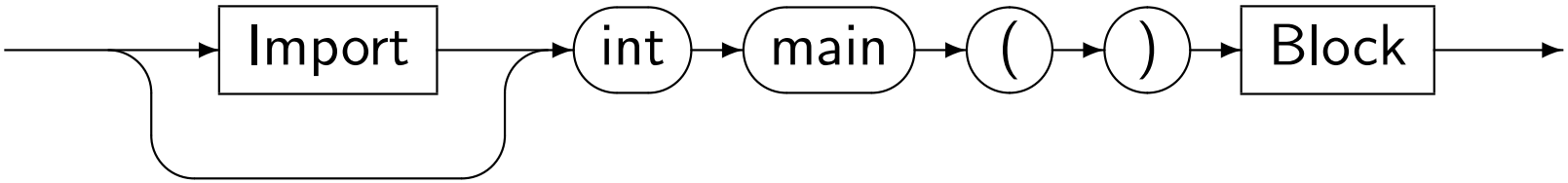
Ein konkretes, einfaches Programm

```
#include <stdio.h>

int main()
{  int n,s,i;
   scanf("%d",&n);
   s=0;
   i=1;
   while (i<=n)
      {  s=s+i*i;
         i=i+1;
      }
   printf("%d",s);
   return 0;
}
```

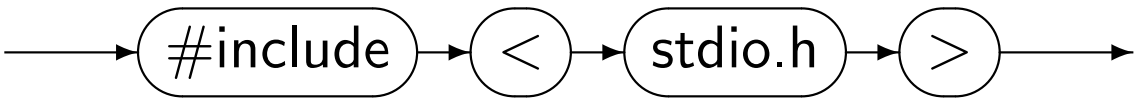
- Deklarationen, zur Zuordnung Variablennamen $\leftrightarrow$ Speicherzellen
- einfache Anweisungen:
  - Ein-/Ausgabeoperationen
  - Zuweisungen
- Ausdrücke (enthalten Rechenoperationen)
  - als Teil von Zuweisungen
  - für Tests
- komplexe Anweisungen, Kontrollstrukturen:
  - Sequenz
  - Verzweigungen
  - Wiederholungen („Schleifen“)
  - Schachtelung

Globaler Aufbau eines C-Programms (für den Moment)

