Algorithmisches Denken und imperative Programmierung

11. Vorlesung

Janis Voigtländer

Universität Bonn

Wintersemester 2012/13

Einschränkung von C zu einer Teilsprache C_0

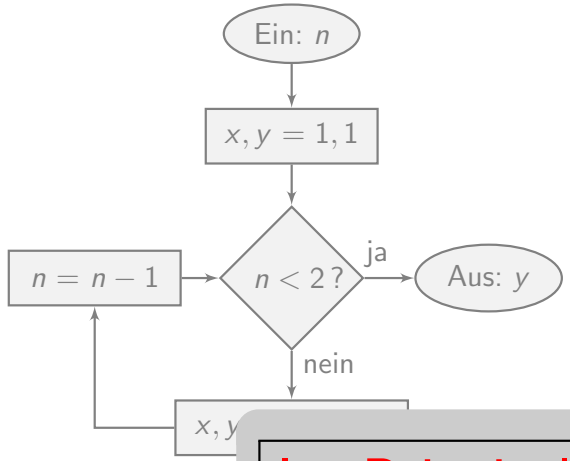
Motivation:

- Wir wollen C-Programmen eine formale Bedeutung geben (neben Hoare-Kalkül), ...
- außerdem allgemeine Prinzipien eines Compilers betrachten, ...
- um die Dinge einfach zu halten, jedoch nicht alle Sprachkonstrukte behandeln.

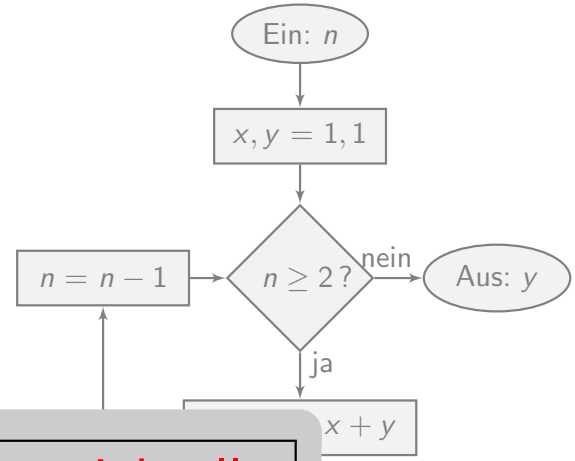
Einschränkungen:

- nur **int** als Typ
  (keine Structs, keine Pointer, keine Arrays)
- keine Funktionsdefinitionen (außer main)
- keine globalen Variablendeklarationen,
  keine Konstantendeklarationen
- lediglich **while**-Schleifen und **if**-Verzweigungen als Kontrollstrukturen
- keine Kommentare; und bestimmte andere syntaktische Freiheitsgrade ebenfalls entfernt

While-Programme — am Fibonacci-Beispiel



$\Rightarrow$



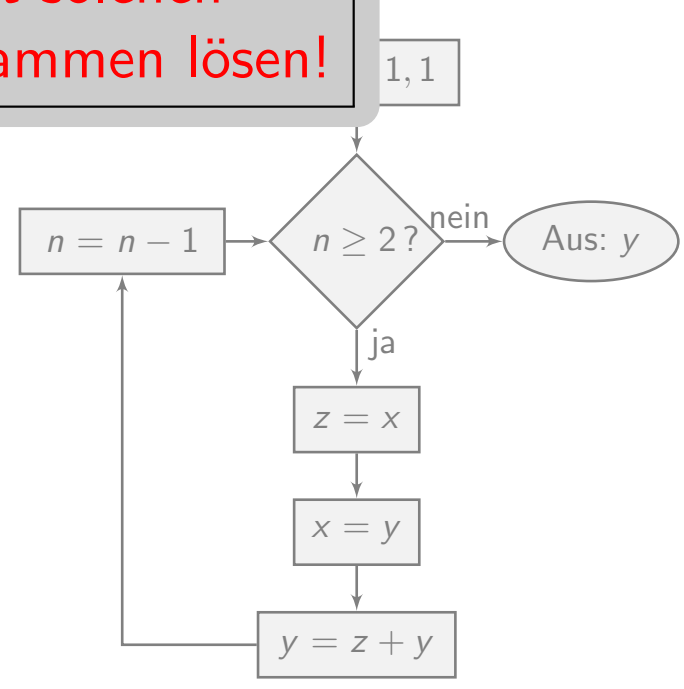
Im Prinzip lassen sich alle überhaupt berechenbaren Probleme mit solchen While-Programmen lösen!

Programm:

```
#include <stdio.h>

int main()
{ int n,x,y,z;
  scanf("%d",&n);
  x=1;
  y=1;
  while (n>=2)
    {z=x; x=y; y=z+y; n=n-1;}
  printf("%d",y);
  return 0; }
```

