Übersetzung imperativer Sprachen – Teil 2

Inhalt

- Grundbegriffe
- Sichtbarkeit von Namen und Bindung
- Speicherorganisation für Prozeduren
- Adressierung von Variablen
  - Globale Variablen
  - Prozeduraufruf
- Berechnung der Adressumgebungen
- Prozedureintritt und Prozedurverlassen
  - Prozedurdeklaration
  - Prozeduraufruf
  - Parameterübergabe
  - Zugriff auf Variablen und formale Parameter
- Formale Prozeduren
- Hauptprogramm

Lernziele

- Die prinzipiellen Probleme, die durch sich dynamisch ändernde Stukturierungsmöglichkeiten (z.B. Prozeduren) entstehen, kennen und erklären können, wie man sie prinzipiell löst; insbesondere wie man
  - Variable adressiert
  - die jeweilige Adressumgebung berechnet
  - das Zusammenspiel zwischen Prozedurdeklaration und -aufruf regelt
  - verschiedene Parameterübergabemechanismen umsetzt
- Beschreiben können, wie man das Problem von Prozeduren als Parameter löst

Vorkommen von Prozeduren

Ziel der folgenden Überlegungen

- Codefunktionen für Prozedurdeklaration und -aufruf
  - Funktionen (als Spezialfall) mitbehandelt
  - Wichtigster Aspekt: Zusammenspiel zwischen aufrufender und aufgerufener Prozedur

Zu unterscheiden (bei Prozeduren und Funktionen)

- **Definierendes Vorkommen** = Deklaration mit
  - Namen der Prozedur (oder Funktion)
  - Spezifikation (= Namen und Typen) der formalen Parameter
  - (Angabe des Ergebnistyps bei Funktionen)
  - (lokale) Deklarationen (für Variable, Funktionen oder Prozeduren)
  - Anweisungsteil („Rumpf“)
- **Angewandtes Vorkommen**
  - Aufruf (mit aktuellen Parametern)

Anders als etwa bei C können auch Prozeduren lokal (zu anderen Prozeduren) deklariert werden.
Damit: beliebige Schachtelung von Prozeduren möglich

Aufrufbaum (1/2)

Zu beachten (bei Prozeduren)

- Aufgerufene (noch nicht beendete) Prozedur kann ihrerseits andere Prozedur (oder sich selbst) aufrufen

Begriffe (zur Beschreibung dieses Phänomens)

- **Aufrufbaum** (eines Programmlaufs)
  Geordneter Baum, der bei Ausführung eines Programms entstehenden Prozeduraufrufe („Aufrufstruktur“)
- **Inkarnation** (einer Prozedur p)
  Vorkommen von p im Aufrufbaum

Können auch mehrere sein

- **Inkarnationsweg** (einer Inkarnation)
  Weg von der Wurzel des Aufrufbaums zu dieser Inkarnation

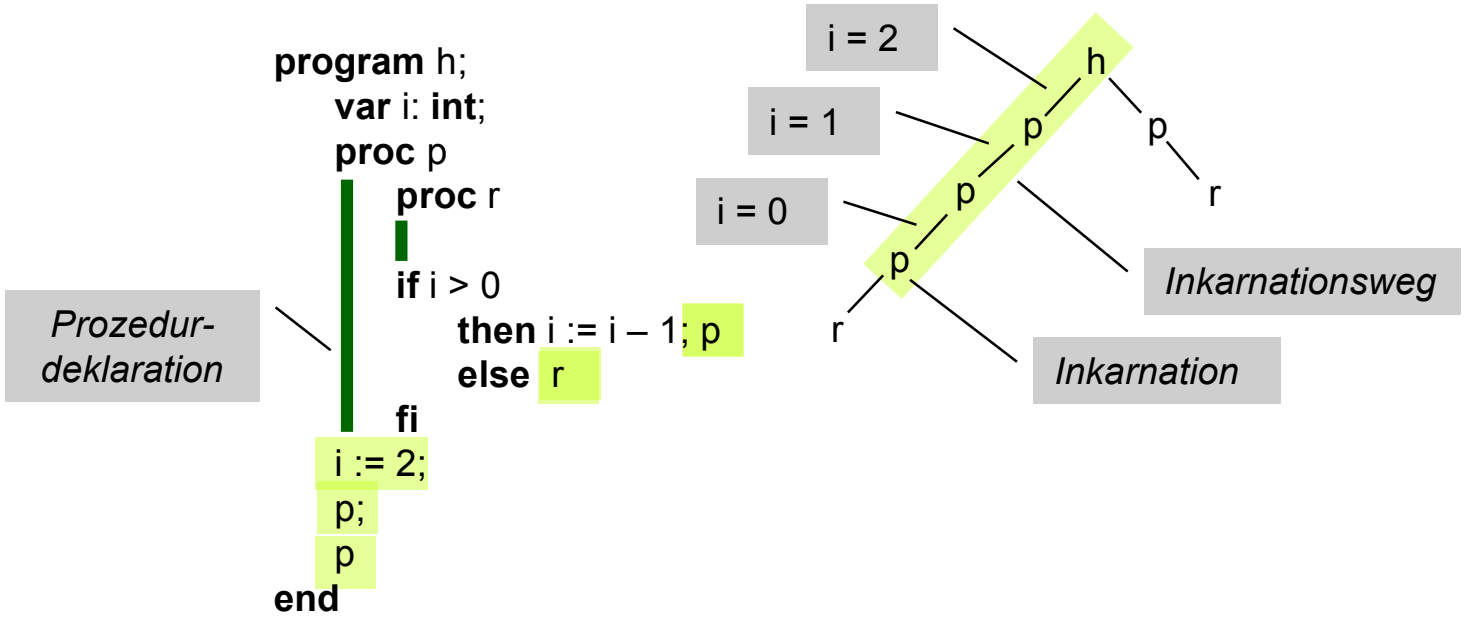
„Aufruf-Geschichte“

- **Lebendige Inkarnation**
  Zugehörige Prozedur ist aufgerufen, aber nicht beendet

Ist von der betrachteten Programmstelle abhängig

Aufrufbaum (2/2)

Beispiel (Programm und Aufrufbaum)



Beachte

- Mehr als ein Aufrufbaum für ein Programm
  (abhängig von Eingabe / Wertebelegung)
  — *Besonderheit im obigen Beispiel: genau ein endlicher Aufrufbaum*
- Unendliche Aufrufbäume
  — *z.B. bei Nicht-Terminierung*

Prozeduren und Namen

In Prozeduren auftretende Namen

„relative“ Begriffe, abhängig vom Bezugspunkt

- Lokale Namen
  - Formale Parameter oder lokale Deklarationen
  - Dafür: neue Inkarnationen bei Prozeduraufruf
    ⇒ entsprechend Speicherplatz vorsehen
  - Lebensdauer (dieser Inkarnationen) =
    Lebensdauer der zugehörigen Prozedurinkarnation
    ⇒ kellerartige Speicherverwaltung
- Globale Namen
  - Angewandt auftretende Namen, die zur betrachteten Prozedur nicht lokal sind

Beispiel

```
program h;
    var i: int;
    proc p
        proc r
        |
        if i > 0
            then i := i – 1; p
            else  r
        fi
    i := 2;
    p;
    p
end
```

Hier:
r *lokal zu* p
i *global zu* p
p, i *lokal zu* h

Problem

- Auf welches definierende Vorkommen beziehen sich angewandt auftretende globale Namen

Sichtbarkeits- / Gültigkeitsregel (Bestandteil der Sprachsemantik)