Program



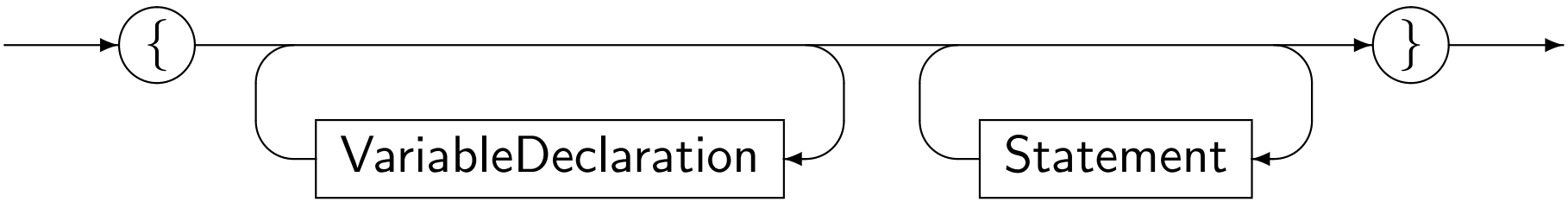
Import



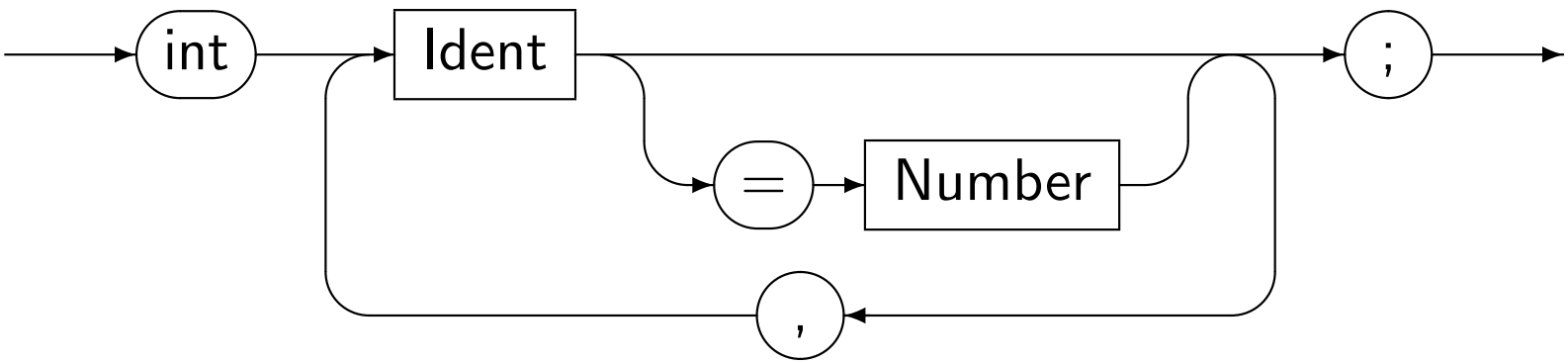
- Import der Standardbibliothek für Ein-/Ausgabe
- main - „Funktion“ enthält auszuführenden Code.
- Rückgabewert (eine Zahl) erlaubt Rückmeldung über Erfolg der Programmausführung an Betriebssystem.

Globaler Aufbau eines C-Programms (für den Moment)

Block



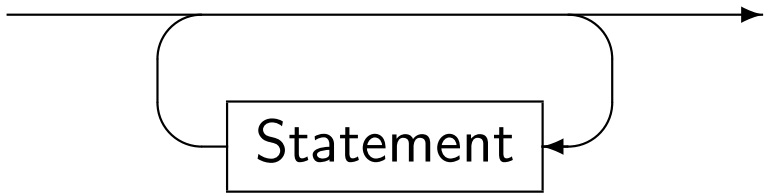
VariableDeclaration



- Festlegung der benutzten Speicherstellen/Variablen, gegebenenfalls mit Initialwerten

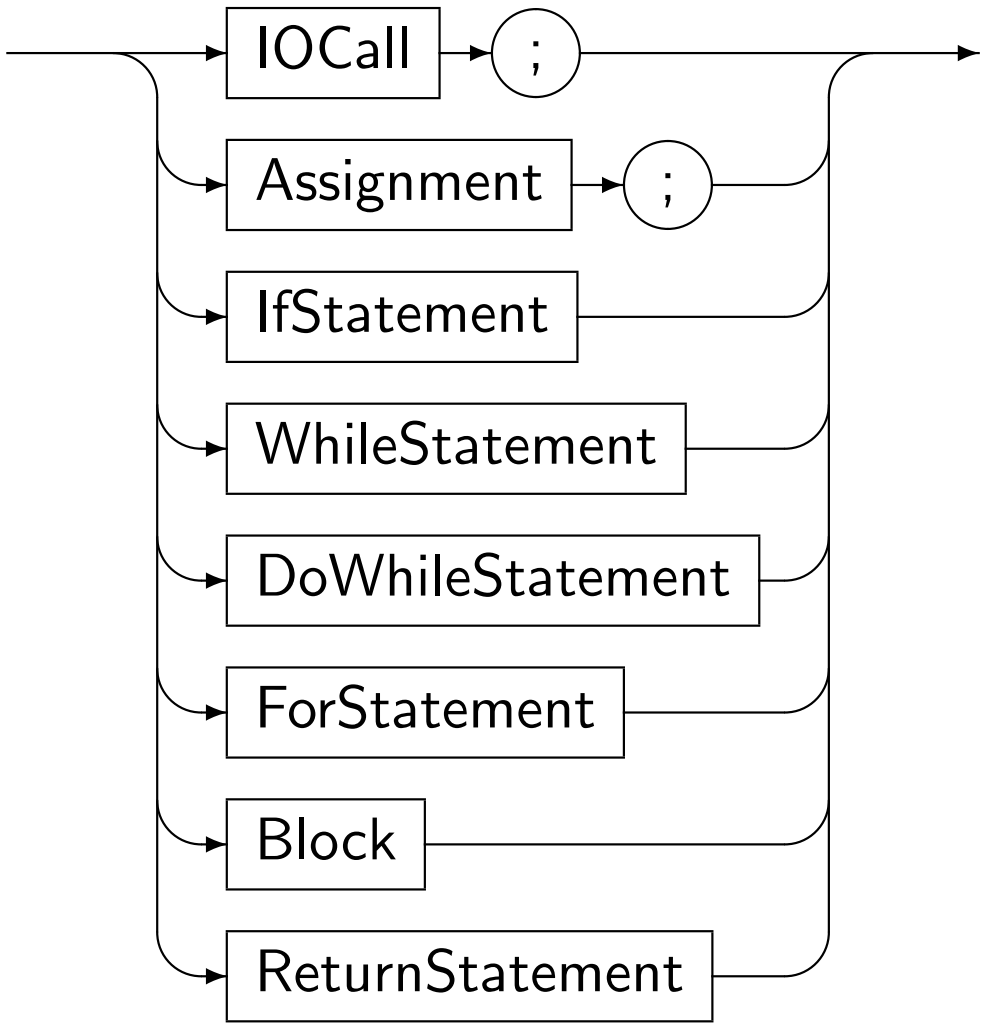
Zeit, Anweisungen zu geben

```
#include <stdio.h>

int main()
{  int n,s,i;
   scanf("%d",&n);
   s=0;
   i=1;
   while (i<=n)
      {  s=s+i*i;
         i=i+1;
      }
   printf("%d",s);
   return 0;
}
```



