

# PROGRAMMIERUNG

## ÜBUNG 10: $C_0$ UND ABSTRAKTE MASCHINE $AM_0$

---

Eric Kunze

`eric.kunze@tu-dresden.de`

1. Funktionale Programmierung
  - 1.1 Einführung in Haskell: Listen
  - 1.2 Algebraische Datentypen
  - 1.3 Funktionen höherer Ordnung
  - 1.4 Typpolymorphie & Unifikation
  - 1.5 Beweis von Programmeigenschaften
  - 1.6  $\lambda$ -Kalkül
2. Logikprogrammierung
3. Implementierung einer imperativen Programmiersprache
  - 3.1 **Implementierung von  $C_0$**
  - 3.2 Implementierung von  $C_1$
4. Verifikation von Programmeigenschaften
5.  $H_0$  – ein einfacher Kern von Haskell

# Implementierung von $C_0$ und abstrakte Maschine $AM_0$

---

- ▶ **Ziel:** Implementierung einer einfachen Programmiersprache  $C_1 \subset C$

- ▶ **Ziel:** Implementierung einer einfachen Programmiersprache  $C_1 \subset C$
- ▶ **Hier:** zunächst Einschränkung auf  $C_0 \subset C_1$ 
  - ▶ genau eine main-Funktion
  - ▶ Zugriff auf `stdio` durch `#include`
  - ▶ einzig zugelassene Datenstruktur: `int`, Konstanten
  - ▶ Kontrollstrukturen: Ein-/Ausgabebefehle, Zuweisungen, Sequenzen, Verzweigungen, bedingte Schleifen

- ▶ **Ziel:** Implementierung einer einfachen Programmiersprache  $C_1 \subset C$
- ▶ **Hier:** zunächst Einschränkung auf  $C_0 \subset C_1$ 
  - ▶ genau eine main-Funktion
  - ▶ Zugriff auf `stdio` durch `#include`
  - ▶ einzig zugelassene Datenstruktur: `int`, Konstanten
  - ▶ Kontrollstrukturen: Ein-/Ausgabebefehle, Zuweisungen, Sequenzen, Verzweigungen, bedingte Schleifen
- ▶ **Implementierung** durch
  - ▶ Syntax von  $C_0$
  - ▶ Befehle und Semantik einer abstrakten Maschine  $AM_0$
  - ▶ Übersetzer  $C_0 \leftrightarrow AM_0$

Wir bauen eine abstrakte Maschine  $AM_0$ , die unsere Berechnungen ausführen kann. Wir benötigen dafür:

- ▶ ein Ein- und Ausgabeband,
- ▶ einen Datenkeller,
- ▶ einen Hauptspeicher und
- ▶ einen Befehlszähler

Nun müssen aber auch Aktionen ausgeführt werden, wie zum Beispiel das Einlesen vom Eingabeband in den Hauptspeicher. Dafür gibt es folgende Befehle:





Den Zustand der abstrakten Maschine beschreiben wir durch die Zustände der 5 Komponenten, also als 5-Tupel

$$(m, d, h, inp, out)$$

= (Befehlszähler, Datenkeller, Hauptspeicher, Input, Output)

Jeder Befehl verändert den Zustand der Maschine – er verändert also die Einträge in diesem Tupel.

$$\begin{aligned} C[\text{SUB}](m, d, h, inp, out) &:= \\ &\text{if } d = d.1 : d.2 : \dots : d.n \\ &\text{then } (m + 1, (d.2 - d.1) : d.3 : \dots : d.n, inp, out) \end{aligned}$$

# SEMANTIK DER BEFEHLE

... gibt zu jedem Befehl der  $AM_0$  an, welche Zustandstransformation der  $AM_0$  dieser Befehl bewirkt. Dabei schreiben wir statt  $C[\cdot](\gamma)$  nun  $C[\cdot](\gamma)$ .

