# Semantik 2:

# Nachweisbar korrekte Implementierung

**Bezogen auf Kapitel 3 aus:**

**Semantics with applications, a formal introduction**
**von Hanne Riis Nielson und Flemming Nielson**
**http://www.daimi.au.dk/~hrn/**

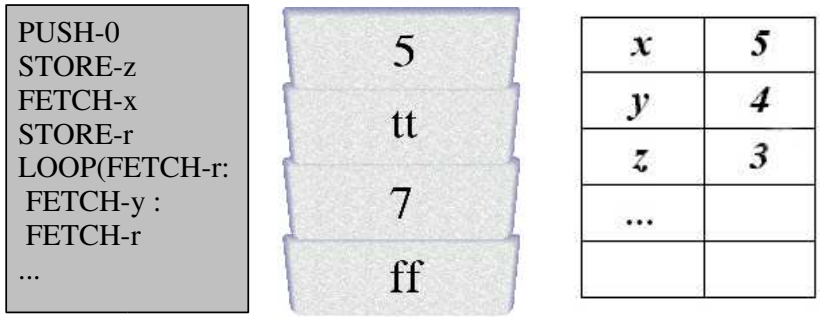
## Gliederung

- **Die abstrakte Maschine**
  - Semantik
  - AM ist deterministisch
- **Spezifizierung der Übersetzung**
  - Arithmetische und boolsche Ausdrücke
  - Statements
- **Korrektheit**
  - Ausdrücke
  - Statements

## Die Abstrakte Maschine (AM):
Semantik (1)

Konfiguration der Form:

<c, e, s> ∈ Code × Stack × State




**c** **e** **s**

## AM: Semantik (2)

**Tabellarisch:**

| | c | e | s |
|---|---|---|---|
| aus Menge | Code | Stack= (Z∪T)* | State= Var → Z |
| ist konkret | Programmcode | Abarbeitungsstapel | Speicher |

**Code**: -ist die Kombination aller Möglichen Befehle zu einem Programm
**Stack:** -ist eine Sequenz bestehend aus Ziffern und boolschen Werten
**State:** -ist die Abbildung von Variablen auf Werte

# AM: Semantik(3)

**Die Befehle von AM werden nun über die abstrakte Syntax festgelegt**:

inst ::= PUSH-n | ADD | MULT | SUB
| TRUE | FALSE | EQ | LE | AND | NEG
| FETCH-x | STORE-x
| NOOP | BRANCH(c, c) | LOOP(c, c)

c ::= □ | *inst*: c

□ ist die leere Sequenz,
c wird als Metavariable über den Code betrachtet.

# AM: Semantik(4)

Tabelle 1:
*operationale Semantik der AM*

Die **Transitionsrelation** der AM wird durch

<c, e, s> ☮ <c´, e´, s´>

ausgedrückt.