Beschränkung (zunächst) auf einige Anweisungsarten

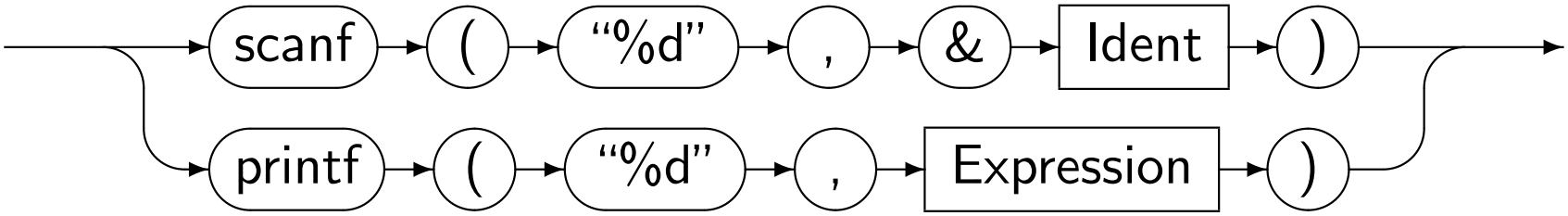
Statement



Nein, ich habe keine Semikolons vergessen.

Ein- und Ausgabe

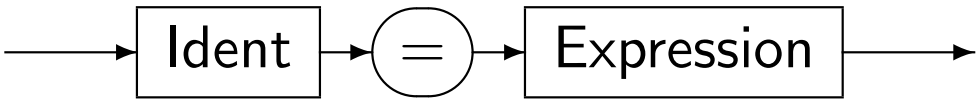
IOCall



- scanf liest (von der Tastatur) einen Zahlenwert ein und speichert ihn in angegebener Variable.
- printf gibt Wert eines angegebenen arithmetischen Ausdrucks aus (auf dem Bildschirm/Konsole).
- Ein- und Ausgabe anderer Arten von Daten auch möglich, im Moment aber nicht relevant.
- Hier Annahme, dass Eingaben immer sinnvoll (kein Nutzerfehler).