$\Leftarrow$



Beispiel für ein C_0-Programm

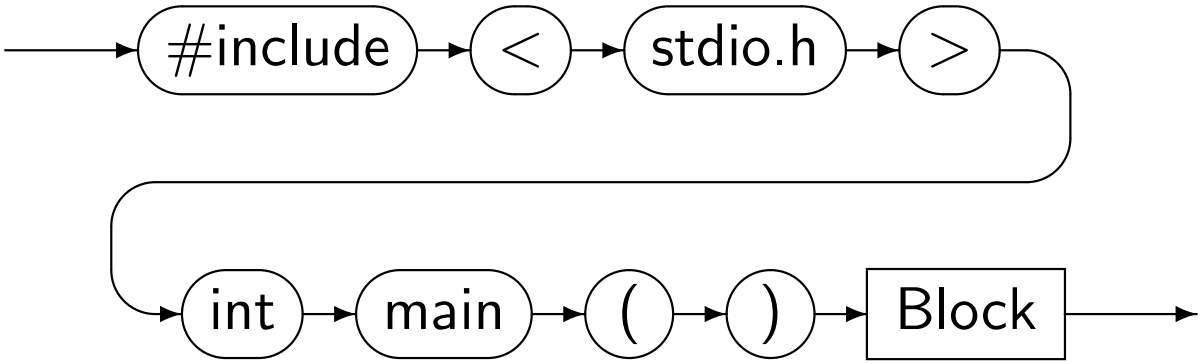
```
#include <stdio.h>

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

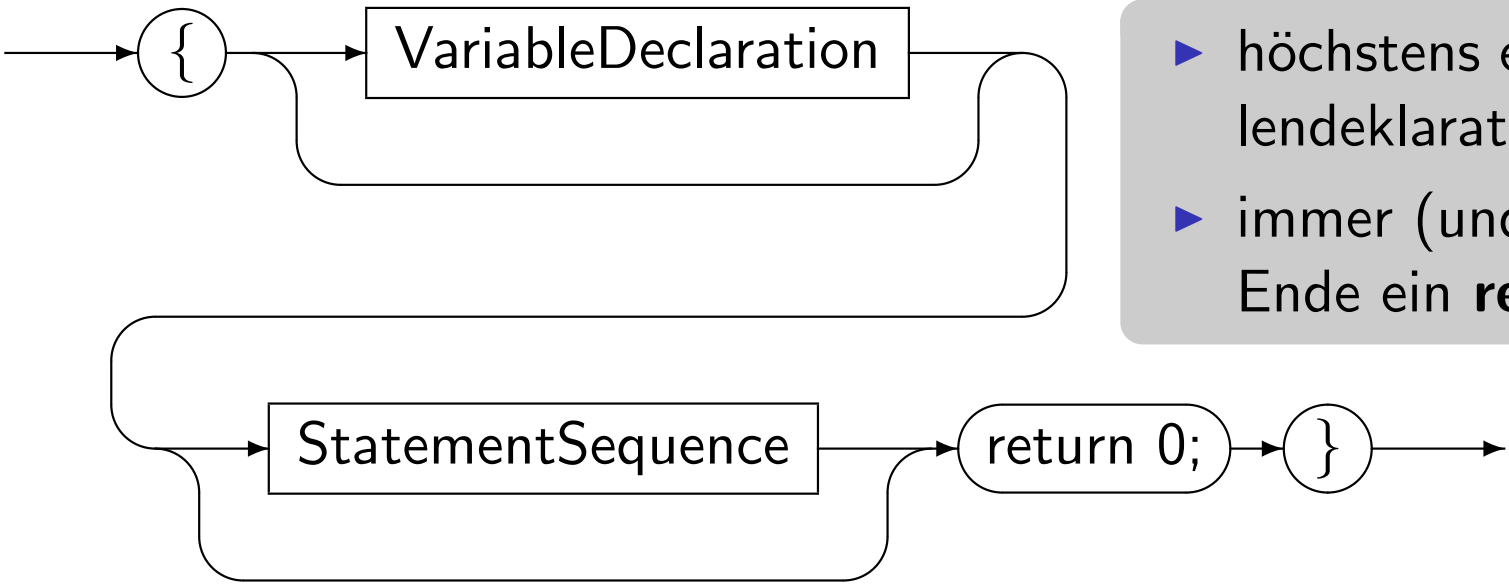
Formale Beschreibung: Syntaxdiagramme

Syntaxdiagramme für C_0: Programmstruktur (I)

Program



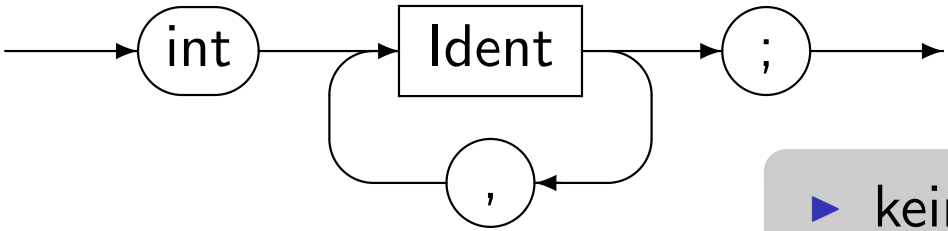
Block



- höchstens eine Variablendeklarationszeile
- immer (und nur) am Ende ein **return** 0;

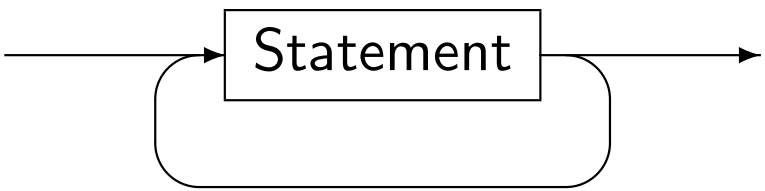
Syntaxdiagramme für C_0: Programmstruktur (II)

VariableDeclaration



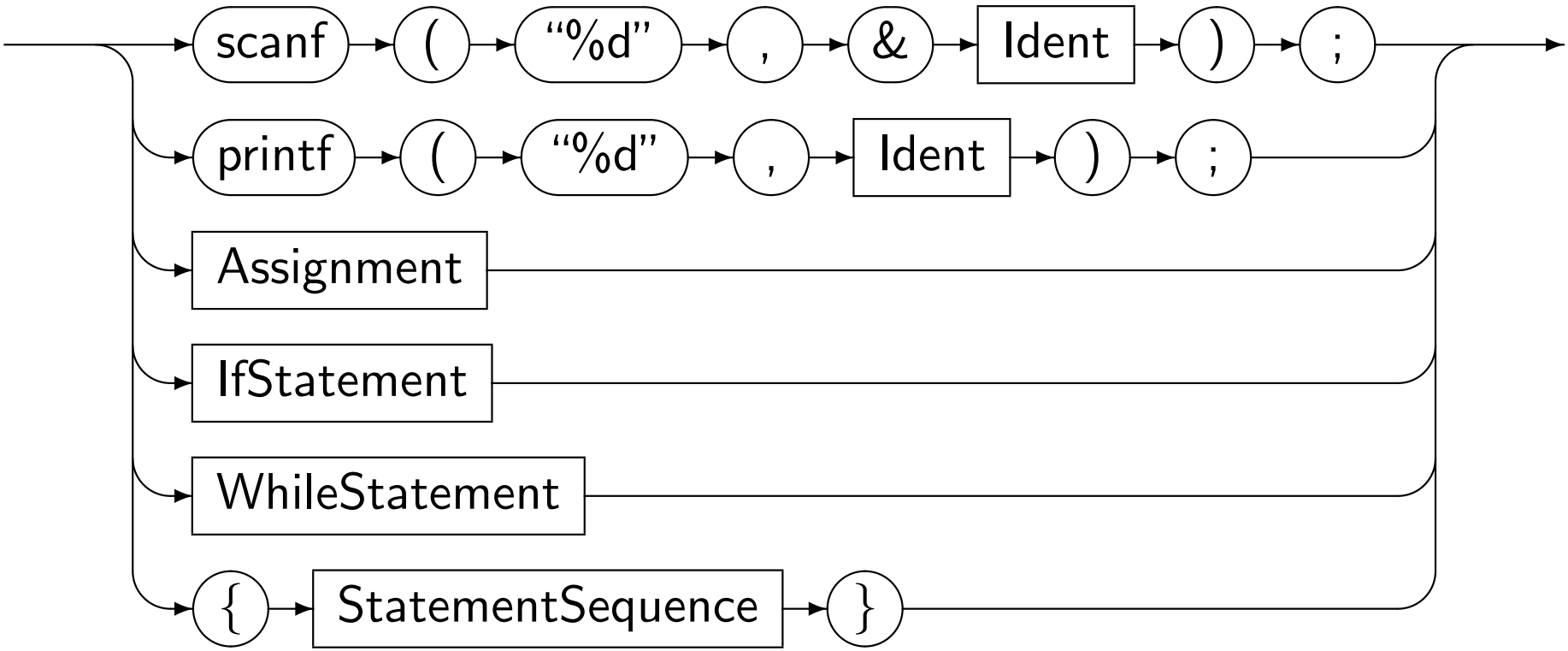
- keine Initialisierung von Variablen bei Deklaration

StatementSequence



Syntaxdiagramme für C_0: Programmstruktur (III)

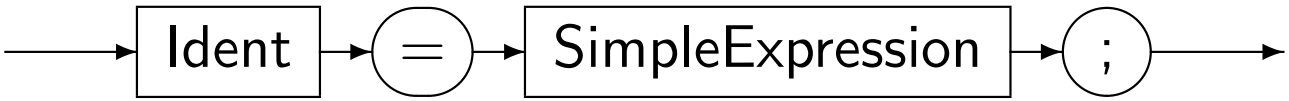
Statement



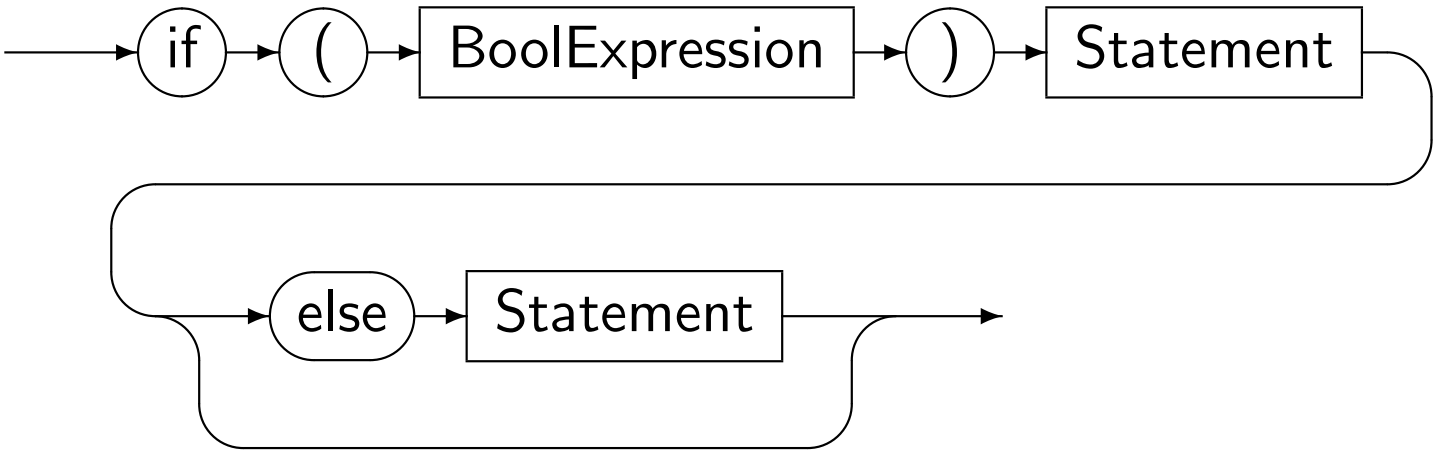
- bei printf kein beliebiger Ausdruck als auszugebendes Argument
- keine **do-while**-Schleife, keine **for**-Schleife

Syntaxdiagramme für C_0: Programmstruktur (IV)

Assignment



