Helmut Seidl

Compilerbau
+
Abstrakte Maschinen

München

Sommersemester 2005

Organisatorisches

Der erste Abschnitt Die Übersetzung von C ist den Vorlesungen Compilerbau und Abstrakte Maschinen gemeinsam :-)

Vorlesung Compilerbau:	Mo. 12:15-13:45 Uhr
	Mi. 10:15-11:45 Uhr
Vorlesung Abstrakte Maschinen:	Fr. 12:15-13:45 Uhr
	Verma
Übung Compilerbau:	Do. 12:15-13:45 Uhr
	Fr. 10:15-11:45 Uhr
Übung Abstrakte Maschinen:	Di. 10:15-11:45 Uhr

Einordnung:

Diplom-Studierende:

Compilerbau:	Wahlpflichtveranstaltung
Abstrakte Maschinen:	Vertiefende Vorlesung

Bachelor-Studierende:

Compilerbau:	8 ETCS-Punkte
Abstrakte Maschinen:	nicht anrechenbar

Scheinerwerb:

Diplom-Studierende:

- 50% der Punkte;
- zweimal Vorrechnen :-)

Bachelor-Studierende:

- Klausur oder
- mündliche Prüfung.

Material:

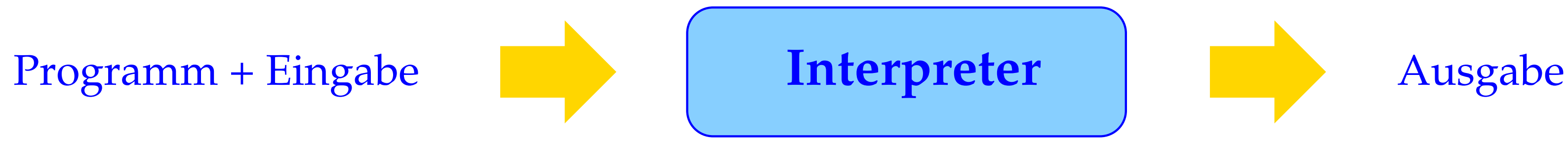
- Literaturliste (im Netz)
- Aufzeichnung der Vorlesungen
  (Folien + Annotationen + Ton + Bild)
- die Folien selbst :-)
- Tools zur Visualisierung der Abstrakten Maschinen :-))
- Tools, um Komponenten eines Compilers zu generieren ...

Weitere Veranstaltungen:

- Seminar: Automatische Deduktion — Mi., 12:00-14:00 Uhr (Verma)
- Wahlpflicht-Praktikum: Compilerbau — Mi., 15:00-17:00 Uhr (Petter)

0 Einführung

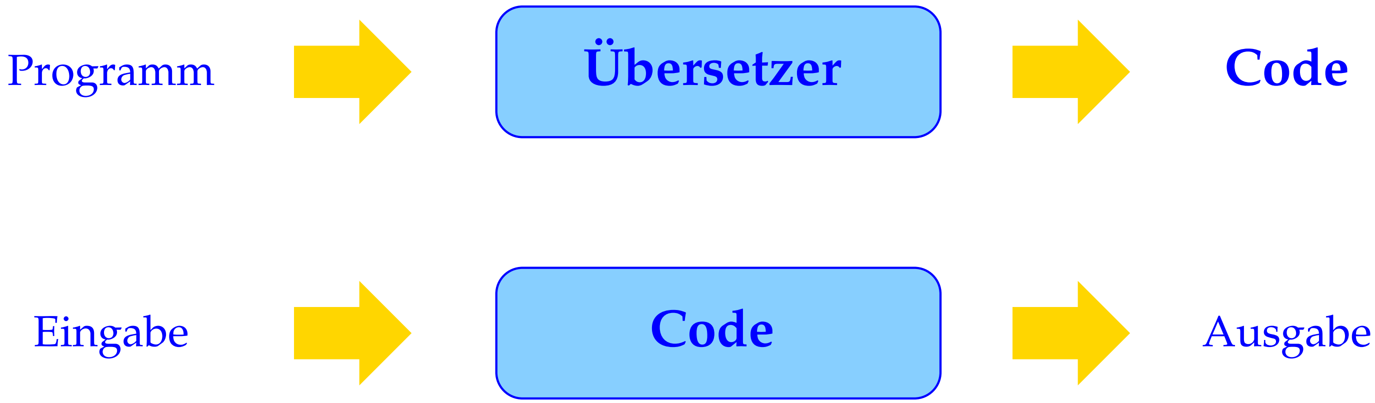
Prinzip eines Interpreters:



Vorteil: Keine Vorberechnung auf dem Programmtext erforderlich $\Longrightarrow$ keine/geringe Startup-Zeit :-)

Nachteil: Während der Ausführung werden die Programm-Bestandteile immer wieder analysiert $\Longrightarrow$ längere Laufzeit :-(

Prinzip eines Übersetzers:



Zwei Phasen:

- Übersetzung des Programm-Texts in ein Maschinen-Programm;
- Ausführung des Maschinen-Programms auf der Eingabe.

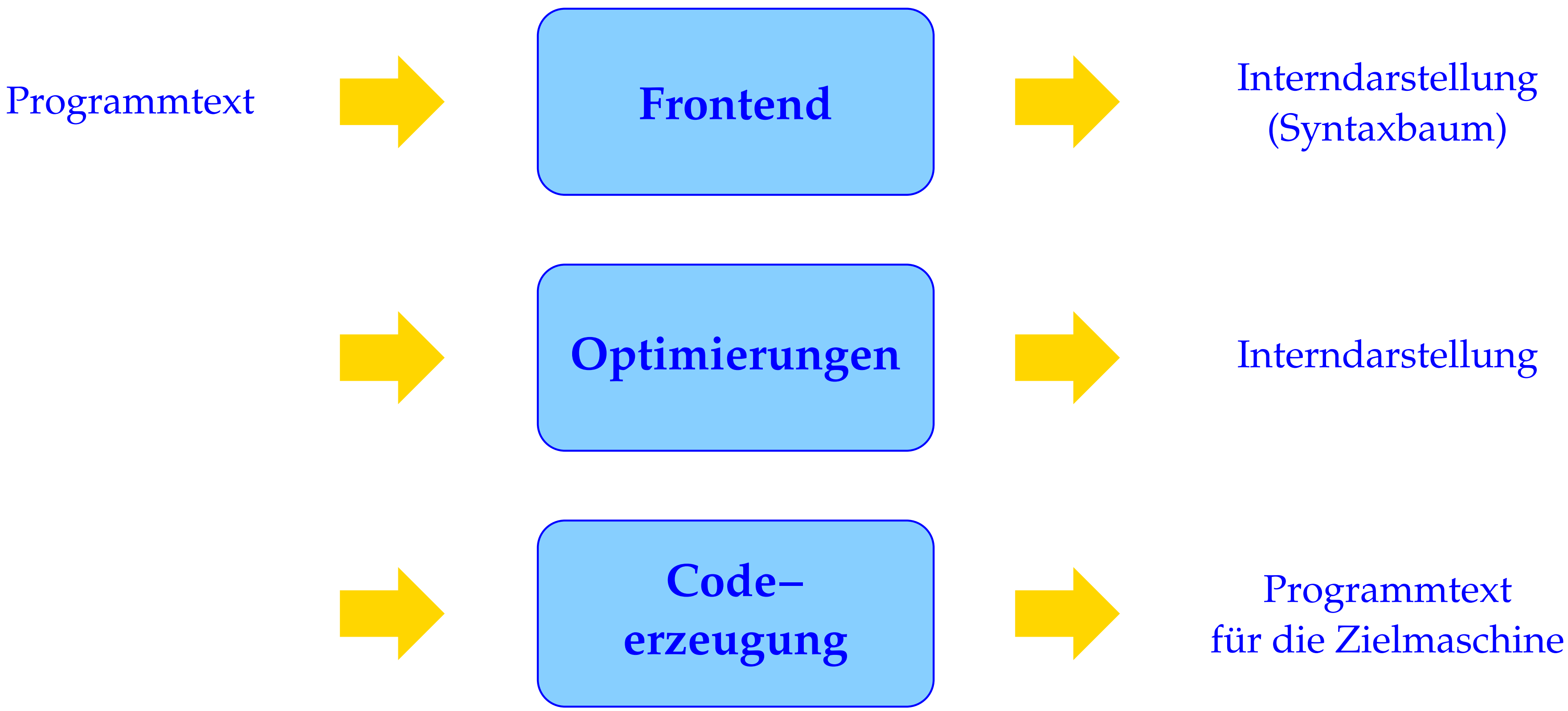
Eine Vorberechnung auf dem Programm gestattet u.a.

