Übersetzung imperativer Sprachen – Teil 1

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

Grundlagen (1/2)

Wesentliche Zielsetzung

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

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

Dabei zu klären

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

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

- Als *rekursive Funktion*, unter Verwendung geeigneter *Hilfsfunktionen*

  $\ddot{U}_L(p_L) =_{def}$ code $(p_L, \varnothing, 0)$ (bzw. in klammerfreier Schreibweise: code $p_L \varnothing 0$) mit

  code: L × (Id → (Adr × ST)) × ST → M

  Id → (Adr × ST): *Adresszuordnung*
  ST: *Schachtelungstiefe*

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

Grundlagen (2/2)

Warum „rekursive Funktion“?

- Nähe zur semantischen Definition („denotationelle Semantik“)
- Rekursionsmechanismus sorgt (automatisch) für die Bereitstellung der relevanten Kontextinformation (z.B. korrekte Adresszuordnung für Variablen im Rahmen von Blockschachtelung)

Orientiert an Syntax:
Anw. ::= if-Anw. | ...
If-Anw. ::= **if** ... **then** Anw.
else Anw. **fi**

Warum „Hilfsfunktionen“?

- Für viele L-Konstrukte werden zusätzliche Informationen (z.B. *Adresszuordnung* ☛, *Schachtelungstiefe* ☛) aus dem jeweiligen Kontext zur korrekten Übersetzung benötigt

Zunächst verwendet

- Vereinfachte Form der Funktion code

  code: L × (Id → Adr) → M

Parameter ST bleibt vorläufig konstant

- Funktion code übersetzt Folgen (von L-Konstrukten) elementweise in Folgen (von M-Konstrukten):

  code () ρ $=_{def}$ ()
  code (st_1; sts) ρ $=_{def}$ code st_1 ρ; code sts ρ

Konstrukte der Quellsprache

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
- Konstante
- Ausdrücke
- Anweisungen
  - Zuweisung
  - Kontrollanweisungen
    - Sprünge (**goto**)
    - Bedingte (**if** - **fi**, **case**)
    - Iterierte (**while**, **repeat**, **for**)
- Strukturierungsmöglichkeiten
  - Prozeduren
  - Blöcke

Die P-Maschine

P-Maschine

„Pascal-Maschine“

- Abstrakte (oder virtuelle) Maschine (die von 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

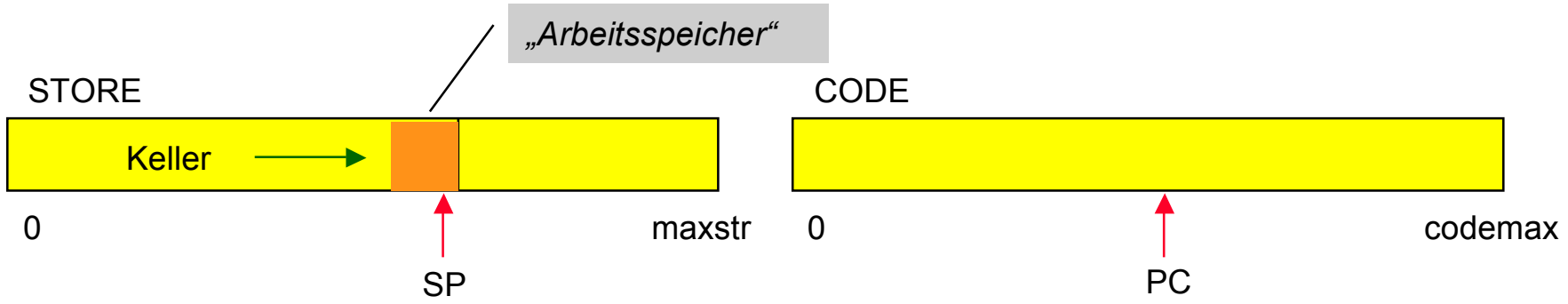
Architektur der P-Maschine

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

zunächst benutzt

- *Verarbeitungswerk*, das die einzelnen Zellen des Programmspeichers als Maschinenbefehle interpretiert und ausführt



Verarbeitungsprinzip

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
  - Modifikatoren, Adressen: natürliche Zahlen (z.T. eingeschränkt, z.B. Adressen ≤ maxstr)

Befehle ohne Adressangabe wirken auf die obersten Kellerzellen

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

Transportbefehle

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
ldo T q	SP := SP+1; STORE[SP] := STORE[q]	q ∈ [0..maxstr]	(T)
ldc T q	SP := SP+1; STORE[SP] := q	Typ(q) = T	(T)
ind T	STORE[SP] := STORE[STORE[SP]]	(a)	(T)
sro T q	STORE[q] := STORE[SP] SP := SP–1;	(T) q ∈ [0..maxstr]	
sto T	STORE[STORE[SP–1]] := STORE[SP] SP := SP–2;	(a, T)	

load **o**bject

load **c**onstant

load **ind**irectly

store object

store

Typ von STORE[SP-1]

