Script generated by TTT

Title: Simon: Compilerbau (20.06.2012)

Date: Wed Jun 20 14:20:34 CEST 2012

Duration: 72:22 min

Pages: 58

Teiltypen

- Auf den arithmetischen Basistypen **char**, **int**, **long**, etc. gibt es i.a. eine reichhaltige Teiltypen-Beziehungen.
- Dabei bedeutet $t_1 \leq t_2$, dass die Menge der Werte vom Typ t_1
  1. eine Teilmenge der Werte vom Typ t_2 sind;
  2. in einen Wert vom Typ t_2 konvertiert werden können;
  3. die Anforderungen an Werte vom Typ t_2 erfüllen.

int ≤ float double

Teiltypen

- Auf den arithmetischen Basistypen **char**, **int**, **long**, etc. gibt es i.a. eine reichhaltige Teiltypen-Beziehungen.
- Dabei bedeutet $t_1 \leq t_2$, dass die Menge der Werte vom Typ t_1
  1. eine Teilmenge der Werte vom Typ t_2 sind;
  2. in einen Wert vom Typ t_2 konvertiert werden können;
  3. die Anforderungen an Werte vom Typ t_2 erfüllen.

Erweitere Teiltypen-Beziehungen der Basistypen auf komplexe Typen

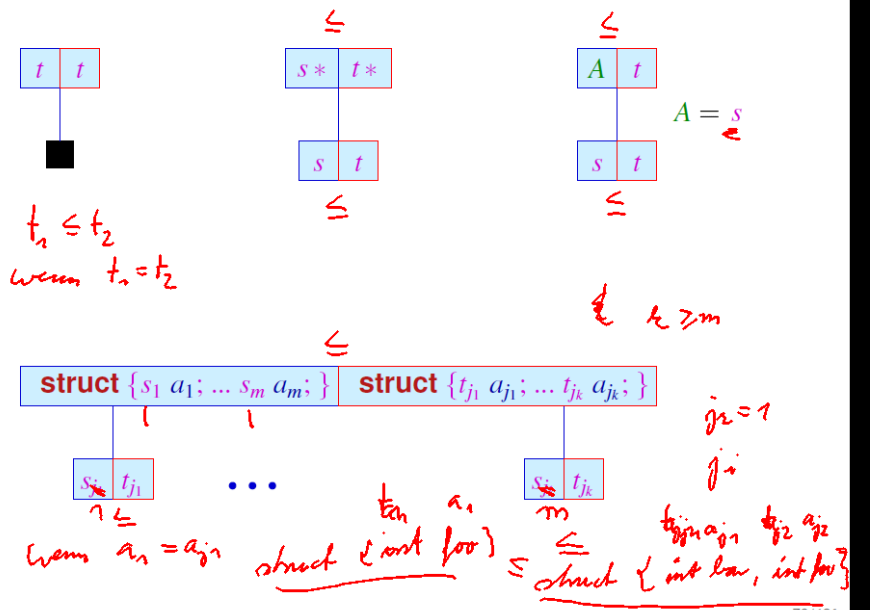
Beispiel: Teiltypen

Betrachte:

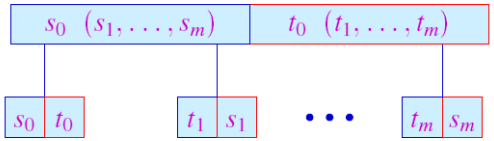
```
string extractInfo( struct { string info; } x) {
  return x.info;
}
```

- Wir möchten dass `extractInfo` für alle Argument-Strukturen funktioniert, die eine Komponente `string info` besitzen
- Wann $t_1 \leq t_2$ gelten soll, beschreiben wir durch Regeln
- Die Idee ist vergleichbar zur Anwendbarkeit auf Unterklassen (aber allgemeiner)

Regeln für Wohlgetyptheit von Teiltypen



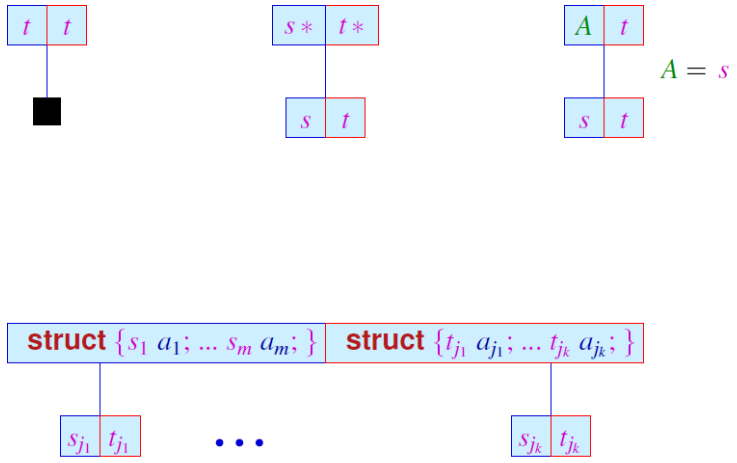
Regeln und Beispiel für Teiltypen



Beispiele:

struct {**int** a; **int** b;}	**struct** {**float** a;}
int (**int**)	**float** (**float**)
int (**float**)	**float** (**int**)

Regeln für Wohlgetyptheit von Teiltypen



Teiltypen

- Auf den arithmetischen Basistypen **`char`**, **`int`**, **`long`**, etc. gibt es i.a. eine reichhaltige Teiltypen-Beziehungen.
- Dabei bedeutet $t_1 \leq t_2$, dass die Menge der Werte vom Typ t_1
  1. eine Teilmenge der Werte vom Typ t_2 sind;
  2. in einen Wert vom Typ t_2 konvertiert werden können;
  3. die Anforderungen an Werte vom Typ t_2 erfüllen.

