AM_0

AM_0-Ablaufprotokoll

- Ablaufprotokoll mit Tabelle mit den Spalten: Befehlszähler(BZ), Datenkeller(DK), Hauptspeicher(HS), Eingabeband(IN) und Ausgabeband(OUT)
- der Datenkeller wächst von rechts nach links
- im Hauptspeicher steht an einer Adresse ein Wert (Adresse/Wert)

Befehl	**Auswirkung**
arithmetische Befehle (*ADD, MUL, SUB, DIV, MOD*)	Nimmt die zwei obersten Elemente vom DK und legt den berechneten Wert wieder auf den DK. Das oberste Element entspricht dem 2. Argument. (Wird bei *SUB* z.B. abgezogen vom Zweitobersten.) + inkrementiert BZ
logische Befehle (*EQ, NE, LT, GT, LE, GE*)	Nimmt die zwei obersten Elemente vom DK und legt den entsprechenden Wert (1 für true, 0 für false) wieder auf den DK. Das oberste Element entspricht dem 2. Argument. (Wird bei *GT* z.B. geprüft, ob es kleiner gleich dem Zweitobersten ist. + inkrementiert BZ
LIT a	Legt *a* auf den DK. + inkrementiert BZ
LOAD a	Lädt Wert von gegebener Adresse *a* des HS auf den DK. + inkrementiert BZ
STORE a	Nimmt obersten Wert von DK und schreibt diesen in den HS auf gegebene Adresse *a*. + inkrementiert BZ
JMP a	Setzt BZ auf *a*.
JMC a	Nimmt oberstes Element des DK. Wenn 1 ist: wird BZ inkrementiert. Wenn 0: setzen des BZ auf *a*.
READ a	Nimmt ersten IN-Wert und schreibt diesen im HS in gegebene Adresse *a*. + inkrementiert BZ
WRITE a	Schreibt Wert des HS von gegebener Adresse *a* auf OUT. + inkrementiert BZ

- Gegeben wird meist eine Anfangskombination als Tupel. Das erste Tupelelement

entspricht damit dem BZ-Wert, das zweite dem DK, etc.

- Ist nach $\mathscr{P}[\![Prog]\!] \underbrace{(0)}_{\text{Inputwert}}$ gefragt, entspricht das $proj^5{}_5(I[\![Prog]\!](1, \epsilon, [], 0, \epsilon))$, was nichts anderes meint, als den Output vom Input $(1, \epsilon, [], 0, \epsilon)$.
- Wenn der BZ außerhalb des Programmbereiches zeigt endet die Ausführung.
- Wenn sich aus der vorherigen Zeile der Wert nicht ändert dann kann der Eintrag leer bleiben. (siehe Bsp. zweite Tabellenhälfte)
- z.B.:

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

Berechnen Sie $\mathscr{P}[\![Prog]\!](0)$.

BZ	DK	HS	IN	OUT
1	ϵ	[]	0	ϵ
2	ϵ	[2/0]	ϵ	ϵ
3	0	[2/0]	ϵ	ϵ
4	ϵ	[1/0, 2/0]	ϵ	ϵ
5	0	[1/0, 2/0]	ϵ	ϵ
6	5:0			
7	0			
17	ϵ			
18	ϵ	[1/0, 2/0]	ϵ	0

Damit ist $\mathscr{P}[\![Prog]\!](0) = 0$.

$C_0 \rightarrow AM_0$

- Variablen werden in Deklarationsreihenfolge in eine Symboltabelle eingetragen. In ihr steht zu jeder Variable die HS-Adresse. (z.B. tab=[x/(var/1), y/(var/2)])
- Für einen C_0-Befehl sind meist mehrere AM_0-Sequenzen nötig.

