Script generated by TTT

Title: Seidl: Informatik_1 (12.11.2012)

Date: Mon Nov 12 18:13:56 CET 2012

Duration: 89:01 min

Pages: 59

Bemerkungen:

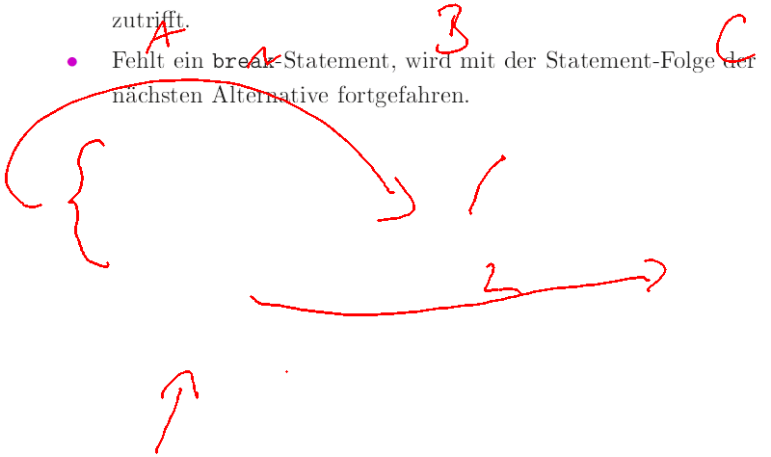
- `move()` ist rekursiv, aber nicht end-rekursiv.
- Sei $N(h)$ die Anzahl der ausgegebenen Moves für einen Turm der Höhe $h \geq 0$. Dann ist

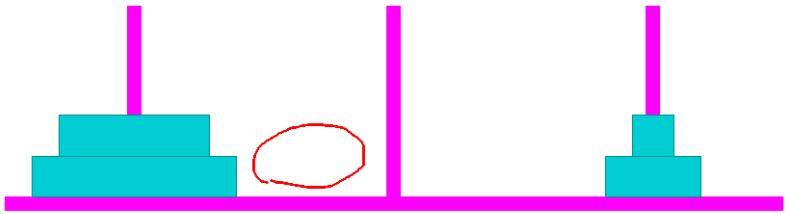
$$\begin{aligned} N(0) &= 0 \qquad \text{und für } h > 0, \\ N(h) &= 1 + 2 \cdot N(h-1) \end{aligned}$$

- Folglich ist $N(h) = 2^h - 1$.
- Bei genauerer Analyse des Problems lässt sich auch ein nicht ganz so einfacher nicht-rekursiver Algorithmus finden ... (wie könnte der aussehen?)

  Hinweis: Offenbar rückt die kleinste Scheibe in jedem zweiten Schritt eine Position weiter ...

- `default` beschreibt den Fall, bei dem keiner der Konstanten zutrifft.
- Fehlt ein `break`-Statement, wird mit der Statement-Folge der nächsten Alternative fortgefahren.





$N(1) = 1$

$N(2)$

Bemerkungen:

- `move()` ist rekursiv, aber nicht end-rekursiv.
- Sei $N(h)$ die Anzahl der ausgegebenen Moves für einen Turm der Höhe $h \geq 0$. Dann ist

$$\begin{aligned} N(0) &= 0 \qquad \text{und für } h > 0, \\ N(h) &= 1 + 2 \cdot N(h-1) \end{aligned}$$

- Folglich ist $N(h) = 2^h - 1$.
- Bei genauerer Analyse des Problems lässt sich auch ein nicht ganz so einfacher nicht-rekursiver Algorithmus finden ... (wie könnte der aussehen?)

  Hinweis: Offenbar rückt die kleinste Scheibe in jedem zweiten Schritt eine Position weiter ...