Typ von STORE[SP]

Befehle für Ausdrücke

Auswertung von Ausdrücken (generelle Idee)

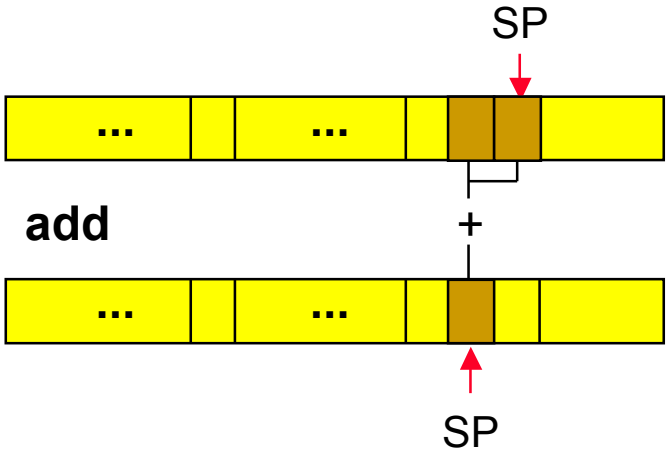
- Operanden (oben) auf den Keller legen
- Auswertung über den obersten Kellerzellen (Anzahl durch jeweilige Operation festgelegt)

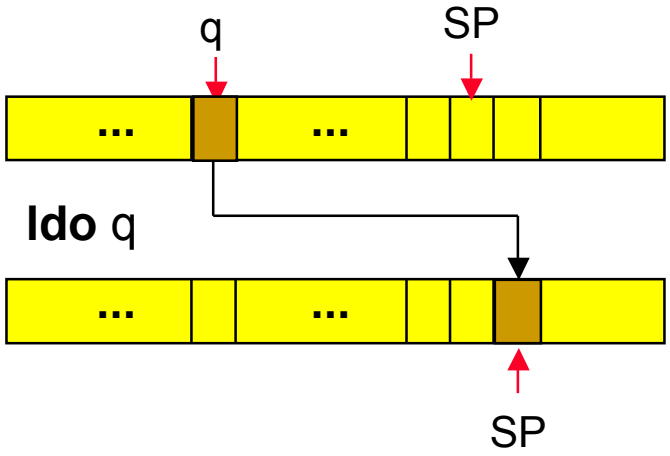
Befehle für Ausdrücke

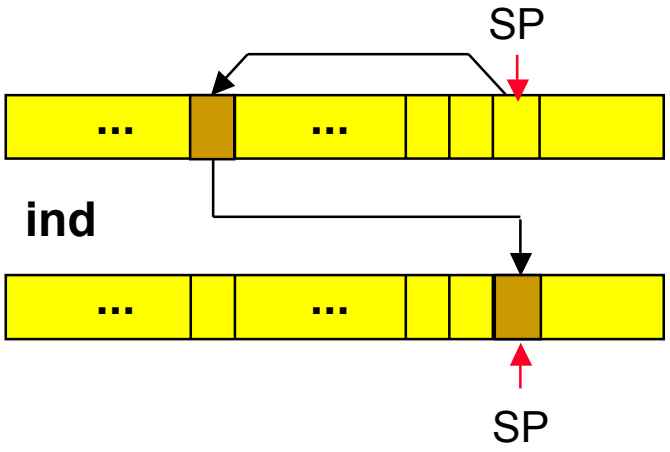
Befehl	Bedeutung	Bedingung	Ergebnis
add N	STORE[SP–1] := STORE[SP–1] $+_N$ STORE[SP]; SP := SP–1;	(N, N)	(N)
sub N	STORE[SP–1] := STORE[SP–1] $-_N$ STORE[SP]; SP := SP–1;	(N, N)	(N)
mul N	STORE[SP–1] := STORE[SP–1] $\times_N$ STORE[SP]; SP := SP–1;	(N, N)	(N)
div N	STORE[SP–1] := STORE[SP–1] $/_N$ STORE[SP];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

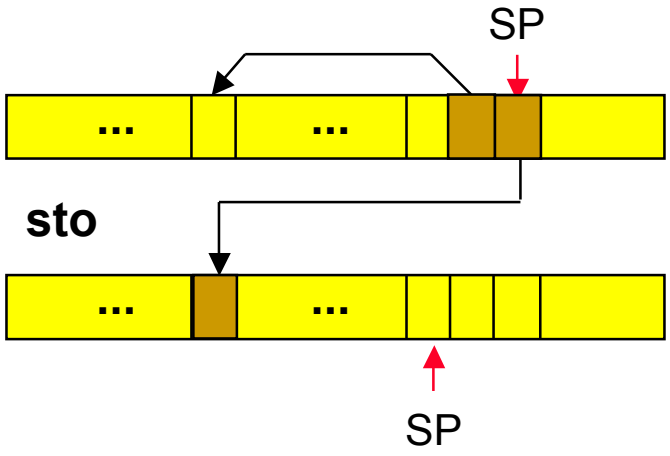
Wirkung von Befehlen

Wirkung einiger Befehle









Wertzuweisung (1/3)