C_0	AM_0
`scanf("%i", &a);`	`READ` Adresse von a `;`
`printf("%d", a);`	`WRITE` Adresse von a `;`

`if(a > b) then {`*then*`}` `else {`*else*`}`
 	
 Es können auch komplexere Ausdrücke oder Zahlen an Stelle der Variablen stehen. Diese müssen entsprechend behandelt werden.
	`LOAD` Adresse von a `;`
	 `LOAD` Adresse von b `;`
	 `GT ;`
	 `JMC` AM_0-Adresse des *else*-Zweiges `;`
	 then
	 `JMP` Adresse nach dem *else*-Zweig `;`
	 else

`while(a > b) {`*then*`}`	`LOAD` Adresse von a `;`
	 `LOAD` Adresse von b `;`
	 `GT ;`
	 `JMC` AM_0-Adresse des *else*-Zweiges `;`
	 then
	 `JMP` AM_0-Adresse des `while ;`

Wertzuweisungen wie `a = 0;`	`LIT 0;` `STORE` Adresse von a`;`

Wertzuweisungen wie `a = a - b;`	`LOAD` Adresse von a `;`
	 `LOAD` Adresse von b `;`
	 `SUB;`
	 `STORE` Adresse von a`;`

- Im Falle einer linearen Adressierung die Adressen erstmal leer lassen und am Ende eintragen.
- Im Falle der baumstrukturierten Adressierung können die Adressen gleich eingetragen werden. Mehrere unterschiedliche Adressen können die selbe Stelle adressieren.

- Am einfachsten baumstrukturierte Adressen im C_0-Code markieren. Die erste Teiladresse ist immer 1. Der zweite Teil ist die Nummer des Befehles in der Mainfunktion. Sollte es sich um ein `if-then-else` oder ein `while` handeln, gibt es einen dritten Teil und so weiter bei gestaffelten Statments

- Bei `if-then-else` mit der Adresse a:
  `if(`Bed.`)` `JMC` a.1
  statements `JMP` a.3 (innerhalb mit a.2 weiter)
  `else`
  a.1 statements (innerhalb mit a.4 weiter)
  a.3

- Bei `while` mit der Adresse a:
  a `while(`Bed.`)` `JMC` a.1 (innerhalb mit a.2 weiter)
  statements `JMP` a
  a.1

- Z.B. ein AM_0-Programm mit baumstrukturierten Adressen für folgendes C_0-Programm:

```
#include <stdio.h>
int main(){
     int a, b, max;
1.1  scanf("%i", &a);
1.2  scanf("%i", &b);
1.3  if(a > b) 1.3.2 max = a;
1.3.1  else max = b;
1.3.3 = 1.4 printf("%d", max);
      return 0;
}
```

```
READ 1;     ≃ scanf("%i", &a);
READ 2;     ≃ scanf("%i", &b);
LOAD 1;     ≃ if(a > b)
LOAD 2;
GT;
JMC 1.3.1;

                    LOAD 1;     ≃ then max = a;
                    STORE 3;
                    JMP 1.3.3;
1.3.1 :     LOAD 2;     ≃ else max = b;
                    STORE 3;
1.3.3 :     WRITE 3;    ≃ printf("%d", max);
```

$\mathsf{AM}_0 \rightarrow \mathsf{C}_0$

- Man sucht in den AM_0-Befehlen nach Mustern, die C_0-Strukturen entsprechen.
- Auffällig sind vor allem `while`s und `if`s durch die `JMP`s und `JMC`s. Außerdem verwenden sie meist logische Operatoren.
- z.B.: Geben sie für den Ausschnitt aus einem AM_0-Programm die zugehörigen C_0-Statements an, deren Übersetzung (bis auf eine eventuelle Verschiebung der Befehlsadressen) zu dieser AM_0-Befehlsfolge führt. Vergeben Sie dabei für den Speicherplatz i die Variable `xi`.

```
1  ...
2  LOAD 1;
3  LOAD 2;
4  LOAD 3;
5  ADD;
6  LE;
7  JMC 15;
8  LIT 0;
9  LOAD 1;
10 GT;
11 JMC 14;
12 WRITE 2;
13 JMP 8;
14 JMP 16;
15 WRITE 1;
16 ...
```