- eine geschickte(re) Verwaltung der Variablen;
- Erkennung und Umsetzung globaler Optimierungsmöglichkeiten.

Nachteil: Die Übersetzung selbst dauert einige Zeit :-(

Vorteil: Die Ausführung des Programme wird effizienter $\Longrightarrow$ lohnt sich bei aufwendigen Programmen und solchen, die mehrmals laufen ...

Aufbau eines Übersetzers:



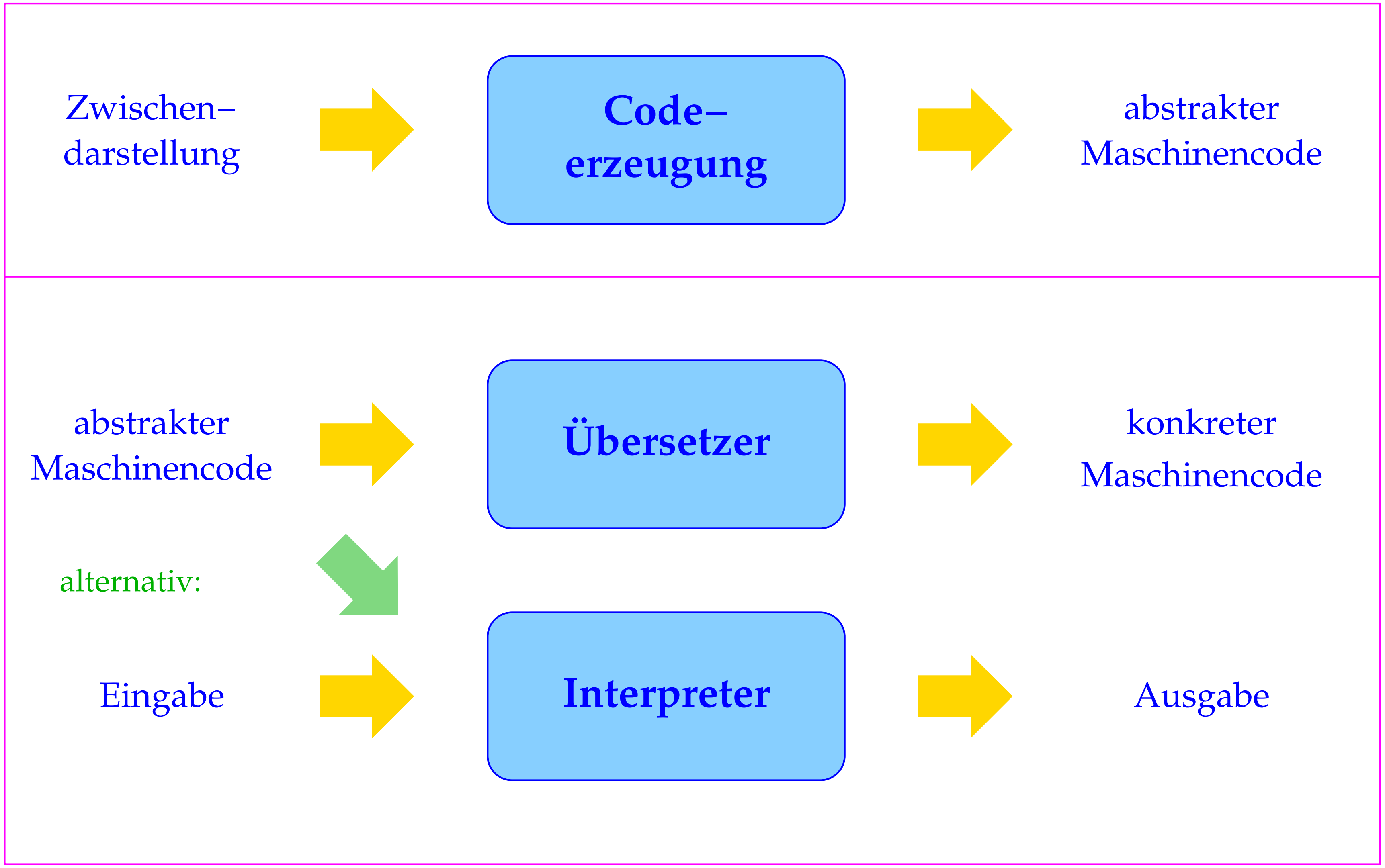
Aufgaben der Code-Erzeugung:

Ziel ist eine geschickte Ausnutzung der Möglichkeiten der Hardware. Das heißt u.a.:

1. Instruction Selection: Auswahl geeigneter Instruktionen;
2. Registerverteilung: optimale Nutzung der vorhandenen (evt. spezialisierten) Register;
3. Instruction Scheduling: Anordnung von Instruktionen (etwa zum Füllen einer Pipeline).

