

Inhalt

- Grundlagen
- Die P-Maschine
- Einfache Befehle
 - Transportbefehle
 - Ausdrücke
 - Wertzuweisung
- Kontrollanweisungen
 - Bedingte Anweisungen
 - Fallunterscheidung
 - Iterierte Anweisungen
- Speicherbelegung
 - Einfache Variablen
 - Statische Felder
 - Dynamische Felder
 - Verbunde
 - Zeiger
 - Zusammengesetzte Bezeichner

Lernziele

- Das Grundprinzip der Übersetzung einer imperativen Sprache mithilfe einer abstrakten Kellermaschine beschreiben können
- Erklären können, wie man verschiedene Arten von Anweisungen übersetzt
- Die wichtigsten Techniken der Speicherzuordnung für Variable wiedergeben können

Wesentliche Zielsetzung

- Behandlung der „Essenz“ des Übersetzungsprozesses (für imperative Sprachen), d.h. Abbildung $\ddot{U}_L: L \rightarrow M$ von „Quellprogrammen“ p_L auf „Zielprogramme“ p_M

Dabei zu klären

- Was ist „Quellsprache“ L ?
- Was ist „Zielsprache“ M ?
- Wie sieht die Abbildung $\ddot{U}_L$ aus?

d.h. Übersetzer-Teilaufgabe
„Codeerzeugung“ auf abstrakter Ebene

Prinzip der Definition der Abbildung $\ddot{U}_L: L \rightarrow M$

- Als *rekursive Funktion*, unter Verwendung geeigneter *Hilfsfunktionen*
 $\ddot{U}_L(p_L) =_{\text{def}} \text{code}(p_L, \emptyset, 0)$ (bzw. in klammerfreier Schreibweise: $\text{code } p_L \emptyset 0$) mit
 $\text{code}: L \times (\text{Id} \rightarrow (\text{Adr} \times \text{ST})) \times \text{ST} \rightarrow M$

$\text{Id} \rightarrow (\text{Adr} \times \text{ST})$: Adresszuordnung
ST: Schachtelungstiefe

- wobei
 - „Quellprogramm“ p_L : Folge (syntaktisch + semantisch) korrekter L -Konstrukte („abstrakte Syntax“)
 - „Zielprogramm“: Folge syntaktisch und semantisch korrekter M -Konstrukte
 - $\text{Id} = \{\text{Namen}\}$; Adr , $\text{ST} = \{\text{natürliche Zahlen}\}$

Quellsprache

Konkrete Syntax für das Verständnis irrelevant

- Traditionelle imperative Sprache (Syntax ähnlich zu Pascal oder Modula)
- Wichtigste Konzepte (insbesondere Semantik) als (intuitiv) bekannt vorausgesetzt

Prinzipieller Aufbau (eines Quellsprachenprogramms)

- *Deklarationsteil*: Definition von Namen für Konstante, Variablen, Zeiger, Prozeduren
- *Anweisungsteil*: Änderung des *Programmzustands* (= Werte aller Variablen zu bestimmtem Zeitpunkt) durch Folgen von Anweisungen

Sprachkonstrukte

- Variablen: „Behälter“ für Datenobjekte
 - Einfache
 - Strukturierte:
 - Felder, statisch u. dynamisch
 - Verbunde
 - Anonyme: Zeiger
- Ausdrücke
- Anweisungen
 - Zuweisung
 - Kontrollanweisungen
 - Sprünge (**goto**)
 - Bedingte (**if - fi**, **case**)
 - Iterierte (**while**, **repeat**, **for**)
- Strukturierungsmöglichkeiten
 - Prozeduren
 - Blöcke

P-Maschine — „Pascal-Maschine“

- Abstrakte (oder virtuelle) Maschine (die von verschiedenen realen Gegebenheiten abstrahiert)
- Einfacher (gegenüber realen Maschinen) idealisierter Aufbau
 - „Modell“ verschiedener realer Maschinen
 - Eingeschränkter (größtenteils sprachspezifischer) Befehlsvorrat
- Ziel: Portable Implementierung

Prinzip der Arbeitsweise

- *Deklarationsteil*: Zuordnung von Variablenbezeichnungen zu Speicherzellen
- *Anweisungsteil*: Veränderung der Variablenwerte
 - Durch Abänderung der zugeordneten Speicherzelleninhalte
 - Mithilfe geeigneter Maschinenbefehle

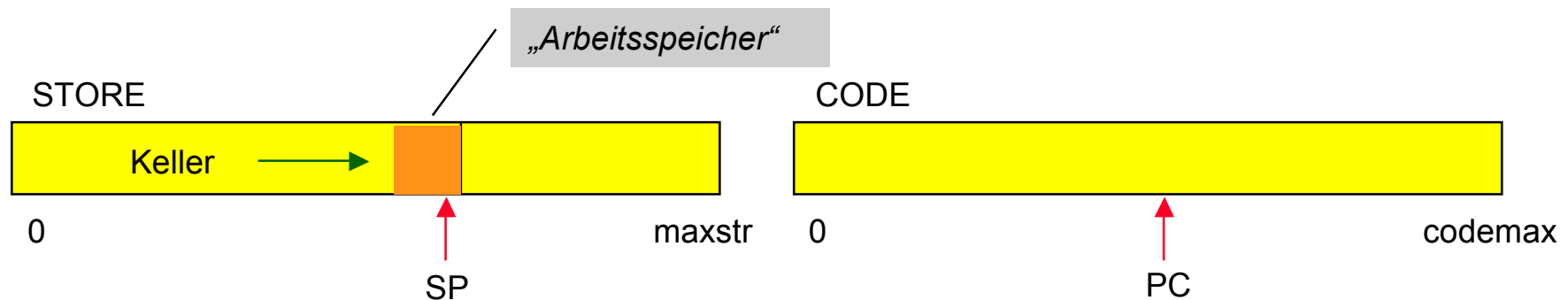
Für traditionelle imperative Sprachen charakteristisch

- Dynamische Existenz von Namen (z.B. rekursive Prozeduren mit lokalen Variablen)
 - ⇒ neue „Inkarnationen“ von Namen (die entstehen und verschwinden)
 - ⇒ kellerartige Speicherverwaltung

Aufbau der P-Maschine

- **Speicher**, aufgeteilt in
 - STORE Datenspeicher (später unterteilt in Keller und Halde)
 - CODE Programmspeicher (für Maschinenbefehle)
- **Register** (spezielle Speicherzellen des Verarbeitungswerks)
 - SP stack pointer „oberste“ belegte Kellerzelle
 - PC program counter Befehlszähler
 - NP new pointer „unterste“ belegte Zelle der Halde
 - EP extreme stack pointer größtmöglicher Wert von SP
 - MP mark pointer Anfang des Kellerrahmens der aktuellen Inkarnation
- **Verarbeitungswerk**, das die einzelnen Zellen des Programmspeichers als Maschinenbefehle interpretiert und ausführt

zunächst
benutzt



Arbeitsweise der P-Maschine

- **Befehlszyklus** (PC mit 0 initialisiert!)
 - do
 - PC := PC+1;
 - führe Befehl in Zelle CODE[PC-1] aus
 - od

Beachte:

*Erst PC erhöhen;
Dann Befehl ausführen*

Befehlsformat

- **<Bezeichner> {<Typ>} {<Modifikator>} {<Adresse>}**

optional

- wobei

- Bezeichner: **add, equ, ldo, sto, ...**
- Typen (zur Differenzierung von Maschinenbefehlen)
 - T beliebiger Typ
 - N numerisch
 - i integer
 - a Adresse
 - b boolean

*Befehle ohne Adressangabe wirken
auf die obersten Kellerzellen*

*nötig, da die Wirkungsweise
mancher Befehle vom Typ der
Operanden abhängig*

- Modifikatoren, Adressen: natürliche Zahlen (z.T. eingeschränkt, z.B. Adressen $\leq \text{maxstr}$)