  - Das erste `JMC 15;` in Zeile 7 weist auf auf ein `while` oder ein `if` hin.
  - Das zweite `JMC 14;` in Zeile 11 ebenfalls.
  - An Hand der `JMP`s erkennt man, dass es sich um ein `while` in einem `if` handelt.
  - Zwischenstand: (blau = C_0, schwarz = AM_0)

```
...                 LE;){               }
if( LOAD 1;         while( LIT 0;       }else{
LOAD 2;             LOAD 1;             WRITE 1;}
LOAD 3;             GT;){               ...
ADD;                WRITE 2;
```

  - Das `ADD;` auflösen;

```
...                 while( LIT 0;       }
if( LOAD 1;         LOAD 1;             }else{
x2 + x3             GT;){               WRITE 1;}
LE;){               WRITE 2;            ...
```

– Das `LE;` und `GT;` auflösen.

```
...
if(x1 < x2+x3){
while(0 > x1){
WRITE 2;
}
}else{
WRITE 1;}
...
```

– `WRITE`s auflösen.

```
...
if(x1 < x2+x3){
      while(0 > x1){
            printf("%d", x2);
      }
}else{
      printf("%d", x1);
}
...
```

AM_1

Symboltabelle

- Die Symboltabelle beinhaltet jetzt nicht nur Variablen, sondern auch Parameter und Funktionen
- Funktionen: *Funktionsname* / (proc, *einmalige Funktionsnummer, beginnend bei 1*)
- globale Variablen: *Variablenname* / (var, global, *einmalige globale Variablennummer, beginnend bei 1*)
- lokale Variablen: *Variablenname* / (var, lokal, *in Funktion einmalige Nummer, beginnend bei 1*)
- Parameter: *Variablenname* / (var, lokal, *Parameternummer*)
- Referenzparameter: *Variablenname* / (var-ref, *Parameternummer*)
- Parameternummer:
  - Die Funktion habe n Parameter.
  - Die Parameter werden von hinten abwärts vergeben, beginnend bei -2. Mit anderen Worten der letzte Parameter hat die Nummer -2, der vorletzte -3 usw. der erste hat $-1 - n$.

AM_1-Ablaufprotokoll

- die Tabelle besteht jetzt aus: BZ, DK, Laufzeitkeller(LK), Referenzzeiger(REF), IN, OUT
- Der LK wächst von links nach rechts.
- es gibt neue Befehle und alte ändern sich:

Befehl	**Auswirkung**
arithmetische Befehle (*ADD, MUL, SUB, DIV, MOD*)	ändern sich nicht
logische Befehle (*EQ, NE, LT, GT, LE, GE*)	ändern sich nicht
Sprungbefehle (*JMP, JMC*)	ändern sich nicht
LIT a	ändert sich nicht
LOAD a b	Lädt Wert von gegebener Adresse *b* des HS auf den DK. *a* kann die Werte lokal oder global annehmen. + inkrementiert BZ
WRITE a b	Schreibt Wert des HS von gegebener Adresse *b* auf OUT. *a* kann die Werte lokal oder global annehmen. + inkrementiert BZ
LOADI a	Schaut nach, welcher Wert an gegebener Adresse *a* im LK steht. Dieser Wert ist die Adresse des Wertes, der auf den DK geschrieben wird. + BZ inkrementieren
WRITEI a	Schaut nach, welcher Wert an gegebener Adresse *a* im LK steht. Dieser Wert ist die Adresse des Wertes, der auf OUT geschrieben wird. + BZ inkrementieren
LOADA a b	Schreibt die globale Adresse einer gegebenen Adresse *b* auf den DK. *a* kann die Werte lokal oder global annehmen. (Ist *a* global, wird *b* auf den DK geschrieben) + BZ inkrementieren