Weitere gegebenenfalls auszunutzende spezielle Hardware-Features können mehrfache Recheneinheiten sein, verschiedene Caches, ...

Weil konkrete Hardware so vielgestaltig ist, wird die Code-Erzeugung oft erneut in zwei Phasen geteilt:


Zwischen–
darstellung
Code–
erzeugung
abstrakter
Maschinencode
abstrakter
Maschinencode
Übersetzer
konkreter
Maschinencode
alternativ:
Eingabe
Interpreter
Ausgabe

Eine abstrakte Maschine ist eine idealisierte Hardware, für die sich einerseits “leicht” Code erzeugen lässt, die sich andererseits aber auch “leicht” auf realer Hardware implementieren lässt.

Vorteile:

- Die Portierung auf neue Zielarchitekturen vereinfacht sich;
- der Compiler wird flexibler;
- die Realisierung der Programmkonstrukte wird von der Aufgabe entkoppelt, Hardware-Features auszunutzen.

Programmiersprachen, deren Übersetzungen auf abstrakten Maschinen beruhen:

Pascal	$\longrightarrow$	P-Maschine	
Smalltalk	$\longrightarrow$	Bytecode	
Prolog	$\longrightarrow$	WAM	(“Warren Abstract Machine”)
SML, Haskell	$\longrightarrow$	STGM	
Java	$\longrightarrow$	JVM	

Hier werden folgende Sprachen und abstrakte Maschinen betrachtet:

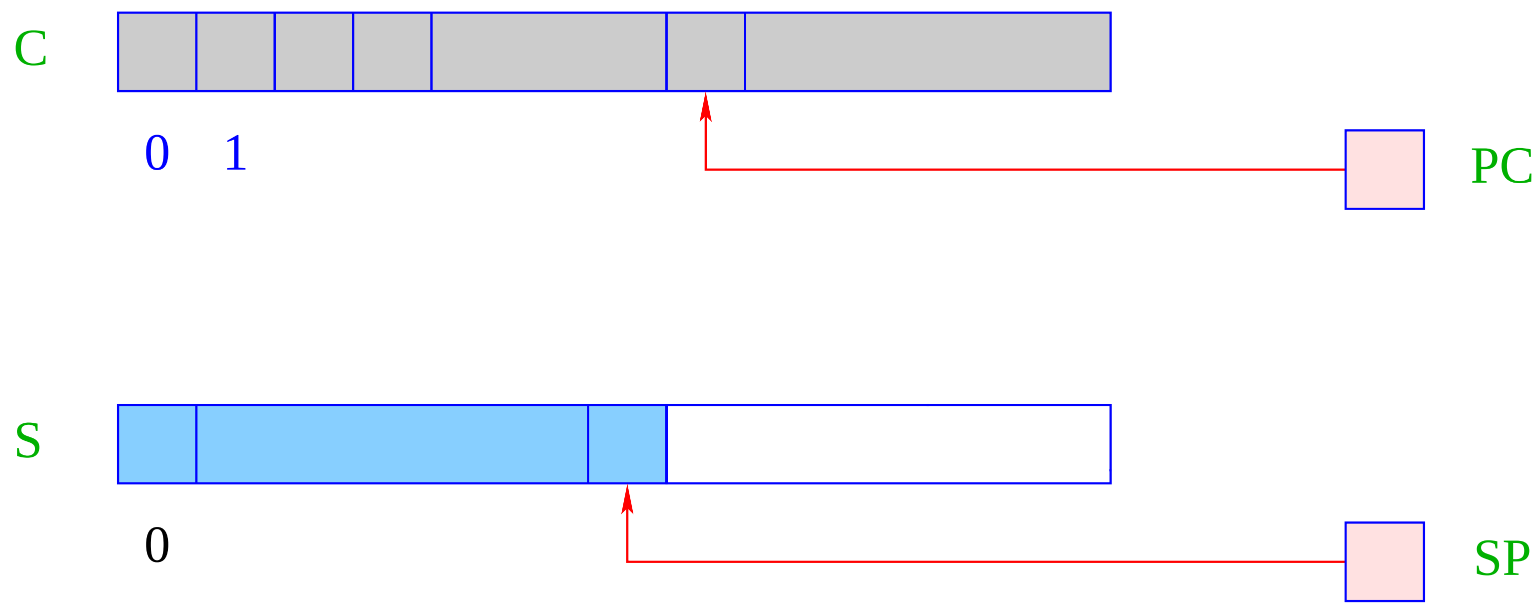
C	$\longrightarrow$	CMa	//	*imperativ*
PuF	$\longrightarrow$	MaMa	//	*funktional*
PuP	$\longrightarrow$	WiM	//	*logikbasiert*