Elementare Befehle

- *Transportbefehle* (Laden und Speichern)
- *Auswertung von Ausdrücken*

Befehle zum Laden und Speichern

Pseudocode zur Beschreibung der Semantik

Typ von STORE[SP]

	Befehl	Bedeutung	Bedingung	Ergebnis
load object	ldo T q	SP := SP+1; STORE[SP] := STORE[q]	$q \in [0..\text{maxstr}]$	(T)
load constant	ldc T q	SP := SP+1; STORE[SP] := q	Typ(q) = T	(T)
load indirectly	ind T	STORE[SP] := STORE[STORE[SP]]	(a)	(T)
store object	sro T q	STORE[q] := STORE[SP] SP := SP-1;	(T) $q \in [0..\text{maxstr}]$	
store	sto T	STORE[STORE[SP-1]] := STORE[SP] SP := SP-2;	(a, T)	

Typ von STORE[SP-1]

Typ von STORE[SP]

Auswertung von Ausdrücken (generelle Idee)

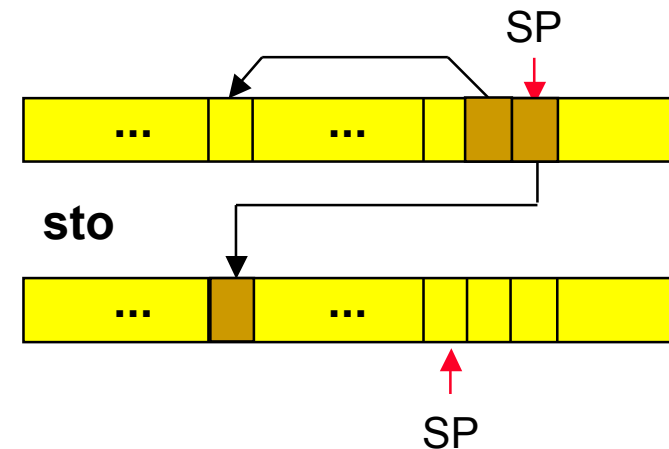
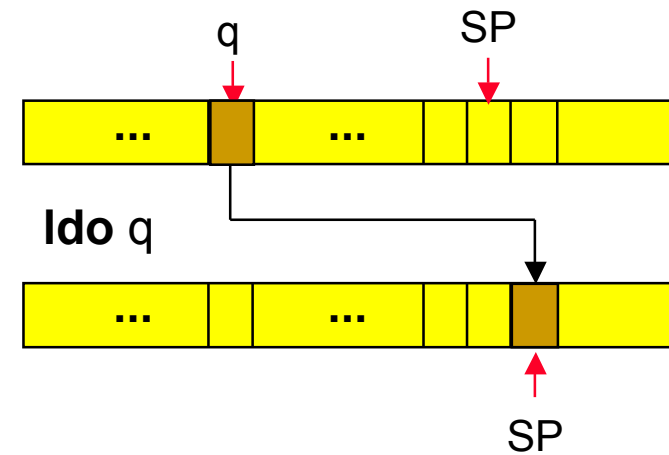
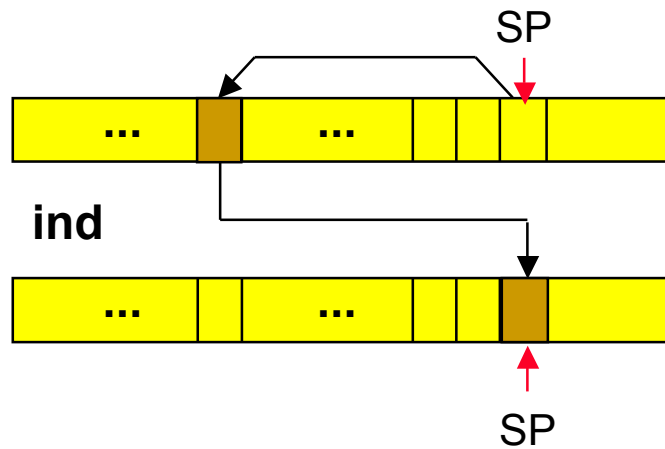
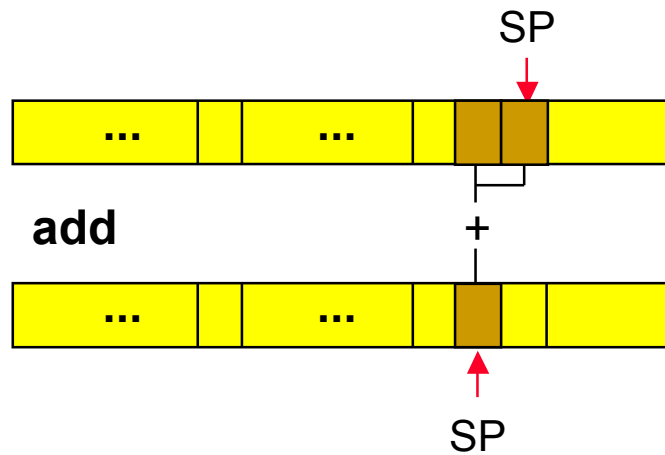
- Operanden (oben) auf den Keller legen
- Auswertung über den obersten Kellerzellen (Anzahl durch jeweilige Operation festgelegt)

Befehle für Ausdrücke

Befehl	Bedeutung	Bedingung	Ergebnis
add N	$\text{STORE}[\text{SP}-1] := \text{STORE}[\text{SP}-1] +_N \text{STORE}[\text{SP}]; \text{SP} := \text{SP}-1;$	(N, N)	(N)
sub N	$\text{STORE}[\text{SP}-1] := \text{STORE}[\text{SP}-1] -_N \text{STORE}[\text{SP}]; \text{SP} := \text{SP}-1;$	(N, N)	(N)
mul N	$\text{STORE}[\text{SP}-1] := \text{STORE}[\text{SP}-1] \times_N \text{STORE}[\text{SP}]; \text{SP} := \text{SP}-1;$	(N, N)	(N)
div N	$\text{STORE}[\text{SP}-1] := \text{STORE}[\text{SP}-1] /_N \text{STORE}[\text{SP}]; \text{SP} := \text{SP}-1;$	(N, N)	(N)
neg N	$\text{STORE}[\text{SP}] := -_N \text{STORE}[\text{SP}];$	(N)	(N)
and	$\text{STORE}[\text{SP}-1] := \text{STORE}[\text{SP}-1] \text{ and } \text{STORE}[\text{SP}]; \text{SP} := \text{SP}-1;$	(b, b)	(b)
or	$\text{STORE}[\text{SP}-1] := \text{STORE}[\text{SP}-1] \text{ or } \text{STORE}[\text{SP}]; \text{SP} := \text{SP}-1;$	(b, b)	(b)
not	$\text{STORE}[\text{SP}] := \text{not } \text{STORE}[\text{SP}];$	(b)	(b)
equ T	$\text{STORE}[\text{SP}-1] := \text{STORE}[\text{SP}-1] =_T \text{STORE}[\text{SP}]; \text{SP} := \text{SP}-1;$	(T, T)	(b)
neq T	$\text{STORE}[\text{SP}-1] := \text{STORE}[\text{SP}-1] \neq_T \text{STORE}[\text{SP}]; \text{SP} := \text{SP}-1;$	(T, T)	(b)
grt T	$\text{STORE}[\text{SP}-1] := \text{STORE}[\text{SP}-1] >_T \text{STORE}[\text{SP}]; \text{SP} := \text{SP}-1;$	(T, T)	(b)
...			