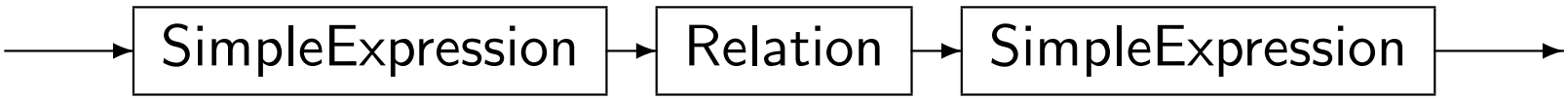
IfStatement



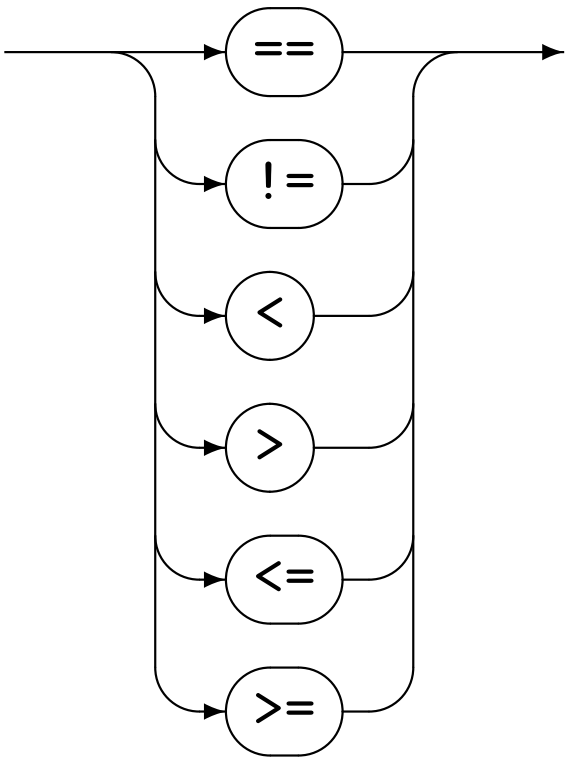
WhileStatement

Syntaxdiagramme für C_0: Ausdrücke (I)

BoolExpression



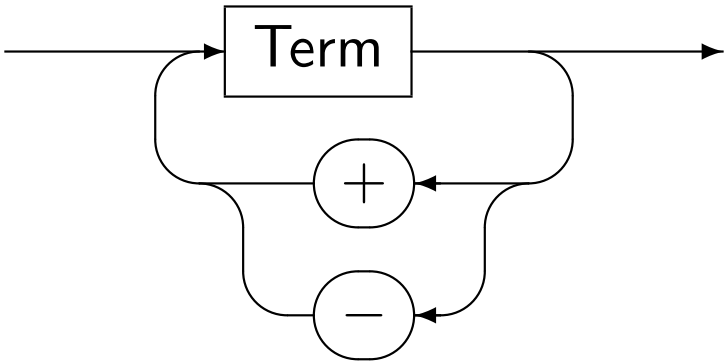
Relation



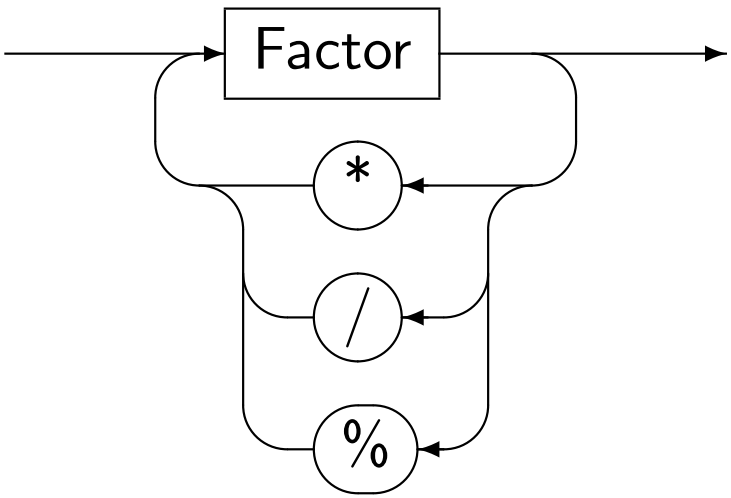
- nicht mal logische Verknüpfungs-operationen eingebaut

Syntaxdiagramme für C_0: Ausdrücke (II)

SimpleExpression

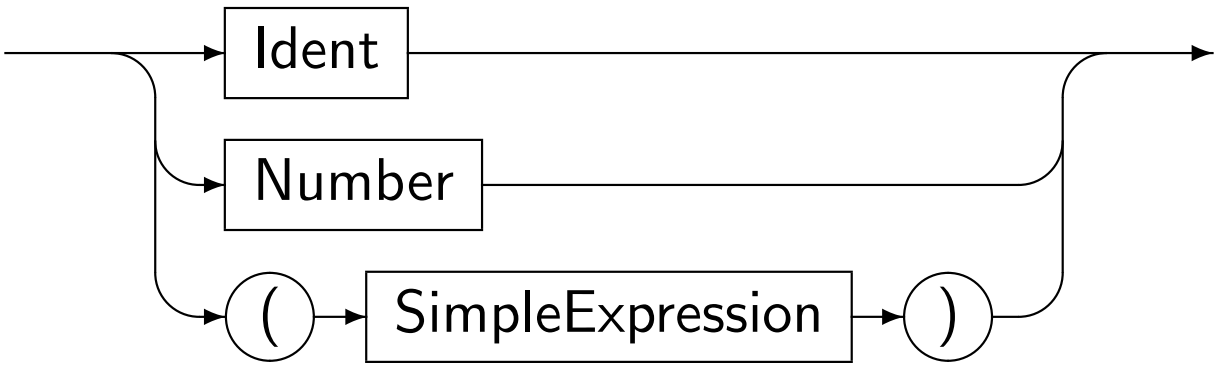


Term



Syntaxdiagramme für C_0: Ausdrücke (III)

Factor



Ident

...

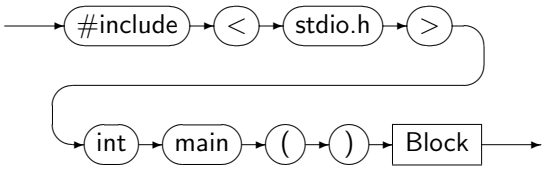
Number

...

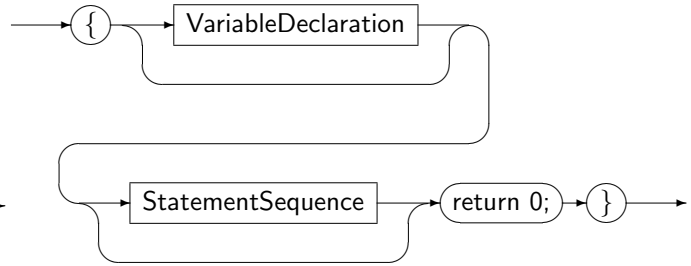
(lexikalische Syntax wie in C, natürlich nur ganze Zahlen)

Zusammenfassung Syntax C_0

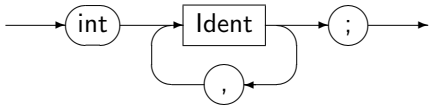
Program



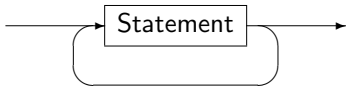
Block



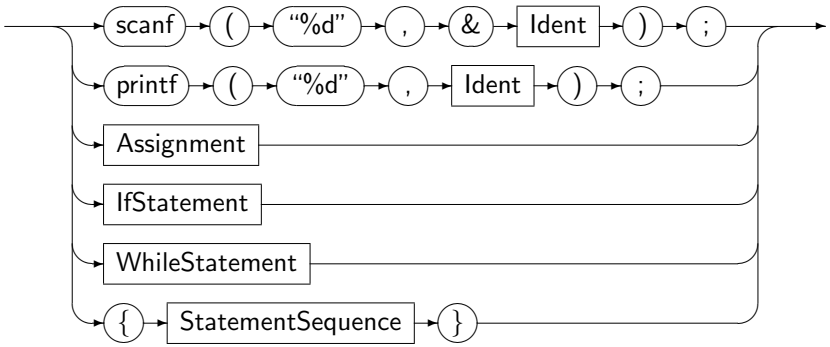
VariableDeclaration



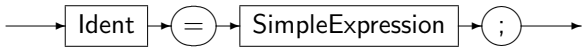
StatementSequence



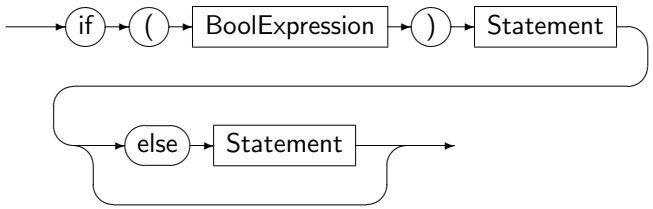
Statement



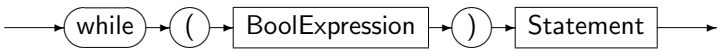