Die Endkonfiguration hat die Gestalt:
<□ , e, s>

$$
\begin{array}{lcll}
\langle \text{PUSH-}n{:}c,\ e,\ s\rangle & \triangleright & \langle c, \mathcal{N}[\![n]\!]{:}e,\ s\rangle & \\
\langle \text{ADD}{:}c,\ z_1{:}z_2{:}e,\ s\rangle & \triangleright & \langle c, (z_1{+}z_2){:}e,\ s\rangle & \text{if } z_1, z_2 \in \mathbf{Z} \\
\langle \text{MULT}{:}c,\ z_1{:}z_2{:}e,\ s\rangle & \triangleright & \langle c, (z_1{\star}z_2){:}e,\ s\rangle & \text{if } z_1, z_2 \in \mathbf{Z} \\
\langle \text{SUB}{:}c,\ z_1{:}z_2{:}e,\ s\rangle & \triangleright & \langle c, (z_1{-}z_2){:}e,\ s\rangle & \text{if } z_1, z_2 \in \mathbf{Z} \\
\langle \text{TRUE}{:}c,\ e,\ s\rangle & \triangleright & \langle c, \mathbf{tt}{:}e,\ s\rangle & \\
\langle \text{FALSE}{:}c,\ e,\ s\rangle & \triangleright & \langle c, \mathbf{ff}{:}e,\ s\rangle & \\
\langle \text{EQ}{:}c,\ z_1{:}z_2{:}e,\ s\rangle & \triangleright & \langle c, (z_1{=}z_2){:}e,\ s\rangle & \text{if } z_1, z_2 \in \mathbf{Z} \\
\langle \text{LE}{:}c,\ z_1{:}z_2{:}e,\ s\rangle & \triangleright & \langle c, (z_1{\leq}z_2){:}e,\ s\rangle & \text{if } z_1, z_2 \in \mathbf{Z} \\
\langle \text{AND}{:}c,\ t_1{:}t_2{:}e,\ s\rangle & \triangleright & & \\
 & & \begin{cases} \langle c, \mathbf{tt} : e, s\rangle & \text{if } t_1{=}\mathbf{tt} \text{ and } t_2{=}\mathbf{tt} \\ \langle c, \mathbf{ff} : e, s\rangle & \text{if } t_1{=}\mathbf{ff} \text{ or } t_2{=}\mathbf{ff},\ t_1, t_2 \in \mathbf{T} \end{cases} & \\
\langle \text{NEG}{:}c,\ t{:}e,\ s\rangle & \triangleright & \begin{cases} \langle c, \mathbf{ff} : e, s\rangle & \text{if } t{=}\mathbf{tt} \\ \langle c, \mathbf{tt} : e, s\rangle & \text{if } t{=}\mathbf{ff} \end{cases} & \\
\langle \text{FETCH-}x{:}c,\ e,\ s\rangle & \triangleright & \langle c, (s\ x){:}e,\ s\rangle & \\
\langle \text{STORE-}x{:}c,\ z{:}e,\ s\rangle & \triangleright & \langle c, e, s[x{\mapsto}z]\rangle & \text{if } z \in \mathbf{Z} \\
\langle \text{NOOP}{:}c,\ e,\ s\rangle & \triangleright & \langle c, e, s\rangle & \\
\langle \text{BRANCH}(c_1, c_2){:}c,\ t{:}e,\ s\rangle & \triangleright & \begin{cases} \langle c_1 : c, e, s\rangle & \text{if } t{=}\mathbf{tt} \\ \langle c_2 : c, e, s\rangle & \text{if } t{=}\mathbf{ff} \end{cases} & \\
\langle \text{LOOP}(c_1, c_2){:}c,\ e,\ s\rangle & \triangleright & & \\
\multicolumn{4}{c}{\langle c_1{:}\text{BRANCH}(c_2{:}\text{LOOP}(c_1, c_2), \text{NOOP}){:}c,\ e,\ s\rangle}
\end{array}
$$

# AM: Semantik(5)

## Definition Ausführungssequenz für AM:

Befehlssequenz und ein Speicher s. Ausführungssequenz für c und s ist

- eine **endliche** Sequenz

  $\gamma_0, \gamma_1, \gamma_2, \dots, \gamma_\kappa$

  (für die gilt: $\gamma_0$ **= <c, □ , s>** und $\gamma_i$ ☮ $\gamma_{i+1}$ mit $0 \leq i < k$, $k \neq 0$
  und es gibt **kein** $\gamma$, so dass $\gamma_k$ ☮ $\gamma$)

- eine **unendliche** Sequenz

  $\gamma_0, \gamma_1, \gamma_2, \dots$

  (für die gilt: $\gamma_0$ = <c, □ , s> und $\gamma_i$ ☮ $\gamma_{i+1}$ mit $0 \leq i$) .

Der **Initialisierungsstapel** ist immer die **leere Sequenz**. Eine **Ausführungssequenz terminiert** nur dann, wenn sie **finit** ist, anderenfalls durchläuft sie sich endlos.
Eine **terminierende Ausführungssequenz** kann in eine Konfiguration mit leerem Code enden (**<□, e, s>**) oder in einer unbefriedigten Konfiguration (**z.B. <SUB, □, s>**).

# AM: Semantik(6)

**Beispiele:**

***(1)*** Wir betrachten folgende Befehlssequenz:

PUSH-1:FETCH-x:ADD:STORE-x

Vorgabe: s x=3, wir bekommen also:

<PUSH-1:FETCH-x:ADD:STORE-x, □, s>
☮<FETCH-x:ADD:STORE-x, 1, s>
☮<ADD:STORE-x, 3:1, s>
☮<STORE-x, 4, s>
☮<□, □, s[x↦4]>

Die Abarbeitung endet hier mangels nächsten Ausführungsschrittes. Somit handelt es sich um ein **terminierendes** Beispiel.

# AM: Semantik(7)

*(2)* Gegeben sei dieser Code:

LOOP(TRUE, NOOP)

$$\langle \text{BRANCH}(c_1, c_2):c, t:e, s\rangle \triangleright \begin{cases} \langle c_1 : c, e, s\rangle & \text{if } t=\mathbf{tt} \\ \langle c_2 : c, e, s\rangle & \text{if } t=\mathbf{ff} \end{cases}$$

$$\langle \text{LOOP}(c_1, c_2):c, e, s\rangle \triangleright \langle c_1:\text{BRANCH}(c_2:\text{LOOP}(c_1, c_2), \text{NOOP}):c, e, s\rangle$$

Abzuarbeiten:

<LOOP(TRUE, NOOP), □, s>
↷<TRUE:BRANCH(NOOP:LOOP(TRUE, NOOP), NOOP), □, s>
↷<BRANCH(NOOP:LOOP(TRUE, NOOP), NOOP), tt, s>
↷<NOOP:LOOP(TRUE, NOOP), □, s>
↷...