Wirkung einiger Befehle



Übersetzung einer Wertzuweisung $x := y$

- Intuitive Bedeutung
 - y auswerten
 - Wert an der durch x bezeichneten Speicherzelle ablegen
- D.h. Variablenbezeichnung links von „:=“ ist anders zu übersetzen als rechts von „:=“
 - Links: Adresse (**L-Wert**) von x
 - Rechts: Wert (**R-Wert**) von y

Somit

- $\text{code}(x := y) \rho =_{\text{def}} \text{code}_L x \rho; \text{code}_R y \rho; \text{sto } T$
- wobei
 - $\text{code}_L x \rho =_{\text{def}} \text{ldc } a \rho(x)$
 - $\text{code}_R y \rho =_{\text{def}} \text{ldc } T y$, falls y Konstante vom Typ T (andernfalls entsprechend Tabelle, s.u.)

Zweitoberste Kellerzelle:
Adresse von x

Oberste Kellerzelle:
Wert von y

d.h.
– L -Wert von x
– R -Wert von y
– **sto**

Übersetzung von Ausdrücken und Wertzuweisung

Funktion	Bedingung
$\text{code}_R (e_1 = e_2) \rho =_{\text{def}} \text{code}_R e_1 \rho; \text{code}_R e_2 \rho; \mathbf{equ} T$	$\text{Typ}(e_1) = \text{Typ}(e_2) = T$
$\text{code}_R (e_1 \neq e_2) \rho =_{\text{def}} \text{code}_R e_1 \rho; \text{code}_R e_2 \rho; \mathbf{neq} T$	$\text{Typ}(e_1) = \text{Typ}(e_2) = T$
...	
$\text{code}_R (e_1 + e_2) \rho =_{\text{def}} \text{code}_R e_1 \rho; \text{code}_R e_2 \rho; \mathbf{add} N$	$\text{Typ}(e_1) = \text{Typ}(e_2) = N$
$\text{code}_R (e_1 - e_2) \rho =_{\text{def}} \text{code}_R e_1 \rho; \text{code}_R e_2 \rho; \mathbf{sub} N$	$\text{Typ}(e_1) = \text{Typ}(e_2) = N$
$\text{code}_R (e_1 \times e_2) \rho =_{\text{def}} \text{code}_R e_1 \rho; \text{code}_R e_2 \rho; \mathbf{mul} N$	$\text{Typ}(e_1) = \text{Typ}(e_2) = N$
$\text{code}_R (e_1 / e_2) \rho =_{\text{def}} \text{code}_R e_1 \rho; \text{code}_R e_2 \rho; \mathbf{div} N$	$\text{Typ}(e_1) = \text{Typ}(e_2) = N$
$\text{code}_R (-e) \rho =_{\text{def}} \text{code}_R e \rho; \mathbf{neg} N$	$\text{Typ}(e) = N$
$\text{code}_R c \rho =_{\text{def}} \mathbf{ldc} T c$	c Konstante vom Typ T
$\text{code}_R x \rho =_{\text{def}} \text{code}_L x \rho; \mathbf{ind} T$	x Variablenbezeichnung vom Typ T
$\text{code}_L x \rho =_{\text{def}} \mathbf{ldc} a \rho(x)$	x Variablenbezeichnung
$\text{code} (x := e) \rho =_{\text{def}} \text{code}_L x \rho; \text{code}_R e \rho; \mathbf{sto} T$	$\text{Typ}(e) = T$ x Variablenbezeichnung

Einfache
Variable,
Feldkomponente,
Verbund-
komponente



Beispiel

- geg.: $\rho(x) = 5, \rho(y) = 6, \rho(z) = 7$

Annahme:

x, y, z vom Typ integer

- `code (x := (y + (y × z))) ρ =`

`codeL x ρ; codeR (y + (y × z)) ρ; sto i =`

`ldc a 5; codeR (y + (y × z)) ρ; sto i =`

`ldc a 5; codeR (y) ρ; codeR(y × z) ρ; add i; sto i =`

`ldc a 5; ldc a 6; ind i; codeR(y × z) ρ; add i; sto i =`

`ldc a 5; ldc a 6; ind i; codeR(y) ρ; codeR(z) ρ; mul i; add i; sto i =`

`ldc a 5; ldc a 6; ind i; ldc a 6; ind i; codeR(z) ρ; mul i; add i; sto i =`

`ldc a 5; ldc a 6; ind i; ldc a 6; ind i; ldc a 7; ind i; mul i; add i; sto i`

Aufeinander folgende Anweisungen („Anweisungsfolgen“)

- Folgt unmittelbar aus allgemeiner Definition von code
 $\text{code } (st_1; st_2) \rho =_{\text{def}} \text{code } st_1 \rho; \text{code } st_2 \rho$

Sprünge

- $\text{code } (\text{goto } l) \rho =_{\text{def}} \text{ujp } l$

	Befehl	Bedeutung	Kommentar	Bedingung
unconditioned jump	ujp q	PC := q	unbedingter Sprung	$q \in [0, \text{codemax}]$
false jump	fjp q	if STORE[SP] = false then PC := q fi SP := SP-1	bedingter Sprung	(b), $q \in [0, \text{codemax}]$

Neu

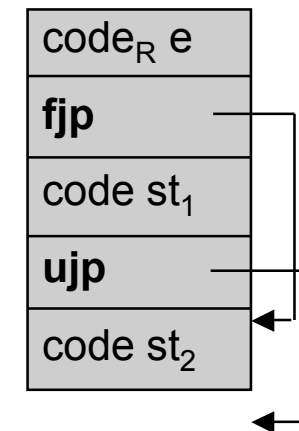
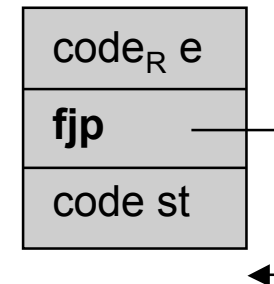
- Markierung von Stellen im Zielprogramm durch Namen („Marken“)
- Bedeutung solcher „Marken“: Adresse des Befehls einsetzen, der diese Marke trägt, z.B.
 - In separatem Lauf über erzeugten Code
 - Oder durch „back-patching“

Grundidee bei der Übersetzung anderer Kontrollanweisungen

- Rückführung (auf Quellsprachen-Niveau) auf Zuweisungen + (bedingte/unbedingte) Sprünge
- Umsetzung in P-Code

Beispiel

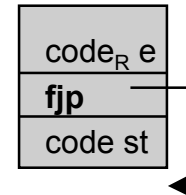
- „Einarmige“ bedingte Anweisung
if e then st fi →
if $\neg e$ then goto l fi; st; l: ...
- „Zweiarmige“ bedingte Anweisung
if e then st₁ else st₂ fi →
if $\neg e$ then goto l₁ fi; st₁; goto l₂; l₁: st₂; l₂: ...



Übersetzung von bedingten Anweisungen

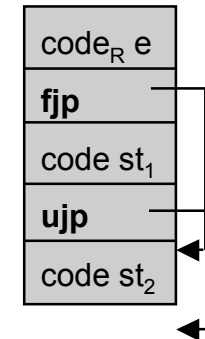
- „einarmige“

code (if e then st fi) $\rho =_{\text{def}}$
code_R e ρ ; **fjp** l; code st ρ ; l:



- „zweiarmige“

code (if e then st₁ else st₂ fi) $\rho =_{\text{def}}$
code_R e ρ ; **fjp** l₁; code st₁ ρ ; **ujp** l₂; l₁: code st₂ ρ ; l₂:



Beispiel

- geg.: $\rho(x) = 5, \rho(y) = 6, \rho(z) = 7,$
- code (if x > y then z := x else z := y fi) =
ldc a 5; ind i; ldc a 6; ind i; grt i;
fjp l₁;
ldc a 7; ldc a 5; ind i; sto i;
ujp l₂;
l₁: ldc a 7; ldc a 6; ind i; sto i;
l₂:...