Assignment



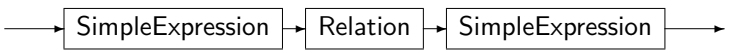
IfStatement



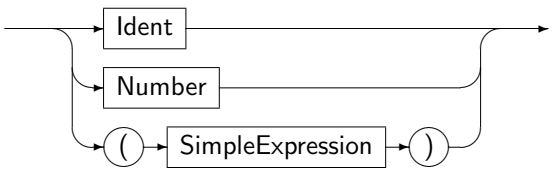
WhileStatement



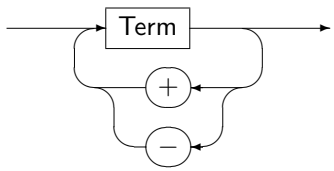
BoolExpression



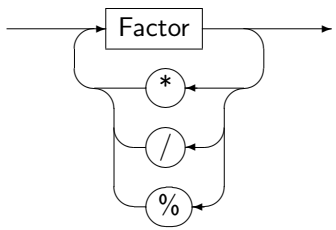
Factor



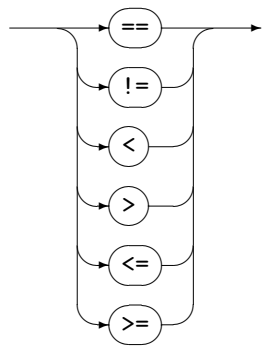
SimpleExpression



Term



Relation



Ident . . . **Number . . .**

Kontextsensitive Nebenbedingungen in C_0

Nicht durch Syntaxdiagramme ausdrückbare Forderungen an C-Programme gelten weiterhin, insbesondere:

- Jeder Bezeichner darf höchstens einmal in der eventuell vorhandenen VariableDeclaration deklariert sein.
- Wenn ein Bezeichner in der eventuell vorhandenen StatementSequence des Blocks auftritt, so muss er in der (dann unbedingt vorhandenen) VariableDeclaration des Blocks deklariert sein.

Insbesondere ist somit jedes C_0-Programm auch ein gültiges C-Programm.

Realisierung in „Maschinensprache“

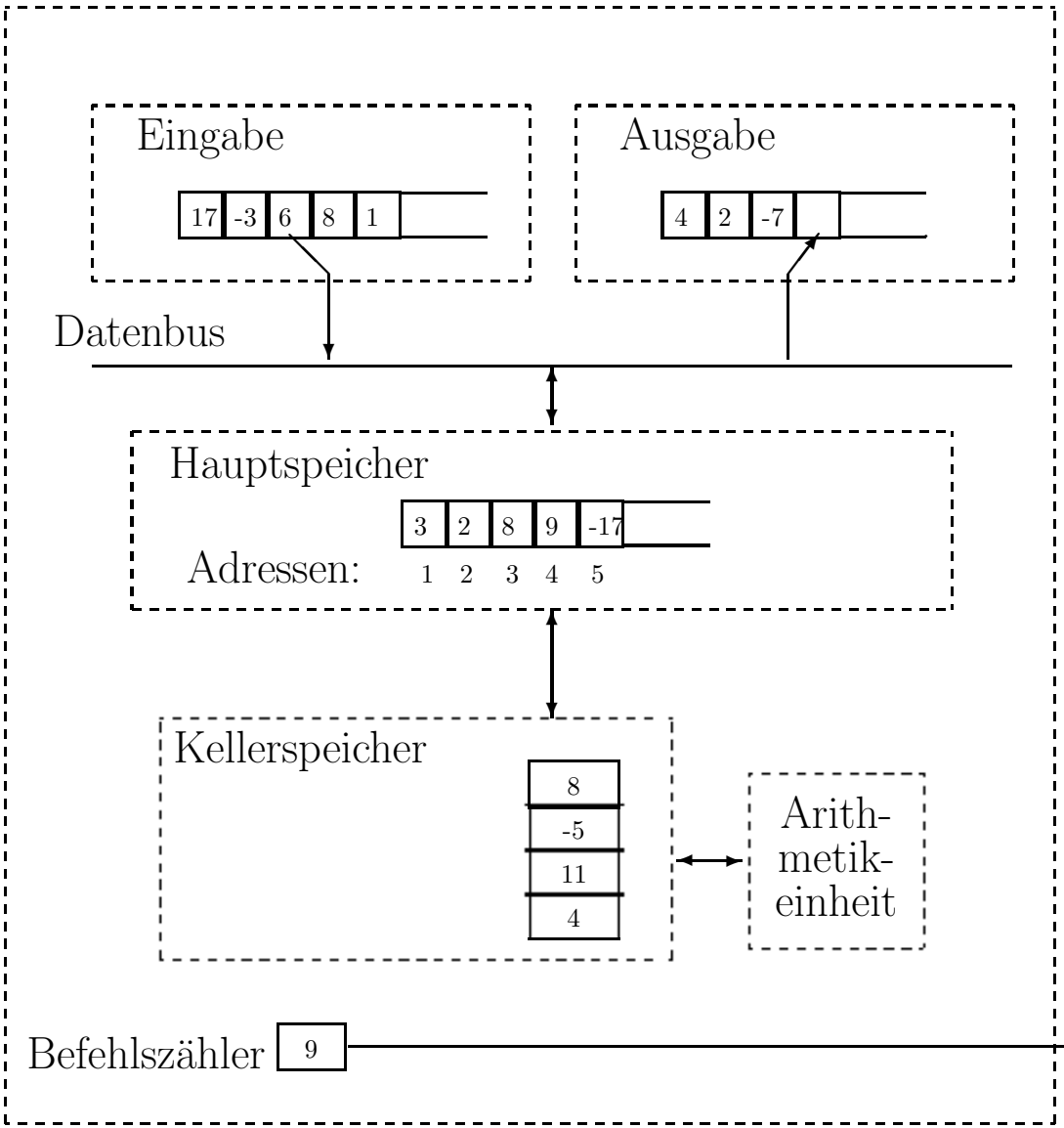
Modellierung/Umsetzung von C_0-Sprachfeatures