Zuweisungen und Ausdrücke

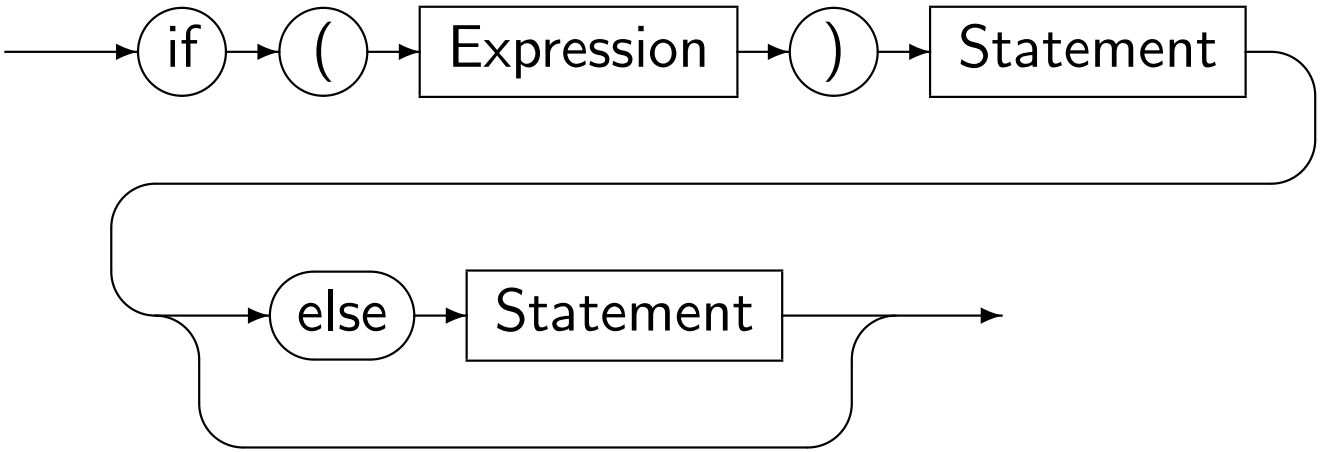
Assignment



- Wert des Ausdrucks (Expression) wird berechnet und der durch Ident bezeichneten Variable zugewiesen.
- Die Zielvariable kann selbst im Ausdruck vorkommen.
  $\longrightarrow$ Berechnung neuen Werts unter Nutzung alten Werts.
- Ausdrücke setzen sich zusammen aus:
  - festen Werten
  - Variablen
  - Operationen $+$, $-$, $*$, $/$, %
    (Standardpräzedenzen; gegebenenfalls Klammerung)
  - Vergleichsrelationen $==$, $!=$, $<$, $>$, $<=$, $>=$
  - logischen Verknüpfungen/Operationen && („und“), $||$ („oder“), ! („nicht“)
- Achtung: nicht $=$ (Zuweisung) und $==$ (Vergleich) verwechseln!