Die Übersetzung von C

1 Die Architektur der CMa

- Jede abstrakte Maschine stellt einen Satz abstrakter Instruktionen zur Verfügung.
- Instruktionen werden auf der abstrakten Hardware ausgeführt.
- Die abstrakte Hardware fassen wir als eine Menge von Datenstrukturen auf, auf die die Instruktionen zugreifen
- ... und die vom Laufzeitsystem verwaltet werden.

Für die CMa benötigen wir:


C
0
1
PC
S
0
SP

- S ist der (Daten-)Speicher, auf dem nach dem LIFO-Prinzip neue Zellen allokiert werden können $\Longrightarrow$ Keller/Stack.
- SP ($\widehat{=}$ Stack Pointer) ist ein Register, das die Adresse der obersten belegten Zelle enthält.

  Vereinfachung: Alle Daten passen jeweils in eine Zelle von S.
- C ist der Code-Speicher, der das Programm enthält.

  Jede Zelle des Felds C kann exakt einen abstrakten Befehl aufnehmen.
- PC ($\widehat{=}$ Program Counter) ist ein Register, das die Adresse des nächsten auszuführenden Befehls enthält.
- Vor Programmausführung enthält der PC die Adresse 0

  $\Longrightarrow$ C[0] enthält den ersten auszuführenden Befehl.

Die Ausführung von Programmen:

- Die Maschine lädt die Instruktion aus C[PC] in ein Instruktions-Register IR und führt sie aus.
- Vor der Ausführung eines Befehls wird der PC um 1 erhöht.

```
while (true) {
    IR = C[PC]; PC++;
    execute (IR);
}
```

- Der PC muss vor der Ausführung der Instruktion erhöht werden, da diese möglicherweise den PC überschreibt :-)
- Die Schleife (der Maschinen-Zyklus) wird durch Ausführung der Instruktion halt verlassen, die die Kontrolle an das Betriebssystem zurückgibt.
- Die weiteren Instruktionen führen wir nach Bedarf ein :-)

2 Einfache Ausdrücke und Wertzuweisungen

Aufgabe: werte den Ausdruck $(1 + 7) * 3$ aus!

Das heißt: erzeuge eine Instruktionsfolge, die

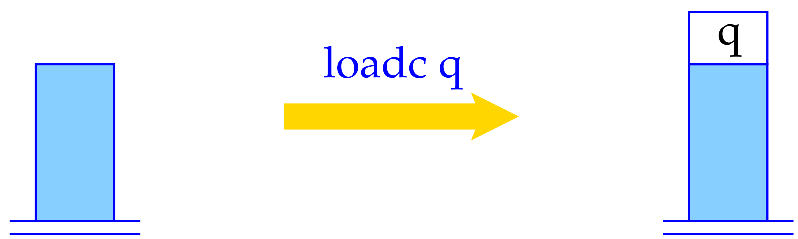
- den Wert des Ausdrucks ermittelt und dann
- oben auf dem Keller ablegt...

Idee:

- berechne erst die Werte für die Teilausdrücke;
- merke diese Zwischenergebnisse oben auf dem Keller;
- wende dann den Operator an!