$N(1) = 1$

$N(2) = 1 + 2 \cdot N(1) = 3$

$N(3) = 1 + 2 \cdot N(2)$

$= 1 + 2 \cdot (1$

Bemerkungen:

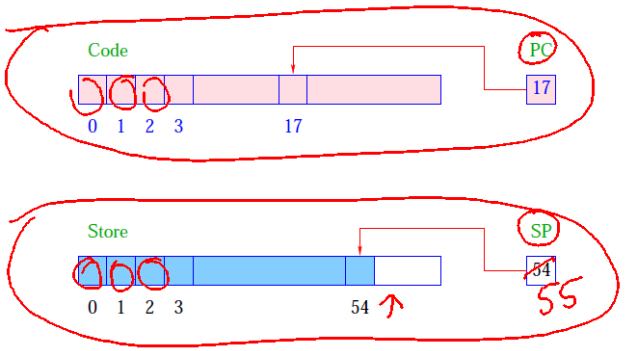
- `move()` ist rekursiv, aber nicht end-rekursiv.
- Sei $N(h)$ die Anzahl der ausgegebenen Moves für einen Turm der Höhe $h \geq 0$. Dann ist

$$\begin{aligned} N(0) &= 0 \quad \text{und für } h > 0, \\ N(h) &= 1 + 2 \cdot N(h-1) \end{aligned}$$

- Folglich ist $N(h) = 2^h - 1$.
- Bei genauerer Analyse des Problems lässt sich auch ein nicht ganz so einfacher nicht-rekursiver Algorithmus finden ... (wie könnte der aussehen?)
- Hinweis: Offenbar rückt die kleinste Scheibe in jedem zweiten Schritt eine Position weiter ...

9 Von MiniJava zur JVM

Architektur der JVM:

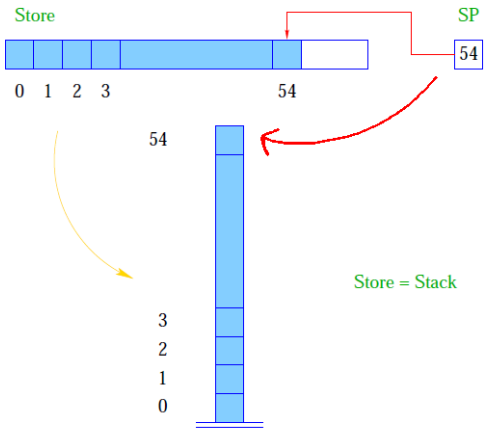


Code	=	enthält JVM-Programm;
		jede Zelle enthält einen Befehl;
PC	=	Program Counter –
		zeigt auf nächsten auszuführenden Befehl;
Store	=	Speicher für Daten;
		jede Zelle kann einen Wert aufnehmen;
SP	=	Stack-Pointer –
		zeigt auf oberste belegte Zelle.

Achtung:

- Programm wie Daten liegen im Speicher – aber in verschiedenen Abschnitten.
- Programm-Ausführung holt nacheinander Befehle aus Code und führt die entsprechenden Operationen auf Store aus.

Konvention:



Befehle der JVM:

`int`-Operatoren:	NEG, ADD, SUB, MUL, DIV, MOD
`boolean`-Operatoren:	NOT, AND, OR
Vergleichs-Operatoren:	LESS, LEQ, EQ, NEQ
Laden von Konstanten:	CONST i, TRUE, FALSE
Speicher-Operationen:	LOAD i, STORE i
Sprung-Befehle:	JUMP i, FJUMP i
IO-Befehle:	READ, WRITE
Reservierung von Speicher:	ALLOC i
Beendung des Programms:	HALT

Befehle der JVM:

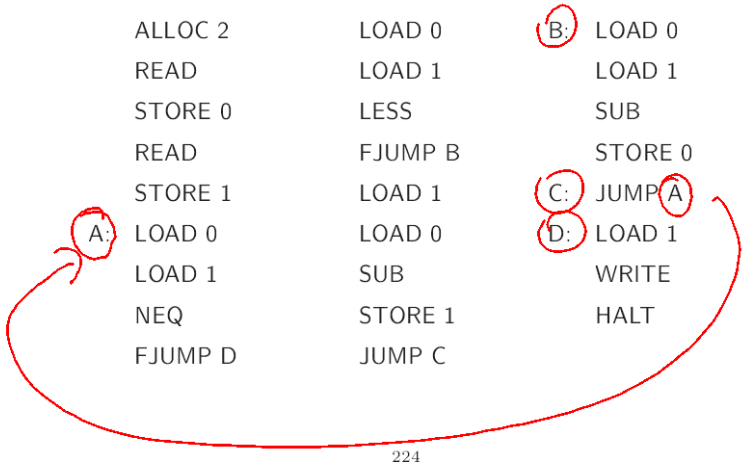
`int`-Operatoren:	NEG, ADD, SUB, MUL, DIV, MOD
`boolean`-Operatoren:	NOT, AND, OR
Vergleichs-Operatoren:	LESS, LEQ, EQ, NEQ
Laden von Konstanten:	CONST i, TRUE, FALSE
Speicher-Operationen:	LOAD i, STORE i
Sprung-Befehle:	JUMP i, FJUMP i
IO-Befehle:	READ, WRITE
Reservierung von Speicher:	ALLOC i
Beendung des Programms:	HALT