Erweitere Teiltypen-Beziehungen der Basistypen auf komplexe Typen

Beispiel: Teiltypen

Betrachte:

```
string extractInfo( struct { string info; } x) {
  return x.info;
}
```

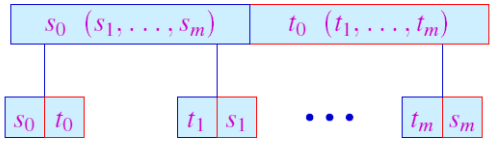
- Wir möchten dass `extractInfo` für alle Argument-Strukturen funktioniert, die eine Komponente `string info` besitzen
- Wann $t_1 \leq t_2$ gelten soll, beschreiben wir durch Regeln
- Die Idee ist vergleichbar zur Anwendbarkeit auf Unterklassen (aber allgemeiner)

Teiltypen

- Auf den arithmetischen Basistypen **`char`**, **`int`**, **`long`**, etc. gibt es i.a. eine reichhaltige Teiltypen-Beziehungen.
- Dabei bedeutet $t_1 \leq t_2$, dass die Menge der Werte vom Typ t_1
  1. eine Teilmenge der Werte vom Typ t_2 sind;
  2. in einen Wert vom Typ t_2 konvertiert werden können;
  3. die Anforderungen an Werte vom Typ t_2 erfüllen.

Erweitere Teiltypen-Beziehungen der Basistypen auf komplexe Typen

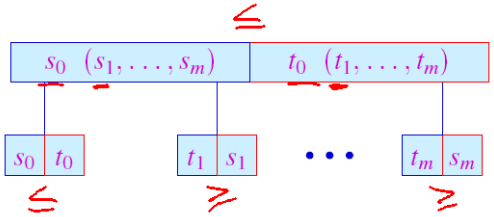
Regeln und Beispiel für Teiltypen



Beispiele:

struct {**int** a; **int** b;}	$\leq$	**struct** {**float** a;}
int (**int**)		**float** (**float**)
int (**float**)	$\leq$	**float** (**int**)

Regeln und Beispiel für Teiltypen

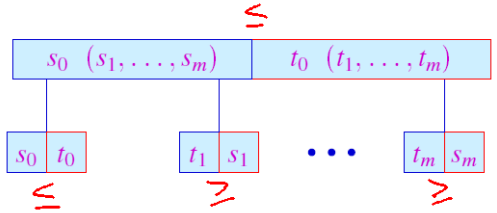


Beispiele:

struct {**int** a; **int** b;}	**struct** {**float** a;}
int (**int**)	**float** (**float**)
int (**float**)	**float** (**int**)

Achtung:

Regeln und Beispiel für Teiltypen



Beispiele:

struct {**int** a; **int** b;}	$\leq$	**struct** {**float** a;}
int (**int**)	$\not\leq$	**float** (**float**)
int (**float**)	$\leq$	**float** (**int**)

Achtung:

- Bei Funktionen gilt:
- Rückgabewerte sind in normaler Teiltypen-Beziehung
- bei Argumenten dreht sich die Anordnung der Typen um

Ko- und Kontravarianz

Definition

Sei $s_0(s_1, \ldots s_n) \leq t_0(t_1, \ldots t_n)$ zwei Funktionstypen, in Teiltyp-Beziehung. Dann gilt:

- Kovarianz der Rückgabewerte $s_0 \leq t_0$
- Kontravarianz der Argumente $s_i \geq t_i$ für $1 < i \leq n$

Ko- und Kontravarianz

Definition

Sei $s_0(s_1, \ldots s_n) \leq t_0(t_1, \ldots t_n)$ zwei Funktionstypen, in Teiltyp-Beziehung. Dann gilt:

- Kovarianz der Rückgabewerte $s_0 \leq t_0$
- Kontravarianz der Argumente $s_i \geq t_i$ für $1 < i \leq n$

Betrachte Beispiel aus funktionalen Sprachen:

int $\rightarrow$ (**float** $\rightarrow$ **int**) $\leq$ **int** $\rightarrow$ (**int** $\rightarrow$ **float**)

① (float → int) ≤ (int → float)

② int ≤ int

float ≥ int

int ≤ float

Ko- und Kontravarianz

Definition

Sei $s_0(s_1, \ldots s_n) \leq t_0(t_1, \ldots t_n)$ zwei Funktionstypen, in Teiltyp-Beziehung. Dann gilt:

- Kovarianz der Rückgabewerte $s_0 \leq t_0$
- Kontravarianz der Argumente $s_i \geq t_i$ für $1 < i \leq n$

Betrachte Beispiel aus funktionalen Sprachen:

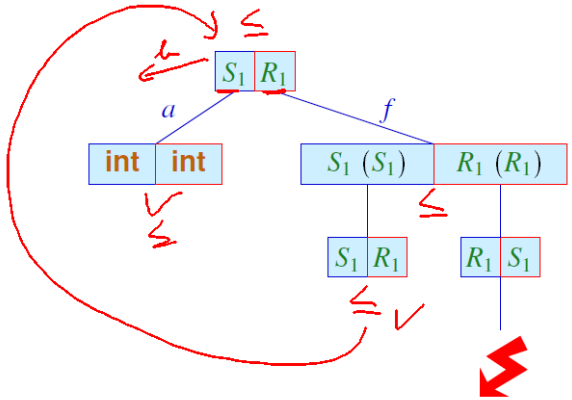
int $\rightarrow$ **float** $\rightarrow$ **int** $\leq$ **int** $\rightarrow$ **int** $\rightarrow$ **float**

Diese Regeln können wir direkt benutzen, um auch für rekursive Typen die Teiltyp-Relation zu entscheiden

Teiltypen: Anwendung der Regeln (I)

Prüfe ob $S_1 \leq R_1$:

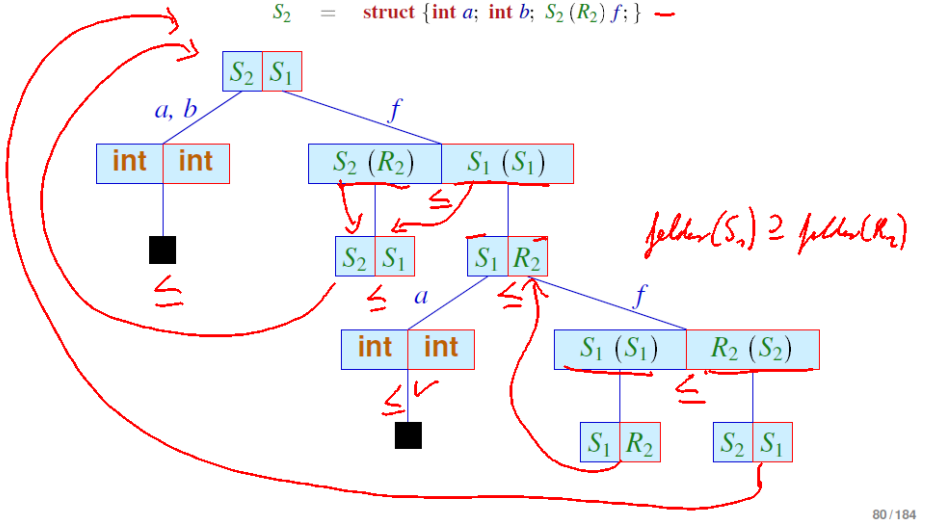
$$
\begin{array}{lcl}
R_1 & = & \textbf{struct}\ \{\textbf{int}\ a;\ R_1\,(R_1)\,f;\} \\
S_1 & = & \textbf{struct}\ \{\textbf{int}\ a;\ \textbf{int}\ b;\ S_1\,(S_1)\,f;\} \\
R_2 & = & \textbf{struct}\ \{\textbf{int}\ a;\ R_2\,(S_2)\,f;\} \\
S_2 & = & \textbf{struct}\ \{\textbf{int}\ a;\ \textbf{int}\ b;\ S_2\,(R_2)\,f;\}
\end{array}
$$



Teiltypen: Anwendung der Regeln (II)

Prüfe ob $S_2 \leq S_1$:

$$
\begin{array}{lcl}
R_1 & = & \textbf{struct}\ \{\textbf{int}\ a;\ R_1\,(R_1)\,f;\} \\
S_1 & = & \textbf{struct}\ \{\textbf{int}\ a;\ \textbf{int}\ b;\ S_1\,(S_1)\,f;\} \\
R_2 & = & \textbf{struct}\ \{\textbf{int}\ a;\ R_2\,(S_2)\,f;\} \\
S_2 & = & \textbf{struct}\ \{\textbf{int}\ a;\ \textbf{int}\ b;\ S_2\,(R_2)\,f;\}
\end{array}
$$



Diskussion

- Um die Beweisbäume nicht in den Himmel wachsen zu lassen, werden Zwischenknoten oft ausgelassen
- Strukturelle Teiltypen sind sehr mächtig und deshalb nicht ganz leicht zu durchschauen.
- Java verallgemeinert Strukturen zu Objekten / Klassen.
- Teiltyp-Beziehungen zwischen Klassen müssen explizit deklariert werden
- Durch Vererbung wird sichergestellt, dass Unterklassen über die (sichtbaren) Komponenten der Oberklasse verfügen
- Überdecken einer Komponente mit einer anderen gleichen Namens ist möglich — aber nur, wenn diese keine Methode ist

Themengebiet:

Die Synthesephase

Codegenerierung: Überblick

Vom AST des Programs erzeugen wir induktiv Instruktionen:

- für jedes Nicht-Terminal Symbol in der Grammatik gibt es eine Regel zur Generierung von Machineninstruktionen
- der Code ist nur ein weiteres Attribut im Syntaxbaum
- die Codegenerierung benutzt die schon berechneten Attribute

Codegenerierung: Überblick

Vom AST des Programs erzeugen wir induktiv Instruktionen:

- für jedes Nicht-Terminal Symbol in der Grammatik gibt es eine Regel zur Generierung von Machineninstruktionen
- der Code ist nur ein weiteres Attribut im Syntaxbaum
- die Codegenerierung benutzt die schon berechneten Attribute

Um die Codeerzeugung zu spezifizieren, benötigt man

- die Semantik der Ausgangssprache (C Standard)
- die Semantik der Maschineninstruktionen

84 / 184

Kapitel 1:
Die Register C-Maschine

Die Register C-Maschine (R-CMa)

Wir erzeugen Code für die C-Maschine.
Die Register C-Maschine ist eine virtuelle Machine.

