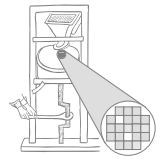


Optimierende Compiler

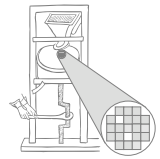
4. Laufzeitumgebung

Andreas Koch

FG Eingebettete Systeme und ihre Anwendungen
Informatik, TU Darmstadt

Sommersemester 2007

Organisatorisches 1

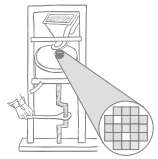


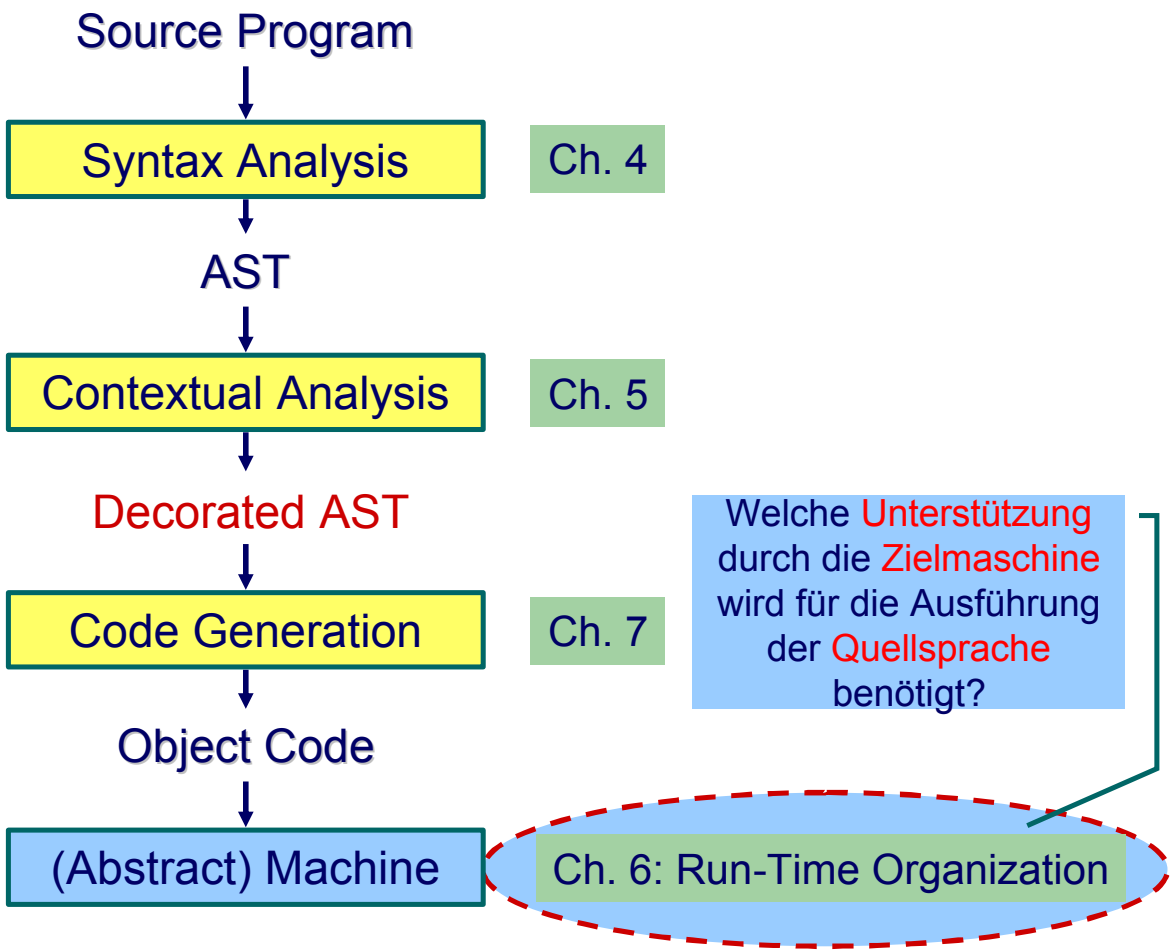
Klausuren

- Mittwoch, 6.6.2007, von 17:15 - 18:30 Uhr in C205
  - Für alle Teilnehmer (IV4 und V3)
- Mittwoch, 18.7.2007 von 16:00 -18:00 Uhr in S3 06 | 051
  - Nur für Theoretiker (V3)

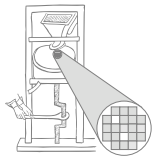
➥Rechtzeitig anmelden!