Ein Beispiel-Programm:

```
    ALLOC 2
    READ
    STORE 0
    READ
    STORE 1
A:  LOAD 0
    LOAD 1
    NEQ
    FJUMP D
    LOAD 0
    LOAD 1
    LESS
    FJUMP B
    LOAD 1
    LOAD 0
    SUB
    STORE 1
    JUMP C
B:  LOAD 0
    LOAD 1
    SUB
    STORE 0
C:  JUMP A
D:  LOAD 1
    WRITE
    HALT
```

Ein Beispiel-Programm:

```
      ALLOC 2          LOAD 0          B:  LOAD 0
      READ             LOAD 1              LOAD 1
      STORE 0          LESS                SUB
      READ             FJUMP B             STORE 0
      STORE 1          LOAD 1          C:  JUMP A
A:    LOAD 0           LOAD 0          D:  LOAD 1
      LOAD 1           SUB                 WRITE
      NEQ              STORE 1             HALT
      FJUMP D          JUMP C
```



- Das Programm berechnet den GGT !
- Die Marken (Labels) A, B, C, D bezeichnen symbolisch die Adressen der zugehörigen Befehle:

  A = 5
  B = 18
  C = 22
  D = 23

- ... können vom Compiler leicht in die entsprechenden Adressen umgesetzt werden (wir benutzen sie aber, um uns besser im Programm zurechtzufinden.

- Das Programm berechnet den GGT !
- Die Marken (Labels) A, B, C, D bezeichnen symbolisch die Adressen der zugehörigen Befehle:

  A = 5
  B = 18
  C = 22
  D = 23

- ... können vom Compiler leicht in die entsprechenden Adressen umgesetzt werden (wir benutzen sie aber, um uns besser im Programm zurechtzufinden.

Ein Beispiel-Programm:

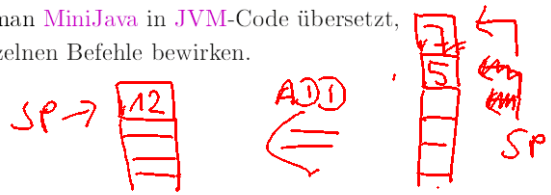
```
      ALLOC 2          LOAD 0          B:  LOAD 0
      READ             LOAD 1              LOAD 1
      STORE 0          LESS                SUB
      READ             FJUMP B             STORE 0
      STORE 1          LOAD 1          C:  JUMP A
A:    LOAD 0           LOAD 0          D:  LOAD 1
      LOAD 1           SUB                 WRITE
      NEQ              STORE 1             HALT
      FJUMP D          JUMP C
```

- Das Programm berechnet den GGT !
- Die Marken (Labels) A, B, C, D bezeichnen symbolisch die Adressen der zugehörigen Befehle:

  A = 5
  B = 18
  C = 22
  D = 23

- ... können vom Compiler leicht in die entsprechenden Adressen umgesetzt werden (wir benutzen sie aber, um uns besser im Programm zurechtzufinden.

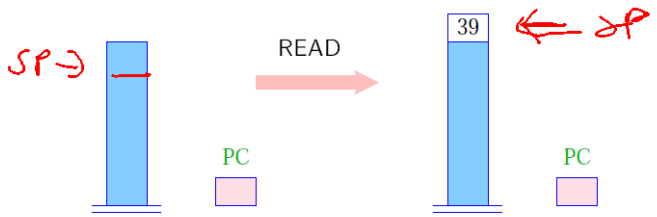
Bevor wir erklären, wie man MiniJava in JVM-Code übersetzt, erklären wir, was die einzelnen Befehle bewirken.