- Legt Korrespondenz zwischen definierenden und angewandten Vorkommen fest

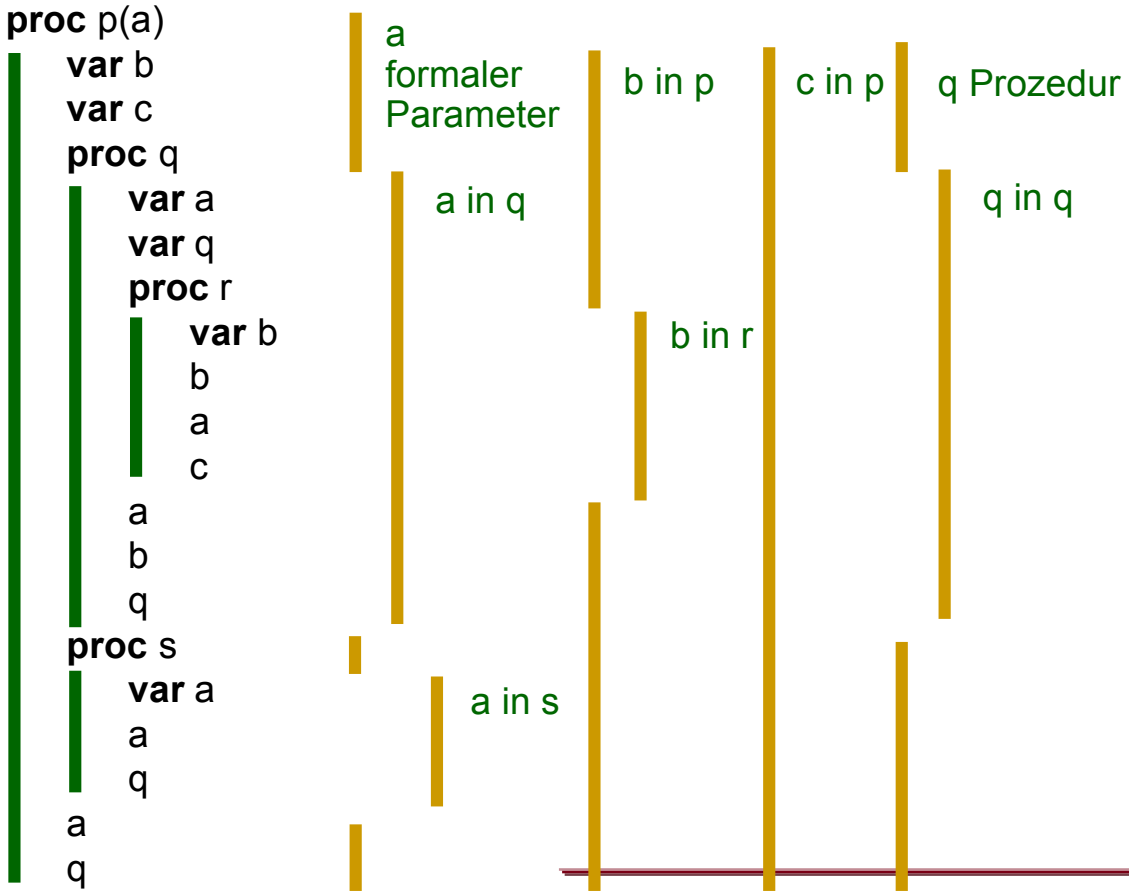
Sichtbarkeit von Namen

Sichtbarkeitsregel (in Algol-ähnlichen Sprachen)

Fast alle imperativen Sprachen
Ausnahme: Fortran

- Definierendes Vorkommen eines Namens ist sichtbar in der Prozedur (Block), deren Deklarations-/Spezifikationsteil die Definition enthält, abzüglich aller echt umfassten Prozeduren (Blöcke), die eine Neudefinition enthalten

Sichtbar-keits-bereiche



Vorgehensweise
(für Namen x)
1. *Def. Vorkommen von x suchen*
2. *Umfassenden Block bestimmen*
3. *Innere Blöcke mit def. Vorkommen von x entfernen*

Statische und dynamische Bindung (1/2)

Verschiedene Möglichkeiten (der Zuordnung zwischen def. und angew. Vorkommen)

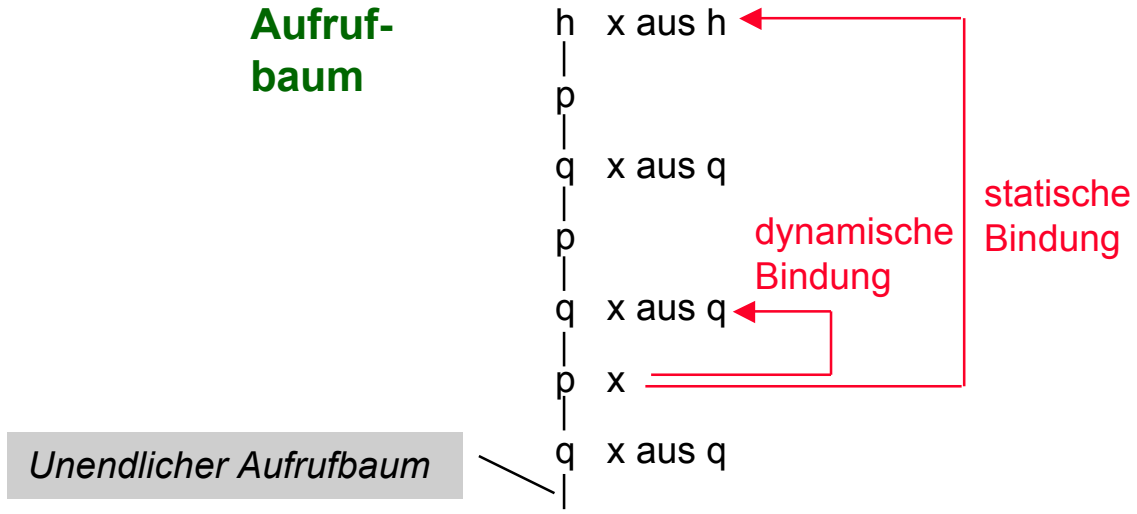
- **Statische Bindung**
  - Zuordnung gemäß obiger Sichtbarkeitsregel (statisch = nur von Programmtext abhängig): angewandtes Vorkommen (bei Ausführung) korrespondiert mit zuletzt kreierter Inkarnation des statisch zugeordneten definierenden Vorkommens
- **Dynamische Bindung**
  - Angewandtes Vorkommen (bei Ausführung) korrespondiert mit zuletzt kreierter Inkarnation (unabhängig davon, wo definierendes Vorkommen auftritt)

Beispiel (statische und dynamische Bindung anhand des Aufrufbaums)

Beispiel-Programm

```
program h
    var x
    proc p
        proc q
            var x
            p
        x
        q
    p
```

Aufruf-baum

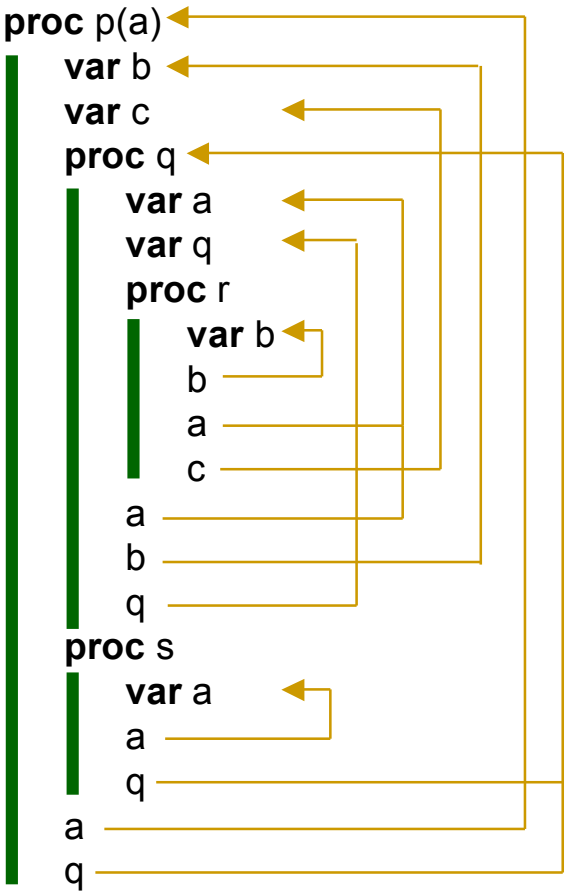


Statische und dynamische Bindung (2/2)

Ermittlung eines definierenden Vorkommens (über Inkarnationsweg)

- *Dynamische Bindung*
  Erstes definierendes Vorkommen im Aufrufbaum auf dem Rückweg zur Wurzel (s.o.)
- *Statische Bindung*
  Definierendes Vorkommen in letzter Inkarnation der innersten umfassenden Prozedur
- **Fazit**: Aufrufbaum ungeeignet zum effizienten Auffinden der „richtigen“ Inkarnation bei statischer Bindung
  ⇒ anderes Konzept erforderlich (s.u.)