- $C[\text{ADD}](m, d, h, \text{inp}, \text{out}) :=$   
if  $d = d.1 : d.2 : d.3 : \dots : d.n$  mit  $n \geq 2$  then  $(m + 1, (d.2 + d.1) : d.3 : \dots : d.n, h, \text{inp}, \text{out})$   
für MUL analog

$C[\text{SUB}](m, d, h, \text{inp}, \text{out}) :=$   
if  $d = d.1 : d.2 : d.3 : \dots : d.n$  mit  $n \geq 2$  then  $(m + 1, (d.2 - d.1) : d.3 : \dots : d.n, h, \text{inp}, \text{out})$   
für DIV und MOD analog

$C[\text{LT}](m, d, h, \text{inp}, \text{out}) :=$   
if  $d = d.1 : d.2 : d.3 : \dots : d.n$  mit  $n \geq 2$  then  $(m + 1, b : d.3 : \dots : d.n, h, \text{inp}, \text{out})$   
wobei  $b = 1$ , falls  $d.2 < d.1$ , und  $b = 0$ , falls  $d.2 \geq d.1$ ,  
d. h. für den Wert *true* (bzw. *false*) wird 1 (bzw. 0) abgelegt  
für EQ, NE, GT, LE und GE analog

- $C[\text{LOAD } n](m, d, h, \text{inp}, \text{out}) :=$   
if  $h(n) \in \mathbb{Z}$  then  $(m + 1, h(n) : d, h, \text{inp}, \text{out})$   
 $C[\text{LIT } z](m, d, h, \text{inp}, \text{out}) := (m + 1, z : d, h, \text{inp}, \text{out})$   
 $C[\text{STORE } n](m, d, h, \text{inp}, \text{out}) :=$   
if  $d = d.1 : d'$  then  $(m + 1, d', h[n/d.1], \text{inp}, \text{out})$   
wobei  $h[n/d.1](k) = \begin{cases} d.1 & \text{falls } k = n \\ h(k) & \text{sonst} \end{cases}$

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

$C[\text{JMC } e](m, d, h, \text{inp}, \text{out}) :=$   
if  $d = 0 : d.2 : \dots : d.n$  mit  $n \geq 1$  then  $(e, d.2 : \dots : d.n, h, \text{inp}, \text{out})$   
if  $d = 1 : d.2 : \dots : d.n$  mit  $n \geq 1$  then  $(m + 1, d.2 : \dots : d.n, h, \text{inp}, \text{out})$

Es wird also zum Befehl mit der Nummer  $e$  gesprungen, wenn das oberste Kellerelement gleich 0 ist; die 0 repräsentiert den Wert *false*. Wenn das oberste Kellerelement gleich 1 ist (und damit den Wert *true* repräsentiert), dann wird der Befehlszähler um 1 inkrementiert.

- $C[\text{READ } n](m, d, h, \text{inp}, \text{out}) :=$   
if  $\text{inp} = \text{first}(\text{inp}).\text{rest}(\text{inp})$  then  $(m + 1, d, h[n/\text{first}(\text{inp})], \text{rest}(\text{inp}), \text{out})$   
wobei für jedes  $n \in \mathbb{Z}$  und  $w \in \mathbb{Z}^*$  gilt:  $\text{first}(n : w) = n$  und  $\text{rest}(n : w) = w$

```
sttrans(if (exp) stat1 else stat2, tab, a) :=  
    boolexptrans(exp, tab)  
    JMC a.1;  
    sttrans(stat1, tab, a.2)  
    JMP a.3;  
a.1 : sttrans(stat2, tab, a.4)  
a.3 :
```

für alle  $exp \in W(\langle \text{BoolExpression} \rangle)$ ,  $stat_1, stat_2 \in W(\langle \text{Statement} \rangle)$ ,  
 $tab \in \text{Tab}$  und  $a \in \mathbb{N}^*$ .