Annahme:

Wie oben, d.h. x, y, z vom Typ integer

x > y

z := x

z := y

Fallunterscheidung (case-Anweisung)

- Ausgangsform der **case**-Anweisung
case e of 0: st₀; 1: st₁; ... k: st_k end

Vereinfachende Annahme:

- e liefert nur Selektoren zwischen 0 und k
- keine „Lücken“

- **1. Idee:** Rückführung auf bedingte Anweisungen

```
if e = 0 then st0; goto l fi;  
if e = 1 then st1; goto l fi; ...  
if e = k then stk; goto l fi;  
l: ...
```

- **Problem:** ineffizient wegen (eventueller) Mehrfachauswertung von e
- **2. Idee:** Einmalige Auswertung von e und „Vorausberechnung“ von l_e mit e ∈ {0, ..., k}

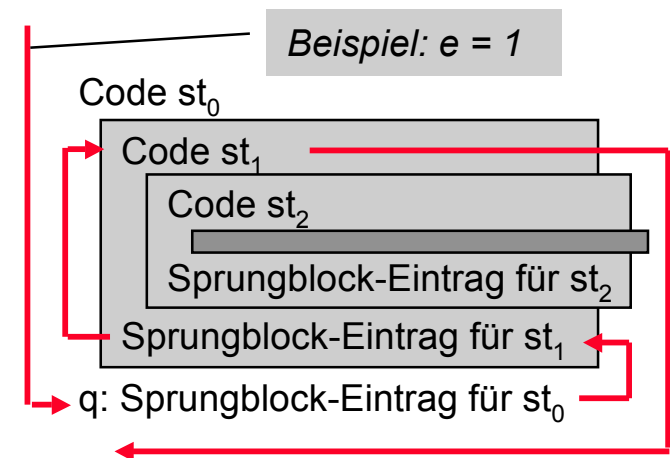
```
„goto le“;  
l0: st0; goto l;  
l1: st1; goto l; ...  
lk: stk; goto l;  
l: ...
```

- **Problem:** Wie „l_e“ berechnen?

„Berechnung“ von „ I_e “

- **Grundidee:** Neuer Maschinenbefehl (*indizierter Sprung* 🐼)
 - Ausdruck e auswerten (d.h. Wert = oberstes Kellerelement)
 - Sprungziel durch Wert des obersten Kellerelements bestimmt
- **Problem:** Direkte Sprünge auf einzelne Alternativen nicht möglich (da Anzahl der Alternativen und Länge der jeweiligen Anweisungen nicht bekannt!)
- **Idee:** „Indirekte“ Sprünge („Dreisprung“)
 - (unbedingter) Sprung auf (mit q markierten) *Sprungblock* 🐼 (hinter dem Code für die einzelnen Alternativen), (bedingter) „Rücksprung“ auf entsprechenden Sprungblockeintrag (in Abhängigkeit von e), (unbedingter) Sprung auf Alternative (gemäß Sprungblockeintrag)
 - Abarbeitung der entsprechenden Alternative
 - (unbedingter) Sprung hinter den Sprungblock
- Dabei
 - „Umgekehrte“ Reihenfolge der Sprünge im Sprungblock
 - D.h. zwischen dem Code für die Alternative st_i und deren zugehörigem Eintrag im Sprungblock liegt die Übersetzung des „Rests“ der **case**-Anweisung

Genauer:
Erst am Ende der
case-Anweisung
bekannt



Code für Fallunterscheidung

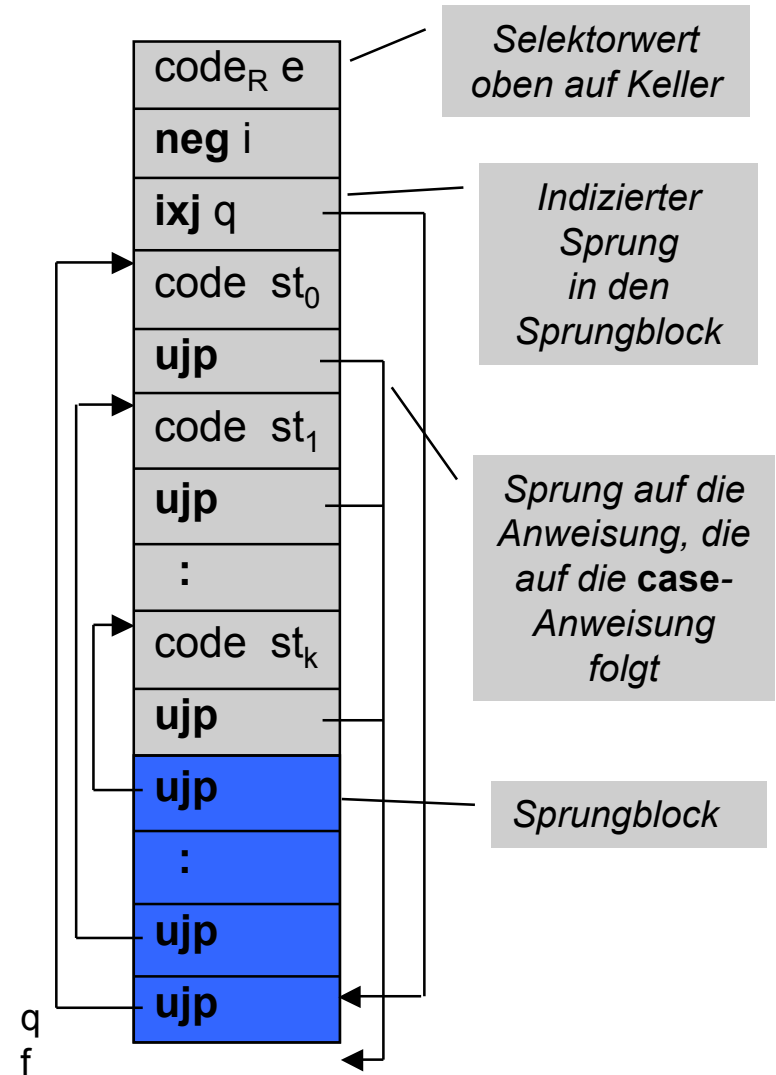
- code (case e of 0: st₀; 1: st₁; ... k: st_k end) ρ =_{def}
 code_R e ρ; neg i; ixj q;
 l₀: code st₀ ρ; ujp f;
 l₁: code st₁ ρ; ujp f; ...
 l_k: code st_k ρ; ujp f;
 ujp l_k; ...
 ujp l₁;
 q: ujp l₀;
 f: ...

Schönheitsfehler: „...“
Vollständige Form: rekursiv

Indizierter Sprung

Befehl	Bedeutung	Bedingung	Ergebnis
ixj q	PC := STORE[SP] + q; SP := SP-1	(i)	

indexed jump





Beispiel

- Geg.:

- $\rho(x) = 5, \rho(y) = 6, \rho(z) = 7,$
- code (**case** $x - y$ **of**
0: $z := z + 1;$
1: $z := x;$
2: $z := y$ **end**)

- Erzeugter Code

**ldc a 5; ind i; ldc a 6; ind i; sub i;
neg i; ixj q;**

l_0 : ldc a 7; ldc a 7; ind i; ldc i 1; add i; sto i; ujp f;

l_1 : ldc a 7; ldc a 5; ind i; sto i; ujp f;

l_2 : ldc a 7; ldc a 6; ind i; sto i; ujp f;

ujp l_2 ;

ujp l_1 ;

q: ujp l_0 ;

f: ...

$\text{code}_R e \rho$

Einzelne Fälle

Sprungblock

Rückführung auf Quellsprachen-Niveau

- **while-Schleife**

while *e* **do** *st* **od** $\rightarrow l_1$: **if** *e* **then** *st*; **goto** *l*₁ **fi** $\rightarrow l_1$: **if** $\neg e$ **then** **goto** *l*₂ **fi**; *st*; **goto** *l*₁; *l*₂: ...