Beispiel (statische Bindung)



Statischer Vorgänger

Statischer Vorgänger (einer Prozedur p)

- Letzte Inkarnation einer p direkt umfassenden Prozedur

Baum der statischen Vorgänger (zu einem Inkarnationsweg)

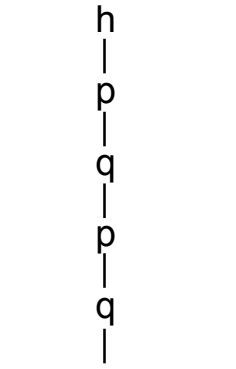
- Knoten: Inkarnationen
- p ist Sohn von q: q ist letzte Inkarnation der p direkt umfassenden Prozedur

Damit (im Baum der statischen Vorgänger)

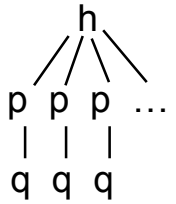
- „richtige“ Inkarnationen (globaler Variablen) eines angewandten Vorkommens auf Weg von zugehöriger Inkarnation zur Wurzel

Beispiele

```
program h
    var x
    proc p
        proc q
            var x
            p
        x
        q
    p
```



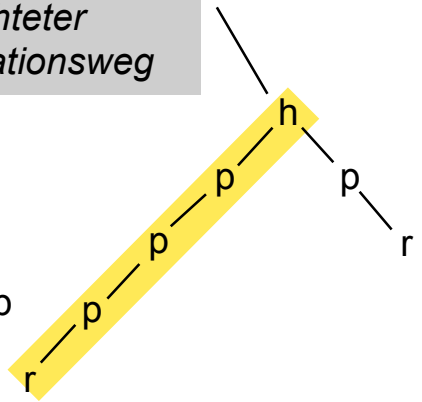
Aufrufbaum



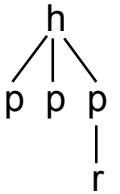
Baum der Statischen Vorgänger

```
program h;
    var i: int;
    proc p
        proc r
        if i > 0
            then i := i – 1; p
            else  r
        fi
    i := 2; p;p
end
```

Betrachteter Inkarnationsweg



Aufrufbaum



Baum der Statischen Vorgänger

Speicherorganisation für Prozeduren (1/3)

Laufzeitkeller

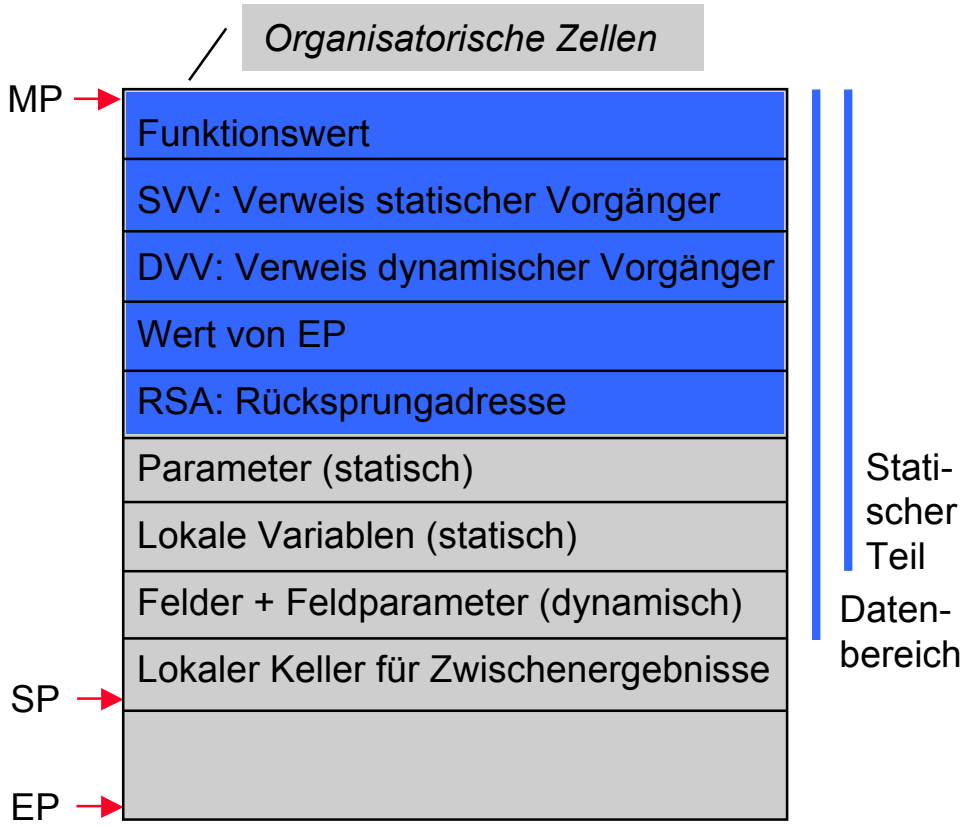
- Gleichzeitige Realisierung zweier Strukturen
  - *Inkarnationsweg*: als Folge von konsekutiven Speicherbereichen für lebende Inkarnationen
  - *Statischer Vorgängerbaum* (zum Inkarnationsweg): durch Zeiger

Für jede Inkarnation einer Prozedur

Kellerrahmen, bestehend aus

- Datenbereich (der Inkarnation)
  - Statischer Teil
    - Organisatorische Zellen, je eine für
      - Ergebnis (für Funktionen)
      - Verweis auf statischen Vorgänger
      - Verweis auf dynamischen Vorgänger
      - Stand von EP (s.u.)
      - Rücksprungadresse
    - Statische Parameter (inkl. Deskriptoren)
    - (stat.) lokale Variablen (inkl. Deskriptoren)
  - Dynamischer Teil (Felder + Feldparameter)
- Lokaler Keller

  (für die Auswertung des Rumpfes)



Speicherorganisation für Prozeduren (2/3)

Verwendete Register

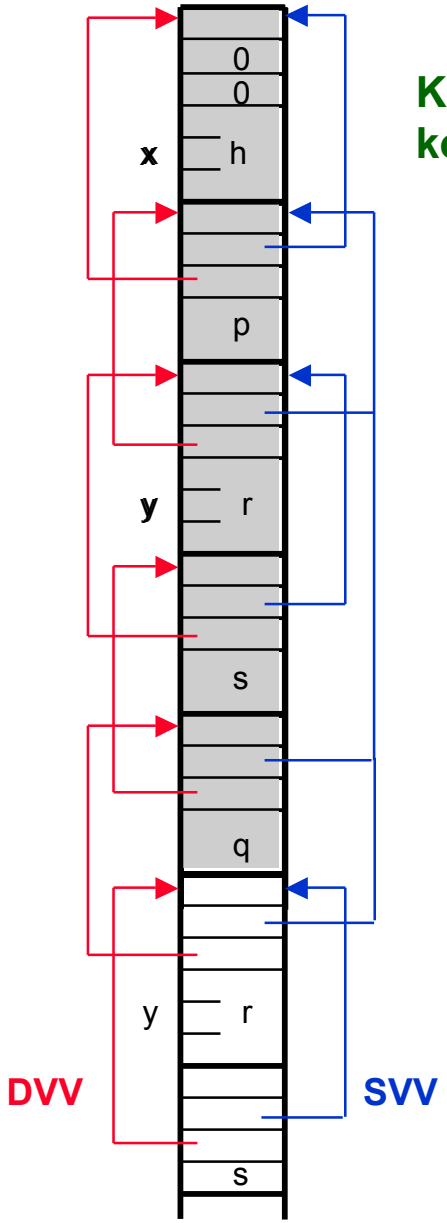
- MP („mark pointer“)
  Zeiger auf Anfang des Kellerrahmens der aktuellen Inkarnation

Neu: Basis für die relative Adressierung innerhalb einer Prozedur

- SP („stack pointer“)
  Zeiger auf „oberste“ belegte Zelle des lokalen Kellers
- EP („extreme stack pointer“)
  Zeiger auf „höchste“ (während Ausführung der Prozedur) belegte Kellerzelle
  (entspricht maximalem Platzbedarf zur Auswertung des Prozedurrumpfes)

wie gehabt

Speicherorganisation für Prozeduren (3/3)

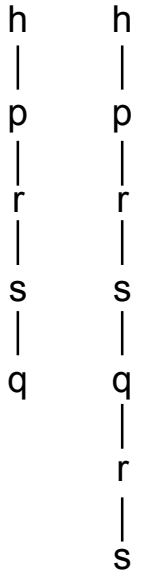


Keller-konfiguration

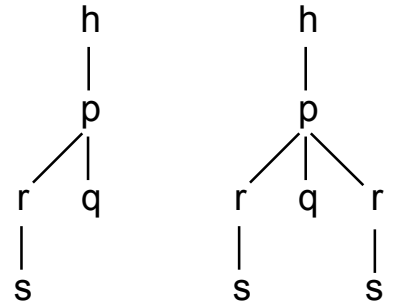
Programm

```
program h
    var x
    proc p
        :
        proc q
            :
            r
            :
        proc r
            var y
            proc s
                x + y
            q
        s
    r
p
```

Aufrufbaum / Inkarnationsweg



Baum der statischen Vorgänger



Adressierung von Variablen

Offensichtlich

- Wegen geschachtelter Sichtbarkeitsbereiche und dynamisch kreierter neuer Inkarnationen keine statischen, absoluten Adressen zuordenbar

Stattdessen

- Lokale Variablen
  - Statische Relativadresse, relativ zum Anfang des Kellerrahmens
  - Zugriff über MP-Register
- Globale Variablen
  - Zugriff über SV-Verweis
  - Problem
    - Mehrfach-Verfolgen von SV-Verweisen
    - Frage: wie oft?