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

- code (x := y) ρ =$_{def}$ code$_L$ x ρ; code$_R$ y ρ; **sto** T
- wobei

  code$_L$ x ρ =$_{def}$ **ldc** a ρ(x)

  code$_R$ y ρ =$_{def}$ **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**

Wertzuweisung (2/3)

Übersetzung von Ausdrücken und Wertzuweisung

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

Einfache Variable, Feldkomponente, Verbund-komponente

Wertzuweisung (3/3)

Beispiel

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

Annahme:
x, y, z *vom Typ integer*

- code $(x := (y + (y \times z)))\ \rho$ =

$code_L\ x\ \rho$; $code_R\ (y + (y \times z))\ \rho$; **sto** i =

ldc a 5; $code_R\ (y + (y \times z))\ \rho$; **sto** i =

ldc a 5; $code_R\ (y)\ \rho$; $code_R(y \times z)\ \rho$; **add** i; **sto** i =

ldc a 5; **ldc** a 6; **ind** i; $code_R(y \times z)\ \rho$; **add** i; **sto** i =

ldc a 5; **ldc** a 6; **ind** i; $code_R(y)\ \rho$; $code_R(z)\ \rho$; **mul** i; **add** i; **sto** i =

ldc a 5; **ldc** a 6; **ind** i; **ldc** a 6; **ind** i; $code_R(z)\ \rho$; **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

Kontrollanweisungen (1/2)

Aufeinander folgende Anweisungen („Anweisungsfolgen“)

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

Sprünge

- code (**goto** l) ρ =$_{def}$ **ujp** l

unconditioned jum**p**

false **j**um**p**

Befehl	Bedeutung	Kommentar	Bedingung
ujp q	PC := q	unbedingter Sprung	q ∈ [0, codemax]
fjp q	**if** STORE[SP] = **false then** PC := q **fi** SP := SP–1	bedingter Sprung	(b), q ∈ [0, 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“

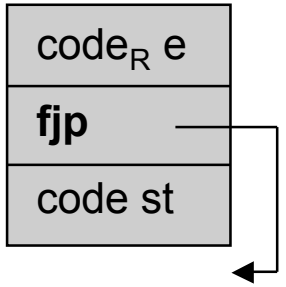
Kontrollanweisungen (2/2)

Grundidee bei der Übersetzung anderer Kontrollanweisungen

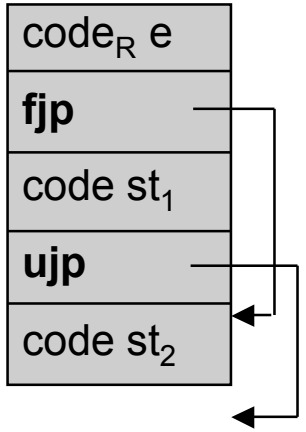
- Rückführung (auf Quellsprachen-Niveau) auf Zuweisungen + (bedingte/unbedingte) Sprünge
- Umsetzung in P-Code

Beispiel

- „Einarmige“ bedingte Anweisung
  if e **then** st **fi** ➔
  if ¬e **then goto** l **fi**; st; l: ...



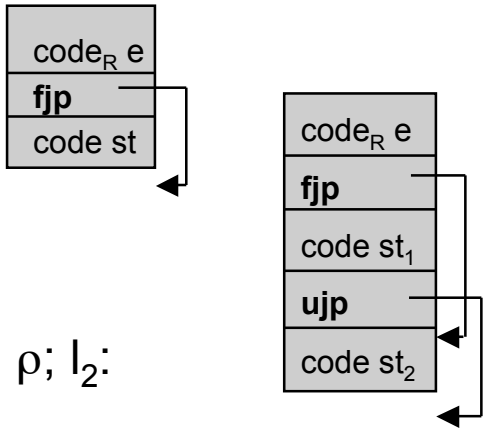
- „Zweiarmige“ bedingte Anweisung
  if e **then** st_1 **else** st_2 **fi** ➔
  if ¬e **then goto** l_1 **fi**; st_1; **goto** l_2; l_1: st_2; l_2: ...



Bedingte Anweisungen

Übersetzung von bedingten Anweisungen

- „einarmige“
  code (**if** e **then** st **fi**) ρ $=_{def}$
  $code_R$ e ρ; **fjp** l; code st ρ; l:
- „zweiarmige“
  code (**if** e **then** st_1 **else** st_2 **fi**) ρ $=_{def}$
  $code_R$ e ρ; **fjp** l_1; code st_1 ρ; **ujp** l_2; l_1: code st_2 ρ; l_2:



Beispiel

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

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

- code (**if** x > y **then** z := x **else** z := y **fi**) =
  ldc a 5; **ind** i; **ldc** a 6; **ind** i; **grt** i;
  fjp l_1;
  ldc a 7; **ldc** a 5; **ind** i; **sto** i;
  ujp l_2;
  l_1: **ldc** a 7; **ldc** a 6; **ind** i; **sto** i;
  l_2:...

x > y

z := x

z := y

Fallunterscheidung (1/4)

Fallunterscheidung (**case**-Anweisung)

- Ausgangsform der **case**-Anweisung
  case e **of** 0: st_0; 1: st_1; ... 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** st_0; **goto** l **fi**;
  if e = 1 **then** st_1; **goto** l **fi**; ...
  if e = k **then** st_k; **goto** l **fi**;
  l: ...