- es existiert kein Prozessor, der die Instruktionen der CMA ausführen kann
- aber es steht ein Interpreter zur Verfügung
- es seht auch eine Visualisierungsumgebung zur Verfügung
- die R-CMa kennt keine `double`, `float`, `char`, `short` oder `long` Typen
- die R-CMa hat kennt keine Instruktionen um mit dem Betriebssystem zu kommunizieren (Eingabe/Ausgabe)
- die R-CMa hat eine unendliche Anzahl an Registern

Die Register C-Maschine (R-CMa)

Wir erzeugen Code für die C-Maschine.

Die Register C-Maschine ist eine virtuelle Machine.

- es existiert kein Prozessor, der die Instruktionen der CMA ausführen kann
- aber es steht ein Interpreter zur Verfügung
- es seht auch eine Visualisierungsumgebung zur Verfügung
- die R-CMa kennt keine **`double`**, **`float`**, **`char`**, **`short`** oder **`long`** Typen
- die R-CMa hat kennt keine Instruktionen um mit dem Betriebssystem zu kommunizieren (Eingabe/Ausgabe)
- die R-CMa hat eine unendliche Anzahl an Registern

Die R-CMa ist realistischer als es auf den ersten Blick erscheint:

- die Einschränkungen können mit wenig Aufwand aufgehoben werden
- die Java Virtual Machine (JVM) ist sehr ähnlich zur R-CMa ohne Register
- ein Interpreter für die R-CMa läuft auf jeder Platform

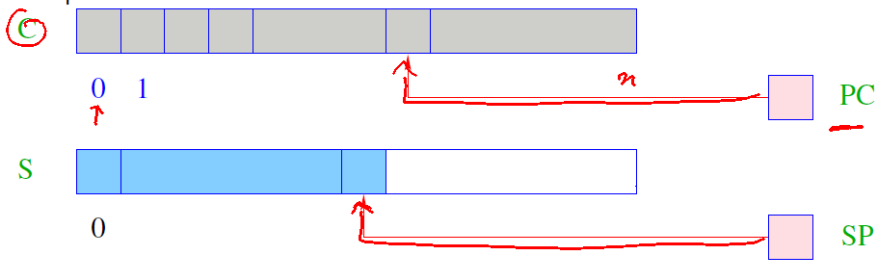
Virtuelle Maschinen

Ein virtuelle Maschine definiert sich wie folgt:

- Jede virtuelle Maschine stellt einen Satz von Instruktionen zur Verfügung.
- Instruktionen werden auf der virtuellen Hardware ausgeführt.
- Die virtuelle Hardware ist eine Menge von Datenstrukturen, auf die die Instruktionen zugreifen
- ... und die vom Laufzeitsystem verwaltet werden.
- der Interpreter ist Teil des Laufzeitsystems

Komponenten einer virtuellen Maschine

Beispiel für Java:

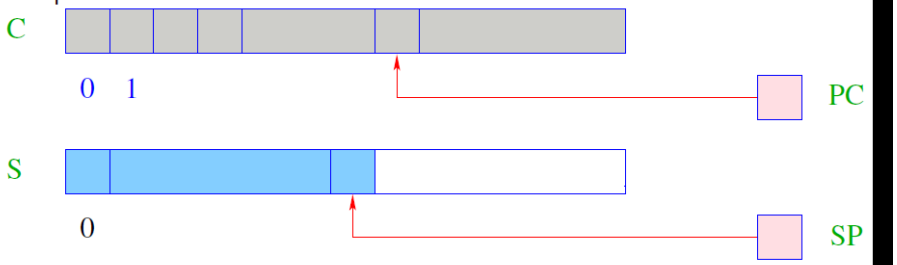


Eine virtuelle Maschine wie die JVM hat folgende Strukturen:

- S: (Daten-)Speicher, auf dem nach dem LIFO-Prinzip neue Zellen allokiert werden können ⇝ Keller/Stack.
- SP: (≙ Stack Pointer) oberste belegte Zelle in S
- am oberen Ende von S liegt der Heap

Komponenten einer virtuellen Maschine

Beispiel für Java:



Eine virtuelle Maschine wie die JVM hat folgende Strukturen:

- S: (Daten-)Speicher, auf dem nach dem LIFO-Prinzip neue Zellen allokiert werden können ⇝ Keller/Stack.
- SP: (≙ Stack Pointer) oberste belegte Zelle in S
- am oberen Ende von S liegt der Heap
- C ist der Code-Speicher
  - Jede Zelle von C kann exakt einen virtuellen Befehl aufnehmen
  - C kann nur *gelesen* werden
- PC (≙ Program Counter) Adresse des nächsten auszuführenden Befehls
- PC enthält zu Begin 0

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 wird durch Ausführung der Instruktion halt verlassen, die die Kontrolle an das Betriebssystem zurückgibt