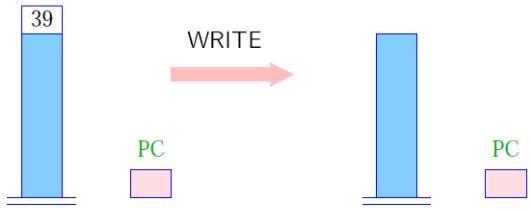
Idee:

- Befehle, die Argumente benötigen, erwarten sie am oberen Ende des Stack.
- Nach ihrer Benutzung werden die Argumente vom Stack herunter geworfen.
- Mögliche Ergebnisse werden oben auf dem Stack abgelegt.

Betrachten wir als Beispiele die IO-Befehle READ und WRITE.



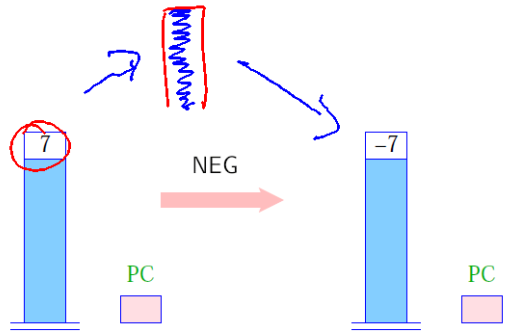
... falls 39 eingegeben wurde

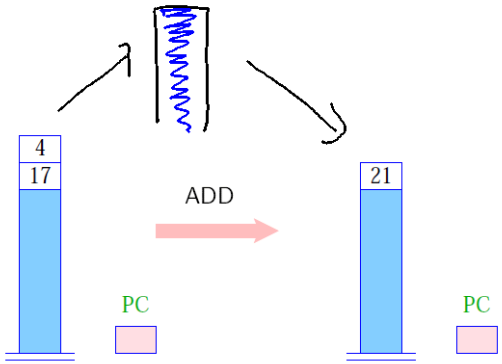


... wobei 39 ausgegeben wird

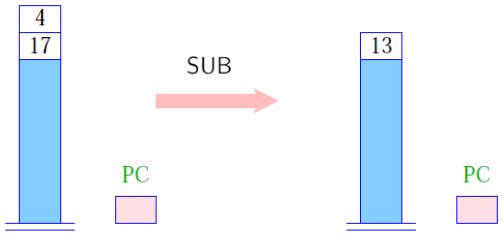
Arithmetik

- Unäre Operatoren modifizieren die oberste Zelle.
- Binäre Operatoren verkürzen den Stack.

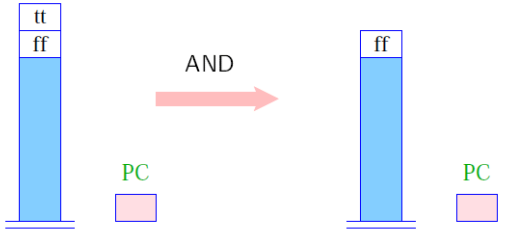




17 − 4

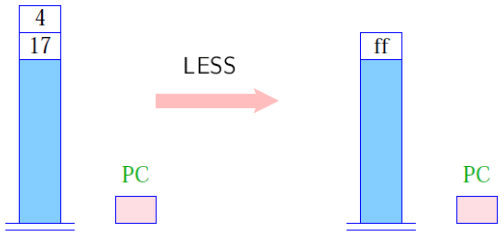


- Die übrigen arithmetischen Operationen MUL, DIV, MOD funktionieren völlig analog.
- Die logischen Operationen NOT, AND, OR ebenfalls – mit dem Unterschied, dass sie statt mit ganzen Zahlen, mit Intern-Darstellungen von `true` und `false` arbeiten (hier: "tt" und "ff").
- Auch die Vergleiche arbeiten so – nur konsumieren sie ganze Zahlen und liefern einen logischen Wert.