Bedingte Anweisung / Verzweigung

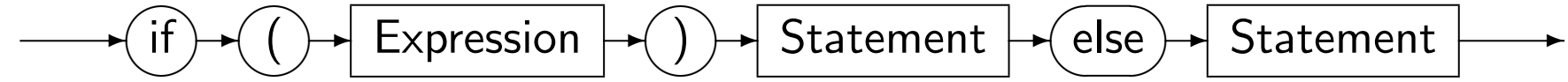
IfStatement

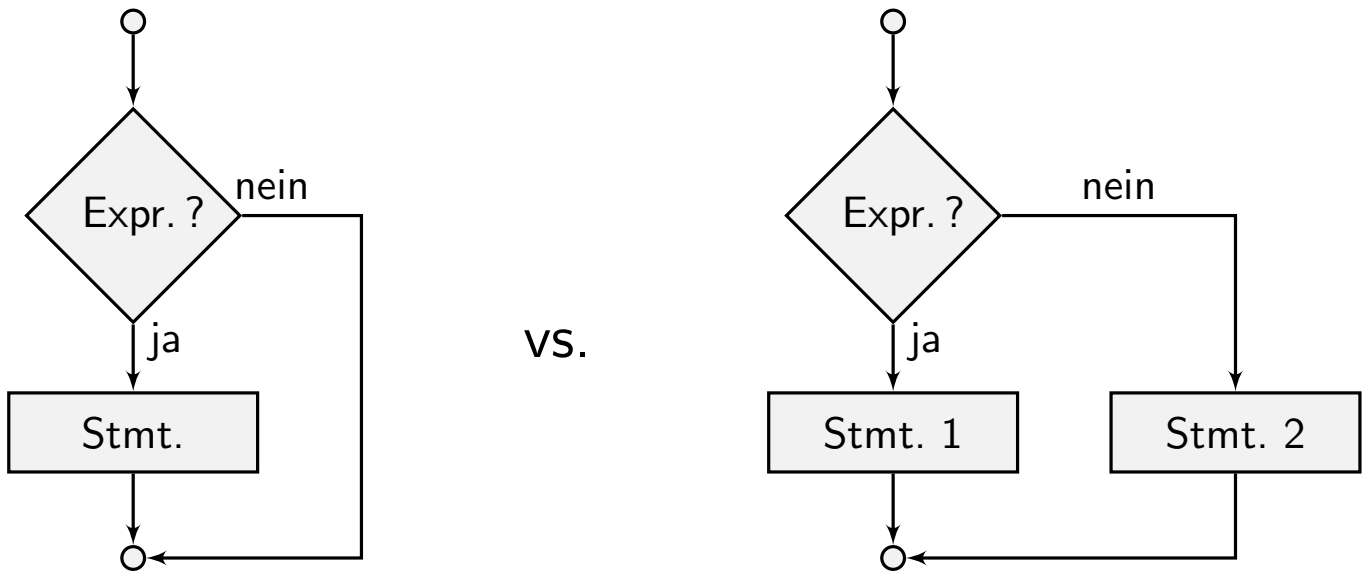


- Ausdruck (Bedingung) wird zu Wahrheitswert ausgewertet.
- Wenn Bedingung erfüllt, wird die erste Anweisung ausgeführt.
- Wenn Bedingung nicht erfüllt und keine zweite Anweisung vorhanden, gar keine Aktion.
- Wenn Bedingung nicht erfüllt und zweite Anweisung vorhanden, wird diese ausgeführt.

Bedingte Anweisung / Verzweigung

vs.



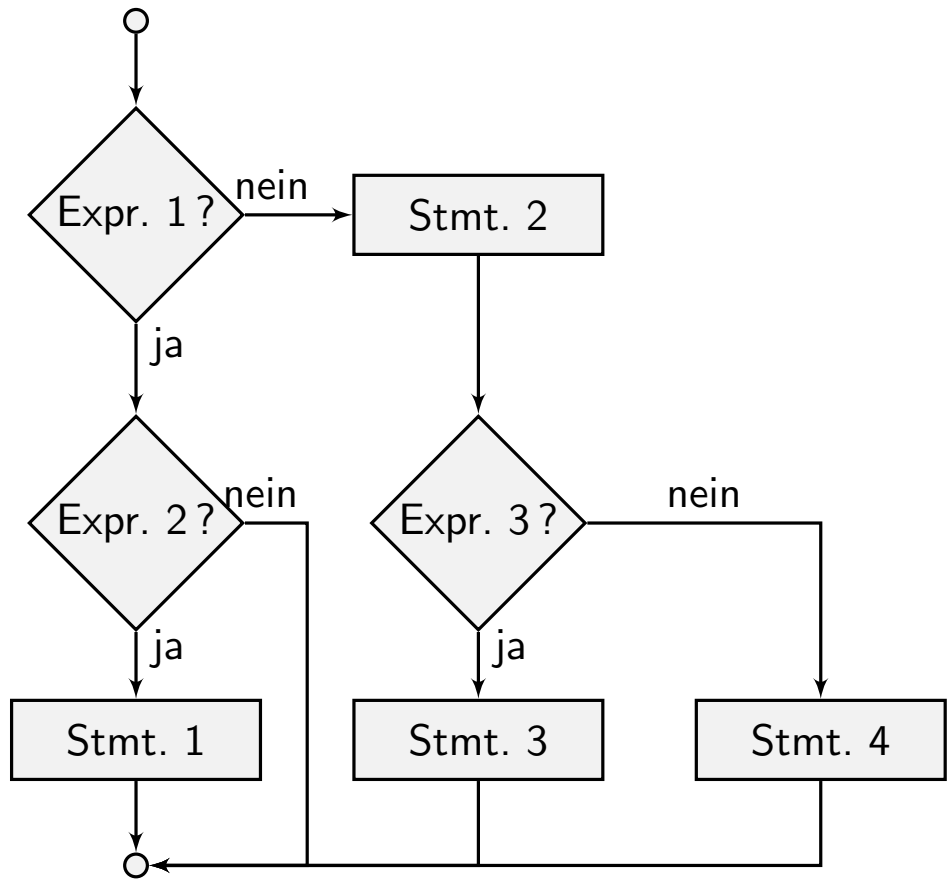


- Die Statements könnten natürlich mittels Block gebildet sein, also selbst mehrere Statements (inklusive **if**) enthalten.