Die Schleife wird endlos durchlaufen und Ausführungsequenz **terminiert nicht**.

# AM ist deterministisch

**Definition von Determinismus bzgl. Eines Programms:**

**Zu jedem Zeitpunkt** während der **Ausführung** eines **deterministischen Programms** existiert genau **eine einzige mögliche Instruktion**. Wählt man genau eine **Instruktion $I_K$** in einem deterministischen Programm, so kann **eindeutig** bestimmt werden, welche Anweisung als **nächste nach $I_K$** ausgeführt werden wird. Wird diese **Regel verletzt**, wird ein Programm **nichtdeterministisch** genannt. Daraus folgt, dass ein nichtdeterministisches Programm bei aufeinanderfolgenden Programmausführungen verschiedene Ergebnisse produzieren kann, auch wenn es die gleichen Eingaben erhält.

**AM** hat eine **deterministische Programmausführung**, da nach jeder Zustandsübersetzung die Eigenschaften der Vorzustände erhalten bleiben. Wollte man den Beweis dazu erbringen, würde dieser mittels vollständiger Induktion über die Länge der Berechnungssequenzen passieren. Der Beweis wird hier allerdings der Kürze wegen nicht erbracht.

# **Spezifizierung der Übersetzung:**
## Arithmetische und boolsche Ausdrücke (1)

Arithmetische und boolsche Ausdrücke werden auf dem Ausarbeitungsstapel der abstrakten Maschine ausgewertet. Damit der Code dieser Anforderung genügt werden folgende totale Funktionen definiert:

CA: Aexp □ Code und CB: Bexp □ Code

Tabelle 2:
*Übersezung von boolschen und arithmetischen Ausdrücken*

| | | |
|---|---|---|
| $\mathcal{CA}[\![n]\!]$ | = | PUSH-$n$ |
| $\mathcal{CA}[\![x]\!]$ | = | FETCH-$x$ |
| $\mathcal{CA}[\![a_1{+}a_2]\!]$ | = | $\mathcal{CA}[\![a_2]\!]$:$\mathcal{CA}[\![a_1]\!]$:ADD |
| $\mathcal{CA}[\![a_1 \star a_2]\!]$ | = | $\mathcal{CA}[\![a_2]\!]$:$\mathcal{CA}[\![a_1]\!]$:MULT |
| $\mathcal{CA}[\![a_1{-}a_2]\!]$ | = | $\mathcal{CA}[\![a_2]\!]$:$\mathcal{CA}[\![a_1]\!]$:SUB |
| $\mathcal{CB}[\![\mathtt{true}]\!]$ | = | TRUE |
| $\mathcal{CB}[\![\mathtt{false}]\!]$ | = | FALSE |
| $\mathcal{CB}[\![a_1 = a_2]\!]$ | = | $\mathcal{CA}[\![a_2]\!]$:$\mathcal{CA}[\![a_1]\!]$:EQ |
| $\mathcal{CB}[\![a_1 \leq a_2]\!]$ | = | $\mathcal{CA}[\![a_2]\!]$:$\mathcal{CA}[\![a_1]\!]$:LE |
| $\mathcal{CB}[\![\neg b]\!]$ | = | $\mathcal{CB}[\![b]\!]$:NEG |
| $\mathcal{CB}[\![b_1 \wedge b_2]\!]$ | = | $\mathcal{CB}[\![b_2]\!]$:$\mathcal{CB}[\![b_1]\!]$:AND |

# Arithmetische und boolsche Ausdrücke (2)

**Bemerkung:**

Bei binären Ausdrücken wird die Reihenfolge so geändert, dass zuerst das rechte gefolgt vom linken Argument und schließlich der Operator stehen. Auf diese Weise ist die Reihenfolge auf dem Evaluierungstapel richtig.

**Beispiel** :

Gegeben sei x+1, ein arithmetischer Ausdruck. Dieser wird folgendermaßen umgewandelt:

CA(x+1) =
CA(1):CA(x):ADD =
PUSH-1:FETCH-x:ADD

# Statements (1)

Die Übersetzung von Statements wird duch die totale Funktion

CS: Stm → Code

$$
\begin{array}{lcl}
\mathcal{CS}[\![x := a]\!] & = & \mathcal{CA}[\![a]\!]\text{:STORE-}x \\
\mathcal{CS}[\![\texttt{skip}]\!] & = & \text{NOOP} \\
\mathcal{CS}[\![S_1;S_2]\!] & = & \mathcal{CS}[\![S_1]\!]\text{:}\mathcal{CS}[\![S_2]\!] \\
\mathcal{CS}[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!] & = & \mathcal{CB}[\![b]\!]\text{:BRANCH}(\mathcal{CS}[\![S_1]\!],\mathcal{CS}[\![S_2]\!]) \\
\mathcal{CS}[\![\texttt{while } b \texttt{ do } S]\!] & = & \text{LOOP}(\mathcal{CB}[\![b]\!],\mathcal{CS}[\![S]\!])
\end{array}
$$