Übersicht



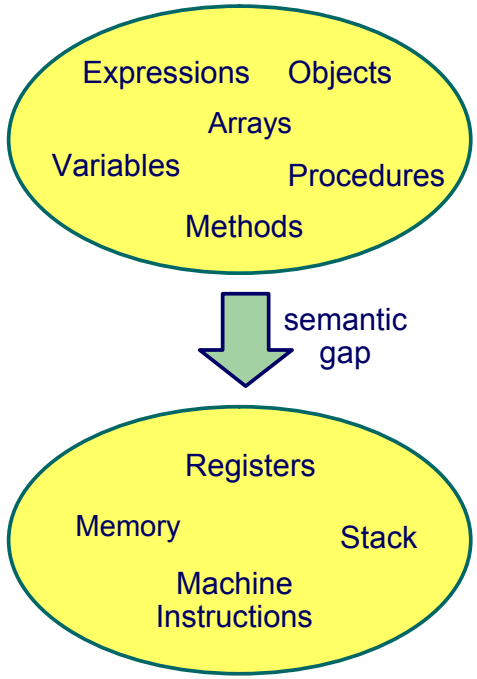

Source Program
Syntax Analysis
Ch. 4
AST
Contextual Analysis
Ch. 5
Decorated AST
Code Generation
Ch. 7
Object Code
(Abstract) Machine
Welche Unterstützung durch die Zielmaschine wird für die Ausführung der Quellsprache benötigt?
Ch. 6: Run-Time Organization

Laufzeitorganisation 1

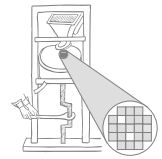


A. Koch

- Compiler übersetzt Hochsprachenprogramm in äquivalentes Maschinenprogramm
- Laufzeitorganisation beschreibt Darstellung von abstrakten Strukturen der Hochsprache auf Maschinenebene
- Instruktionen und Speicherinhalte



Laufzeitorganisation 2



Wichtige Aspekte

Datendarstellung der Werte jedes Typs der Eingabesprache

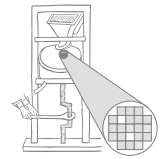
Auswertung von Ausdrücken und Handhabung von Zwischenergebnissen

Speicherverwaltung verschiedener Daten: Global, lokal und Heap

Routinen zur Implementierung von Prozeduren, Funktionen und ihre Datenübergabe

Erweiterung auf OO-Sprachen Objekte, Methoden, Klassen und Vererbung

Triangle Abstract Machine (TAM)

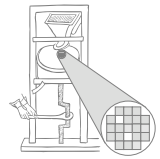


- Zwei getrennte Speicherbereiche
- Datenspeicher: 16b Worte
- Instruktionsspeicher: 32b Worte

➥Harvard-Architektur

Adressbereiche über CPU-Register adressiert

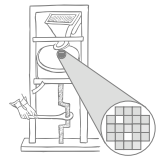
Adressierung des Instruktionsspeichers



A. Koch

Programm	CB	Code Base (konstant)
	CT	Code Top (konstant)
	CP	Code Pointer (variabel)
Intrinsics	PB	Primitive Base (konstant)
	PT	Primitive Top (konstant)