Zur Adressierung globaler Variablen erforderlich

- Schachtelungstiefe (eines Programmkonstrukts)

Schachtelungstiefe

Schachtelungstiefe (eines Programmkonstrukts)

- Hauptprogramm: 0
- Definierendes (angewandtes) Vorkommen eines Namens im Deklarations-(Anweisungs-) teil einer Einheit mit Schachtelungstiefe n: n + 1

Beispiel

```
program h
    var x
    proc p
        :
        proc q
            :
            r
            :
        proc r
            var y
            proc s
                x + y
                q
            s
        r
    p
0   1   2   3   4
```

Schachtelungstiefen

Definierende Vorkommen		Angewandte Vorkommen	
p	1	p	1
q	2	q	4
r	2	r (in p)	2
		r (in q)	3
s	3	s	3
x	1	x	4
y	3	y	4

Angewandte Vorkommen + Schachtelungstiefe

Invariante (ISV)

- Zu jedem Zeitpunkt der Ausführung eines Programms gilt
  - In jedem auf dem Keller angelegten Rahmen für eine Inkarnation einer Prozedur p zeigt der SV-Verweis auf den Kellerrahmen der „richtigen“ Inkarnation der p direkt umfassenden Einheit, d.h.
    - In Programmen ohne formale Prozeduren:
      jeweils jüngste noch lebende Inkarnation der p direkt umfassenden Programmeinheit
    - In Programmen mit formalen Prozeduren
      s.u.

Vorgehensweise

- Zugriff auf globale Variablen: Adressberechnung mit ISV
- Prozeduraufruf: Sicherstellen von ISV für neu anzulegenden Rahmen

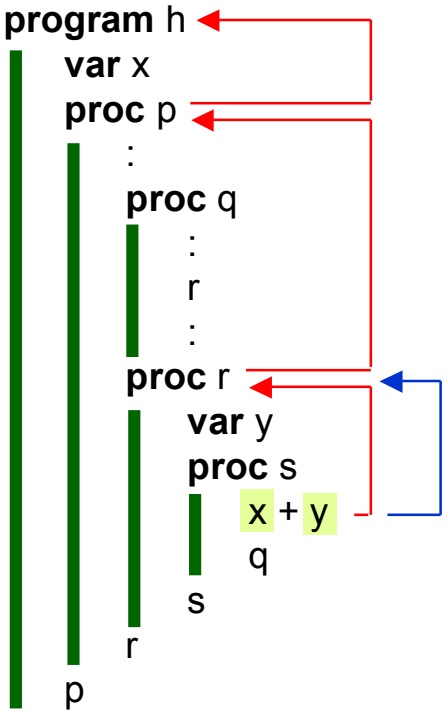
Zugriff auf angewandte Vorkommen (1/2)

Zugriff auf globale Variablen

- Voraussetzung: ISV gilt
- Zugriff auf globale Namen (Variablen und Prozeduren)
  - Situation
    - Angewandtes Vorkommen auf ST n,
    - Definierendes Vorkommen auf ST m (m ≤ n)
  - (n-m)-maliges Verfolgen des SV-Verweises ergibt AA des gesuchten Rahmens

Beispiele

- Zugriff auf x in s
  - ST dieses (angew.) Vorkommens: 4
  - ST des zugehörigen def. Vorkommens in h: 1
  - ⇒ 3-maliges Verfolgen des SV-Verweises ergibt Anfang des zugehörigen Kellerrahmens (von h)
- Zugriff auf y in s
  - ST dieses (angew.) Vorkommens: 4
  - ST des zugehörigen def. Vorkommens in r: 3
  - ⇒ 1-maliges Verfolgen des SV-Verweises ergibt Anfang des zugehörigen Kellerrahmens (der letzten Inkarnation von r)



Zugriff auf angewandte Vorkommen (2/2)

Neue P-Befehle (zum Zugriff über Schachtelungstiefe)

Befehl	Bedeutung	Kommentar
lod T p q	SP := SP+1; STORE[SP] := STORE[base(p, MP) + q]	p Differenz der Schachtelungstiefen q Relativadresse
lda p q	SP := SP+1; STORE[SP] := base(p, MP) + q	
str T p q	STORE[base(p, MP) + q] := STORE[SP] SP := SP–1;	

entspricht
- **ldo**
- **ldc** a
- **sro**

Unterschied:

Dabei

- base(p, a) =$_{def}$ **if** p = 0 **then** a **else** base(p-1, STORE[a + 1]) **fi**

Differenz der ST

Aktueller MP

AA des Keller-rahmens der Prozedur, wo die Deklaration zu finden ist

entspricht lokaler Variable

SV-*Verweis*

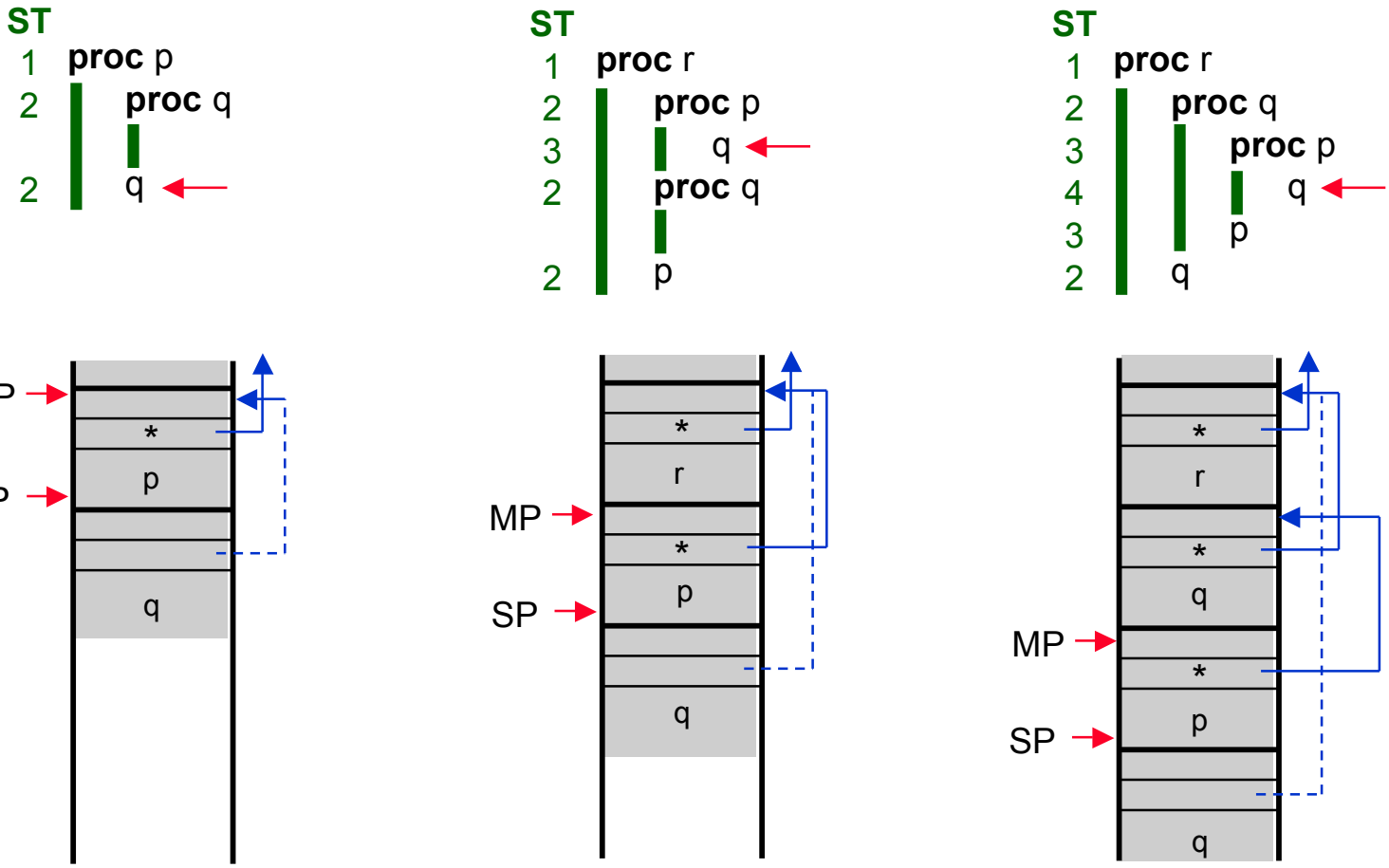
Prozeduraufruf (1/2)