Befehl	**Auswirkung**
STORE a b	Nimmt obersten Wert von DK und schreibt diesen in den LK in gegebene Adresse *b*. *a* kann die Werte lokal oder global annehmen. + inkrementiert BZ
READ a b	Nimmt ersten IN-Wert und schreibt diesen im LK in gegebene Adresse *b*. *a* kann die Werte lokal oder global annehmen. + inkrementiert BZ
STOREI a	Schaut nach, welcher Wert an gegebener Adresse *a* im LK steht. Dieser Wert ist die Adresse in die der oberste Wert von DK geschrieben wird. + inkrementiert BZ
READI a	Schaut nach, welcher Wert an gegebener Adresse *a* im LK steht. Dieser Wert ist die Adresse in die der erste IN-Wert geschrieben wird. + inkrementiert BZ
PUSH	Legt oberstes Element vom DK auf den LK. + inkrementiert BZ
CALL a	Legt den eigentlich folgenden BZ-Wert auf den LK. Setzt BZ auf *a*. Legt aktuellen REF auf den LK. Neuer REF wird die jetzige Länge des LK. (**Hier ist die Reihenfolge wichtig!**)
INIT	Legt gegeben viele 0en auf den LK.
RET a	Den LK nach dem REF-Pointer löschen. Jetzt obersten Wert vom LK nehmen. Diesen Wert zum neuen REF Wert machen. Jetzt obersten Wert vom LK nehmen. BZ auf diesen Wert setzen. *a* Elemente vom LK nehmen. (**Hier ist die Reihenfolge wichtig!**)

- Speicherzugriffe:
  - Wenn REF = 7 und LK:

9	:	3	:	0	:	0	:	1	:	17	:	3	:	4
1		2		3		4		5		6		7		8

  - `LOAD (global, 1)` greift auf globale Adresse 1 zu. Ergebnis: 9

9	:	3	:	0	:	0	:	1	:	17	:	3	:	4
1		2		3		4		5		6		7		8

- `LOAD (lokal, 1)` greift auf lokale Adresse 1 zu. Das heißt auf das Element 1 rechts des REF-Pointers zu. Ergebnis: 4

  9 : 3 : 0 : 0 : 1 : 17 : 3 : 4

  1 2 3 4 5 6 7 8

- `LOAD (lokal, -2)` greift auf lokale Adresse -2 (einen Parameter) zu. Das heißt auf das Element 2 links des REF-Pointers zu. Ergebnis: 1

  9 : 3 : 0 : 0 : 1 : 17 : 3 : 4

  1 2 3 4 5 6 7 8

- `LOADI (-2)` greift auf lokale Adresse -2 (einen Parameter). Das heißt auf das Element 2 links des REF-Pointers zu. Danach wird das Ergebnis als Adresse genutzt, auf die zugegriffen wird. Ergebnis: 1

  9 : 3 : 0 : 0 : 1 : 17 : 3 : 4

  1 2 3 4 5 6 7 8

- `LOADA (global, 3)` globale Adresse der globalen Adresse 3 wird geladen. Ergebnis: 3

  9 : 3 : 0 : 0 : 1 : 17 : 3 : 4

  1 2 3 4 5 6 7 8

- `LOADA (lokal, 1)` globale Adresse der lokalen Adresse 1 wird geladen. Das heisst die Adresse 1 rechts neben dem REF-Pointer. Ergebnis: 8

  9 : 3 : 0 : 0 : 1 : 17 : 3 : 4

  1 2 3 4 5 6 7 8

- Beispiel:

```
1  INIT 1;                7  LOAD (global, 1);  13 READ (global, 1);
2  CALL 12;               8  STORE (lokal, 1);  14 LOADA (global, 1);
3  JMP 0;                 9  LIT 5;             15 PUSH;
4  INIT 2;                10 STORE (global, 1); 16 CALL 4;
5  LOADI (-2);            11 RET 1;             17 WRITE (global, 1);
6  STORE (lokal, 2);      12 INIT 1;            18 RET 0;
```

Die Maschine befindet sich im Zustand $(12, \epsilon, 0 : 3 : 0, 3, 9, \epsilon)$. Lassen sie die Maschine so lange laufen, bis sie stoppt. Notieren sie den Zustand nach jedem Befehl.