# AUFGABE 1

Wir betrachten das  $C_0$ -Programm *Max*:

```
1 #include <stdio.h>           7     if (a > b)
2                               8         max = a;
3 int main( ) {                9     else max = b;
4     int a, b, max;           10    printf("%d", max);
5     scanf("%i", &a);         11    return 0;
6     scanf("%i", &a);         12 }
```

- (a) Berechnen Sie schrittweise das baumstrukturierte Programm  $bMax_0 = \text{trans}(Max)$  mit Hilfe der in der Vorlesung angegebenen Übersetzungsfunktionen. Dokumentieren Sie dabei jeden rekursiven Funktionsaufruf.

# AUFGABE 1 – TEIL (A)

## Baumstrukturierte Adressen:

```
    READ 1;  
    READ 2;  
    LOAD 1;  
    LOAD 2;  
    GT;  
    JMC 1.3.1;  
    LOAD 1;  
    STORE 3;  
    JMP 1.3.3;  
1.3.1  LOAD 2;  
       STORE 3;  
1.3.3  WRITE 3;
```

# AUFGABE 1 – TEIL (A)

## Baumstrukturierte Adressen:

```
    READ 1;  
    READ 2;  
    LOAD 1;  
    LOAD 2;  
    GT;  
    JMC 1.3.1;  
    LOAD 1;  
    STORE 3;  
    JMP 1.3.3;  
1.3.1  LOAD 2;  
        STORE 3;  
1.3.3  WRITE 3;
```

## Linearisierte Adressen:

```
1 READ 1;  
2 READ 2;  
3 LOAD 1;  
4 LOAD 2;  
5 GT;  
6 JMC 10;  
7 LOAD 1;  
8 STORE 3;  
9 JMP 12;  
10 LOAD 2;  
11 STORE 3;  
12 WRITE 3;
```

## AUFGABE 1 – TEIL (B)

### Ablauf der abstrakten Maschine:

	BZ	,	DK	,	HS	,	Inp	,	Out	
(	1	,	$\varepsilon$	,	[ ]	,	5:7	,	$\varepsilon$	)
(	2	,	$\varepsilon$	,	[1/5]	,	7	,	$\varepsilon$	)
(	3	,	$\varepsilon$	,	[1/5, 2/7]	,	$\varepsilon$	,	$\varepsilon$	)
(	4	,	5	,	[1/5, 2/7]	,	$\varepsilon$	,	$\varepsilon$	)
(	5	,	7:5	,	[1/5, 2/7]	,	$\varepsilon$	,	$\varepsilon$	)
(	6	,	0	,	[1/5, 2/7]	,	$\varepsilon$	,	$\varepsilon$	)
(	10	,	$\varepsilon$	,	[1/5, 2/7]	,	$\varepsilon$	,	$\varepsilon$	)
(	11	,	7	,	[1/5, 2/7]	,	$\varepsilon$	,	$\varepsilon$	)
(	12	,	$\varepsilon$	,	[1/5, 2/7, 3/7]	,	$\varepsilon$	,	$\varepsilon$	)
(	13	,	$\varepsilon$	,	[1/5, 2/7, 3/7]	,	$\varepsilon$	,	7	)

## AUFGABE 1 – TEIL (B)

### Ablauf der abstrakten Maschine:

	BZ	,	DK	,	HS	,	Inp	,	Out
(	1	,	$\varepsilon$	,	[ ]	,	5:7	,	$\varepsilon$ )
(	2	,	$\varepsilon$	,	[1/5]	,	7	,	$\varepsilon$ )
(	3	,	$\varepsilon$	,	[1/5, 2/7]	,	$\varepsilon$	,	$\varepsilon$ )
(	4	,	5	,	[1/5, 2/7]	,	$\varepsilon$	,	$\varepsilon$ )
(	5	,	7:5	,	[1/5, 2/7]	,	$\varepsilon$	,	$\varepsilon$ )
(	6	,	0	,	[1/5, 2/7]	,	$\varepsilon$	,	$\varepsilon$ )
(	10	,	$\varepsilon$	,	[1/5, 2/7]	,	$\varepsilon$	,	$\varepsilon$ )
(	11	,	7	,	[1/5, 2/7]	,	$\varepsilon$	,	$\varepsilon$ )
(	12	,	$\varepsilon$	,	[1/5, 2/7, 3/7]	,	$\varepsilon$	,	$\varepsilon$ )
(	13	,	$\varepsilon$	,	[1/5, 2/7, 3/7]	,	$\varepsilon$	,	7)