- **code** (**while** *e* **do** *st* **od**) $\rho =_{\text{def}}$ *l*₁: **code**_R *e* ρ ; **fjp** *l*₂; **code** *st* ρ ; **ujp** *l*₁; *l*₂:

- **repeat-until-Schleife**

repeat *st* **until** *e* $\rightarrow l$: *st*; **if** $\neg e$ **then** **goto** *l* **fi**

- **code** (**repeat** *st* **until** *e*) $\rho =_{\text{def}}$ *l*: **code** *st* ρ ; **code**_R *e* ρ ; **fjp** *l*;

Beispiel

- geg.: $\rho(x) = 5, \rho(y) = 6, \rho(z) = 7,$

- **code** (**while** *x* > *y* **do** *z* := *z*+1; *x* := *x* – *y* **od**) =

*l*₁: **ldc** *a* 5; **ind** *i*; **ldc** *a* 6; **ind** *i*; **grt** *i*;

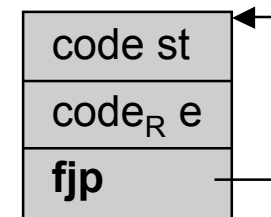
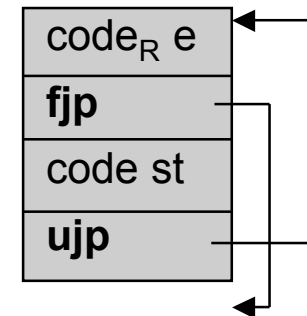
fjp *l*₂:

ldc *a* 7; **ldc** *a* 7; **ind** *i*; **ldc** *i* 1; **add** *i*; **sto** *i*;

ldc *a* 5; **ldc** *a* 5; **ind** *i*; **ldc** *a* 6; **ind** *i*; **sub** *i*; **sto** *i*;

ujp *l*₁;

*l*₂:...



Begriffe

- **Übersetzungszeit**
Zeitpunkt der Übersetzung
- **Laufzeit**
Zeitpunkt der Ausführung des (übersetzten) Programms
- **Statische Information**
zur Übersetzungszeit ersichtlich/berechenbar (z.B. Zieladressen von Sprüngen)
- **Dynamische Information**
erst zur Laufzeit mit Eingabedaten verfügbar
(z.B. Werte von Variablen und Ausdrücken, Kontrollfluss in bedingten Anweisungen)

Vereinfachende Annahme

- 1 Speicherzelle pro (einfache) Variable (d.h. Typ = integer, bool, char, ...)

Prinzip der Speicherzuteilung

- Konsekutiv, in der Reihenfolge des Auftretens im Deklarationsteil
- Wegen Prozeduren und Blöcken: Relativadressen (ab Zelle 5, siehe später)

Zuordnungsfunktion

- $\rho: \text{Id} \rightarrow (\text{Adr} \times \text{ST})$ bzw. (vorläufig)
 $\rho: \text{Id} \rightarrow \text{Adr}$

Adresszuordnung (für einfache Variablen)

- Geg.: **var** $n_1: t_1; n_2: t_2; \dots n_m: t_m;$
wobei t_i = einfacher Typ
- Relativadressen
 $\rho(n_i) =_{\text{def}} 5 + (i-1) \quad \text{für } 1 \leq i \leq m$

Problem

- Linearer Speicher
- Mehrdimensionale Felder müssen linear abgelegt werden

Möglichkeiten

- Zeilenweise
- Spaltenweise

Beispiel

- Geg.: **var a: array [-5..5, 1..9] of integer**
- Zeilenweise Ablage

~~a[-5, 1], a[-5, 2], ..., a[-5, 9],~~

~~a[-4, 1], a[-4, 2], ..., a[-4, 9],~~

....

~~a[5, 1], a[5, 2], ..., a[5, 9],~~

Hier: 2. Index variiert am häufigsten
Allgemein: letzter Index variiert am häufigsten

Allgemeiner Fall

- Geg.: **var b: array** $[u_1..o_1, \dots, u_k..o_k]$ **of integer**

- Zeilenweise Ablage

$b[u_1, \dots, u_{k-1}, u_k], \quad b[u_1, \dots, u_{k-1}, u_k+1], \dots, \quad b[u_1, \dots, u_{k-1}, o_k],$
 $b[u_1, \dots, u_{k-1}+1, u_k], \quad b[u_1, \dots, u_{k-1}+1, u_k+1], \dots, \quad b[u_1, \dots, u_{k-1}+1, o_k],$
...
 $b[o_1, \dots, o_{k-1}, u_k], \quad b[o_1, \dots, o_{k-1}, u_k+1], \dots, \quad b[o_1, \dots, o_{k-1}, o_k],$

Adresszuordnung

- Adresse des Feldes = Adresse der 1. Zelle
- Außerdem (für konsekutive Zuteilung): Größe des Feldes relevant

Berechnung der Größe

- $gr: \text{Typ} \rightarrow \mathbb{N}$ mit
- $gr(t) =_{\text{def}} 1$ für alle Variablen vom Typ integer, real, char, bool, Zeiger und vom Aufzählungs- und Mengentyp
- $gr(t) =_{\text{def}} \prod_{i=1..k} (o_i - u_i + 1)$ für statische Felder mit elementaren Komponenten

Adresszuordnung (allgemeiner Fall)

- Geg.: **var** $n_1: t_1; n_2: t_2; \dots n_m: t_m;$
- Relativadressen (für einfache Variablen und Felder) abhängig vom jeweiligen Typ
 $\rho(n_i) =_{\text{def}} 5 + \sum_{j=1..i-1} \text{gr}(t_j)$ für $1 \leq i \leq m$

d.h.
 $\rho(n_1) = 5,$
 $\rho(n_2) = 5 + \text{gr}(t_1),$
 $\rho(n_3) = 5 + \text{gr}(t_1) + \text{gr}(t_2), \dots$

Beachte

- Alle zur Adresszuordnung notwendige Information ist statisch

Problem

- Zugriff auf Feldkomponenten, deren Indices Variablen oder Ausdrücke (über Variablen) sind
- Zu lösende Aufgabe: Beim Übersetzen Befehle erzeugen, wie aktuelle Indices aus aktuellen Variablenwerten und Feldanfangsadresse zu berechnen sind

Beispiel

- Geg.: **var a: array [-5..5, 1..9] of integer**
- Zeilenweise Ablage

$\rho(a) \rightarrow a[-5, 1], a[-5, 2], \dots, a[-5, 9],$
 $a[-4, 1], a[-4, 2], \dots, a[-4, 9],$
 $\dots$
 $a[5, 1], a[5, 2], \dots, a[5, 9]$

- Beispiel: Adresse für $a[-4, 2]$:

$$\rho(a) + (\text{„Zeilennummer“} - 1) \times \text{Zeilenlänge} + (\text{„Spaltennummer“} - 1) =$$

$$\rho(a) + 1 \times 9 + 1 =$$

$$\rho(a) + (-4 - (-5)) \times 9 + (2 - 1)$$

„Zeilennummer“-1 bzw. „Spaltennummer“-1 ergibt sich stets aus der Differenz des jeweiligen aktuellen Index und dessen zugehöriger unterer Grenze

- Allgemeiner Fall: Adresse für $a[i, j]$:

$$\rho(a) + (i - (-5)) \times 9 + (j - 1) =$$

$$\rho(a) + (i - u_1) \times \text{gr}(\text{array}[1..9] \text{ of integer}) + (j - u_2)$$

Idee für den allgemeinen Fall

Allgemeiner Fall (beliebige Dimension, elementare Feldkomponenten)