```
#include <stdio.h>

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

Wir brauchen: Ein- und Ausgabe, Speicherplätze, Auswertung von Ausdrücken, Zuweisungen, Kontrollfluss

Eine abstrakte Maschine (AM)

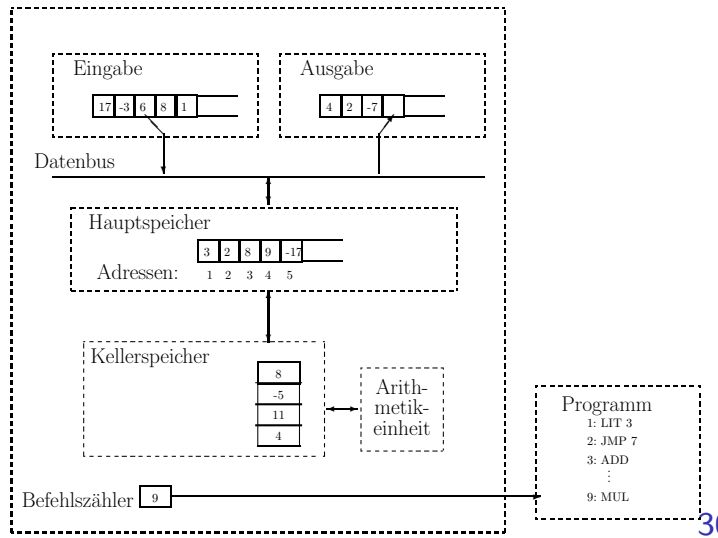


Essentiell:

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

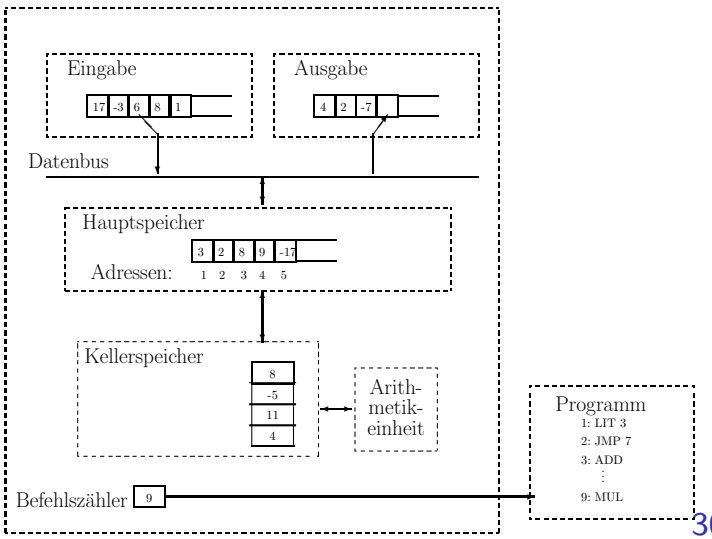
Modellierung des Hauptspeichers

- statt benannter Speicherplätze (Variablen) lediglich numerische Adressen
- einzig möglicher Speicherinhalt: ganze Zahl (**int**)
- konkreter Hauptspeicherinhalt modelliert als partielle Abbildung von Adressen auf Inhalte
- mathematisch: $\mathrm{HS} = \{h \mid h : \mathbb{N}_+ \rightharpoonup \mathbb{Z}\}$
- Notation: etwa $h = [1/3,\ 2/2,\ 3/5]$
- Update: $h[n/d]$;
  $h[n/d](n') = d$, falls $n' = n$, sonst $h[n/d](n') = h(n')$
- Beispiele für $h = [1/3,\ 2/2,\ 3/5]$:
  $h[5/7] = [1/3,\ 2/2,\ 3/5,\ 5/7]$ und
  $h[2/7] = [1/3,\ 2/7,\ 3/5]$
- Befehle zur Kommunikation mit:
  - Ein- und Ausgabe
  - „Berechnungseinheit“



Modellierung von Ein- und Ausgabe

- lediglich Ein- und Ausgabe ganzer Zahlen (**int**)
- Abstraktion von interaktiver Ein- und Ausgabe; stattdessen: Eingabeband und Ausgabeband
- Bandinhalte modelliert als endliche Listen
- mathematisch: $\mathsf{Inp} = \mathsf{Out} = \mathbb{Z}^*$
- Notation: etwa $\mathit{inp} = 1.13.5$ oder $\mathit{out} = \varepsilon$
- READ n: Lesen/Konsumieren erster Position von *inp* und entsprechendes Update der Position n des HS
- WRITE n: Ausgabe des HS-Inhalts von Position n am Ende von *out*



Auswertung von Ausdrücken, Zuweisungen

Problem:

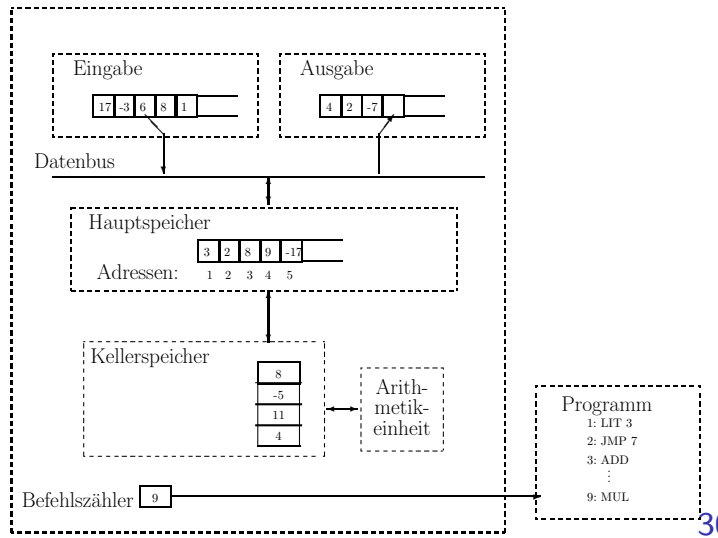
- Wir wollen strukturierte Ausdrücke wie (3+5)∗(7−2) oder j>2∗i+1 auswerten.
- Dies soll jedoch in „linearisierter Form“ erfolgen, insbesondere soll eine Operation immer nur auf zwei Operanden wirken.
- Folglich müssen Zwischenergebnisse berechnet und in geeigneter Weise vorrätig gehalten werden.