Generelles Prinzip:

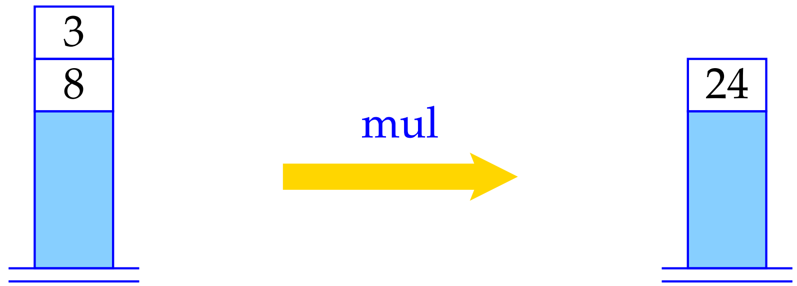
- die Argumente für Instruktionen werden oben auf dem Keller erwartet;
- die Ausführung einer Instruktion konsumiert ihre Argumente;
- möglicherweise berechnete Ergebnisse werden oben auf dem Keller wieder abgelegt.



SP++;

S[SP] = q;

Die Instruktion loadc q benötigt keine Argumente, legt dafür aber als Wert die Konstante q oben auf dem Stack ab.



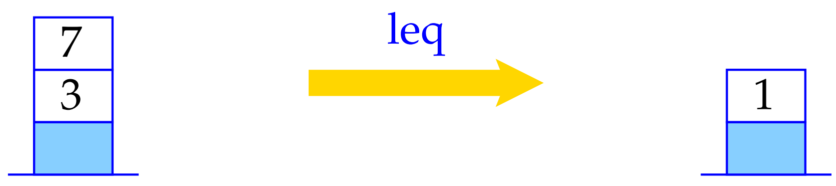
SP--;

S[SP] = S[SP] $*$ S[SP+1];

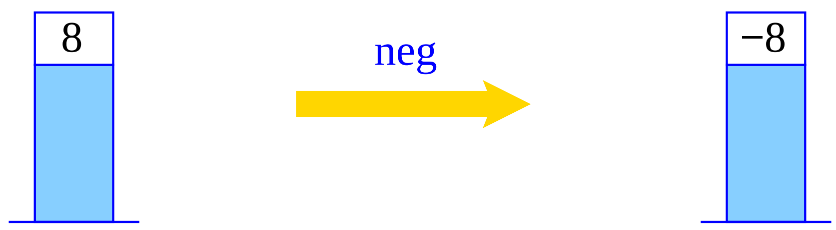
mul erwartet zwei Argumente oben auf dem Stack, konsumiert sie und legt sein Ergebnis oben auf dem Stack ab.

... analog arbeiten auch die übrigen binären arithmetischen und logischen Instruktionen add, sub, div, mod, and, or und xor, wie auch die Vergleiche eq, neq, le, leq, gr und geq.

Beispiel: Der Operator leq



Einstellige Operatoren wie neg und not konsumieren dagegen ein Argument und erzeugen einen Wert:

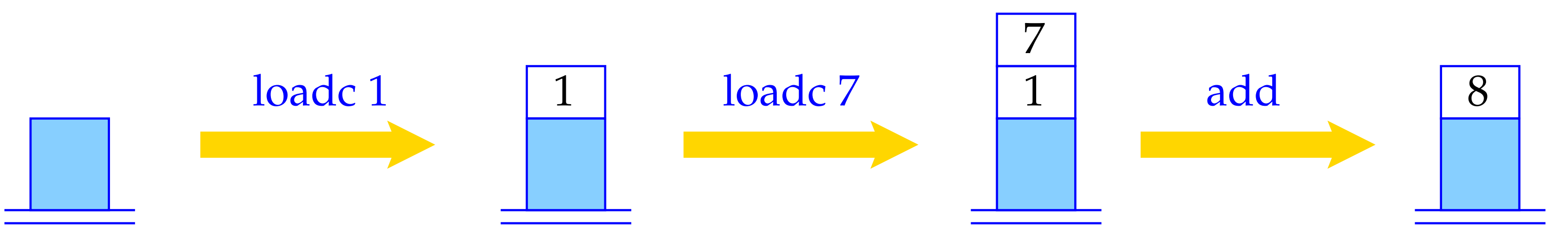


S[SP] = – S[SP];

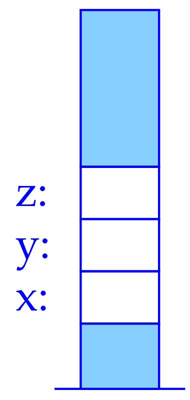
Beispiel: Code für $1 + 7$:

loadc 1 loadc 7 add

Ausführung dieses Codes:



Variablen ordnen wir Speicherzellen in S zu:



Die Übersetzungsfunktionen benötigen als weiteres Argument eine Funktion ρ, die für jede Variable x die (Relativ-)Adresse von x liefert. Die Funktion ρ heißt Adress-Umgebung (Address Environment).