$$\mathcal{P}[\llbracket Max_0 \rrbracket](5 : 7) = \text{proj}_5^{(5)} \left( \mathcal{I}[\llbracket Max_0 \rrbracket](1, \varepsilon, [], 5 : 7, \varepsilon) \right) = 7$$



## AUFGABE 2 – TEIL (A)

```
1 #include <stdio.h>
2
3 int main() {
4     int x1, x2;
5     scanf("%i", &x1);
6     scanf("%i", &x2);
7     while (x1 > 0){
8         x1 = x2 - x1;
9         if (x2 > x1)
10             x2 = x2 / 2;
11     }
12     printf("%d", x1);
13     return 0;
14 }
```

Übersetzen Sie das Programm mittels `trans` in  $AM_0$ -Code mit linearen Adressen. Geben Sie nur das Endergebnis der Übersetzung (keine Zwischenschritte) an!

## AUFGABE 2 – TEIL (A)

1 READ 1;	6 JMC 20;	11 LOAD 2;	16 LIT 2;
2 READ 2;	7 LOAD 2;	12 LOAD 1;	17 DIV;
3 LOAD 1;	8 LOAD 1;	13 GT;	18 STORE 2;
4 LIT 0;	9 SUB;	14 JMC 19;	19 JMP 3;
5 GT;	10 STORE 1;	15 LOAD 2;	20 WRITE 1;

## AUFGABE 2 – TEIL (B)

```
3 LOAD 2;      6 JMC 14;      9 LIT 2;      12 STORE 2;  
4 LIT 5;      7 LOAD 1;      10 MUL;      13 JMP 3;  
5 LT;        8 LOAD 2;      11 ADD;      14 WRITE 1;
```

Erstellen Sie ein Ablaufprotokoll für dieses Programmfragment, bis die  $AM_0$  terminiert. Die Startkonfiguration ist  $(7, \varepsilon, [1/3, 2/1], \varepsilon, \varepsilon)$ .

## AUFGABE 2 – TEIL (B)

### Ablauf der abstrakten Maschine:

	BZ	,	DK	,	HS	,	Inp	,	Out
(	7	,	$\epsilon$	,	[1/3, 2/1]	,	$\epsilon$	,	$\epsilon$ )
(	8	,	3	,	[1/3, 2/1]	,	$\epsilon$	,	$\epsilon$ )
(	9	,	1:3	,	[1/3, 2/1]	,	$\epsilon$	,	$\epsilon$ )
(	10	,	2:1:3	,	[1/3, 2/1]	,	$\epsilon$	,	$\epsilon$ )
(	11	,	2:3	,	[1/3, 2/1]	,	$\epsilon$	,	$\epsilon$ )
(	12	,	5	,	[1/3, 2/1]	,	$\epsilon$	,	$\epsilon$ )
(	13	,	$\epsilon$	,	[1/3, 2/5]	,	$\epsilon$	,	$\epsilon$ )
(	3	,	$\epsilon$	,	[1/3, 2/5]	,	$\epsilon$	,	$\epsilon$ )
(	4	,	5	,	[1/3, 2/5]	,	$\epsilon$	,	$\epsilon$ )
(	5	,	5:5	,	[1/3, 2/5]	,	$\epsilon$	,	$\epsilon$ )
(	6	,	0	,	[1/3, 2/5]	,	$\epsilon$	,	$\epsilon$ )
(	14	,	$\epsilon$	,	[1/3, 2/5]	,	$\epsilon$	,	$\epsilon$ )
(	15	,	$\epsilon$	,	[1/3, 2/5]	,	$\epsilon$	,	3)