x = 17
y = 4

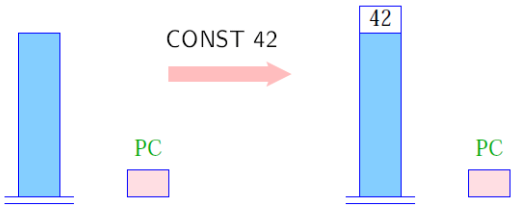
x < y

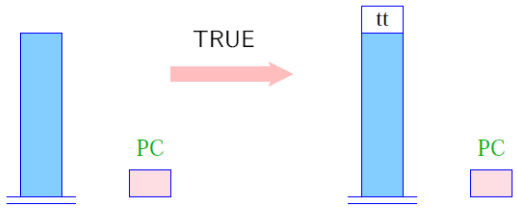


Laden und Speichern

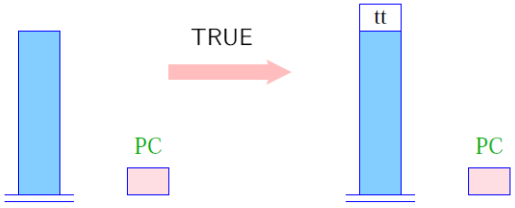
1
2 1
4 2

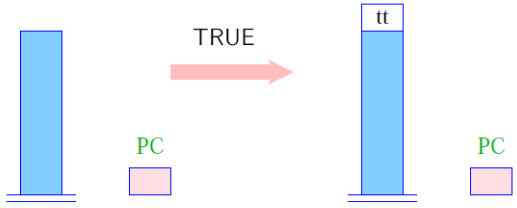
- Konstanten-Lade-Befehle legen einen neuen Wert oben auf dem Stack ab.
- LOAD i legt dagegen den Wert aus der i-ten Zelle oben auf dem Stack ab.
- STORE i speichert den obersten Wert in der i-ten Zelle ab.

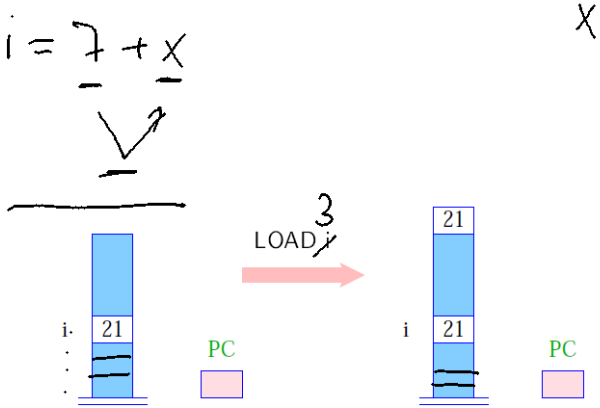


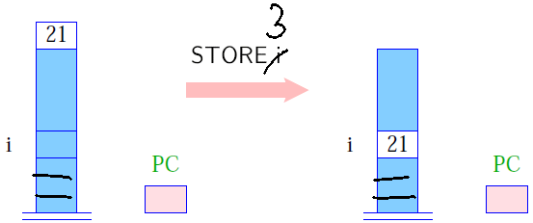


FALSE



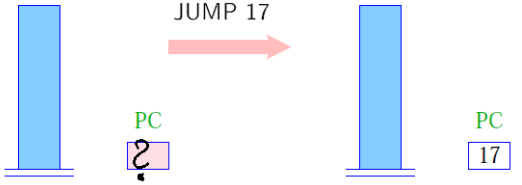


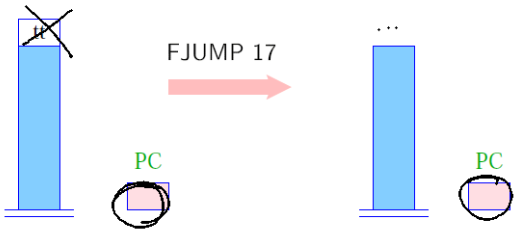


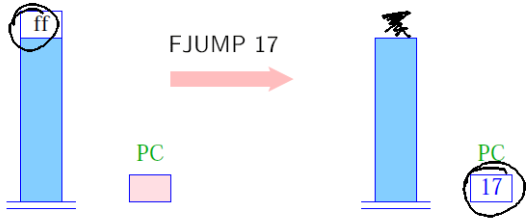


Sprünge

- Sprünge verändern die Reihenfolge, in der die Befehle abgearbeitet werden, indem sie den PC modifizieren.
- Ein unbedingter Sprung überschreibt einfach den alten Wert des PC mit einem neuen.
- Ein bedingter Sprung tut dies nur, sofern eine geeignete Bedingung erfüllt ist.