Prozeduraufruf

- Voraussetzung: ISV gilt
- Sicherstellen von ISV bei Abarbeitung Prozeduraufruf
  (d.h. Besetzung SV-Verweis in anzulegendem Kellerrahmen)
  - Situation
    - Aufruf von q in p auf ST n
    - Definierendes Vorkommen von q auf ST m (m ≤ n)
  - Statischer Vorgänger von q durch (n – m)-maliges Verfolgen des SV-Verweises in p
    - m = n: angewandtes und definierendes Vorkommen von q auf gleicher Stufe (d.h. in p)
      ⇒ statischer Vorgänger von q = p
    - m < n: statischer Vorgänger von q auch (direkter oder indirekter) Vorgänger von p
      ⇒ statischer Vorgänger ergibt sich durch (n-m)-maliges Verfolgen der SV-Kette, die in p beginnt

Prozeduraufruf (2/2)

Aufruf- und Kellersituationen (Aufrufer: p; Aufgerufener: q)



Berechnung der Adressumgebungen

Erweiterung des bisherigen Konzepts

- Adressen und Schachtelungstiefen
- Berücksichtigung der Sichtbarkeitsregel

Adressen und Schachtelungstiefen

- Bindung aller definierenden Vorkommen von Namen
  - Variablennamen: an Relativadresse + Schachtelungstiefe
  - Prozedurnamen: an symbolische Marke + Schachtelungstiefe
- Somit
  - Adr_Umg = Id → Adr × ST
  - wobei Adr
    - Für Variablennamen: Relativadresse in Kellerrahmen
    - Für Prozedurnamen: (absolute) Adresse in CODE

Berücksichtigung der Sichtbarkeitsregel

- Verarbeitung des Deklarationsteils einer Prozedur von außen nach innen
- Lokale Adressumgebung mit „Überschreiben“ nicht sichtbarer Namen
  (Vorteil der rekursiven Definition der code-Funktion: nach Abarbeitung der Prozedurdeklaration liegt wieder „alte“ Adressumgebung vor)

Parameterspezifikationen (1/2)

Verarbeitung von Parameterspezifikationen (= Veränderung der Adressumgebung)

- elab_specs: Spec* × Adr_Umg × Adr × ST → Adr_Umg × Adr

  1. „freie“ Adresse

  nächste „freie“ Adresse

- mit
  - ρ[(n_a, st) / x] identisch mit ρ bis auf Wert n_a für x

    d.h. ergänzen oder überschreiben
- hat man
  - elab_specs () ρ n_a st $=_{def}$ (ρ, n_a)
  - elab_specs (**var** x: t; specs) ρ n_a st $=_{def}$
    elab_specs specs ρ[(n_a, st) / x] (n_a + 1) st
  - elab_specs (**value** x: t; specs) ρ n_a st $=_{def}$
    elab_specs specs ρ[(n_a, st) / x] (n_a + gr(t)) st für statische Typen t
  - elab_specs (**value** x: **array** [$u_1..o_1$, ..., $u_k..o_k$] **of** t; specs) ρ n_a st $=_{def}$
    elab_specs specs ρ' (n_a + 3k + 2) st
    mit ρ' = ρ[(n_a, st) / x][(n_a + 2i + 1, st) / u_i]$_{i=1..k}$ [(n_a + 2i + 2, st) / o_i]$_{i=1..k}$

Platzbedarf

- Variable: 1 Speicherplatz
- (value-) Feld: 3k+2 Speicherplätze für Deskriptor
- (sonstiger) value-Parameter: Speicherplatz entsprechender Größe

Parameterspezifikationen (2/2)

Beispiel

- Geg.: **var** x, f, k: **integer**; z: **boolean**;
  proc p (**value** k: **integer**; **var** x: **boolean**; **value** f: **array** [m..n, s..t] **of char**)
- Annahme: st = 1
- Verarbeitung der Parameterspezifikationen

  Prozedurbezeichner p *zugeordnete symbolische Marke (Details, s.u.)*

  - ρ = {(x → (5,1), f → (6,1), k → (7,1), z → (8,1)}, n_a = 9, st = 1
  - ρ = {(x → (5,1), f → (6,1), k → (7,1), z → (8,1), p → (l, 1)},
    n_a = 5, st = 2

    n_a *wird bei Verarbeitung einer Prozedurdeklaration zurückgesetzt*, st *erhöht*
  - ρ = {(x → (6,2), f → (6,1), k → (5,2), z → (8,1), p → (l, 1)}, n_a = 7, st = 2
  - ρ = {(x → (6,2), f → (7,2), k → (5,2), z → (8, 1), p → (l, 1),
    (m → (10,2), n → (11,2), s → (12,2), t → (13,2)}, n_a = 15, st = 2
- Auf Relativadressen 7 bis 14: Deskriptor für f
- „fehlende“ Einträge für Deskriptor
  - (8,2): Feldgröße
  - (9,2): Subtrahend für fiktive AA
  - (14,2): Spanne d_2

Lokale Variablen (1/2)

Verarbeitung der Deklarationen lokaler Variablen

- elab_vdecls: Vdecl* × Adr_Umg × Adr × ST → Adr_Umg × Adr
- mit
  - elab_vdecls () ρ n_a st $=_{def}$ (ρ, n_a)
  - elab_vdecls (**var** x: t; vdecls) ρ n_a st $=_{def}$
    elab_vdecls vdecls ρ[(n_a, st) / x] (n_a + gr(t)) st für nicht-Feld-Typen t
  - elab_vdecls (**var** x: **array** [$u_1..o_1$, ..., $u_k..o_k$] **of** t; vdecls) ρ n_a st $=_{def}$
    elab_vdecls vdecls ρ[(n_a, st) / x] (n_a + 3k + 2 + $\Pi_{i=1..k}$ ($o_i - u_i + 1$) × gr(t)) st
    falls x statisches Feld ist

Bei Parametern: 1 Speicherplatz für Adresse
Hier: Speicherplatz entsprechend Größe

Platz für Deskriptor

Platz für Feldkomponenten

Platzbedarf

- Variable: Speicherplatz entsprechender Größe
- (statisches) Feld: Platz für Deskriptor und Komponenten

Lokale Variablen (2/2)

Zusätzlich (zu lokalen Variablen)

- Verbundkomponenten: wie früher
- Dynamische Felder
  - Deskriptor in statischem Teil des Kellerrahmens
  - Bindung des Feldnamens an Anfangsadresse Deskriptor
  - Feld selbst im dynamischen Teil des Kellerrahmens abgelegt durch entsprechende Befehlsfolge bei Prozeduraufruf

Anders als früher

- Deskriptoren für alle Felder
  (Deskriptorinformation wird benötigt, falls Feld aktueller Parameter einer Prozedur ist)
- Statische Felder
  Deskriptor und Feld nacheinander (Eintragung durch entsprechende Befehlsfolge)

Lokale Prozedurdeklarationen

Verarbeitung lokaler Prozedurdeklarationen

- elab_pdecls: pdecl* × Adr_Umg × ST → Adr_Umg × M

  Folge von P-Befehlen

- mit
  - elab_pdecls () ρ st $=_{def}$ (ρ, ())
  - elab_pdecls (**proc** p_1(...); ...; ... **proc** p_k(...); ...;) ρ st $=_{def}$

    (ρ', l_1: code (**proc** p_1(...); ...) ρ', st+1; ... l_k: code (**proc** p_k(...); ...) ρ', st+1)

    wobei ρ' = ρ[(l_1, st) / p_1, ... , (l_k, st) / p_k]
- Behandlung angewandter Vorkommen von Prozeduren vor definierendem Vorkommen durch Bindung der Prozedurnamen an symbolische Marken l_i

  (wegen möglicher Unterprogrammsprünge in p_i auf p_j mit j > i und noch nicht bekanntem p_j)

Beachte

- In code (**proc** ...) berechnete Adressumgebungen sind nach außen nicht bekannt

Prozedureintritt und Prozedurverlassen

Prozedureintritt

- Situation
  - p sei momentan aktive Prozedur; alle Verweise seien richtig gesetzt; p rufe Prozedur q auf

Aktionen für Eintritt in q

- Durch aufrufende Prozedur p
  - SV-Verweis auf richtige Inkarnation der q direkt umfassenden Prozedur setzen
  - DV-Verweis auf Anfang Kellerrahmen von p setzen
  - Aktueller Stand EP-Register retten
  - Parameterübergabe (→ später: code $_A$);

  } *mit* **mst**

  - MP-Register auf neuen Kellerrahmen setzen
  - Rücksprungadresse speichern
  - Sprung auf erste Instruktion der Übersetzung von q ausführen

  } *mit* **cup**

- Durch aufgerufene Prozedur q
  - SP auf Anfang des lokalen Kellers von q setzen

  } *mit* **ssp**

  - Kopien der aktuellen value-Feldparameter anlegen (→ später: code$_S$)
  - EP setzen (einschließlich Kollision Keller - Halde prüfen)