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

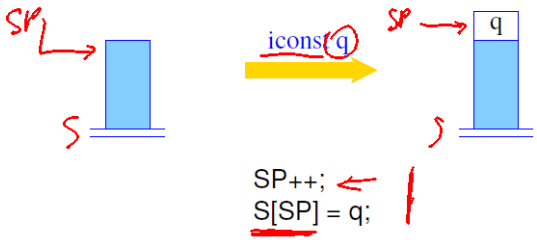
Einfache Ausdrücke und Wertzuweisungen

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

Generelles Prinzip

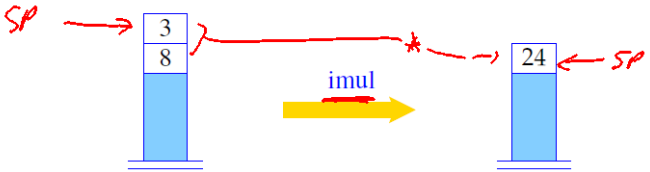
- die Argumente für Instruktionen werden oben auf dem Keller erwartet;
- die Ausführung einer Instruktion konsumiert ihre Argumente;
- möglicherweise berechnete Ergebnisse werden oben auf dem Keller wieder abgelegt.



Die Instruktion iconst q legt die `int`-Konstante q auf dem Stack ab.

Binäre Operatoren

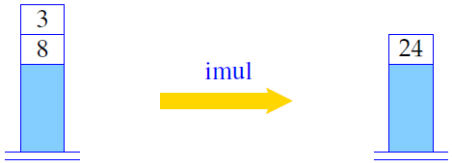
Operatoren mit zwei Argumenten werden wie folgt angewandt:




```
SP--;
S[SP] = S[SP] * S[SP+1];
```


Binäre Operatoren

Operatoren mit zwei Argumenten werden wie folgt angewandt:

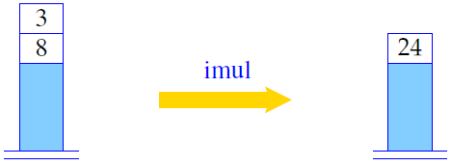



```
SP--;
S[SP] = S[SP] * S[SP+1];
```

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

Binäre Operatoren

Operatoren mit zwei Argumenten werden wie folgt angewandt:




```
SP--;
S[SP] = S[SP] * S[SP+1];
```

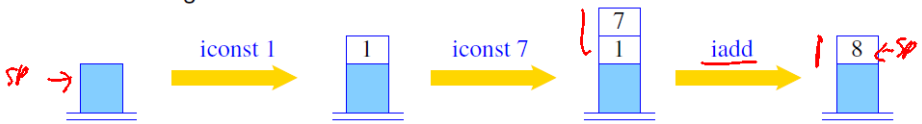
- imul erwartet zwei Argumente oben auf dem Stack, konsumiert sie und legt sein Ergebnis oben auf dem Stack ab.
- analog arbeiten auch die übrigen binären arithmetischen und logischen Instruktionen iadd, isub, idiv, imod, etc.

Zusammensetzung von Instruktionen

Beispiel: Erzeuge Code für 1 + 7:

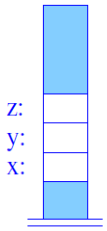
```
iconst 1
iconst 7
iadd
```

Ausführung dieses Codes:



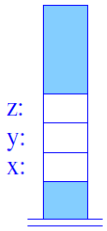
Ausdrücke mit Variablen

Variablen haben eine Speicherzellen in S:



Ausdrücke mit Variablen

Variablen haben eine Speicherzellen in S:

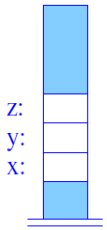


- Die Zuordnung von Adressen zu Variablen kann während der Erstellung der Symboltabelle erfolgen. Die Adresse wird an dem *Deklarationsknoten* der Variable gespeichert.

95 / 184

Ausdrücke mit Variablen

Variablen haben eine Speicherzellen in S:

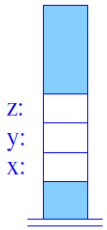


- Die Zuordnung von Adressen zu Variablen kann während der Erstellung der Symboltabelle erfolgen. Die Adresse wird an dem *Deklarationsknoten* der Variable gespeichert.
- Jede *Benutzung* einer Variable hat als Attribut einen Zeiger auf den Deklarationsknoten der Variable.

95 / 184

Ausdrücke mit Variablen

Variablen haben eine Speicherzellen in S:



- Die Zuordnung von Adressen zu Variablen kann während der Erstellung der Symboltabelle erfolgen. Die Adresse wird an dem *Deklarationsknoten* der Variable gespeichert.
- Jede *Benutzung* einer Variable hat als Attribut einen Zeiger auf den Deklarationsknoten der Variable.
- Im folgenden benutzen wir Übersetzungsfunktionen die eine Funktion ρ nehmen, die für jede Variable x die (Relativ-)Adresse von x liefert. Die Funktion ρ heißt *Adress-Umgebung* (Address Environment).

95 / 184

Lesen von Variablen

Die Instruktion iload k lädt den Wert der Speicherzelle k, relativ zur Spitze vom Stapel.

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

Beispiel: Berechne $x + 2$ wobei $\rho = \{x \mapsto 1\}$.

```
iload 1
iconst 2
iadd
```

Motivation für die Register C-Maschine

Ein moderner RISC-Prozessor besitzt eine fixe Anzahl von Universalregistern.

Motivation für die Register C-Maschine

Ein moderner RISC-Prozessor besitzt eine fixe Anzahl von Universalregistern.