Lösung:

- Verwendung eines „Datenkellers“
- Ablage und Entnahme jeweils nur an Kellerspitze
- mathematisch: DK $= \mathbb{Z}^*$
- Notation: etwa $d = 8{:}7{:}2$
- LIT z: Ablage einer Konstante
- LOAD n: Ablage des HS-Inhalts von Position n
- ADD, MUL, . . . , LT, EQ, NE, . . . : Berechnungen
- STORE n: Entnahme und entsprechendes Update der Position n des HS

Kontrollfluss — Abarbeitungsreihenfolge

- Durchnummerierung aller Befehle in einem Programm
- aktuelle Position im Befehlszähler gespeichert ($m \in \mathrm{BZ} = \mathbb{N}$)
- normalerweise einfache Erhöhung nach jeder Befehlsabarbeitung
- JMP n: Sprung an Position n
- JMC n: bedingter Sprung an Position n;
  wenn Spitze des Datenkellers gleich 0, dann Sprung;
  wenn Spitze des Datenkellers gleich 1, dann weiter mit nächstem Befehl



Also, vollständige Befehlsübersicht:

- READ n: Lesen von Eingabeband in Hauptspeicher
- WRITE n: Ausgabe aus Hauptspeicher auf Ausgabeband
- LOAD n: Ablage aus Hauptspeicher auf Datenkeller
- STORE n: Entnahme aus Datenkeller in Hauptspeicher
- LIT z: Ablage einer Konstante auf Datenkeller
- ADD, MUL, SUB, DIV, MOD, LT, EQ, NE, GT, LE, GE: Berechnungen und Vergleiche (auf Datenkeller)
- JMP n: Sprung
- JMC n: bedingter Sprung abhängig von Wert auf Datenkeller

Formale Modellierung

- $\mathsf{AM} = \mathsf{BZ} \times \mathsf{DK} \times \mathsf{HS} \times \mathsf{Inp} \times \mathsf{Out}$, mit:

BZ	$= \mathbb{N}$	Befehlszähler
DK	$= \mathbb{Z}^*$	Datenkeller
HS	$= \{h \mid h : \mathbb{N}_+ \rightharpoonup \mathbb{Z}\}$	Hauptspeicher
Inp	$= \mathbb{Z}^*$	Eingabeband
Out	$= \mathbb{Z}^*$	Ausgabeband

- Maschine ist jederzeit in einem Zustand $s = (m, d, h, inp, out) \in \mathsf{AM}$.
- Ein Programm ist eine partielle, endlich definierte Abbildung P von $\mathbb{N}$ auf die Menge Γ aller Befehle.
- Notation: 1: P(1);
  2: P(2);
  3: P(3);
  ...
- Programmabarbeitung besteht aus wiederholter Anwendung von Befehlen auf einen Zustand.

Befehlssemantik (I)

$$\mathcal{C}[\![.]\!] : \Gamma \longrightarrow (\mathrm{AM} \rightharpoonup \mathrm{AM})$$

$$\mathcal{C}[\![\mathrm{READ}\ n]\!](m, d, h, inp, out) =$$
$$\quad \text{wenn } inp = first(inp).rest(inp) \text{ mit } first(inp) \in \mathbb{Z}, rest(inp) \in \mathbb{Z}^*,$$
$$\quad \text{dann } (m+1, d, h[n/first(inp)], rest(inp), out)$$

$$\mathcal{C}[\![\mathrm{WRITE}\ n]\!](m, d, h, inp, out) =$$
$$\quad \text{wenn } h(n) \in \mathbb{Z}, \text{ dann } (m+1, d, h, inp, out.h(n))$$

$$\mathcal{C}[\![\mathrm{LOAD}\ n]\!](m, d, h, inp, out) =$$
$$\quad \text{wenn } h(n) \in \mathbb{Z}, \text{ dann } (m+1, h(n) : d, h, inp, out)$$

$$\mathcal{C}[\![\mathrm{STORE}\ n]\!](m, d, h, inp, out) =$$
$$\quad \text{wenn } d = d.1 : d', \text{ dann } (m+1, d', h[n/d.1], inp, out)$$

$$\mathcal{C}[\![\mathrm{LIT}\ z]\!](m, d, h, inp, out) = (m+1, z : d, h, inp, out)$$

Befehlssemantik (II)

$\mathcal{C}[\![\text{ADD}]\!](m, d, h, inp, out) =$
wenn $d = d.1 : d.2 : d'$, dann $(m + 1, (d.2 + d.1) : d', h, inp, out)$

für MUL, SUB, DIV und MOD analog

$\mathcal{C}[\![\text{LT}]\!](m, d, h, inp, out) =$
wenn $d = d.1 : d.2 : d'$, dann $(m + 1, b : d', h, inp, out)$,
wobei $b = 1$, falls $d.2 < d.1$, sonst $b = 0$

für EQ, NE, GT, LE und GE analog

$\mathcal{C}[\![\text{JMP } e]\!](m, d, h, inp, out) = (e, d, h, inp, out)$

$\mathcal{C}[\![\text{JMC } e]\!](m, d, h, inp, out) =$
wenn $d = 0 : d'$, dann (e, d', h, inp, out);
wenn $d = 1 : d'$, dann $(m + 1, d', h, inp, out)$

Programmsemantik — am Beispiel

```
 1: READ 2;      8: LE;          15: STORE 3;
 2: LIT 1;       9: JMC 21;      16: LOAD 1;
 3: STORE 1;    10: LOAD 3;      17: LIT 1;
 4: LIT 0;      11: LOAD 1;      18: ADD;
 5: STORE 3;    12: LOAD 1;      19: STORE 1;
 6: LOAD 1;     13: MUL;         20: JMP 6;
 7: LOAD 2;     14: ADD;         21: WRITE 3;
```

$$
\begin{array}{llll}
(& 1 , & \varepsilon , [] & , 2 , \varepsilon) \\
(& 2 , & \varepsilon , [2/2] & , \varepsilon , \varepsilon)
\end{array}
$$

$\mathcal{C}[\![\text{READ } n]\!](m, d, h, inp, out) =$
wenn $inp = first(inp).rest(inp)$ mit $first(inp) \in \mathbb{Z}, rest(inp) \in \mathbb{Z}^*$,
dann $(m + 1, d, h[n/first(inp)], rest(inp), out)$

$n = 2, \quad m = 1, \quad d = \varepsilon, \quad h = [], \quad inp = 2, \quad out = \varepsilon$