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

  „**goto** l_e“;
  l_0: st_0; **goto** l;
  l_1: st_1; **goto** l; ...
  l_k: st_k; **goto** l;
  l: ...

- **Problem**: Wie „ l_e“ berechnen?

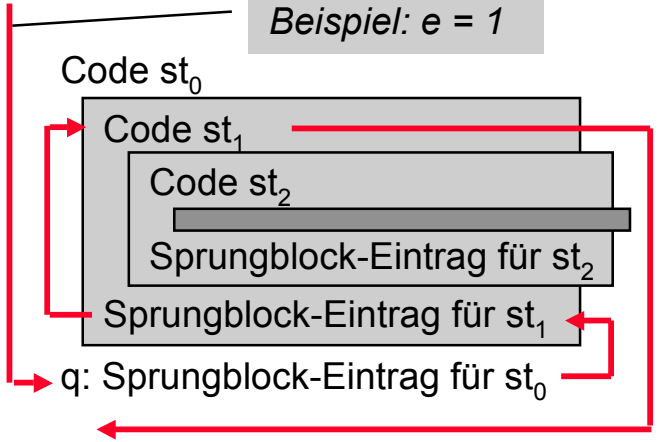
Fallunterscheidung (2/4)

„Berechnung“ von „l_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 („Viersprung“)
  - (unbedingter) Sprung auf (mit q markierten) *Sprungblock* ☛ (hinter dem Code für die Alternativen) ,
  - (bedingter) Sprung auf entsprechenden Sprungblockeintrag (in Abhängigkeit von e),
  - (unbedingter) Sprung auf Alternative (gemäß Sprungblockeintrag) und 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*

Beispiel: e = 1



Fallunterscheidung (3/4)

Code für Fallunterscheidung

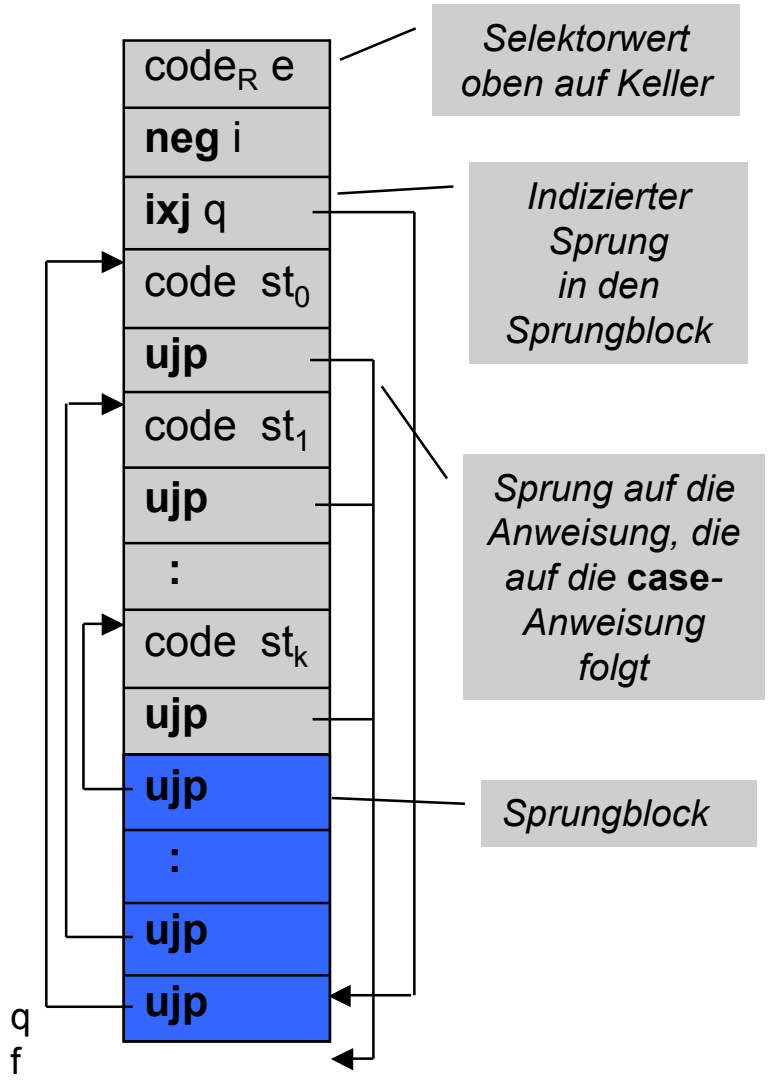
- code (**case** e **of** 0: st_0; 1: st_1; ... k: st_k **end**) ρ =$_{def}$
  $code_R$ e ρ; **neg** i; **ixj** q;
  l_0: code st_0 ρ; **ujp** f;
  l_1: code st_1 ρ; **ujp** f; ...
  l_k: code st_k ρ; **ujp** f;
  ujp l_k; ...
  ujp l_1;
  q: **ujp** l_0;
  f: ...