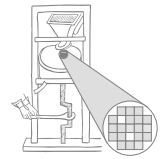
Adressierung des Datenspeichers



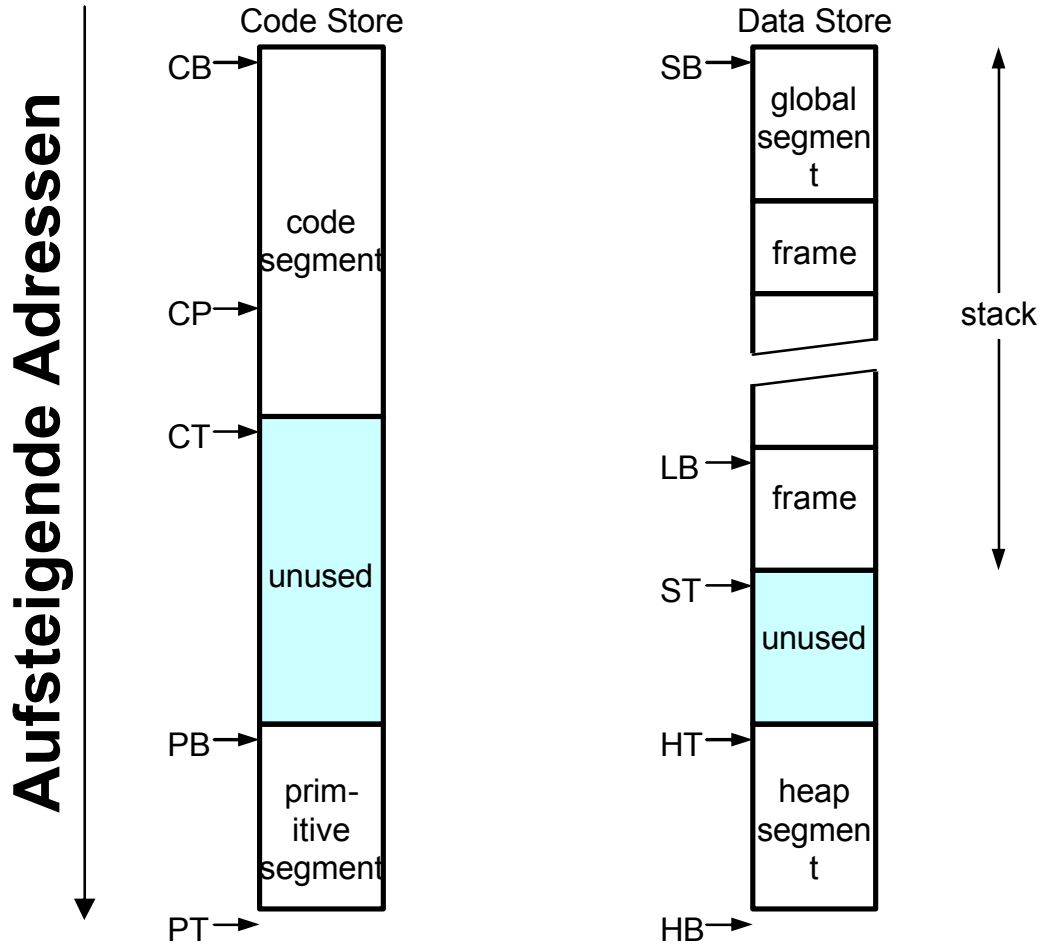
A. Koch

Stack	SB	Stack Base (konstant)
	ST	Stack Top (variabel)
Heap	HB	Heap Base (konstant)
	HT	Heap Top (variabel)
	HF	Heap Free (variabel)

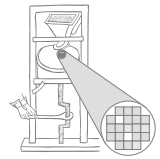
TAM Speicherbereiche



A. Koch



TAM Instruktionen



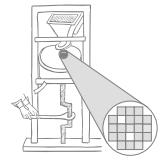
- 32b Worte im Programmspeicher
- op, 4b Art der Instruktion
- r, 4b Registernummer
- n, 8b Operandengröße in Worten
- d, 16b Adressverschiebung (displacement, offset)

Beispiel: LOAD (1) 3[ST]

- op=0 (0000)
- r=5 (1001)
- n=1 (00000001)
- d=3 (0000000000000011)

➥0000 1001 0000 0001 0000 0000 0000 0011

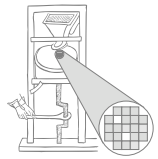
TAM-Befehlssatz



A. Koch

Op.	Mnem.	Effect
0	LOAD(n) d[r]	Fetch an n-word object from the data address and push it onto the stack
1	LOADA d[r]	Push the data address onto the stack
2	LOADI(n)	Pop a data address from the stack, fetch an n-word object from that address, push it onto the stack
3	LOADL d	Push the one-word literal value d onto the stack
4	STORE(n) d[r]	Pop an n-word object from the stack, and store it at the data address
5	STOREI(n)	Pop an address from the stack, then pop an n-word object from the stack and store it at that address
6	CALL(n) d[r]	Call the routine at the code address using the address in register n as the static link
7	CALLI	Pop a closure (static link and code address) from the stack, then call the routine
8	RETURN(n) d	Return from the current routine; pop an n-word result from the stack, then pop the topmost frame, then pop d words of arguments, then push the result back
9	–	(unused)
10	PUSH d	Push d words (uninitialised) onto the stack
11	POP(n) d	Pop an n-word result from the stack, then pop d more words, then push the result back on the stack
12	JUMP d[r]	Jump to code address
13	JUMPI	Pop a code address from the stack, then jump to that address
14	JUMPIF(n) d[r]	Pop a one-word value from the stack, then jump to code address if and only if that value equals n
15	HALT	Stop execution of the program

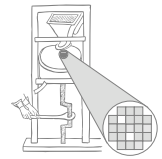
TAM Intrinsics



- Auch Primitive genannt
- “Magische” Adressen im Programmmspeicher
- Führen bei Aufruf als Routine komplexe Operationen aus
- . . . direkt in der abstrakten Maschine (hier: Java)
- Keine TAM-Instruktionen mehr!