Programmsemantik — am Beispiel

```
 1: READ 2;        8: LE;          15: STORE 3;
 2: LIT 1;         9: JMC 21;      16: LOAD 1;
 3: STORE 1;      10: LOAD 3;      17: LIT 1;
 4: LIT 0;        11: LOAD 1;      18: ADD;
 5: STORE 3;      12: LOAD 1;      19: STORE 1;
 6: LOAD 1;       13: MUL;         20: JMP 6;
 7: LOAD 2;       14: ADD;         21: WRITE 3;
```

$$
\begin{array}{lrlll}
(& 7 , & 3 , & [1/3,2/2,3/5] , & \varepsilon \ , \varepsilon) \\
(& 8 , & 2{:}3 , & [1/3,2/2,3/5] , & \varepsilon \ , \varepsilon) \\
(& 9 , & 0 , & [1/3,2/2,3/5] , & \varepsilon \ , \varepsilon) \\
(& 21 , & \varepsilon , & [1/3,2/2,3/5] , & \varepsilon \ , \varepsilon) \\
(& 22 , & \varepsilon , & [1/3,2/2,3/5] , & \varepsilon \ , 5)
\end{array}
$$

$$
\mathcal{C}[\![\text{WRITE } n]\!](m, d, h, inp, out) =
$$
$$
\quad \text{wenn } h(n) \in \mathbb{Z}, \text{ dann } (m+1, d, h, inp, out.h(n))
$$

Programmsemantik — formal

(zur Erinnerung: Befehlssemantik $\mathcal{C}[\![.]\!] : \Gamma \longrightarrow (\mathrm{AM} \dashrightarrow \mathrm{AM})$)

Sei *Prog* die Menge aller AM-Programme.

$\mathcal{I}[\![.]\!] : \mathit{Prog} \longrightarrow (\mathrm{AM} \dashrightarrow \mathrm{AM})$

$$\mathcal{I}[\![\mathrm{P}]\!](m, d, h, \mathit{inp}, \mathit{out}) = \begin{cases} \mathcal{I}[\![\mathrm{P}]\!](\mathcal{C}[\![\mathrm{P(m)}]\!](m, d, h, \mathit{inp}, \mathit{out})), & \text{falls } m \in \mathit{def}(\mathrm{P}) \\ (m, d, h, \mathit{inp}, \mathit{out}), & \text{falls } m \notin \mathit{def}(\mathrm{P}) \end{cases}$$

$\mathcal{P}[\![.]\!] : \mathit{Prog} \longrightarrow (\mathrm{Inp} \dashrightarrow \mathrm{Out})$

$\mathcal{P}[\![\mathrm{P}]\!](\mathit{inp}) = \mathit{proj}_5^{(5)}(\mathcal{I}[\![\mathrm{P}]\!](1, \varepsilon, [], \mathit{inp}, \varepsilon))$

- $\mathit{proj}_5^{(5)}$ meint Projektion auf fünfte Komponente eines Fünftupels.

Weiteres Beispiel

Aufgabe:
- Einlesen einer Folge (Ende: 0) vom Eingabeband
- Ausgabe der Gesamtsumme auf Ausgabeband

```
 1: LIT 0;          6: NE;          11: STORE 2;
 2: STORE 2;        7: JMC 13;      12: JMP 3;
 3: READ 1;         8: LOAD 2;      13: WRITE 2;
 4: LOAD 1;         9: LOAD 1;
 5: LIT 0;         10: ADD;
```

```
(  1 ,   ε , []          , 3.4.2.0 , ε )
(  2 ,   0 , []          , 3.4.2.0 , ε )
(  3 ,   ε , [2/0]       , 3.4.2.0 , ε )
(  4 ,   ε , [1/3,2/0] , 4.2.0   , ε )
(  5 ,   3 , [1/3,2/0] , 4.2.0   , ε )
(  6 , 0:3 , [1/3,2/0] , 4.2.0   , ε )
(  7 ,   1 , [1/3,2/0] , 4.2.0   , ε )
(  8 ,   ε , [1/3,2/0] , 4.2.0   , ε )
```

Weiteres Beispiel

Aufgabe:
- Einlesen einer Folge (Ende: 0) vom Eingabeband
- Ausgabe der Gesamtsumme auf Ausgabeband

```
1: LIT 0;
2: STORE 2;
3: READ 1;
4: LOAD 1;
5: LIT 0;
6: NE;
7: JMC 13;
8: LOAD 2;
9: LOAD 1;
10: ADD;
11: STORE 2;
12: JMP 3;
13: WRITE 2;
```

$$
\begin{array}{llllll}
(& 8 , & \varepsilon , & [1/3,2/0] , & 4.2.0 & , \varepsilon) \\
(& 9 , & 0 , & [1/3,2/0] , & 4.2.0 & , \varepsilon) \\
(& 10 , & 3{:}0 , & [1/3,2/0] , & 4.2.0 & , \varepsilon) \\
(& 11 , & 3 , & [1/3,2/0] , & 4.2.0 & , \varepsilon) \\
(& 12 , & \varepsilon , & [1/3,2/3] , & 4.2.0 & , \varepsilon) \\
(& 3 , & \varepsilon , & [1/3,2/3] , & 4.2.0 & , \varepsilon) \\
(& 4 , & \varepsilon , & [1/4,2/3] , & 2.0 & , \varepsilon) \\
(& 5 , & 4 , & [1/4,2/3] , & 2.0 & , \varepsilon)
\end{array}
$$

Weiteres Beispiel

Aufgabe:

- Einlesen einer Folge (Ende: 0) vom Eingabeband
- Ausgabe der Gesamtsumme auf Ausgabeband

```
 1: LIT 0;          6: NE;          11: STORE 2;
 2: STORE 2;        7: JMC 13;      12: JMP 3;
 3: READ 1;         8: LOAD 2;      13: WRITE 2;
 4: LOAD 1;         9: LOAD 1;
 5: LIT 0;         10: ADD;
```

$$
\begin{array}{lrrrr}
(& 5 , & 4 , & [1/4,2/3] , & 2.0 , \varepsilon) \\
(& 6 , & 0{:}4 , & [1/4,2/3] , & 2.0 , \varepsilon) \\
(& 7 , & 1 , & [1/4,2/3] , & 2.0 , \varepsilon) \\
(& 8 , & \varepsilon , & [1/4,2/3] , & 2.0 , \varepsilon) \\
(& 9 , & 3 , & [1/4,2/3] , & 2.0 , \varepsilon) \\
(& 10 , & 4{:}3 , & [1/4,2/3] , & 2.0 , \varepsilon) \\
(& 11 , & 7 , & [1/4,2/3] , & 2.0 , \varepsilon) \\
(& 12 , & \varepsilon , & [1/4,2/7] , & 2.0 , \varepsilon)
\end{array}
$$