Schönheitsfehler: „...“
Vollständige Form: rekursiv

Indizierter Sprung

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

ixj: **i**ndex**e**d **j**ump



Fallunterscheidung (4/4)

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: ...

$code_R$ e ρ

Einzelne Fälle

Sprungblock

Iterierte Anweisungen

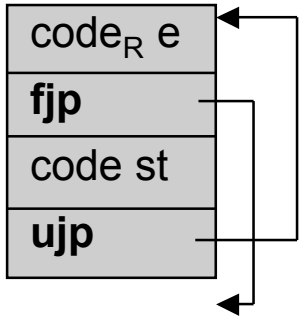
Rückführung auf Quellsprachen-Niveau

- **while**-Schleife

  while e **do** st **od** ➔ l_1: **if** e **then** st; **goto** l_1 **fi** ➔ l_1: **if** ¬e **then goto** l_2 **fi**; st; **goto** l_1; l_2: ...
- code (**while** e **do** st **od**) $\rho =_{def}$ l_1: $code_R$ e ρ; **fjp** l_2; code st ρ; **ujp** l_1; l_2:

- **repeat-until**-Schleife

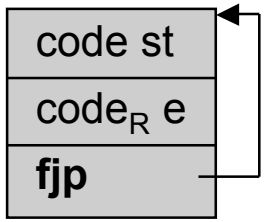
  repeat st **until** e ➔ l: st; **if** ¬e **then goto** l **fi**
- code (**repeat** st **until** e) $\rho =_{def}$ l: code st ρ; $code_R$ e ρ; **fjp** l;



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:...



Speicherbelegung für Variablen

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)

Speicherzuteilung für einfache Variablen

Vereinfachende Annahme

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

Prinzip der Speicherzuteilung

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

Zuordnungsfunktion

- ρ: Id $\rightarrow$ (Adr $\times$ ST) bzw. (vorläufig)
  ρ: Id $\rightarrow$ Adr

Adresszuordnung (für einfache Variablen)

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

Speicherbelegung für statische Felder

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

Zeilenweise Ablage

Allgemeiner Fall

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

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

Adresszuordnung

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

Berechnung der Größe

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

Allgemeine Adresszuordnung

Adresszuordnung (allgemeiner Fall)

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

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

Beachte

- Alle zur Adresszuordnung notwendige Information ist statisch

Zugriff auf Felder (1/4)

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
  - Beispiel: Adresse für a[-4,2]:

    $\rho(a)$ + („Zeilennummer“ - 1) × Zeilenlänge + („Spaltennummer“ - 1) =

    $\rho(a)$ + 1 × 9 + 1 =

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

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

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

    $\rho(a)$ + (i - u_1) × gr(**array**[1..9] **of integer**) + (j - u_2)

$\rho(a)$ → 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]

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

Idee für den allgemeinen Fall

Zugriff auf Felder (2/4)

Allgemeiner Fall (beliebige Dimension, elementare Feldkomponenten)

- Geg.: **var** b: **array** $[u_1..o_1, ..., u_k..o_k]$ **of integer**
- Adresse für $b[i_1, i_2, ..., i_k]$: $\rho(b) + r$ wobei

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

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

$$d_i =_{def} o_i - u_i + 1 \quad (1 \le i \le k)$$
$$r =_{def} (i_1 - u_1) \times d_2 \times d_3 \times ... \times d_k + (i_2 - u_2) \times d_3 \times d_4 \times ... \times d_k + ... + (i_{k-1} - u_{k-1}) \times d_k + (i_k - u_k) \qquad u_k)$$

Vereinfachungen

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

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

dynamisch

statisch

- Zusammenfassen

$$r =_{def} h - d \text{ wobei}$$
$$h =_{def} (i_1 \times D_2 + i_2 \times D_3 + ... + i_{k-1} \times D_k + i_k) = \Sigma_{j=1..k}\, i_j \times D_{j+1} \text{ mit } D_j = \Pi_{l=j..k}\, d_l$$
$$d =_{def} (u_1 \times D_2 + u_2 \times D_3 + ... + u_{k-1} \times D_k + u_k) = \Sigma_{j=1..k}\, u_j \times D_{j+1}$$

Zugriff auf Felder (3/4)

Allgemeiner Fall (beliebige Dimension, beliebige Feldkomponenten)