Addr.	Mnemo.	Arg.	Res.	Effect
. . .				
2[PB]	not	t	t'	t' = !t
. . .				
8[PB]	add	i1, i2	i'	i' = i1 + i2
. . .				
15[PB]	ge	i1, i2	t'	Set t'=true iff i1 $\geq$ i2
. . .				
26[PB]	putint	i	–	Write an integer whose value is i

Datendarstellung (Repräsentation)



Unverwechselbarkeit Unterschiedliche Werte sollen unterschiedliche Darstellungen haben

- Klappt nicht immer (duale Gleitkommadarstellung reeller Zahlen)

Einzigartigkeit Ein Wert wird immer auf die gleiche Weise dargestellt

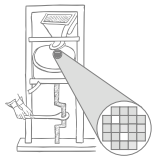
Konstante Größe Alle Werte eines Typs belegen dieselbe Menge an Speicherplatz

Art der Darstellung

Direkt Wert einer Variablen $\mathtt{x}$ kann direkt adressiert werden

Indirekt Wert einer Variablen $\mathtt{x}$ muß über einen Zeiger bzw. *Handle* adressiert werden

Direkte ./. indirekte Repräsentation

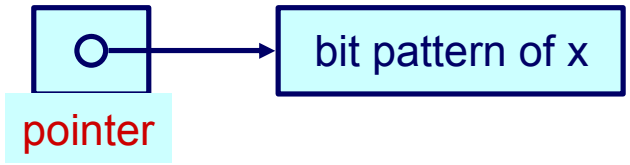


A. Koch

Direkt

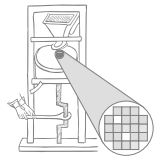
- Effizienter Zugriff, keine Zeiger verfolgen
- Effiziente Abspeicherung
- Implizite Adressierung auf Stack
- Pascal, C/C++, Java (primitive Typen!)

Indirekt



- Für Typen mit variierender Darstellungsgröße
  - Dynamische Arrays
  - Rekursive Typen
  - Objekte
- Zeiger/Handles selber haben konstante Größe
- Lisp, ML, Haskell,

Primitive Typen



Notation

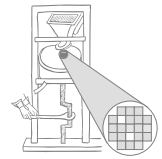
- #[*T*]: Anzahl unterschiedlicher Elemente in *T*
- size[*T*] minimaler Speicherbedarf zur Darstellung eines Wertes aus *T*

Primitive Typen

Können nicht weiter in kleinere Typen zerlegt werden.
Beispiele: `Integer, Char, Boolean`

	#[T]	*size[T]*	*Darstellung*
`Boolean`	2	≥ 1	0 and 1
`Integer`	2^16 or 2^32	16 / 32	2-complement
`Char`	2^8 or 2^16	8 / 16	ASCII/Unicode
`float`	infinite	32 / 64	approximation

Invariante

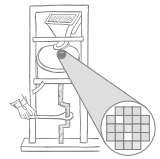


Es muss immer gelten

$$\mathrm{size}[T] \geq \log_2(\ \#[T]\ )$$

wenn $\mathrm{size}[T]$ in Bits gemessen wird.

Darstellung



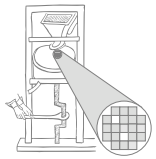
TAM

- **Boolean** 16b (=1 Datenwort): 00..00, 00..01
- **Char** 16b (=1 Datenwort): Unicode
- **Integer** 16b (=1 Datenwort): maxint = $2^{15} - 1$ = 32767

Pentium-basierte Systeme

- **Boolean** 8b (=1 Byte): 00..00, 11..11
- **Char** 8b (=1 Byte): ASCII
- **Integer** 16b oder 32b (=1 word, double word)