  } *mit* **sep**

Prozedurverlassen

- Aktionen nach Abarbeitung von q
  - Restaurieren von MP und EP
  - Kellerrahmen von q freigeben
  - Rücksprung nach p

} *mit* **retf** *bzw.* **retp**

Prozedurdeklaration (1/3)

Befehle für Prozedur-Eintritt

Befehl	Bedeutung	Kommentar
mst p	STORE[SP + 2] := base(p, MP);	Verweis auf statischen Vorgänger
	STORE[SP + 3] := MP;	Verweis auf dynamischen Vorgänger
	STORE[SP + 4] := EP;	Retten von EP
	SP := SP + 5	
cup p q	MP := SP – (p + 4);	p = Platzbedarf für die Parameter
	STORE[MP + 4] := PC;	Retten der Rücksprungadresse
	PC := q	Sprung zur Anfangsadresse q der Prozedur
ssp p	SP := MP + p – 1;	p = Größe statischer Teil des Datenbereichs
sep p	EP := SP + p;	p = maximale Tiefe des lokalen Kellers
	if EP ≥ NP **then** error(„store overflow“) **fi**;	

base(p,a) = **if** p = 0 **then** a else base(p–1, STORE[a+1]) **fi**

mark **st**ack

call **u**ser **p**rocedure

set **sp**

set **ep**

Berechnet AA des Kellerrahmens des statischen Vorgängers

Prozedurdeklaration (2/3)

Befehle für Prozedur-Verlassen

return function

return procedure

Befehl	Bedeutung	Kommentar
retf	SP := MP;	Funktionsergebnis oben im Keller
	PC : = STORE[MP + 4];	Rücksprung
	EP := STORE[MP + 3];	Restaurieren von EP
	if EP ≥ NP **then** error(„store overflow“) **fi**;	
	MP := STORE[MP + 2];	DV-Verweis
retp	SP := MP – 1;	Prozedur ohne Ergebnis
	PC : = STORE[MP + 4];	Rücksprung
	EP := STORE[MP + 3];	Restaurieren von EP
	if EP ≥ NP **then** error(„store overflow“) **fi**;	
	MP := STORE[MP + 2];	DV-Verweis

Aufgerufene Prozedur könnte NP *verändert haben*

Unterschied (zwischen Funktion und Prozedur)

- Funktion: SP auf 1. Rahmenzelle der aufgerufenen Funktion (= Ergebnis)
- Prozedur: SP auf letzte besetzte Adresse des Aufrufers

Prozedurdeklaration (3/3)

Übersetzungsschema für Prozedurdeklaration

- code (**proc** p (specs); vdecls, pdecls; body) ρ st $=_{def}$

ssp n_a";	*Speicherbedarf statischer Teil (= Parameter + lokale Variable)*
$code_S$ specs ρ' st;	*Speicherbedarf dynamischer Teil (Feldkomponenten bei Feldparametern)*
$code_P$ vdecls ρ" st;	*Speicherbedarf dynamischer Teil (Deskriptor für lokale Felder initialisieren)*
sep k;	*Maximale Tiefe des lokalen Kellers*
ujp l;	
proc_code;	*Code für die lokalen Prozeduren*
l: code body ρ'" st;	*Code für den Prozedurrumpf*
retp;	

- wobei
  - (ρ', n_a') = elab_specs specs ρ 5 st
  - (ρ", n_a") = elab_vdecls vdecls ρ' n_a' st
  - (ρ'", proc_code) = elab_pdecls pdecls ρ" st
- bei Funktionen
  - **retf**-Befehl anstelle von **retp**-Befehl (hinterlässt Funktionsergebnis oben auf dem Keller)

Prozeduraufruf

Übersetzungsschema für Prozeduraufruf

- code $p(e_1, ..., e_k)\ \rho\ st =_{def}$

  mst st–st';

  $code_A\ e_1\ \rho\ st$; ...

  $code_A\ e_k\ \rho\ st$;

  cup s l

- wobei
  - s = Platzbedarf für die aktuellen Parameter
  - $\rho(p) = (l, st')$

Differenz der Schachtelungstiefen für SV-Verweis

Parameterübergabe (s.u.)

p *zugeord-nete Marke*

p *zugeordnete Schachtelungstiefe*

Prozedurdeklaration und -aufruf (1/2)

Beispiel

- Geg.: ρ = {p → (m,1)}, st = 2

  proc p (**value** n: **integer**; **var** x: **integer**);
  var k: **integer**; **proc** dupl (**var** x: **integer**); **begin** x := x × 2 **end**;
  begin k := 0; **while** k < n **do** k := k+1; dupl(x) **od end**

- Ges.: code (**proc** p ...) ρ 2
  - (ρ', n_a') = elab_specs (**value** ...) ρ 5 2 = ({p → (m,1), n → (5,2), x → (6,2)}, 7)
  - (ρ", n_a") = elab_vdecls (**var** k ...) ρ' 7 2 = ({p → (m,1), n → (5,2), x → (6,2), k → (7,2)}, 8)
  - (ρ'", proc_code) = elab_pdecls (**proc** dupl ...) ρ" 2 = (ρρ, ll: code (**proc** dupl ...) ρρ 3)
    wobei ρρ = {p → (m,1), n → (5,2), x → (6,2), k → (7,2)), dupl → (ll, 2)}

- Nun ges.: code (**proc** dupl ...) ρρ 3
  - (ρρ', nn_a') = elab_specs (**var** x ...) ρρ 5 3 =
    ({p → (m,1), n → (5,2), x → (5,3), k → (7,2), dupl → (ll, 2}), 6)
  - (ρρ", nn_a") = elab_vdecls () ρρ' 6 3 = (ρρ', 6)
  - (ρρ'", proc_code') = elab_pdecls () ρρ" 3 = (ρρ', ())

Prozedurdeklaration und -aufruf (2/2)

Beispiel (Fortsetzung)

- Damit (für code (**proc** dupl ...) ρρ 3)

```
ssp 6;
sep 3;
ujp lll;
lll: lda 0 5; lda 0 5; ind i; ldc i 2; mult; sto i;
retp
```

Dynamischer Teil (fehlt hier)

Code für lokale Prozeduren (fehlt hier)

Rumpf von dupl

- Insgesamt

```
ssp 8;
sep 3;
ujp l;
ll: «Code für dupl»;
l: lda 0 7; ldc i 0; sto i;
l': lda 0 7; ind i; lda 0 5; ind i; les;
fjp e;
lda 0 7; lda 0 7; ind i; ldc i 1; add; sto i;
mst 0; code_A x ρρ 2; cup 1 ll;
ujp l';
e: retp
```

Dynamischer Teil (fehlt hier)

Wie oben

k := 0

k < n

k := k+1

dupl(x)

while

Rumpf von p

Parameterübergabe (1/4)

Situation

- Verschiedene Arten von Parametern
  - var-Parameter
  - value-Parameter
    - Skalare
    - Verbunde + Felddeskriptoren (statisch)
    - Dynamische Felder
  - Prozeduren (formale Prozeduren)
- Kontext für Parameterübergabe
  - **mst**-Befehl ausgeführt (d.h. organisatorische Zellen besetzt)

Übersetzungsschema (für var-Parameter)

Übergabe der Adresse („call-by-reference“)

- $code_A\ x\ \rho\ st =_{def} code_L\ x\ \rho\ st$
  falls der zu x korrespondierende formale Parameter **var**-Parameter ist

$code_L$ *unten neu definiert*

Übersetzungsschema (für skalare value-Parameter)

Übergabe des Werts („call-by-value“)

- $code_A\ e\ \rho\ st =_{def} code_R\ e\ \rho\ st$
  falls der zu e korrespondierende formale Parameter **value**-Parameter ist

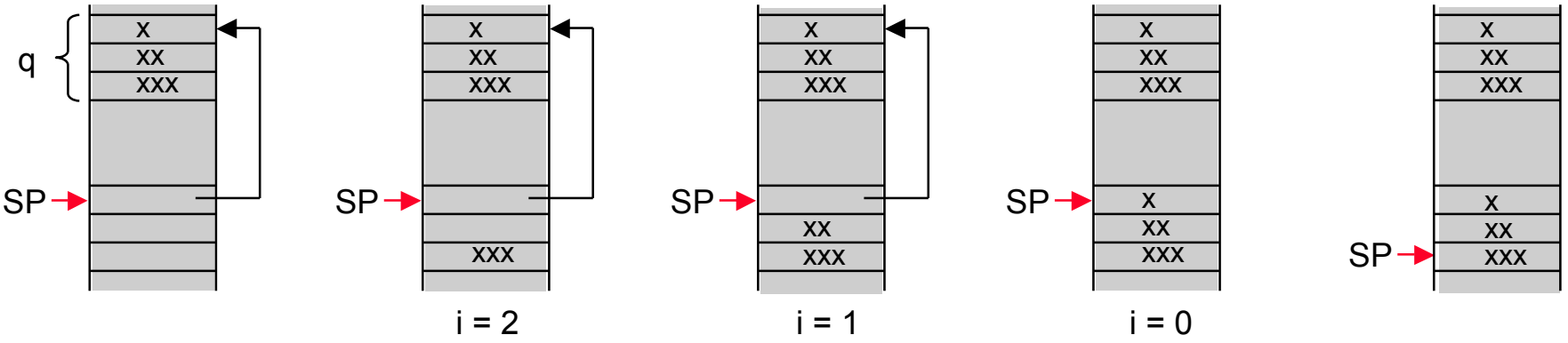
Parameterübergabe (2/4)