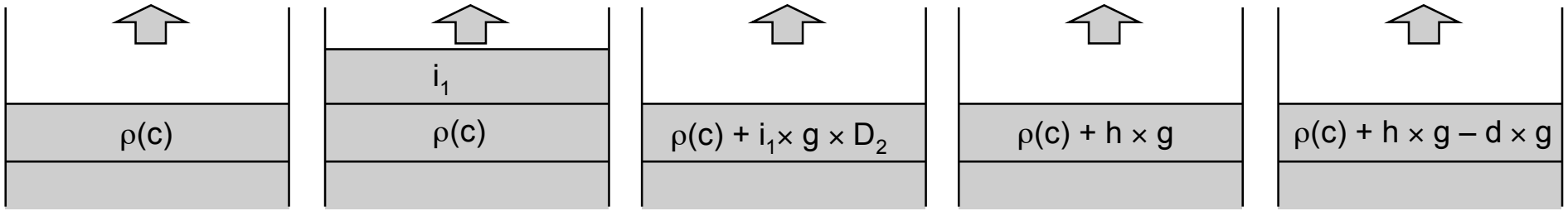
- Adresse für **var** c: **array** [$u_1..o_1$, ..., $u_k..o_k$] **of** t:
  $r =_{def} h \times gr(t) - d \times gr(t)$ (mit h und d wie oben)

Feldzugriff

- Berechnung der Adresse der Feldkomponente $c[i_1, ..., i_k]$
  bei Anfangsadresse $\rho(c)$ und Feldkomponenten vom Typ T mit $g =_{def} gr(T)$:
- $code_R\ c[i_1, ..., i_k]\ \rho =_{def} code_L\ c[i_1, ..., i_k]\ \rho$; **ind** T
- $code_L\ c[i_1, ..., i_k]\ \rho =_{def}$ ldc a $\rho(c)$; $code_l\ [i_1, ..., i_k]\ g\ \rho$
- $code_l\ [i_1, ..., i_k]\ g\ \rho =_{def}$
  $code_R\ i_1\ \rho$; **ixa** $g \times D_2$; $code_R\ i_2\ \rho$; **ixa** $g \times D_3$; ... $code_R\ i_k\ \rho$; **ixa** g; **dec** a $d \times g$; ;

D_i *aus* ρ *oder global*

neuer Befehl, s.u.



Zugriff auf Felder (4/4)

Indizierter Zugriff

index**ed** **a**ccess

Befehl	Bedeutung	Bedingung	Ergebnis
ixa q	STORE[SP–1] := STORE[SP–1] + STORE[SP] × q; SP := SP–1	(a, i) und Typ(q) =i	(a)

Inkrementieren und Dekrementieren

increment

decrement

Befehl	Bedeutung	Bedingung	Ergebnis
inc T q	STORE[SP] := STORE[SP] + q;	(T) und Typ(q) = i	(T)
dec T q	STORE[SP] := STORE[SP] – q;	(T) und Typ(q) = i	(T)

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

Feldzugriff mit Bereichsprüfung (1/2)

Weiteres Problem

- Einhaltung der Indexbereiche
- Lösung: verbessertes Übersetzungsschema

$code_{lc}\,[i_1, ..., i_k]\ (g; u_1, o_1, ..., u_n, o_n)\ \rho =_{def}$
$code_R\ i_1\ \rho$; **chk** $u_1\ o_1$; **ixa** $g \times D_2$;
$code_R\ i_2\ \rho$; **chk** $u_2\ o_2$; **ixa** $g \times D_3$; ...
$code_R\ i_k\ \rho$; **chk** $u_k\ o_k$; **ixa** g;
dec $a\ g \times d$;

check

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

Feldzugriff mit Bereichsprüfung (2/2)

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) ρ =

    ldc a 7;
    ldc a 5; **ind** i; **ldc** i 1; **add** i; **chk** -5 5; **ixa** 9; — $D_2 = \Pi_{j=2..2} d_j = d_2 = 9$
    ldc a 6; **ind** i; **chk** 1 9; **ixa** 1;
    dec a -44;
    ldc i 0; **sto** i

Dynamische Felder

Dynamisches Feld

- Ein Feld c mit **var** c: **array** [$u_1..o_1$, ..., $u_k..o_k$] **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

Zugriff in dynamischen Feldern (1/3)

Situation

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

Offensichtlich

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

  mit

  $r =_{def} h \times g - d \times g$

  $= i_1 \times d_2 \times d_3 \times ... \times d_k + i_2 \times d_3 \times d_4 \times ... \times d_k + ... + i_{k-1} \times d_k + i_k -$

  $(u_1 \times d_2 \times d_3 \times ... \times d_k + u_2 \times d_3 \times d_4 \times ... \times d_k + ... + u_{k-1} \times d_k + u_k)$

  sind bis auf k alle Größen dynamisch

Unterschied (bezüglich „dynamisch“)

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

Zugriff in dynamischen Feldern (2/3)

Daher

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

Damit

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

Zugriff in dynamischen Feldern (3/3)