- arithmetische Operationen können oft nur auf diesen Registern ausgeführt werden
- Zugriffe auf den Speicher werden durch Adressen in Registern vorgenommen
- Register müssen gesichert werden, wenn Funktionen aufgerufen werden

Motivation für die Register C-Maschine

Ein moderner RISC-Prozessor besitzt eine fixe Anzahl von Universalregistern.

- arithmetische Operationen können oft nur auf diesen Registern ausgeführt werden
- Zugriffe auf den Speicher werden durch Adressen in Registern vorgenommen
- Register müssen gesichert werden, wenn Funktionen aufgerufen werden

Die Übersetzung von Code für einen solchen Prozessor muss:

1. Variablen und Funktionsargumente in Registern speichern
2. während eines Funktionsaufrufes die entsprechenden Register auf dem Keller *sichern*
3. beliebige Berechnungen mittels *endlich* vieler Register ausdrücken

Motivation für die Register C-Maschine

Ein moderner RISC-Prozessor besitzt eine fixe Anzahl von Universalregistern.

- arithmetische Operationen können oft nur auf diesen Registern ausgeführt werden
- Zugriffe auf den Speicher werden durch Adressen in Registern vorgenommen
- Register müssen gesichert werden, wenn Funktionen aufgerufen werden

Die Übersetzung von Code für einen solchen Prozessor muss:

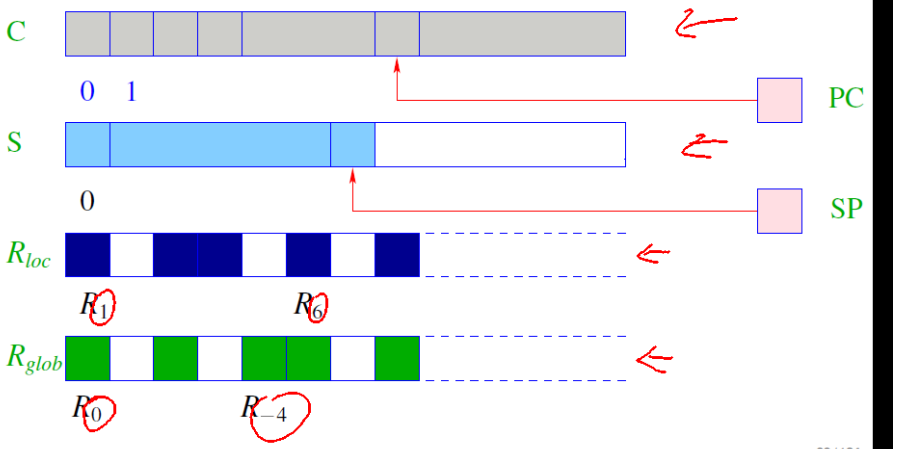
1. Variablen und Funktionsargumente in Registern speichern
2. während eines Funktionsaufrufes die entsprechenden Register auf dem Keller *sichern*
3. beliebige Berechnungen mittels *endlich* vieler Register ausdrücken

$\rightsquigarrow$ betrachte nur die ersten zwei Punkte (und behandle den letzten Punkt als separates Problem)

Prinzip der Register C-Maschine

Die R-CMa besitzt einen Stack/Heap und ein Code-Segment, wie die JVM, und zusätzlich zwei unbegrenzte Mengen an Registern.

- *lokale* Register sind $R_1, R_2, \ldots R_i, \ldots$
- *globale* Register sind $R_0, R_{-1}, \ldots R_j, \ldots$



Die Registersätze der R-CMa

Die zwei Registersätze haben die folgenden Aufgaben:

1. die *lokalen* Register R_i
   - speichern temporäre Zwischenergebnisse
   - speichern lokale Variablen einer Funktion
   - können effizient auf den Stack gerettet werden

Die Registersätze der R-CMa

Die zwei Registersätze haben die folgenden Aufgaben:

1. die *lokalen* Register R_i
   - speichern temporäre Zwischenergebnisse
   - speichern lokale Variablen einer Funktion
   - können effizient auf den Stack gerettet werden
2. die *globalen* Register R_i
   - speichern Parameter einer Funktion
   - speichern den Rückgabewert einer Funktion

Die Registersätze der R-CMa

Die zwei Registersätze haben die folgenden Aufgaben:

1. die *lokalen* Register R_i
   - speichern temporäre Zwischenergebnisse
   - speichern lokale Variablen einer Funktion
   - können effizient auf den Stack gerettet werden
2. die *globalen* Register R_i
   - speichern Parameter einer Funktion
   - speichern den Rückgabewert einer Funktion

Achtung:
zunächst benutzten wir Register nur für Zwischenergebnisse

Idee für Übersetzung: benutze Registerzähler i:

- Register R_j mit $j < i$ sind in Benutzung
- Register R_j mit $j \geq i$ stehen zur Verfügung

Übersetzen von einfachen Anweisungen

Behandlung von Variablen in Registern; laden von Konstanten:

Instruktion	Semantik	Intuition
loadc R_i c	$R_i = c$	lade Konstante
move R_i R_j	$R_i = R_j$	kopiere R_j nach R_i

Übersetzen von einfachen Anweisungen

Behandlung von Variablen in Registern; laden von Konstanten:

Instruktion	Semantik	Intuition
loadc R_i c	$R_i = c$	lade Konstante
move R_i R_j	$R_i = R_j$	kopiere R_j nach R_i