Weiteres Beispiel

Aufgabe:
- Einlesen einer Folge (Ende: 0) vom Eingabeband
- Ausgabe der Gesamtsumme auf Ausgabeband

```
 1: LIT 0;
 2: STORE 2;
 3: READ 1;
 4: LOAD 1;
 5: LIT 0;
 6: NE;
 7: JMC 13;
 8: LOAD 2;
 9: LOAD 1;
10: ADD;
11: STORE 2;
12: JMP 3;
13: WRITE 2;
```

$$
\begin{array}{llll}
(& 12 , & \varepsilon , [1/4,2/7] , & 2.0 \quad , \varepsilon) \\
(& 3 , & \varepsilon , [1/4,2/7] , & 2.0 \quad , \varepsilon) \\
(& 4 , & \varepsilon , [1/2,2/7] , & 0 \quad , \varepsilon) \\
(& 5 , & 2 , [1/2,2/7] , & 0 \quad , \varepsilon) \\
(& 6 , & 0{:}2 , [1/2,2/7] , & 0 \quad , \varepsilon) \\
(& 7 , & 1 , [1/2,2/7] , & 0 \quad , \varepsilon) \\
(& 8 , & \varepsilon , [1/2,2/7] , & 0 \quad , \varepsilon) \\
(& 9 , & 7 , [1/2,2/7] , & 0 \quad , \varepsilon)
\end{array}
$$

Weiteres Beispiel

Aufgabe:

- Einlesen einer Folge (Ende: 0) vom Eingabeband
- Ausgabe der Gesamtsumme auf Ausgabeband

```
1: LIT 0;
2: STORE 2;
3: READ 1;
4: LOAD 1;
5: LIT 0;
6: NE;
7: JMC 13;
8: LOAD 2;
9: LOAD 1;
10: ADD;
11: STORE 2;
12: JMP 3;
13: WRITE 2;
```

$$
\begin{array}{lrlll}
(& 9 , & 7 , & [1/2,2/7] , & 0 , \varepsilon) \\
(& 10 , & 2{:}7 , & [1/2,2/7] , & 0 , \varepsilon) \\
(& 11 , & 9 , & [1/2,2/7] , & 0 , \varepsilon) \\
(& 12 , & \varepsilon , & [1/2,2/9] , & 0 , \varepsilon) \\
(& 3 , & \varepsilon , & [1/2,2/9] , & 0 , \varepsilon) \\
(& 4 , & \varepsilon , & [1/0,2/9] , & \varepsilon , \varepsilon) \\
(& 5 , & 0 , & [1/0,2/9] , & \varepsilon , \varepsilon) \\
(& 6 , & 0{:}0 , & [1/0,2/9] , & \varepsilon , \varepsilon)
\end{array}
$$

Weiteres Beispiel

Aufgabe:

- Einlesen einer Folge (Ende: 0) vom Eingabeband
- Ausgabe der Gesamtsumme auf Ausgabeband

```
1: LIT 0;
2: STORE 2;
3: READ 1;
4: LOAD 1;
5: LIT 0;
6: NE;
7: JMC 13;
8: LOAD 2;
9: LOAD 1;
10: ADD;
11: STORE 2;
12: JMP 3;
13: WRITE 2;
```

$$
\begin{array}{llllll}
(\ 12\ , & \varepsilon\ , & [1/2{,}2/9]\ , & 0 & , & \varepsilon\ ) \\
(\ 3\ , & \varepsilon\ , & [1/2{,}2/9]\ , & 0 & , & \varepsilon\ ) \\
(\ 4\ , & \varepsilon\ , & [1/0{,}2/9]\ , & \varepsilon & , & \varepsilon\ ) \\
(\ 5\ , & 0\ , & [1/0{,}2/9]\ , & \varepsilon & , & \varepsilon\ ) \\
(\ 6\ , & 0{:}0\ , & [1/0{,}2/9]\ , & \varepsilon & , & \varepsilon\ ) \\
(\ 7\ , & 0\ , & [1/0{,}2/9]\ , & \varepsilon & , & \varepsilon\ ) \\
(\ 13\ , & \varepsilon\ , & [1/0{,}2/9]\ , & \varepsilon & , & \varepsilon\ ) \\
(\ 14\ , & \varepsilon\ , & [1/0{,}2/9]\ , & \varepsilon & , & 9\ )
\end{array}
$$

Bzgl. Speicherplatz 2 zeigt sich hier ein sehr spezielles Zugriffsschema, welches man ausnutzen kann, um das Programm zu optimieren.

Optimierung

Idee: statt Ablage im HS, Zwischensumme auf DK lassen

```
1: LIT 0;
2: READ 1;
3: LOAD 1;
4: LIT 0;
5: NE;
6: JMC 10;
7: LOAD 1;
8: ADD;
9: JMP 2;
10: STORE 1;
11: WRITE 1;
```

(1 , ε , [] , 3.4.2.0 , ε)

Optimierung

Idee: statt Ablage im HS, Zwischensumme auf DK lassen

1: LIT 0;	5: NE;	9: JMP 2;
2: READ 1;	6: JMC 10;	10: STORE 1;
3: LOAD 1;	7: LOAD 1;	11: WRITE 1;
4: LIT 0;	8: ADD;	

(8 ,	2:7 ,	[1/2],	0	, ε)
(9 ,	9 ,	[1/2],	0	, ε)
(2 ,	9 ,	[1/2],	0	, ε)
(3 ,	9 ,	[1/0],	ε	, ε)
(4 ,	0:9 ,	[1/0],	ε	, ε)
(5 ,	0:0:9 ,	[1/0],	ε	, ε)
(6 ,	0:9 ,	[1/0],	ε	, ε)
(10 ,	9 ,	[1/0],	ε	, ε)
(11 ,	ε ,	[1/9],	ε	, ε)
(12 ,	ε ,	[1/9],	ε	, 9)

Erweiterung

Aufgabe: statt Summe, nun Durchschnitt zu berechnen

```
 1: LIT 0;
 2: STORE 2;
 3: LIT 0;
 4: READ 1;
 5: LOAD 1;
 6: LIT 0;
 7: NE;
 8: JMC 16;
 9: LOAD 1;
10: ADD;
11: LOAD 2;
12: LIT 1;
13: ADD;
14: STORE 2;
15: JMP 4;
16: LOAD 2;
17: DIV;
18: STORE 1;
19: WRITE 1;
```

(1 , ε , [] , 3.4.2.0 , ε)

Erweiterung

Aufgabe: statt Summe, nun Durchschnitt zu berechnen

```
 1: LIT 0;
 2: STORE 2;
 3: LIT 0;
 4: READ 1;
 5: LOAD 1;
 6: LIT 0;
 7: NE;
 8: JMC 16;
 9: LOAD 1;
10: ADD;
11: LOAD 2;
12: LIT 1;
13: ADD;
14: STORE 2;
15: JMP 4;
16: LOAD 2;
17: DIV;
18: STORE 1;
19: WRITE 1;
```

$$
\begin{array}{lrlll}
(& 7 , & 0{:}0{:}9 , & [1/0{,}2/3] , & \varepsilon \quad , \varepsilon) \\
(& 8 , & 0{:}9 , & [1/0{,}2/3] , & \varepsilon \quad , \varepsilon) \\
(& 16 , & 9 , & [1/0{,}2/3] , & \varepsilon \quad , \varepsilon) \\
(& 17 , & 3{:}9 , & [1/0{,}2/3] , & \varepsilon \quad , \varepsilon) \\
(& 18 , & 3 , & [1/0{,}2/3] , & \varepsilon \quad , \varepsilon) \\
(& 19 , & \varepsilon , & [1/3{,}2/3] , & \varepsilon \quad , \varepsilon) \\
(& 20 , & \varepsilon , & [1/3{,}2/3] , & \varepsilon \quad , 3)
\end{array}
$$

Programmieren in AM:

- ist ja ganz nett
- erlaubt Ausnutzen von in C/C_0 gar nicht sichtbaren Details der Maschine
- ist aber auch umständlich (z.B. keine Strukturierung)

Daher: Übersetzung $C_0 \rightarrow$ AM