Records 1

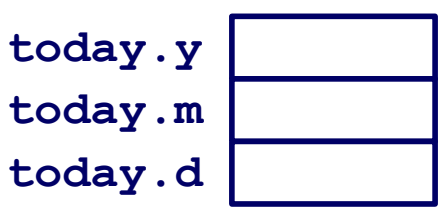


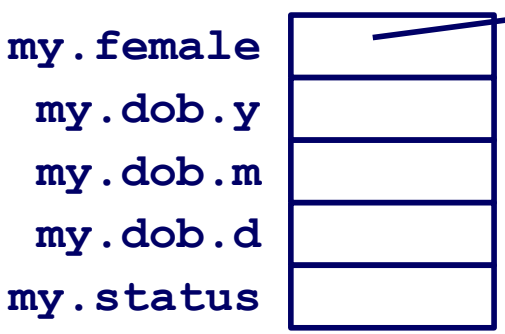
A. Koch

```
type Date ~ record
    y : Integer,
    m : Integer,
    d : Integer
end;
type Details ~ record
    female : Boolean,
    dob : Date,
    status : Char
end;
var today: Date;
var my: Details
```

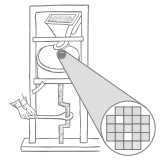
Üblicherweise wird ein Record durch die Anreihung der Darstellungen seiner Komponenten repräsentiert.

Im Beispiel wird angenommen, das ganze Wörter adressiert werden. Verschwenderisch für Boolean!



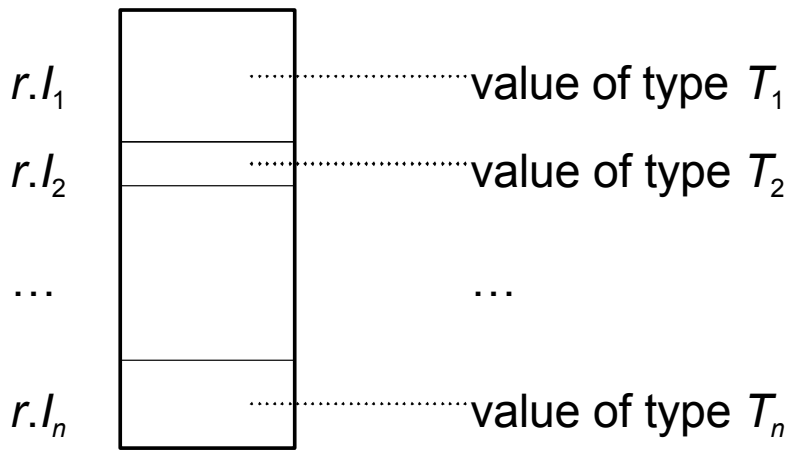


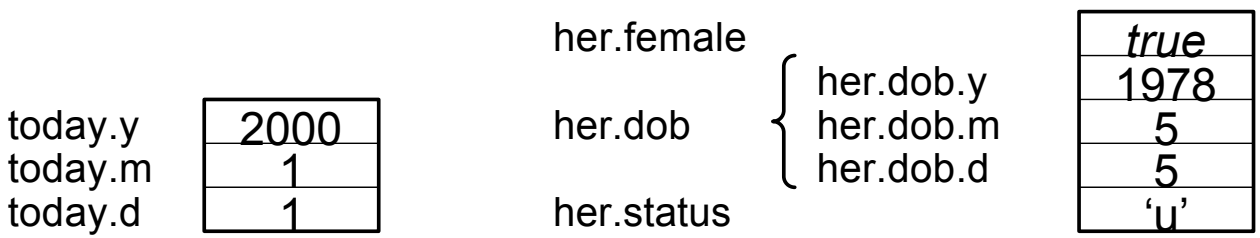
Records 2



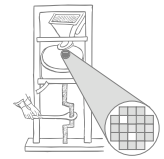
Anordnung im Speicher

A. Koch





Records 3



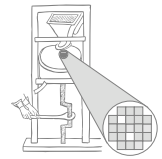
Speicherbedarf und Adressierung
Wo genau liegen die einzelnen Daten im Speicher?