Wir definieren folgende Übersetzungsschemata (mit $\rho\, x = a$):

$$
\begin{aligned}
\text{code}^i_{\text{R}}\ c\ \rho &= \text{loadc}\ R_i\ c \\
\text{code}^i_{\text{R}}\ x\ \rho &= \text{move}\ R_i\ R_a \\
\text{code}^i_{\text{R}}\ x = e\ \rho &= \text{code}^i_{\text{R}}\ e\ \rho \\
&\quad\ \text{move}\ R_a\ R_i
\end{aligned}
$$

Übersetzen von einfachen Anweisungen

Behandlung von Variablen in Registern; laden von Konstanten:

Instruktion	Semantik	Intuition
loadc R_i c	$R_i = c$	lade Konstante
move R_i R_j	$R_i = R_j$	kopiere R_j nach R_i

Wir definieren folgende Übersetzungsschemata (mit $\rho\, x = a$):

$$
\begin{aligned}
\text{code}^i_{\text{R}}\ c\ \rho &= \text{loadc}\ R_i\ c \\
\text{code}^i_{\text{R}}\ x\ \rho &= \text{move}\ R_i\ R_a \\
\text{code}^i_{\text{R}}\ x = e\ \rho &= \text{code}^i_{\text{R}}\ e\ \rho \\
&\quad\ \text{move}\ R_a\ R_i
\end{aligned}
$$

Beachte: alle Instruktionen benutzen die Intel-Konvention (im Ggs. zur AT&T Konvention): op dst src_1 $\ldots$ src_n.

Übersetzen von Ausdrücken

Sei $\mathrm{op} = \{add, sub, div, mul, mod, le, gr, eq, leq, geq, and, or\}$. Die R-CMa kennt für jeden Operator op eine Instruktion

$$\mathrm{op}\ R_i\ R_j\ R_k$$

wobei R_i das Zielregister, R_j das erste und R_k das zweite Argument ist.

Entsprechend erzeugen wir den Code wie folgt:

$$\begin{aligned}\mathrm{code}^i_{\mathrm{R}}\ e_1\ \mathrm{op}\ e_2\ \rho \quad = \quad & \mathrm{code}^i_{\mathrm{R}}\ e_1\ \rho \\ & \mathrm{code}^{i+1}_{\mathrm{R}}\ e_2\ \rho \\ & \mathrm{op}\ R_i\ R_i\ R_{i+1}\end{aligned}$$

Übersetzen von Ausdrücken

Sei $\mathrm{op} = \{add, sub, div, mul, mod, le, gr, eq, leq, geq, and, or\}$. Die R-CMa kennt für jeden Operator op eine Instruktion

$$\mathrm{op}\ R_i\ R_j\ R_k$$

wobei R_i das Zielregister, R_j das erste und R_k das zweite Argument ist.

Entsprechend erzeugen wir den Code wie folgt:

$$\begin{aligned}\mathrm{code}^i_{\mathrm{R}}\ e_1\ \mathrm{op}\ e_2\ \rho \quad = \quad & \mathrm{code}^i_{\mathrm{R}}\ e_1\ \rho \\ & \mathrm{code}^{i+1}_{\mathrm{R}}\ e_2\ \rho \\ & \mathrm{op}\ R_i\ R_i\ R_{i+1}\end{aligned}$$

Beispiel: Übersetze $3 \star 4$ mit $i = 4$:

$$\begin{aligned}\mathrm{code}^4_{\mathrm{R}}\ 3 \star 4\ \rho \quad = \quad & \mathrm{code}^4_{\mathrm{R}}\ 3\ \rho \\ & \mathrm{code}^5_{\mathrm{R}}\ 4\ \rho \\ & \mathrm{mul}\ R_4\ R_4\ R_5\end{aligned}$$

Übersetzen von Ausdrücken

Sei $\mathrm{op} = \{add, sub, div, mul, mod, le, gr, eq, leq, geq, and, or\}$. Die R-CMa kennt für jeden Operator op eine Instruktion

$$\mathrm{op}\ R_i\ R_j\ R_k$$

wobei R_i das Zielregister, R_j das erste und R_k das zweite Argument ist.

Entsprechend erzeugen wir den Code wie folgt:

$$\begin{aligned}\mathrm{code}^i_{\mathrm{R}}\ e_1\ \mathrm{op}\ e_2\ \rho \quad = \quad & \mathrm{code}^i_{\mathrm{R}}\ e_1\ \rho \\ & \mathrm{code}^{i+1}_{\mathrm{R}}\ e_2\ \rho \\ & \mathrm{op}\ R_i\ R_i\ R_{i+1}\end{aligned}$$

Beispiel: Übersetze $3 \star 4$ mit $i = 4$:

$$\begin{aligned}\mathrm{code}^4_{\mathrm{R}}\ 3 \star 4\ \rho \quad = \quad & \mathrm{code}^4_{\mathrm{R}}\ 3\ \rho \\ & \mathrm{code}^5_{\mathrm{R}}\ 4\ \rho \\ & \mathrm{mul}\ R_4\ R_4\ R_5\end{aligned}$$