BZ	DK	LK	REF	IN	OUT
		globale Variablen: (0) ; Rücksprungadresse (ra): 3 ; par: 0			
12	ϵ	(0):(3:0:0)	3	9	ϵ
13	ϵ	(0):(3:0:0) ; lokale Variable: 0	3	9	ϵ
14	ϵ	(9):(3:0:0)	3	ϵ	ϵ
15	1	(9):(3:0:0)	3	ϵ	ϵ
		Parameter: 1			
16	ϵ	(9):(3:0:0):(1)	3	ϵ	ϵ
		ra:BZ+1=16+1: 17 ; par: alter REF-Wert: 3 ; Länge des LK: 7			
4	ϵ	(9):(3:0:0):(1:17:3)	7	ϵ	ϵ
5	ϵ	(9):(3:0:0):(1:17:3:0:0)	7	ϵ	ϵ
6	9	(9):(3:0:0):(1:17:3:0:0)	7	ϵ	ϵ
7	ϵ	(9):(3:0:0):(1:17:3:0:9)	7	ϵ	ϵ
8	9	(9):(3:0:0):(1:17:3:0:9)	7	ϵ	ϵ
9	ϵ	(9):(3:0:0):(1:17:3:9:9)	7	ϵ	ϵ
10	5	(9):(3:0:0):(1:17:3:9:9)	7	ϵ	ϵ
11	ϵ	(5):(3:0:0):(1:17:3:9:9)	7	ϵ	ϵ
17	ϵ	(5):(3:0:0)	3	ϵ	ϵ
18	ϵ	(5):(3:0:0)	3	ϵ	5
3	ϵ	(5)	0	ϵ	5
0	ϵ	(5)	0	ϵ	5

$C_1 \rightarrow AM_1$

- Adressierung wie bei AM_0. Nur der erste Teil der Adresse ist nicht immer 1, sondern die Funktionsnummer aus der Symboltabelle.
- Die alten Befehlsäquivalenzen bleiben im Grunde erhalten. Durch Pointer gibt es allerdings ein paar Unterschiede.
- Um auf den Wert, auf den Pointervariablen zeigen, zuzugreifen muss `LOADI, STOREI, WRITEI` oder `READI` verwendet werden.
- Um eine Adresse zu erhalten, die in einen Pointer geschrieben wird, muss `LOADA` verwendet werden.

- Es kommen eine neue Äquivalenz hinzu:

C_0	AM_0
Funktionsaufrufe:	`LOAD` *Adresse von parameter*$_1$`;`
funktionsname`(`*parameter*$_1$,... , *parameter*$_n$`){`	`PUSH;`
LOKALE VARIABLENDEKLARATIONEN	...
DO SOME STUFF`}`	`LOAD` *Adresse von parameter*$_n$`;`
	`PUSH;`
	`CALL` *Funktions-BZ-Adresse*`;`
An der *Funktions-BZ-Adresse* im AM_1-Code	`INIT` *Anzahl lokaler Variablen*`;`
steht SOME STUFF	`RET` n`;`

- z.B.: Übersetzen Sie nachfolgende C_1-Statements in entsprechenden AM_1-Code mit baumstrukturierten Adressen. Zwischenschritte brauchen Sie keine anzugeben. Die zugehörige Symboltabelle ist:
  tab = [f/(proc,1), d/(var, global, 1), x/(var, global, 2)]

```
...
d = 4;
f(d, &x);
printf("%d", x);
...
```

- Ergebnis:

```
LIT 4;              ≃ d = 4;
STORE(global, 1)
LOAD(global, 1);    ≃ f(d, & x); erster Parameter
PUSH;
LOADA(global, 2);   zweiter Parameter
PUSH;
CALL 1;
WRITE(global, 2)    ≃ printf("% d", x);
```