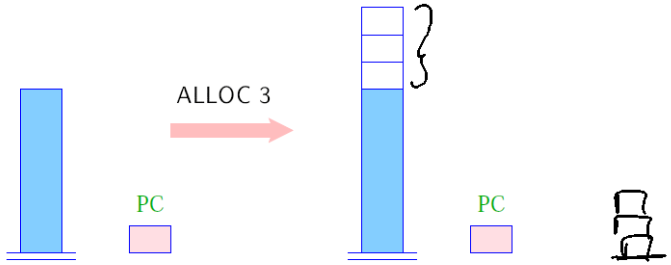
int x, y;

x := 0
y := 1

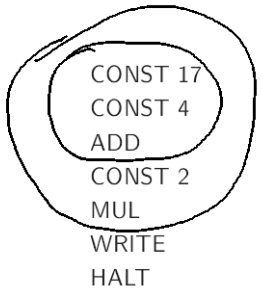
Allokierung von Speicherplatz

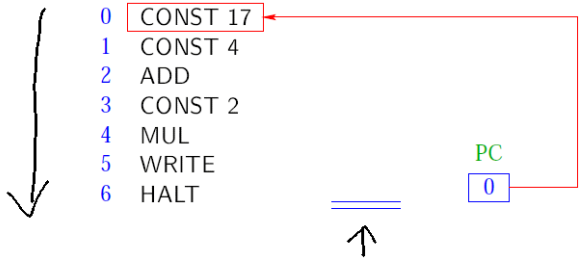
- Wir beabsichtigen, jeder Variablen unseres MiniJava-Programms eine Speicher-Zelle zuzuordnen.
- Um Platz für i Variablen zu schaffen, muss der SP einfach um i erhöht werden.
- Das ist die Aufgabe von ALLOC i.

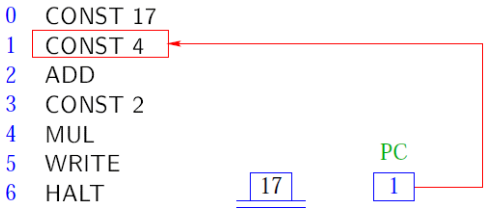
ALLOC 2

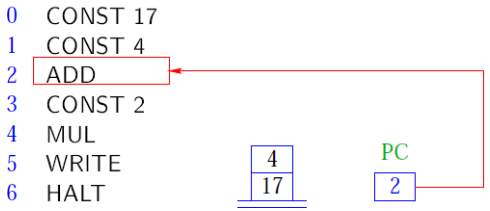


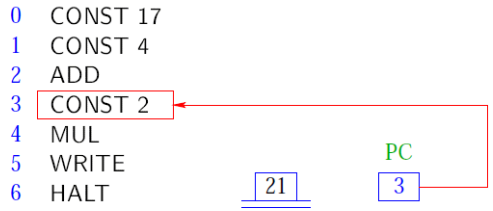
Ein Beispiel-Programm:

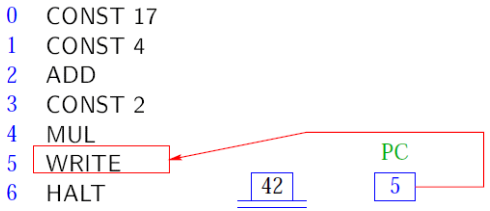












Ausführung eines JVM-Programms:

```
PC = 0;
IR = Code[PC];
while (IR != HALT) {
   PC = PC + 1;
   execute(IR);
   IR = Code[PC];
}
```

JUMP 17

- IR = Instruction Register, d.h. eine Variable, die den nächsten auszuführenden Befehl enthält.
- `execute(IR)` führt den Befehl in IR aus.
- `Code[PC]` liefert den Befehl, der in der Zelle in Code steht, auf die PC zeigt.