A. Koch

```
type Date = record
    y : Integer,
    m : Integer,
    d : Integer
    end;
var today: Date;
```

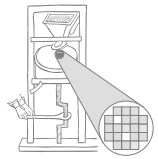
- size[`Date`] = 3 * size[`Integer`] = 3 Worte
- address[`today.y`] = address[`today`]
- address[`today.m`] = address[`today`] + size[`Integer`]
- address[`today.d`] = address[`today`] + 2*size[`Integer`]

Records 4



- Viele reale Prozessoren haben Anforderungen an Adressausrichtung von Daten
  - Beispiel: Es können nur 32b Worte als Einheit adressiert werden
  - Ist schneller, als größere Freiheit zu unterstützen
- Darstellung von Records im Speicher kann ineffizient werden
  - Unter Platzgesichtspunkten (wenn optimal ausgerichtet)
  - Unter Laufzeitgesichtspunkten (wenn optimal gepackt)

Variante Records (disjoint unions) 1

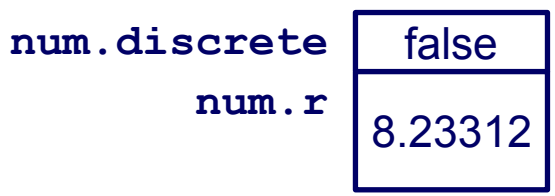


A. Koch

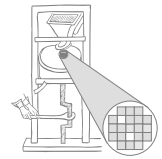
Ähnlich einer Record, aber zu einem Zeitpunkt existiert immer nur eine Untermenge von Komponenten.

- Selektion der aktiven Untermenge durch *type tag*

```
type Number =
   record
      case (discrete:Boolean) of
          true: (i: Integer);
          false: (r: Real)
      end;
var num: Number
```

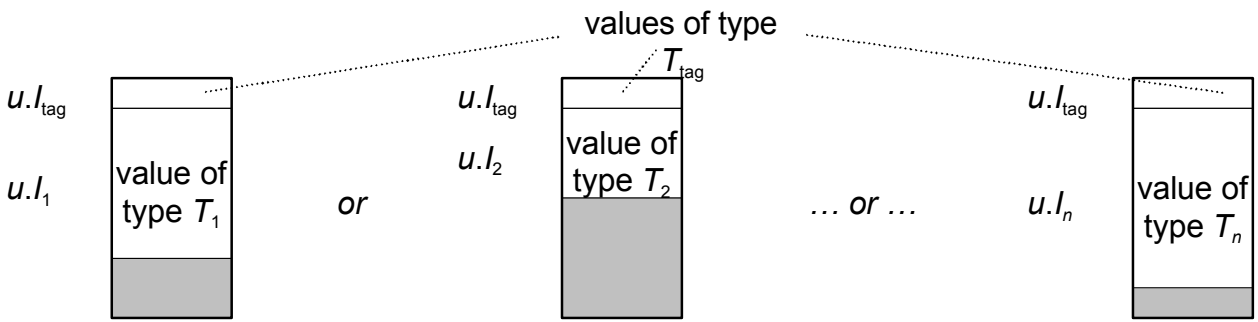


Variante Records 2

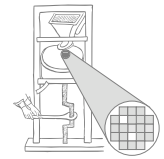


A. Koch

Allgemeiner Aufbau



Variante Records 3



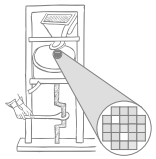
Adressierung: Lege disjunkte Teile im Speicher übereinander

A. Koch

```
type Number = record
    case acc: Boolean of
         true : ( i : Integer );
         false : ( r : Real );
    end;
var num : Number;
```

- size[`Number`] = size[`Boolean`] + max(size[`Integer`], size[`Real`])
- address[`num.acc`] = address[`Number`]
- address[`num.i`] = address[`Number`] + size[`Boolean`]
- address[`num.r`] = address[`Number`] + size[`Boolean`]

Arrays 1



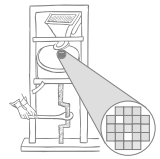
- Zusammengesetzter Typ
- Besteht aus ein oder mehreren Elementen des *gleichen* Typs
  - Unterschied zu Record
- Zugriff über Index, nicht über Namen
- Statische Arrays haben feste, zur Compile-Zeit bekannte Abmessungen
- Dynamische Arrays haben zur Laufzeit variable Abmessungen

```
type Name = array 4 of Char;
var  me: Name;
var  full: array 2 of Name
```

me[0]	'l'
me[1]	'e'
me[2]	'i'
me[3]	'a'

full[0][0]	'h'
full[0][1]	'a'
full[0][2]	'n'
full[0][3]	's'
full[1][0]	'o'
full[1][1]	't'
full[1][2]	't'
full[1][3]	'o'