Bedingte Anweisung / Verzweigung, verschachtelt

```
if (Expr. 1) {if (Expr. 2) Stmt. 1}
 else {
    Stmt. 2
    if (Expr. 3) Stmt. 3
    else Stmt. 4
 }
```

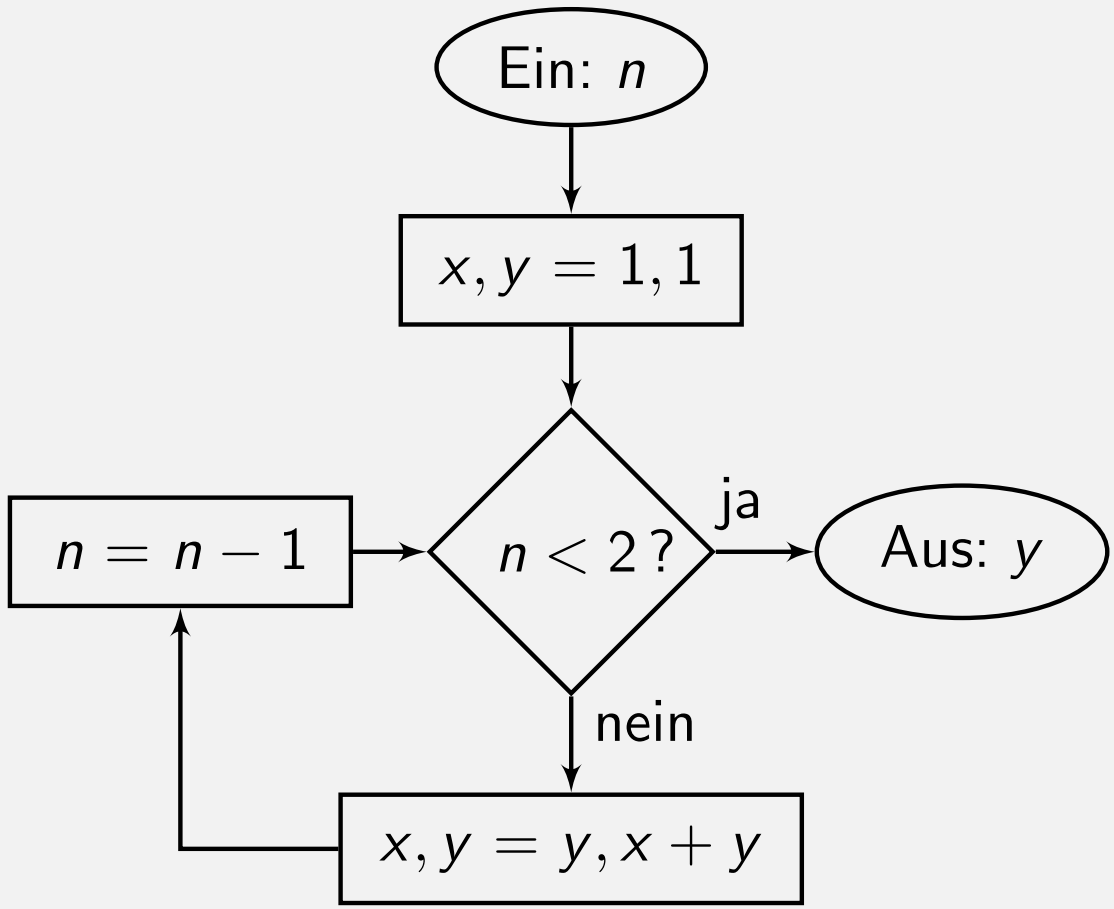
Ohne das erste {...}-Paar wäre es schwer gewesen, dies (so) zu interpretieren!



Bedingte Anweisung / Verzweigung, verschachtelt

```
if (Expr. 1) {if (Expr. 2) Stmt. 1}
 else
    St
    i
    e
 }
```

Allerdings gänzlich unmöglich, sowas umzusetzen:

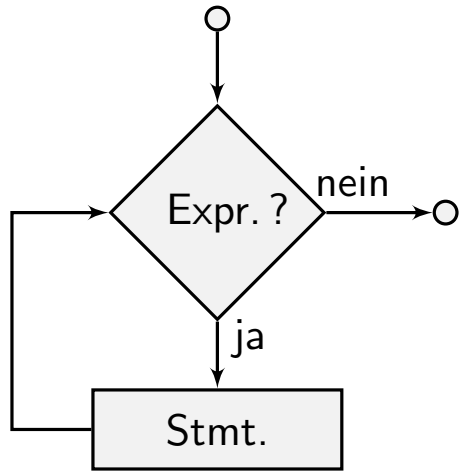


Ohne
wäre e
dies (s

nt. 4

Bedingte Schleife / Iteration

WhileStatement

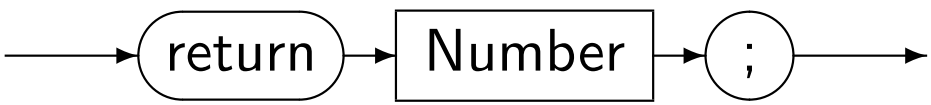


- Ausdruck (Bedingung) wird zu Wahrheitswert ausgewertet.
- Wenn Bedingung erfüllt, wird die Anweisung ausgeführt und danach der Test wiederholt.
- Erst wenn Bedingung nicht (mehr) erfüllt, Abbruch der Schleife.