Codeschema für den Zugriff in dynamischen Feldern

- $code_{Rd}\ b[i_1, ..., i_k]\ \rho =_{def} code_{Ld}\ b[i_1, ..., i_k]\ \rho$; **ind** T
- $code_{Ld}\ b[i_1, ..., i_k]\ \rho =_{def}$ **ldc** a ρ(b); $code_{ld}\ [i_1, ..., i_k]\ g\ \rho$
- $code_{ld}\ [i_1, ..., i_k]\ g\ \rho =_{def}$

  dpl a; **ind** a; **ldc** i 0;

  $code_R\ i_1\ \rho$; **add** i; **ldd** 2k+3; **mul** i;

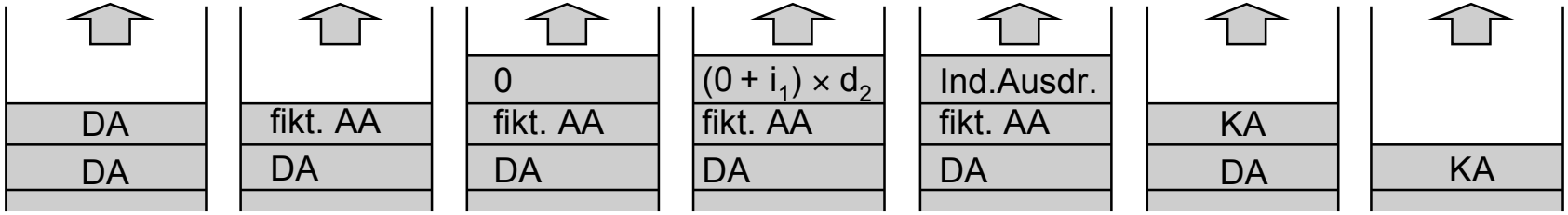
  $code_R\ i_2\ \rho$; **add** i; **ldd** 2k+4; **mul** i; ...

  $code_R\ i_{k-1}\ \rho$; **add** i; **ldd** 3k+1; **mul** i;

  $code_R\ i_k\ \rho$; **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; STORE[SP] := STORE[SP–1]	(T)	(T, T)
ldd q	SP := SP+1; STORE[SP] := STORE[STORE[SP–3]+q]	(a, T_1, T_2)	(a, T_1, T_2, i)
sli T_2	STORE[SP–1] := STORE[SP]; SP := SP–1;	(T_1, T_2)	(T_2)

Speicherbelegung für Verbunde (1/2)

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
  $\rho(i) = 5$, $\rho(j) = 6$, $\rho(v) = 7$, $\rho(a) = 0$, $\rho(b) = 1$

Speicherbelegung für Verbunde (2/2)

Allgemein

- Geg.: **var** v: t mit **type** t = **record** c_1: t_1; c_2: t_2; ... c_k: t_k **end**
- Speicherzuordnung

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

  d.h.
  $\rho(c_1) = 0$
  $\rho(c_2) = gr(t_1)$
  $\rho(c_3) = gr(t_1) + gr(t_2)$
- Größe der Verbundvariablen

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

Beachte

- Größe der Verbundvariablen ist statisch; damit: Felder von Verbunden (oder auch Verbunde mit Verbunden als Komponenten) möglich
- Dynamische Felder als Verbundkomponenten: im Verbund nur Deskriptor ablegen

Adressierung von Verbundkomponenten

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

Adresse von v

Relativadresse der Komponente

Zeiger und dynamische Speicherbelegung (1/3)

Bisher

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

Zeiger

- Verweise auf Speicherzelle („Anonyme Bezeichnungen“ für Objekte)
- 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 (mit Adressen als Werten)

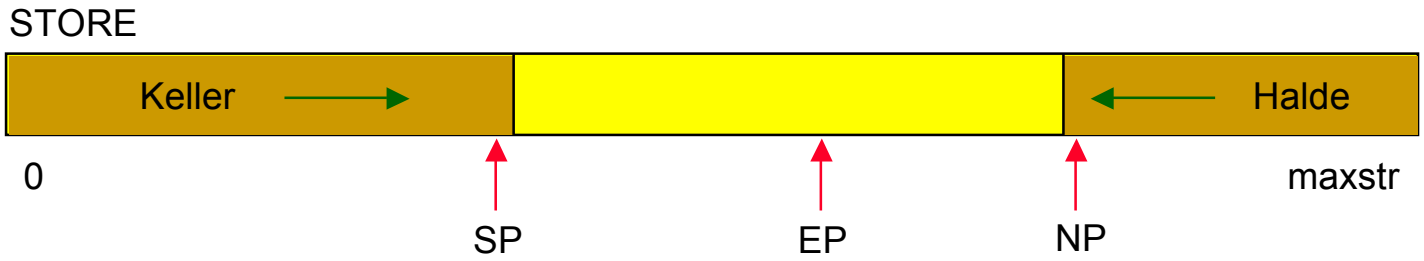
Unterschied

- 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

Zeiger und dynamische Speicherbelegung (2/3)