Übersetzungsschema (für value-Parameter – Verbunde + Deskriptoren von Feldern)

- $code_A\ x\ \rho\ st =_{def} code_L\ x\ \rho\ st$; **movs** g
  falls der zu x korrespondierende formale **value**-Parameter strukturierten Typs t und statischer Größe gr(t) = g ist

AA übergeben Parameter kopieren

Befehl	Bedeutung	Bedingung	Ergebnis
movs q	**for** i:= q – 1 **downto** 0 **do** STORE[SP + i] : = STORE[STORE[SP] + i] **od**; SP := SP + q – 1	(a)	



Parameterübergabe (3/4)

Aktionen bei der Übersetzung von (dynamischen) value-Feldparametern

- Durch den Aufrufer
  - Deskriptor des aktuellen Feldparameters in Parameterbereich der aufgerufenen Prozedur kopieren (mit **movs**)
- Durch die aufgerufene Prozedur
  - Feldinhalt kopieren (mit **movd**)
  - Eintragen der fiktiven Anfangsadresse der Kopie in den Deskriptor

Übersetzungsschema (für die Spezifikation eines formalen value-Feldparameters x)

- Situation
  - Aufrufer hat Deskriptor des aktuellen Parameters bereits kopiert ($\rho(x) = (ra, st)$)
- Codefunktion zur Ablage der Feldkomponenten im dynamischen Teil
  (die bei Übersetzung der Prozedurdeklaration aufgerufen wird)
  - $code_S$ (**value** x: **array** $[u_1..o_1, ..., u_k..o_k]$ **of** t; sp) ρ st $=_{def}$ **movd** ra; $code_S$ sp ρ st
  - $code_S$ (s; sp) ρ st $=_{def}$ $code_S$ sp ρ st falls s kein formaler **value**-Feldparameter ist
  - $code_S$ () ρ st $=_{def}$ ()

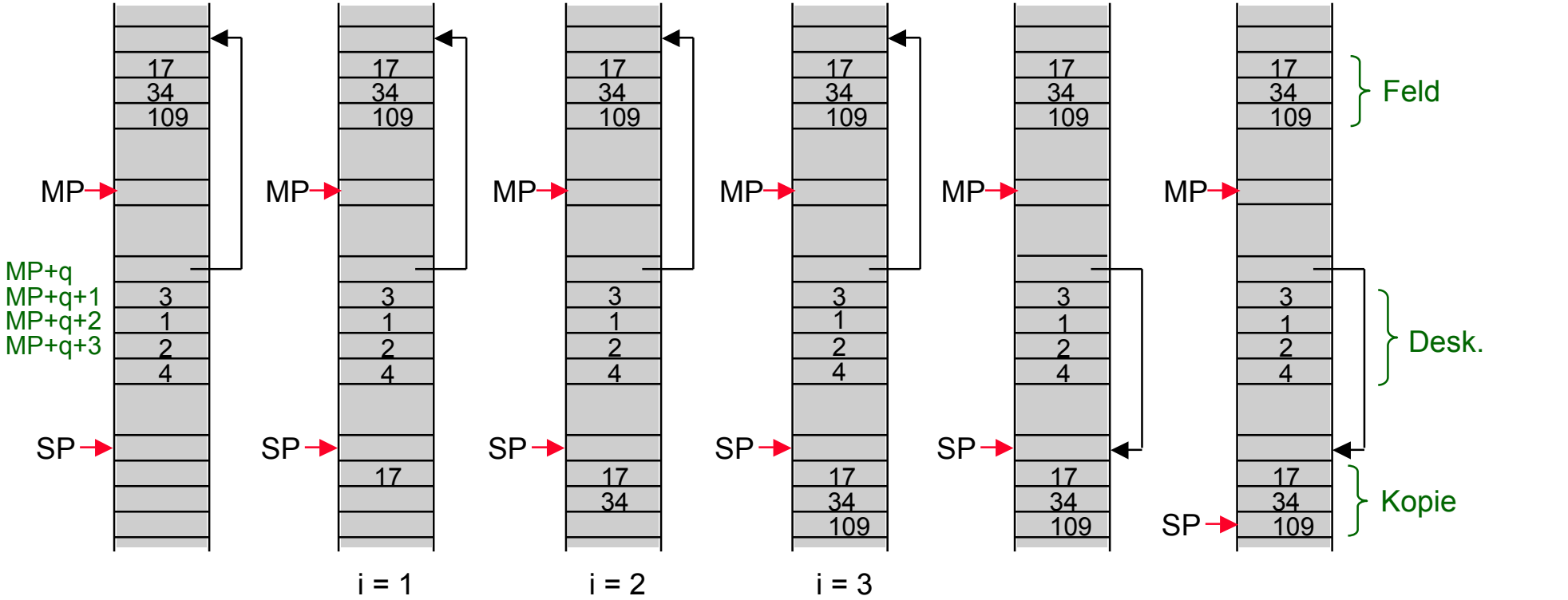
Parameterübergabe (4/4)

Kopieren eines dynamischen Feldes

Im Prinzip wie statisches Kopieren (indirekt über Felddeskriptor)

Befehl	Bedeutung
movd q	**for** i:= 1 **to** STORE[MP + q +1] **do** STORE[SP + i] : = STORE[STORE[MP + q] + STORE[MP + q + 2] + i -1] **od**; STORE[MP + q] := SP + 1 - STORE[MP + q + 2] ; SP := SP + STORE[MP + q + 1];

q: *Relativadr. Deskr.*
STORE[MP+q+1]:
Feldgröße
STORE[MP+q]: *fikt. AA*
STORE[MP+q+2]:
Subtr. für fikt. AA



Zugriff auf Variablen und formale Parameter

Erweiterung von $code_L$ (für Zugriff auf lokale/globale Variablen und formale Parameter)

- Lokale Variablen und formale **value**-Parameter
  direkt: Relativadresse zu MP addieren
- Globale Variablen
  direkt: Basisadresse mittels SV-Verweis berechnen und Relativadresse addieren

Spezialfall von globalen Variablen: Basisadresse = MP

- Formale **var**-Parameter
  indirekt: Relativadresse zu MP addieren

Übersetzungsschema $code_L$ (für angewandte Vorkommen von Namen)

- $code_L\ (x\ r)\ \rho\ st =_{def}$ **lda** $d\ ra;\ code_M\ r\ \rho\ st$
  wobei $\rho(x) = (ra, st')$ und $d = st–st'$, falls x Variable oder formaler **value**-Parameter ist

lda: *Adresse direkt*

- $code_L\ (x\ r)\ \rho\ st =_{def}$ **lod** $d\ ra;\ code_M\ r\ \rho\ st$
  wobei $\rho(x) = (ra, st')$ und $d = st–st'$, falls x formaler **var**-Parameter ist

lod: *Adresse indirekt*

Formale Prozeduren (1/3)

Motivation

- Geg.: Beispiel
- Ges.: SV-Verweis für das Anlegen des Kellerrahmens für die aktuellen Prozeduren (d.h. für g bzw. f bei Aufruf von p)

```
program
    proc p (function h)
        h
    proc q
        function f
            p(g)
        p(f)
    function g
        q
end
```

Wegen statischer Bindung

- Für Aufrufer von p muss aktuelle Prozedur sichtbar sein (d.h. über Kette der statischen Vorgänger erreichbar)

Daher

- Kette der statischen Vorgänger liefert Anfangsadresse der richtigen Inkarnation
- Diese Adresse wird beim Aufruf mitgegeben

Übergabe einer aktuellen Prozedur

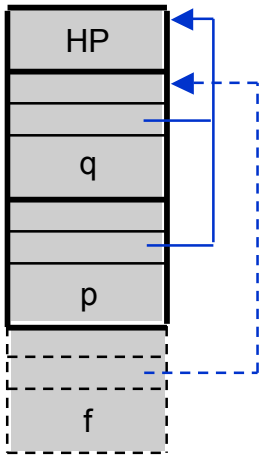
- Im Parameterbereich der aufgerufenen Prozedur werden abgelegt
  - Anfangsadresse der Übersetzung der aktuellen Prozedur
  - Anfangsadresse des Kellerrahmens ihres statischen Vorgängers