Beenden des Programms

ReturnStatement

- Für den Moment, in der Regel einfach nur:



- Gibt Kontrolle an Aufrufer/Betriebssystem zurück.
- Jeder mögliche Ausführungspfad durch das Programm muss ein **return** erreichen.

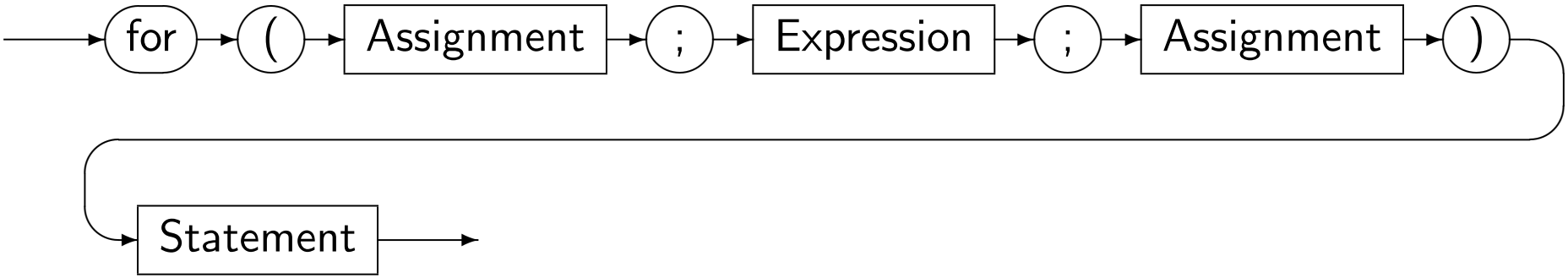
Nochmal unser einfaches Programm

```
#include <stdio.h>

int main()
{  int n,s,i;
   scanf("%d",&n);
   s=0;
   i=1;
   while (i<=n)
      {  s=s+i*i;
         i=i+1;
      }
   printf("%d",s);
   return 0;
}
```

- n=?, s=?, i=?
- Eingabe: 3 (zum Beispiel)
- n=3, s=?, i=?
- n=3, s=0, i=?
- n=3, s=0, i=1
- n=3, s=1, i=1
- n=3, s=1, i=2
- n=3, s=5, i=2
- n=3, s=5, i=3
- n=3, s=14, i=3
- n=3, s=14, i=4
- Ausgabe: 14
- Programmende

Schleifen mit fester Anzahl Durchläufe

ForStatement



- Die erste Zuweisung wird durchgeführt,
  dann der Ausdruck zu Wahrheitswert ausgewertet.
- Wenn erfüllt, wird die Anweisung im „Rumpf“ ausgeführt,
  dann die zweite Zuweisung, und danach der Test wiederholt.
- Sinnvolle Einschränkungen:
  - Der Ausdruck vergleicht die Variable aus der ersten Zuweisung
    mit einem anderen Wert.
  - Dieser wird durch Ausführung des Rumpfes nicht verändert.
  - Die zweite Zuweisung manipuliert auch nur die „Laufvariable“,
    durch festes Inkrement oder Dekrement.
  - Ausführung des Rumpfes verändert auch nicht Wert der Laufvariable.

Entsprechende Alternative unseres Programms

```c
#include <stdio.h>

int main()
{ int n,s,i;
  scanf("%d",&n);
  s=0;
  i=1;
  while (i<=n)
    { s=s+i*i;
      i=i+1;
    }
  printf("%d",s);
  return 0;
}
```

```c
#include <stdio.h>

int main()
{ int n,s,i;
  scanf("%d",&n);
  s=0;
  for (i=1; i<=n; i=i+1)
    s=s+i*i;
  printf("%d",s);
  return 0;
}
```

Geht das denn immer?

```
#include <stdio.h>

int main()
{ int a,b,r;
  printf("Bitte a eingeben: ");
  scanf("%d",&a);
  printf("Bitte b eingeben: ");
  scanf("%d",&b);
  r=a;
  while (r>=b) r=r-b;
  printf("Ergebnis von a %% b ist: %d",r);
  return 0;
}
```

- Anzahl der Durchläufe vor Eintritt in Schleife nicht bekannt.
  (Es sei denn, man kennt im Prinzip schon das Ergebnis.)