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

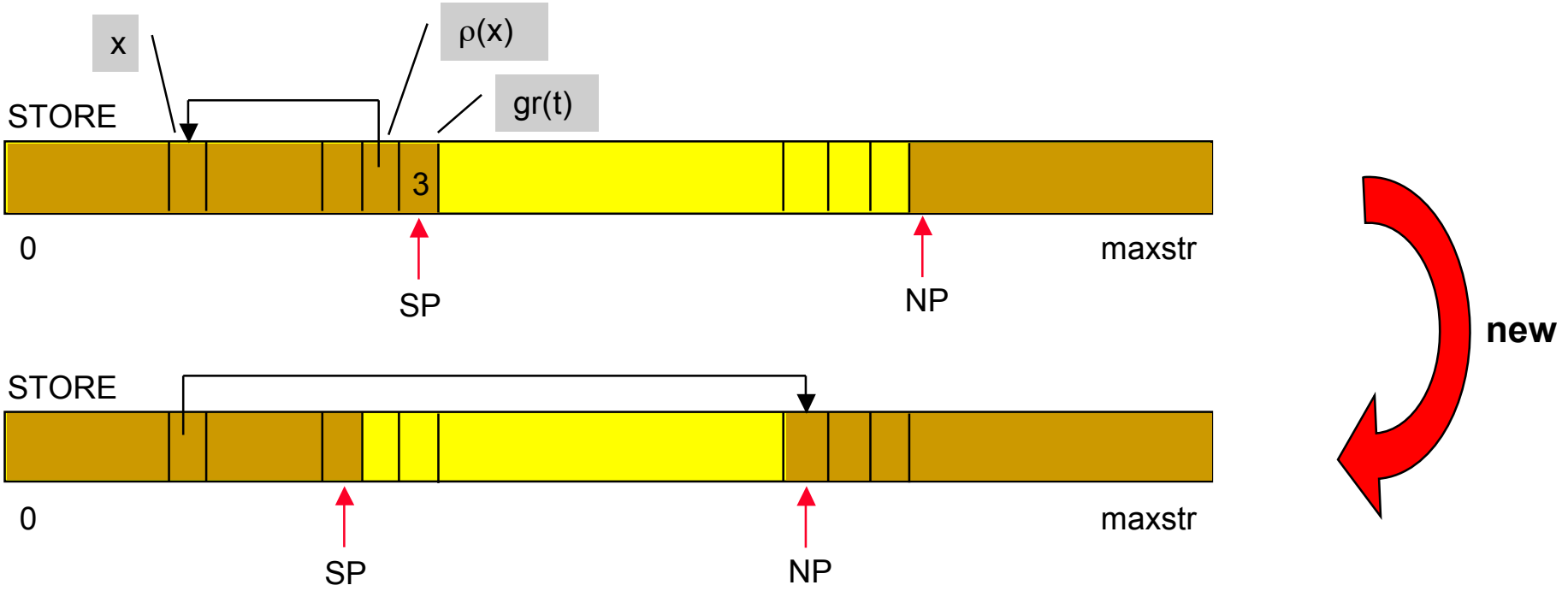
Adresse des Haldenobjekts

Befehl	Bedeutung	Bedingung	Ergebnis
new	**if** NP – STORE[SP] ≤ EP **then** error(„store overflow“) **else** NP := NP–STORE[SP]; STORE[STORE[SP–1]] := NP; SP := SP–2 **fi**	(a,i)	

Zeiger und dynamische Speicherbelegung (3/3)

Codeschema für new-Anweisung

- code **new**(x) ρ =$_{def}$ **ldc** a ρ(x); **ldc** i gr(t); **new** falls x Variable vom Typ ↑t



Adressierung eines anonymen Objekts (Dereferenzierung)

- code x↑ ρ =$_{def}$ **ldc** a ρ(x); **ind** a

Zusammengesetzte Bezeichner (1/2)

Adressberechnung für beliebig kompliziert aufgebaute Bezeichnungen

- $code_R\ x\ \rho =_{def} code_L\ x\ \rho$; **ind**
- $code_L\ (xr)\ \rho =_{def}$ **ldc** a $\rho(x)$; $code_M\ (r)\ \rho$ für Namen x
- $code_M\ (.xr)\ \rho =_{def}$ **inc** a $\rho(x)$; $code_M\ (r)\ \rho$ für Namen x
- $code_M\ (\uparrow r)\ \rho =_{def}$ **ind** a; $code_M\ (r)\ \rho$
- $code_M\ ([i]r)\ \rho =_{def} code_{ld}\ [i]\ g\ \rho$; $code_M\ (r)\ \rho$
  falls g die Komponentengröße des indizierten Felds ist
- $code_M\ (\varepsilon)\ \rho =_{def} \varepsilon$

Invariante

- Falls $code_L\ (uv)\ \rho = b$; $code_M\ (v)\ \rho$, 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

Zusammengesetzte Bezeichner (2/2)

Beispiel

- Geg.:

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

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

- 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

$code_{ld}$ [i+1, j] 1 ρ

Komponentengröße