Variablen können auf zwei Weisen verwendet werden.

Beispiel: $x = y + 1$

Für y sind wir am Inhalt der Zelle, für x an der Adresse interessiert.

$$\begin{array}{lcl} \text{L-Wert von } x & = & \text{Adresse von } x \\ \text{R-Wert von } x & = & \text{Inhalt von } x \end{array}$$

$\text{code}_\text{R}\ e\ \rho$	liefert den Code zur Berechnung des R-Werts von e in der Adress-Umgebung ρ
$\text{code}_\text{L}\ e\ \rho$	analog für den L-Wert

Achtung:

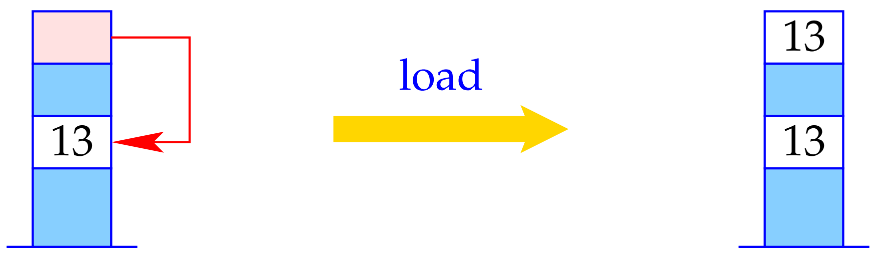
Nicht jeder Ausdruck verfügt über einen L-Wert (Bsp.: $x + 1$).

Wir definieren:

$$
\begin{array}{lcl}
\mathrm{code}_R\ (e_1 + e_2)\ \rho & = & \mathrm{code}_R\ e_1\ \rho \\
 & & \mathrm{code}_R\ e_2\ \rho \\
 & & \mathrm{add} \\
 & & \quad \text{... analog für die anderen binären Operatoren} \\
\mathrm{code}_R\ (-e)\ \rho & = & \mathrm{code}_R\ e\ \rho \\
 & & \mathrm{neg} \\
 & & \quad \text{... analog für andere unäre Operatoren} \\
\mathrm{code}_R\ q\ \rho & = & \mathrm{loadc}\ \mathrm{q} \\
\mathrm{code}_L\ x\ \rho & = & \mathrm{loadc}\ (\rho\, x) \\
 & \ldots &
\end{array}
$$

$$\begin{array}{lll} \mathrm{code_R}\ x\ \rho & = & \mathrm{code_L}\ x\ \rho \\ & & \mathtt{load} \end{array}$$

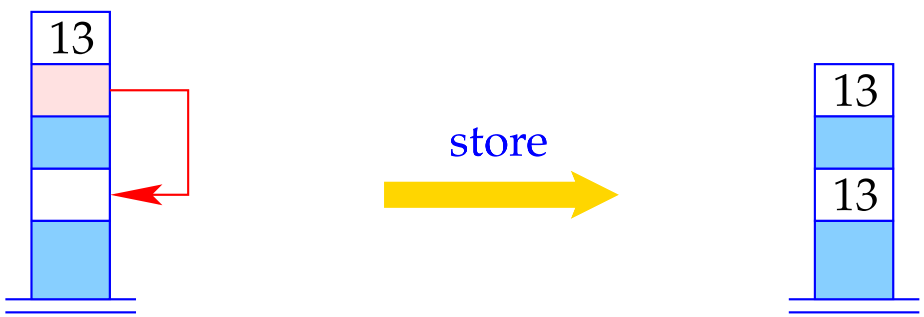
Die Instruktion **load** lädt den Wert der Speicherzelle, deren Adresse oben auf dem Stack liegt.



S[SP] = S[S[SP]];

$$\mathrm{code_R}\ (x = e)\ \rho \quad = \quad \mathrm{code_L}\ x\ \rho$$
$$\mathrm{code_R}\ e\ \rho$$
$$\mathrm{store}$$

Die Instruktion store schreibt den Inhalt der obersten Speicherzelle in die Speicherzelle, deren Adresse darunter auf dem Keller steht, lässt den geschriebenen Wert aber oben auf dem Keller liegen :-)



S[S[SP-1]] = S[SP];

S[SP-1] = S[SP]; SP--;