Verwaltung temporärer Register

Beachte, dass temporäre Register wiederverwendet werden:
Übersetze $3 \star 4 + 3 \star 4$ mit $t = 4$:

$$\begin{aligned}\mathrm{code}^4_{\mathrm{R}}\ (3 \star 4) + (3 \star 4)\ \rho \quad = \quad & \mathrm{code}^4_{\mathrm{R}}\ 3 \star 4\ \rho \\ & \mathrm{code}^5_{\mathrm{R}}\ 3 \star 4\ \rho \\ & \mathrm{add}\ R_4\ R_4\ R_5\end{aligned}$$

mit

$$\begin{aligned}\mathrm{code}^i_{\mathrm{R}}\ 3 \star 4\ \rho \quad = \quad & \mathrm{loadc}\ R_i\ 3 \\ & \mathrm{loadc}\ R_{i+1}\ 4 \\ & \mathrm{mul}\ R_i\ R_i\ R_{i+1}\end{aligned}$$

ergibt sich

$$\begin{aligned}\mathrm{code}^4_{\mathrm{R}}\ 3 \star 4 + 3 \star 4\ \rho \quad = \quad & \mathrm{loadc}\ R_4\ 3 \\ & \mathrm{loadc}\ R_5\ 4 \\ & \mathrm{mul}\ R_4\ R_4\ R_5 \\ & \mathrm{loadc}\ R_5\ 3 \\ & \mathrm{loadc}\ R_6\ 4 \\ & \mathrm{mul}\ R_5\ R_6\ R_6 \\ & \mathrm{add}\ R_4\ R_4\ R_5\end{aligned}$$

Verwaltung temporärer Register

Beachte, dass temporäre Register wiederverwendet werden:
Übersetze $3\star4+3\star4$ mit $t = 4$:

$$\begin{aligned}\text{code}_R^4\ \ 3\star4+3\star4\ \ \rho \quad = \quad & \text{code}_R^4\ \ 3\star4\ \ \rho \\ & \text{code}_R^5\ \ 3\star4\ \ \rho \\ & \text{add } R_4\ R_4\ R_5\end{aligned}$$

mit

$$\begin{aligned}\text{code}_R^i\ \ 3\star4\ \ \rho \quad = \quad & \text{loadc } R_i\ 3 \\ & \text{loadc } R_{i+1}\ 4 \\ & \text{mul } R_i\ R_i\ R_{i+1}\end{aligned}$$

ergibt sich

$$\begin{aligned}\text{code}_R^4\ \ 3\star4+3\star4\ \ \rho \quad = \quad & \text{loadc } R_4\ 3 \\ & \text{loadc } R_5\ 4 \\ & \text{mul } R_4\ R_4\ R_5 \\ & \text{loadc } R_5\ 3 \\ & \text{loadc } R_6\ 4 \\ & \text{mul } R_5\ R_5\ R_6 \\ & \text{add } R_4\ R_4\ R_5\end{aligned}$$

Semantik der Operatoren

Die Operatoren haben die folgende Semantik:

add $R_i\ R_j\ R_k$	$R_i = R_j + R_k$
sub $R_i\ R_j\ R_k$	$R_i = R_j - R_k$
div $R_i\ R_j\ R_k$	$R_i = R_j / R_k$
mul $R_i\ R_j\ R_k$	$R_i = R_j * R_k$
mod $R_i\ R_j\ R_k$	$R_i = sgn(R_k)k$ wobei
	$\lvert R_j\rvert = n\lvert R_k\rvert + k \wedge n \geq 0, 0 \leq k < \lvert R_k\rvert$
le $R_i\ R_j\ R_k$	$R_i =$ if $R_j < R_k$ then 1 else 0
gr $R_i\ R_j\ R_k$	$R_i =$ if $R_j > R_k$ then 1 else 0
eq $R_i\ R_j\ R_k$	$R_i =$ if $R_j = R_k$ then 1 else 0
leq $R_i\ R_j\ R_k$	$R_i =$ if $R_j \leq R_k$ then 1 else 0
geq $R_i\ R_j\ R_k$	$R_i =$ if $R_j \geq R_k$ then 1 else 0
and $R_i\ R_j\ R_k$	$R_i = R_j\ \&\ R_k$ // bit-wise and
or $R_i\ R_j\ R_k$	$R_i = R_j \mid R_k$ // bit-wise or

Semantik der Operatoren

Die Operatoren haben die folgende Semantik:

add $R_i\ R_j\ R_k$	$R_i = R_j + R_k$
sub $R_i\ R_j\ R_k$	$R_i = R_j - R_k$
div $R_i\ R_j\ R_k$	$R_i = R_j / R_k$
mul $R_i\ R_j\ R_k$	$R_i = R_j * R_k$
mod $R_i\ R_j\ R_k$	$R_i = sgn(R_k)k$ wobei
	$\lvert R_j\rvert = n\lvert R_k\rvert + k \wedge n \geq 0, 0 \leq k < \lvert R_k\rvert$
le $R_i\ R_j\ R_k$	$R_i =$ if $R_j < R_k$ then 1 else 0
gr $R_i\ R_j\ R_k$	$R_i =$ if $R_j > R_k$ then 1 else 0
eq $R_i\ R_j\ R_k$	$R_i =$ if $R_j = R_k$ then 1 else 0
leq $R_i\ R_j\ R_k$	$R_i =$ if $R_j \leq R_k$ then 1 else 0
geq $R_i\ R_j\ R_k$	$R_i =$ if $R_j \geq R_k$ then 1 else 0
and $R_i\ R_j\ R_k$	$R_i = R_j\ \&\ R_k$ // bit-wise and
or $R_i\ R_j\ R_k$	$R_i = R_j \mid R_k$ // bit-wise or

Beachte: alle Register und Speicherzellen enthalten Zahlen in $\mathbb{Z}$