Arrays 2



A. Koch

Offensichtliche Darstellung

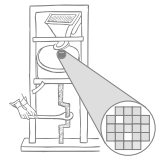
```
type Name = array 6 of Char;
var me : Name;
```

- size[`Name`] = 6 * size[`Char`] = 6 Worte
- address[`me[0]`] = address[`me`]
- address[`me[1]`] = address[`me`] + 1 * size[`Char`]
- address[`me[i]`] = address[`me`] + i * size[`Char`]

Kommentare

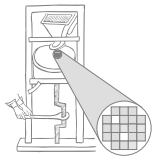
- Annahme hier: Indizes beginnen bei 0 (C, Java)
- `i` nicht notwendigerweise konstant
  ➥Adressberechnung zur Laufzeit

Dynamische Arrays 1



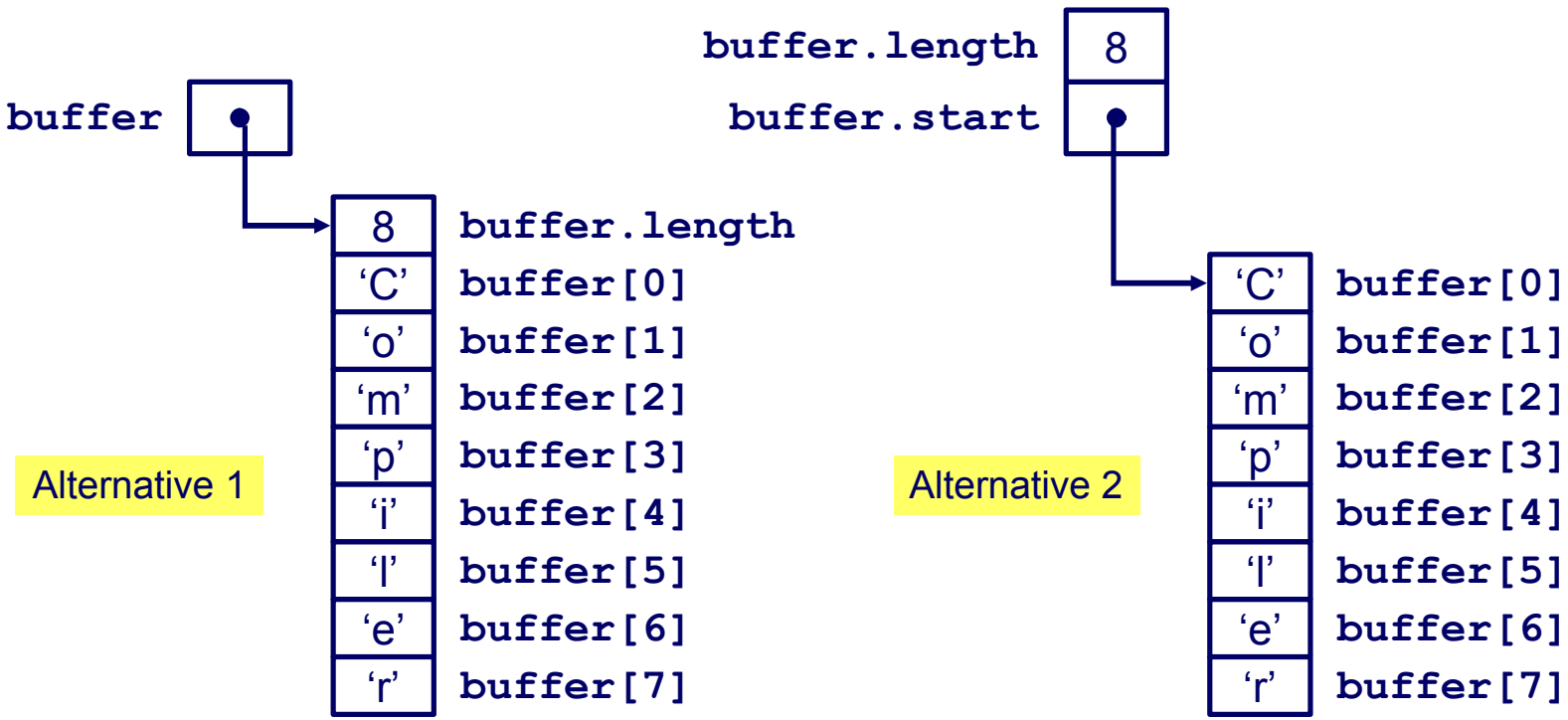
- Grundsätzlich wie statische Arrays
- Aber Abmessungen erst zur Laufzeit bekannt
  - Möglicherweise sogar variabel
- Indirekte Darstellung über Deskriptor
  - Adresse des ersten Elements
  - Abmessungen
- Speicher wird zur Laufzeit angefordert ($\longrightarrow$ Heap)