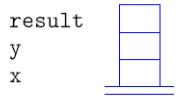
9.1 Übersetzung von Deklarationen

Betrachte Deklaration

```
int x, y, result;
```

Idee:

Wir reservieren der Reihe nach für die Variablen Zellen im Speicher:



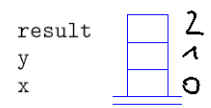
9.1 Übersetzung von Deklarationen

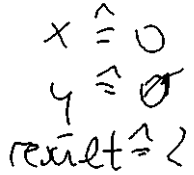
Betrachte Deklaration

```
int x, y, result;
```

Idee:

Wir reservieren der Reihe nach für die Variablen Zellen im Speicher:





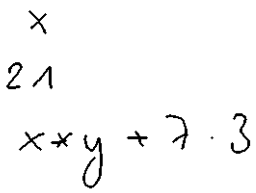
$\Longrightarrow$

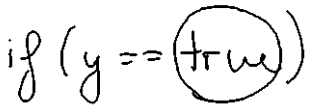
Übersetzung von `int` $x_0, \ldots, x_{n-1}$; = ALLOC n

9.2 Übersetzung von Ausdrücken

Idee:

Übersetze Ausdruck expr in eine Folge von Befehlen, die den Wert von expr berechnet und dann oben auf dem Stack ablegt.

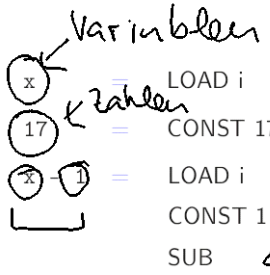




9.2 Übersetzung von Ausdrücken

Idee:

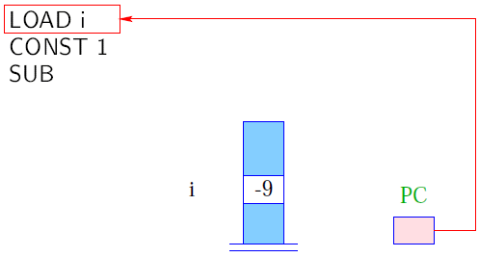
Übersetze Ausdruck expr in eine Folge von Befehlen, die den Wert von expr berechnet und dann oben auf dem Stack ablegt.

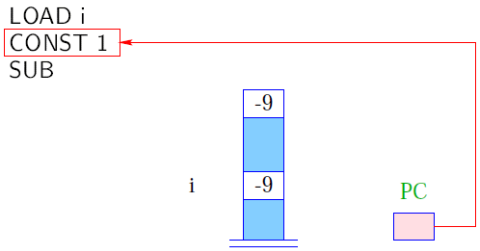


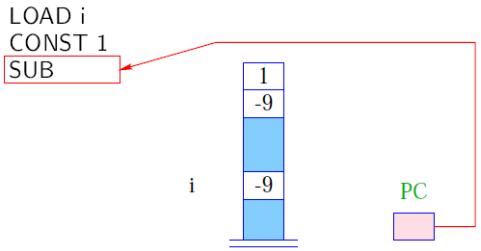
Übersetzung von x = LOAD i — x die i-te Variable

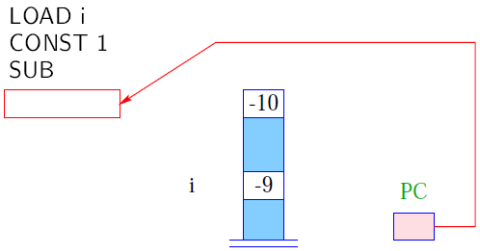
Übersetzung von 17 = CONST 17

Übersetzung von x − 1 = LOAD i
CONST 1
SUB

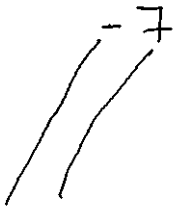








Allgemein:



Übersetzung von - expr = Übersetzung von expr ① ②
NEG

Übersetzung von $expr_1$ + $expr_2$ = Übersetzung von $expr_1$ ①
Übersetzung von $expr_2$ ②
ADD ③

... analog für die anderen Operatoren ...