# Statements (2)

**Beispiel:**

CS( if 0=0 then 3 else 4) =

CB[0=0]: BRANCH(CS(3),CS(4)) =

CA[0]:CA[0]:EQ:BRANCH(CS(3),CS(4)) =

# Korrektheit

Die **Korrektheit der Implementierung** ermöglicht es **zu zeigen**, dass wenn ein Statement zuerst in den **Code für AM** übersetzt wird, und **dieser** dann **ausgeführt** wird, man **dasselbe Ergebnis** erhält wie in der **operationalen Semantik** von **While** spezifiziert.

# Ausdrücke (1)

Die Richtigkeit der Implementierung von arithmetischen Ausdrücken wird durch das folgende Lemma ausgedrückt:

**Lemma 1:**

Für alle arithmetischen Ausdrücke a existiert

$< CA[a], \Box, s> \rightarrow^* <\Box, A[a]s, s>$ .

Weiterhin haben alle dazwischenliegenden Konfigurationen dieser Ausführungssequenz keinen leeren Ausarbeitungsstapel.

# Ausdrücke (2)

**Beweis:**
Der Beweis erfolgt mittels struktureller Induktion über a. Dies soll hier nur an drei Beispielfällen aufgezeigt werden. Die übrigen Fälle sind analog zu vollziehen.

**1. Fall:** n
Gegeben sei CA[n]= PUSH-n. Mit Tabelle 1 ergibt es:

<PUSH-n, ε, s> ☮ <ε, N[n], s>
Wegen A[n]s =N[n] laut Definition der Semantik von arithmetischen Ausdrücken (hier nicht aufgeführt) ist dieser Fall bewiesen.

**2. Fall:** x
Gegeben sei CA[x]= FETCH-n. Mit Tabelle 1 ergibt es:

<FETCH-x, ε, s> ☮ <ε, (s x), s>
Wegen A[x]s = s x ergibt sich die Richtigkeit.

# Ausdrücke (3)

**3. Fall: $a_1$+$a_2$:**

Gegeben sei $CA[a_1+a_2]= CA[a_1]:CA[a_2]:ADD$

Die Induktionshypothesen auf $a_1$ und $a_2$ angewandt ergeben:

$< CA[a_1], ε, s> ☮^* <ε, A[a_1]s, s>$

und

$< CA[a_2], ε, s> ☮^* <ε, A[a_2]s, s>$

In beiden Fällen erhält man keine Zwischenkonfiguration mit einem leeren Evaluierungstapel.
Wir erhalten also:

$< CA[a_2]:CA[a_1]:ADD, ε, s> ☮^* <CA[a_1]:ADD, A[a_2]s, s>$

# Ausdrücke (4)

weiterentwickelt:

$< CA[a_1]:ADD, A[a_2], s> ☮^* < ADD, (A[a_1]s):(A[a_2]s), s>$

Benützt man die Übersetzungsrelation für ADD gegeben in Tabelle 1 erhält man:

$< ADD, (A[a_1]s):(A[a_2]s), s> ☮ <ε, A[a_1]s+ A[a_2]s, s>$

Es ist einfach zu sehen, dass alle Zwischenkonfigurationen keinen leeren

Evaluierungstapel haben.

Da $A[a_1+a_2]s= A[a_1]s+ A[a_2]s$ gilt ergibt sich das gewünschte Ergebnis.

Für die weiteren Fälle ist das Vorgehen ähnlich.

# Statements

Die Richtigkeit des Ergebnisses von statements wird nun mittels der natürlichen Semantik gezeigt.

**Satz:**

Für jedes stament S in While gilt: $S_{ns}(S)= S_{am}(S)$.

Dieses Satz **bindet das Verhalten** eines **Statements in natürlicher Semantik** mit dem Verhalten des Codes für die **Abstrakte Maschine** mit ihrer strukturell **operationalen Semantik**.

**Es gilt somit:**

- Wenn bei der **Ausführung** von S in einem beliebigen Zustand dieses in einer Semantik **terminierend** ist, so ist S auch in der **anderen Semantik** terminierend und das reultierende Ergebnis ist dasselbe.
- Hingegen ist S in einem beliebigen Zustand in einer **Semantik nicht terminierend**, so ist es das **auch nicht in einer anderen**.