- Geg.: **var b: array** $[u_1..o_1, \dots, u_k..o_k]$ **of integer**

- Adresse für $b[i_1, i_2, \dots, i_k]$: $\rho(b) + r$ wobei

$$r =_{\text{def}} (i_1 - u_1) \times \text{gr}(\text{array } [u_2..o_2, \dots, u_k..o_k] \text{ of integer}) + \\ (i_2 - u_2) \times \text{gr}(\text{array } [u_3..o_3, \dots, u_k..o_k] \text{ of integer}) + \dots + \\ (i_{k-1} - u_{k-1}) \times \text{gr}(\text{array } [u_k..o_k] \text{ of integer}) + (i_k - u_k)$$

- gr-Werte statisch über **Spannen** (= Anzahl Elemente pro Dimension) bestimmbar:

$$d_i =_{\text{def}} o_i - u_i + 1 \quad (1 \leq i \leq k)$$

$$r =_{\text{def}} (i_1 - u_1) \times d_2 \times d_3 \times \dots \times d_k + (i_2 - u_2) \times d_3 \times d_4 \times \dots \times d_k + \dots + (i_{k-1} - u_{k-1}) \times d_k + (i_k - u_k) \quad u_k)$$

Vereinfachungen

- Ausmultiplizieren und (nach i- und u-Ausdrücken) aufspalten

$$r = \frac{i_1 \times d_2 \times d_3 \times \dots \times d_k + i_2 \times d_3 \times d_4 \times \dots \times d_k + \dots + i_{k-1} \times d_k + i_k - (u_1 \times d_2 \times d_3 \times \dots \times d_k + u_2 \times d_3 \times d_4 \times \dots \times d_k + \dots + u_{k-1} \times d_k + u_k)}{}$$

dynamisch

statisch

- Zusammenfassen

$$r =_{\text{def}} h - d \text{ wobei}$$

$$h =_{\text{def}} (i_1 \times D_2 + i_2 \times D_3 + \dots + i_{k-1} \times D_k + i_k) = \sum_{j=1..k} i_j \times D_{j+1} \text{ mit } D_j = \prod_{l=j..k} d_l$$

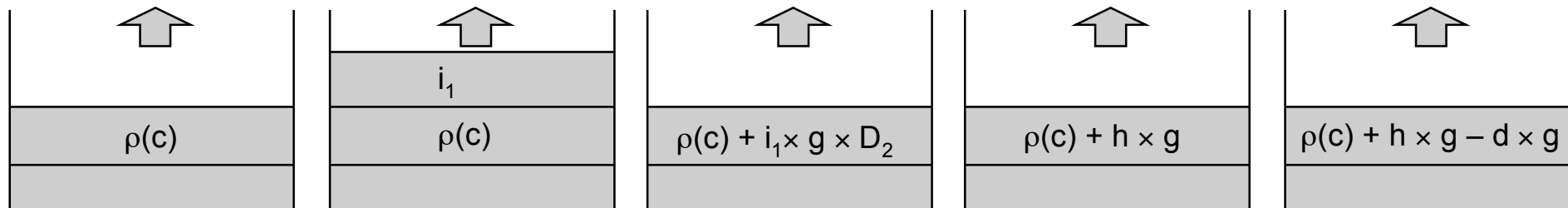
$$d =_{\text{def}} (u_1 \times D_2 + u_2 \times D_3 + \dots + u_{k-1} \times D_k + u_k) = \sum_{j=1..k} u_j \times D_{j+1}$$

Allgemeiner Fall (beliebige Dimension, beliebige Feldkomponenten)

- Adresse für **var c: array** $[u_1..o_1, \dots, u_k..o_k]$ **of** t :
 $r =_{\text{def}} h \times \text{gr}(t) - d \times \text{gr}(t)$ (mit h und d wie oben)

Feldzugriff

- Berechnung der Adresse der Feldkomponente $c[i_1, \dots, i_k]$
bei Anfangsadresse $\rho(c)$ und Feldkomponenten vom Typ T mit $g =_{\text{def}} \text{gr}(T)$:
- $\text{code}_R c[i_1, \dots, i_k] \rho =_{\text{def}} \text{code}_L c[i_1, \dots, i_k] \rho$; **ind** T
- $\text{code}_L c[i_1, \dots, i_k] \rho =_{\text{def}} \text{ldc } a \ \rho(c); \text{code}_l [i_1, \dots, i_k] g \rho$ D_i aus ρ oder global
- $\text{code}_l [i_1, \dots, i_k] g \rho =_{\text{def}}$
 $\text{code}_R i_1 \rho; \text{ixa } g \times D_2; \text{code}_R i_2 \rho; \text{ixa } g \times D_3; \dots \text{code}_R i_k \rho; \text{ixa } g; \text{dec } a \ d \times g; \quad ;$ neuer Befehl, s.u.



Indizierter Zugriff

	Befehl	Bedeutung	Bedingung	Ergebnis
indexed access	ixa q	$\text{STORE}[\text{SP}-1] := \text{STORE}[\text{SP}-1] + \text{STORE}[\text{SP}] \times q;$ $\text{SP} := \text{SP}-1$	(a, i) und $\text{Typ}(q) = i$	(a)

Inkrementieren und Dekrementieren

	Befehl	Bedeutung	Bedingung	Ergebnis
increment	inc T q	$\text{STORE}[\text{SP}] := \text{STORE}[\text{SP}] + q;$	(T) und $\text{Typ}(q) = i$	(T)
decrement	dec T q	$\text{STORE}[\text{SP}] := \text{STORE}[\text{SP}] - q;$	(T) und $\text{Typ}(q) = i$	(T)

Definiert für alle Typen T, auf denen Nachfolger und Vorgänger durch integer-Addition und -Subtraktion definiert sind

Weiteres Problem

- Einhaltung der Indexbereiche
- Lösung: verbessertes Übersetzungsschema

$\text{code}_{lc} [i_1, \dots, i_k] (g; u_1, o_1, \dots, u_n, o_n) \rho =_{\text{def}}$
 $\text{code}_R i_1 \rho; \mathbf{chk} u_1 o_1; \mathbf{ixa} g \times D_2;$
 $\text{code}_R i_2 \rho; \mathbf{chk} u_2 o_2; \mathbf{ixa} g \times D_3; \dots$
 $\text{code}_R i_k \rho; \mathbf{chk} u_k o_k; \mathbf{ixa} g;$
 $\mathbf{dec} a g \times d;$

	Befehl	Bedeutung	Bedingung	Ergebnis
check	chk p q	if (STORE[SP] < p) or (STORE[SP] > q)	(i) und	
		then error(„value out of range“) fi	Typ(p, q) = i	



Beispiel

- geg.: **var** i, j: **integer**; b: **array** [-5..5, 1..9] **of integer**

- dann

- $\rho(i) = 5, \rho(j) = 6, \rho(b) = 7,$

- $d_1 = 11, d_2 = 9$

$$d_i = o_i - u_i + 1$$

- $g = 1$ und $d = -44$

$$d = u_1 \times d_2 + u_2$$

- code (b[i+1, j] := 0) $\rho =$

ldc a 7;

ldc a 5; **ind** i; **ldc** i 1; **add** i; **chk** -5 5; **ixa** 9;

$$D_2 = \prod_{j=2..2} d_j = d_2 = 9$$

ldc a 6; **ind** i; **chk** 1 9; **ixa** 1;

dec a -44;

ldc i 0; **sto** i

Dynamisches Feld

- Ein Feld c mit **var** c : **array** $[u_1..o_1, \dots, u_k..o_k]$ **of** T ist **dynamisch**, wenn nicht alle u_i, o_i konstant sind (sondern Variablen oder formale Parameter)
- Es sind
 - Dynamisch: Grenzen, Spannen, und damit Größe
 - Statisch: Dimension k

Problem

- Anfangsadresse i.a. auch dynamisch

Idee

- Statisch festlegen, **wo** Anfangsadresse zur Laufzeit abgelegt wird

Dazu

- Felddeskriptor** mit
 - Statischer Anfangsadresse
 - Statischer Größe (hängt nur von der Dimension des Feldes ab)