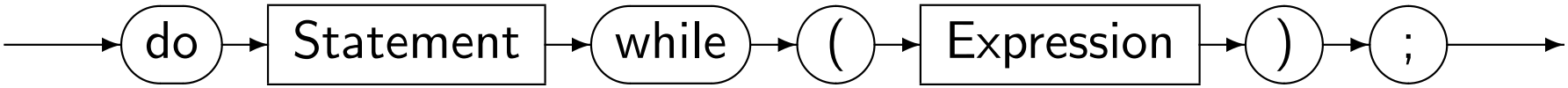
Geht es noch „schlimmer“?

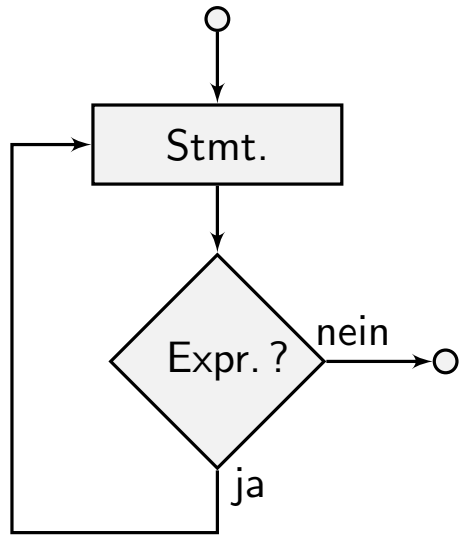
```
#include <stdio.h>

int main()
{ int n;
  printf("Bitte n eingeben: ");
  scanf("%d",&n);
  while (n>1)
    { if (n%2 == 0) n=n/2;
      else n=3*n+1;
      printf("%d\n",n);
    }
  return 0;
}
```

- Es ist ein noch ungelöstes Problem der Mathematik, ob diese Schleife überhaupt immer zum Ende kommt.

Nichtabweisende Schleife

DoWhileStatement

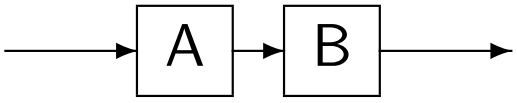




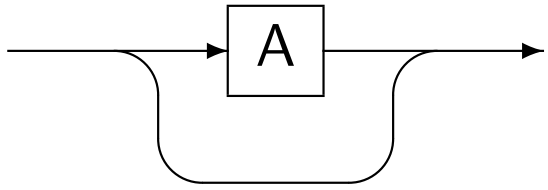
- Die Anweisung wird mindestens einmal ausgeführt.
- Jeweils danach finden Auswertung des Ausdrucks und Test auf Erfülltheit statt.

Etwas grobe Analogien zu Syntaxdiagramm-Mustern

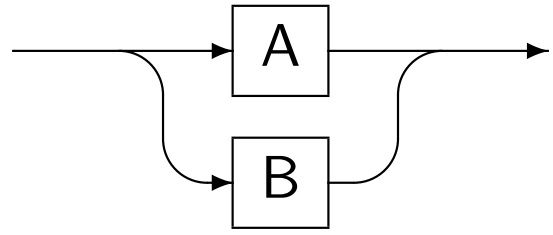
{ ... ; ... }:



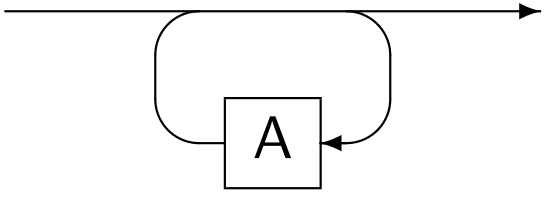
if:



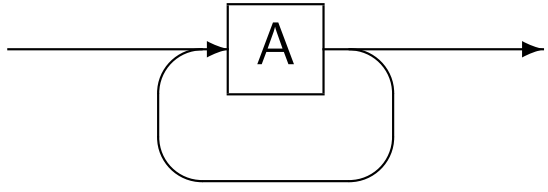
if/**else**:



while (und **for**):



do/**while**:



return:

???:

B