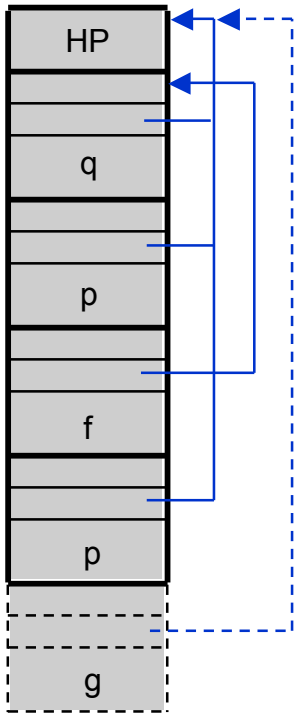
in Prozedurdeskriptor

Formale Prozeduren (2/3)

Beispiel (SV-Verweis für formale Prozeduren)

```
program
    proc p (function h)
        h
    proc q
        function f
            p(g)
        p(f)
    function g
        q
end
```





Formale Prozeduren (3/3)

Zwei mögliche Fälle (bei der Übergabe einer aktuellen Prozedur q in Aufruf p(q))

- Unterschied: Deskriptor existiert bereits oder muss erst angelegt werden
- Aktuelle Prozedur q ist deklariert: Deskriptor muss angelegt werden
  - „Prozedurdeskriptor“ (mit Anfangsadresse der Übersetzung der Prozedur und SV-Verweis) anlegen (mit p als Typ für Adresse in CODE)
  - $code_A\ f\ \rho\ st =_{def}$ falls f deklarierte Prozedur mit $\rho(f) = (adr, st')$ und $d = st – st'$ ist

    ldc p adr; Anfangsadresse der Übersetzung

    lda d 0; SV-Verweis für späteren Aufruf

    } *Deskriptor*
- Aktuelle Prozedur q ist formale Prozedur (d.h. Parameter): Deskriptor existiert bereits
  - Inhalt des Deskriptors der formalen Prozedur kopieren
  - $code_A\ f\ \rho\ st =_{def}$ falls f formale Prozedur mit $\rho(f) = (ra, st')$ und $d = st – st'$ ist

    lda d ra; Lade Deskriptoradresse (ra ist Relativadresse des Deskriptors)

    movs 2; Kopiere Deskriptor

Aufruf einer formalen Prozedur

Aufruf einer formalen Prozedur

- Änderungen (gegenüber Aufruf einer deklarierten Prozedur)
  - SV-Verweis aus 2. Zelle der formalen Prozedur (modifizierter **mst**-Befehl)
  - Unterprogrammsprung indirekt über 1. Zelle der formalen Prozedur (neuer Befehl **cupi**)
- Übersetzungsschema für Aufruf einer formalen Prozedur f
  - code f(e_1, ..., e_k) ρ st =$_{def}$ **mstf** st–st' ra; code$_A$ e_1 ρ st; ... code$_A$ e_k ρ st; **smp** s; **cupi** (st–st') ra
  - wobei ρ(f) = (ra, st') ist und s der Platzbedarf für die aktuellen Parameter

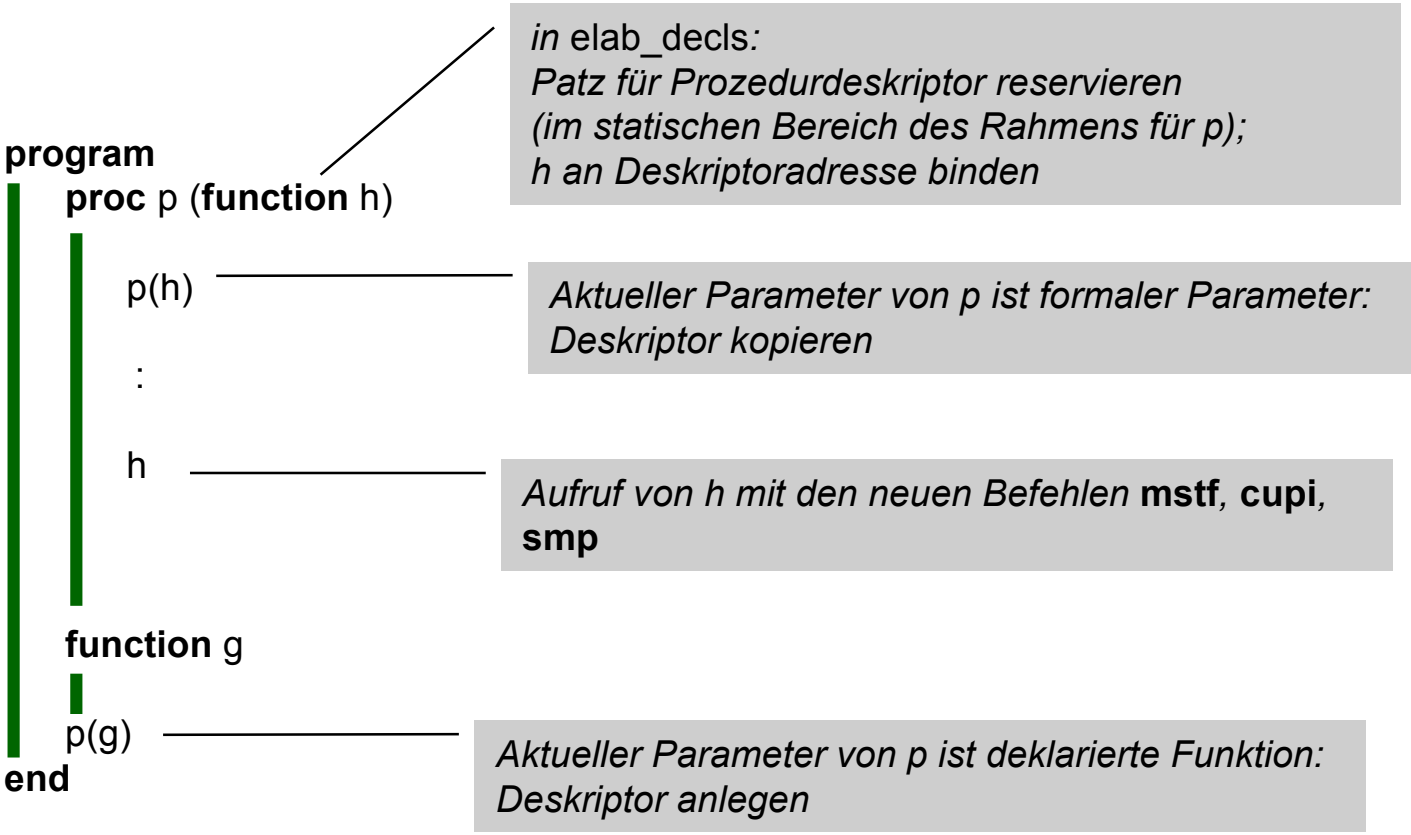
Befehl	Bedeutung	Kommentar
mstf p q	STORE[SP + 2] := STORE[base(p, MP]) + q + 1];	Verweis auf statischen Vorgänger
	STORE[SP + 3] := MP;	Verweis auf dynamischen Vorgänger
	STORE[SP + 4] := EP;	Retten von EP
	SP := SP + 5	
cupi p q	STORE[MP + 4] := PC;	Retten der Rücksprungadresse
	PC := STORE[base(p, STORE[MP + 2]]) + q]	Sprung zur Anfangsadresse q der Prozedur
smp p	MP := SP – (p + 4);	

Erweiterung von elab_specs um formale Prozeduren und Funktionen

- Namen an Relativadressen für zugehörige Deskriptoren binden

Kurzfassung: Formale Prozeduren

Verschiedene Aktivitäten bei der Übersetzung formaler Prozeduren



Hauptprogramm

Ausgangssituation (Initialisierung der Register)

- SP-Register mit -1 initialisiert
- NP-Register mit maxstore+1 initialisiert
- Übrige Register mit 0 initialisiert

Übersetzungschema für das Hauptprogramm

Im Prinzip wie Prozeduren;
Unterschiede:
- feste Startwerte für Speichervergabe
- keine Parameter

- code (**program** vdecls, pdecls; stats) $\varnothing$ 0 $=_{def}$

  ssp n_a;
  $code_P$ vdecls ρ 1;
  sep k;
  ujp l;
  proc_code;
  l: code stats ρ' 1;
  stp;

Erzeugt Code für die Besetzung von Felddeskriptoren

hält die P-Maschine an

- wobei
  - (ρ, n_a) = elab_vdecls vdecls $\varnothing$ 5 1
  - (ρ', proc_code) = elab_pdecls pdecls ρ 1