0	Fiktive Anfangsadresse: a
1	Feldgröße: i
2	Subtr. für fiktive AA: i
3	$u_1: i$
4	$o_1: i$
	:
$2k+1$	$u_k: i$
$2k+2$	$o_k: i$
$2k+3$	$d_2: i$
	:
$3k+1$	$d_k: i$

Situation

- Geg.: Dynamisches Feld **var** b: **array** [$u_1..o_1, \dots, u_k..o_k$] **of integer**
- Ges.: Befehlsfolge für Adressierung von $b[i_1, i_2, \dots, i_k]$

Offensichtlich

- In Formel (zur Bestimmung der Adresse) $\rho(b) + r$
mit

$$\begin{aligned} r &=_{\text{def}} h \times g - d \times g \\ &= i_1 \times d_2 \times d_3 \times \dots \times d_k + i_2 \times d_3 \times d_4 \times \dots \times d_k + \dots + i_{k-1} \times d_k + i_k - \\ &\quad (u_1 \times d_2 \times d_3 \times \dots \times d_k + u_2 \times d_3 \times d_4 \times \dots \times d_k + \dots + u_{k-1} \times d_k + u_k) \end{aligned}$$

sind bis auf k alle Größen dynamisch

Unterschied (bezüglich „dynamisch“)

- $i_1, i_2, \dots, i_k$ von der jeweiligen Indizierung abhängig (d.h. bei jedem Zugriff anders)
- $u_1, u_2, \dots, u_k$ und $d_2, d_3, \dots, d_k$ ergeben sich **einmal** für die gesamte Lebensdauer
 - Formaler Parameter: bei Parameterübergabe
 - Deklariert: bei Anlage des Feldes

Daher

- d hat bei allen Indizierungen denselben Wert und wird nur einmal berechnet (bei Parameterübergabe bzw. Anlage des Feldes)
- dito für g
- Somit
 - $d \times g$ einmal berechnen und von Anfangsadresse AA abziehen („fiktive Anfangsadresse“)
 - Für jede Indizierung nur noch $h \times g$ berechnen und aufaddieren

Damit

- Adressierung von $b[i_1, i_2, \dots, i_k]$: Fiktive Anfangsadresse + $h \times g$
- Effiziente Berechnung von h mit *Horner-Schema*:
$$h = (\dots((i_1 \times d_2 + i_2) \times d_3 + i_3) \times d_4 + \dots) \times d_k + i_k$$
- Ebenfalls aus Effizienzgründen:
Abspeicherung der d_i im Deskriptor

Codeschema für den Zugriff in dynamischen Feldern

- $\text{code}_{\text{Rd}} b[i_1, \dots, i_k] \rho =_{\text{def}} \text{code}_{\text{Ld}} b[i_1, \dots, i_k] \rho; \text{ind } T$
- $\text{code}_{\text{Ld}} b[i_1, \dots, i_k] \rho =_{\text{def}} \text{ldc } a \rho(b); \text{code}_{\text{ld}} [i_1, \dots, i_k] g \rho$
- $\text{code}_{\text{ld}} [i_1, \dots, i_k] g \rho =_{\text{def}}$

dpl a; ind a; ldc i 0;

code_R i₁ ρ; add i; ldd 2k+3; mul i;

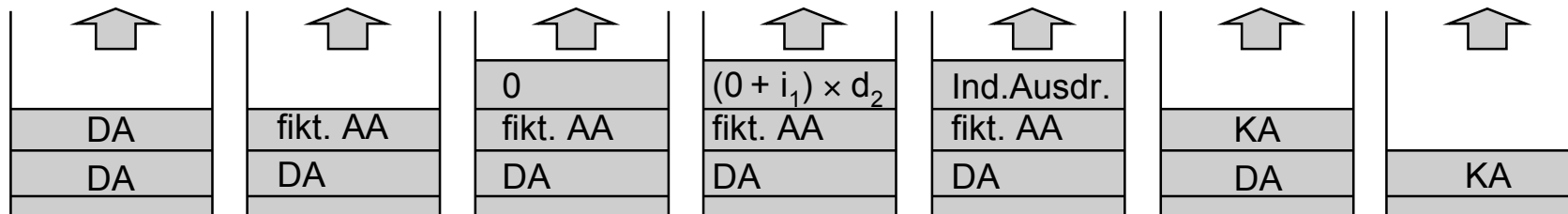
code_R i₂ ρ; add i; ldd 2k+4; mul i; ...

code_R i_{k-1} ρ; add i; ldd 3k+1; mul i;

code_R i_k ρ; add i; ixa g; sli a

*Variante mit Bereichsprüfung
analog zu oben; Zugriff auf
Feldgrenzen indirekt über
Deskriptor (mit ldd)*

0	Fiktive Anfangsadr: a
1	Feldgröße: i
2	Subtr. für fiktive AA: i
3	u ₁ : i
4	o ₁ : i
	:
2k+1	u _k : i
2k+2	o _k : i
2k+3	d ₂ : i
	:
3k+1	d _k : i



Befehl	Bedeutung	Bedingung	Ergebnis
dpl T	$SP := SP+1; \text{STORE}[SP] := \text{STORE}[SP-1]$	(T)	(T, T)
ldd q	$SP := SP+1; \text{STORE}[SP] := \text{STORE}[\text{STORE}[SP-3]+q]$	(a, T ₁ , T ₂)	(a, T ₁ , T ₂ , i)
sli T₂	$\text{STORE}[SP-1] := \text{STORE}[SP]; SP := SP-1;$	(T ₁ , T ₂)	(T ₂)

Situation

- Geg.: **var v: record a: integer; b: bool end**
- Vereinfachende Annahme: Komponentenbezeichner (global) eindeutig

Speicherzuordnung

- v: Adresse der ersten freien Zelle (= nächstmöglicher Speicherplatz)
- a, b: Relativadressen bezüglich v

Beispiel

- Geg.: **var i, j: integer; var v: record a: integer; b: bool end**
- Speicherzuordnung
 $\rho(i) = 5, \rho(j) = 6, \rho(v) = 7, \rho(a) = 0, \rho(b) = 1$

Allgemein

- Geg.: **var** $v: t$ mit **type** $t = \text{record } c_1: t_1; c_2: t_2; \dots c_k: t_k \text{ end}$

- Speicherzuordnung

$$\rho(c_i) =_{\text{def}} \sum_{j=1..i-1} \text{gr}(t_j)$$

- Größe der Verbundvariablen

$$\text{gr}(t) =_{\text{def}} \sum_{j=1..k} \text{gr}(t_j)$$

d.h.

$$\rho(c_1) = 0$$

$$\rho(c_2) = \text{gr}(t_1)$$

$$\rho(c_3) = \text{gr}(t_1) + \text{gr}(t_2)$$

Beachte

- Größe der Verbundvariablen ist statisch; damit: Felder von Verbunden möglich
- Dynamische Felder als Verbundkomponenten: im Verbund nur Deskriptor ablegen

Adressierung von Verbundkomponenten

- code $v.c_i \rho =_{\text{def}} \text{ldc } a \rho(v); \text{inc } a \rho(c_i)$

Adresse von v

Relativadresse der Komponente

Bisher

- Speicherbelegung für deklarierte Objekte (wobei Deklaration = Abb.: Namen → Adressen)

Zeiger

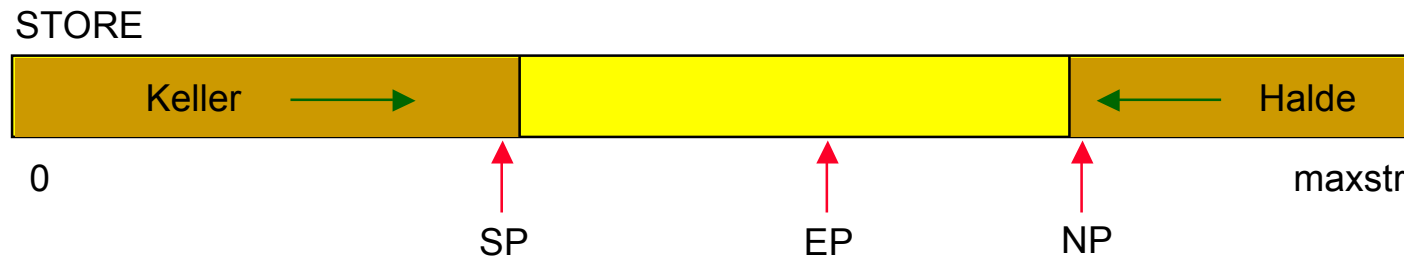
- „Anonyme Bezeichnungen“ für Objekte (Verweise auf Speicherzellen)
- Verwendung in dynamisch sich verändernden verketteten Strukturen
- Kreation solcher Objekte nicht durch Deklaration, sondern durch spezielle Anweisungen (z.B. **new**)
- Zeigervariable: (deklarierte) Variable vom Typ Zeiger

Unterschied

- Durch Deklaration kreierte Objekte (Variable):
Speicherfreigabe nach Ablauf der Lebensdauer ⇒ kellerartige Speicherverwaltung
- Dynamisch kreierte Objekte („Zeiger-Objekte“):
Speicherfreigabe, wenn Zugriff nicht mehr möglich (*garbage collection*)
⇒ i.a. **nicht** kellerartig

Ablage von Zeiger-Objekten

- Eigener Bereich („Halde“) für Zeiger-Objekte am oberen Ende des Datenspeichers



Neue Register

- NP new pointer „unterste“ belegte Zelle der Halde
- EP extreme stack pointer größtmöglicher Wert von SP

Bei Auswertung von Ausdrücken im Anweisungsteil zur Übersetzungszeit berechenbar

Kreation eines neuen Objekts auf der Halde

- new**-Befehl

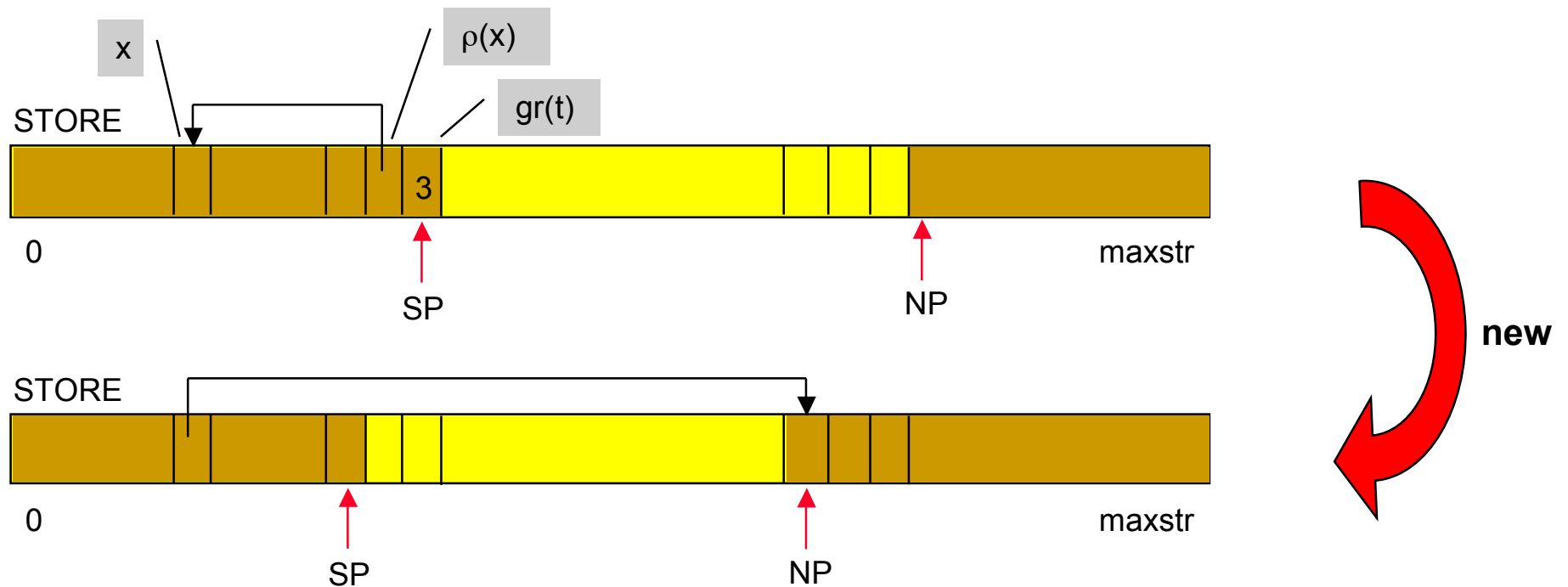
Größe des Haldenobjekts

Adresse des Haldenobjekts

Befehl	Bedeutung	Bedingung	Ergebnis
new	if $NP - \text{STORE}[SP] \leq EP$ then error(„store overflow“) else $NP := NP - \text{STORE}[SP]; \text{STORE}[\text{STORE}[SP-1]] := NP;$ $SP := SP - 2$ fi	(a,i)	

Codeschema für new-Anweisung

- code **new**(x) $\rho =_{\text{def}}$ **ldc a** $\rho(x)$; **ldc i** $\text{gr}(t)$; **new** falls x Variable vom Typ $\uparrow t$



Adressierung eines anonymen Objekts (Dereferenzierung)

- code **x** $\uparrow \rho =_{\text{def}}$ **ldc a** $\rho(x)$; **ind a**

Adressberechnung für beliebig kompliziert aufgebaute Bezeichnungen

- $\text{code}_R x \rho =_{\text{def}} \text{code}_L x \rho; \mathbf{ind}$
- $\text{code}_L (xr) \rho =_{\text{def}} \mathbf{ldc} a \rho(x); \text{code}_M (r) \rho$ für Namen x
- $\text{code}_M (.xr) \rho =_{\text{def}} \mathbf{inc} a \rho(x); \text{code}_M (r) \rho$ für Namen x
- $\text{code}_M (\uparrow r) \rho =_{\text{def}} \mathbf{ind} a; \text{code}_M (r) \rho$
- $\text{code}_M ([i]r) \rho =_{\text{def}} \text{code}_{ld} [i] g \rho; \text{code}_M (r) \rho$
falls g die Komponentengröße des indizierten Felds ist
- $\text{code}_M (\varepsilon) \rho =_{\text{def}} \varepsilon$

Invariante

- Falls $\text{code}_L (uv) \rho = b; \text{code}_M (v) \rho$, dann gilt zur Laufzeit:
Ausführung von b berechnet in oberste Kellerzelle
 - Adresse eines Felddeskriptors, wenn v mit Indizierung beginnt
 - Anfangsadresse der von u bezeichneten Variablen, sonst

Beispiel

- Geg.:

type t = record c: array [-5..5, 1..9] of integer; b: ↑t end;
var i, j: integer; pt: ↑t;

- Voraussetzung

$\rho(i) = 5$, $\rho(j) = 6$, $\rho(pt) = 7$

- Übersetzung von **pt↑.b↑.c[i+1, j]**

ldc a 7;	Lade Adresse von pt
ind a;	Lade Anfangsadresse von Verbund
inc a 99;	Berechne Anfangsadresse von Verbundkomponente b
ind a;	Dereferenziere Zeiger
inc a 0;	Anfangsadresse von Komponente c

$\text{code}_{ld} [i+1, j] \ 1 \ \rho$

Komponentengröße

d.h.
Größe c = 99
Relativadresse c = 0
Relativadresse b = 99