Dynamische Arrays 2

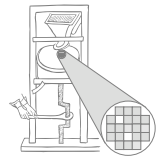


A. Koch

```
char[] buffer;
buffer = new char[len];
```



Rekursive Typen



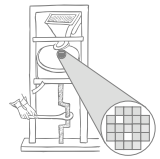
Referenziert sich selbst in seiner eigenen Definition

- Rekursiver Typ *T* hat Komponenten vom Type *T*

```
class IntList {
  int head;
  IntList tail;
}
```

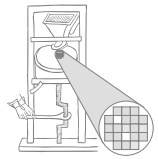
➥In der Regel nur über Zeiger

Auswertung von Ausdrücken



- Beispiel: `a*a + 2*a*b* - 4*a*c`
- Zugrundeliegende Maschine hat Instruktionen für Addition, Multiplikation, (Division), ...
- ...fast immer: Rechnen mit zwei Operanden
  ➥Abarbeiten in Teilausdrücken
- Wie mit Zwischenergebnissen verfahren? Wo abspeichern?
  - Registermaschine: In Registern (nicht ganz einfach ...)
  - Stack-Maschine: Post-Fix Auswertung auf Stack (einfach!)
- Virtuelle Maschine für Triangle TAM ist Stackmaschine

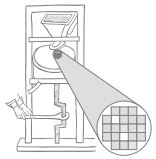
Typische Stack-Maschine



A. Koch

Instr.	***Meaning***
`STORE` *a*	**Pop** the top value off the stack and **store** it at **address** *a*.
`LOAD` *a*	**Fetch** a value from **address** *a* and push it on to the stack.
`LOADL` *n*	**Push** the **literal value** *n* onto the stack.
`ADD`	**Replace** the **two** top values on the top by their **sum**.
`SUB`	**Replace** the **two** top values on the top by their **difference**.
`MUL`	**Replace** the **two** top values on the stack by their **product**.

Beispielauswertung




```
d := a*a + 2*a*b - 4*a*c;
```

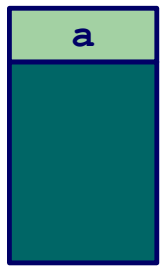
```
LOAD  a
LOAD  a
MUL
LOADL 2
LOAD  a
MUL
LOAD  b
MUL
ADD
LOADL 4
LOAD  a
MUL
LOAD  c
MUL
SUB
STORE d
```

```
STORE a
LOAD  a
LOADL n
ADD
SUB
MUL
```

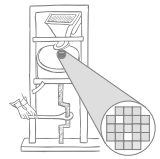
```
d := a*a + 2*a*b
```

```
LOAD  a
LOAD  a
MUL
LOADL 2
LOAD  a
MUL
LOAD  b
MUL
ADD
LOADL 4
LOAD  a
MUL
LOAD  c
MUL
SUB
STORE d
```

...



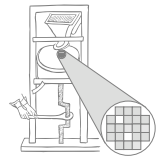
Typische Register-Maschine 1



Sehr schnelle Speicherelemente direkt im Prozessor

- Für Zwischenergebnisse etc.
- In der Regel 8/16/32/64b breit
- Begrenzte Anzahl, üblicherweise 4. . . 32 direkt verwendbar

Typische Register-Maschine 2



A. Koch

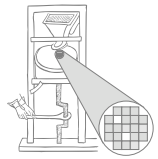
Instr.	*Meaning*
`STORE Ri a`	**Store** the value in `Ri` into memory location `a`.
`LOAD Ri a`	**Load** the value on memory location `a` into `Ri`.
`MULT Ri x`	**Multiply** the values in `Ri` and `x` and store the result in `Ri` (overwriting the old value).
`ADD Ri x`	**Subtract** the value in `x` from `Ri` and store the result in `Ri`.
`...`	

`x` Register `Ri`, oder eine Adresse `a`, oder ein literaler Wert `L`

Nicht immer so allgemein verwendbar, häufig Einschränkungen

- Nur bestimmte Register für bestimmte Operationen
- Nicht alle Arten von Operanden für aller Operationen

Beispielauswertung



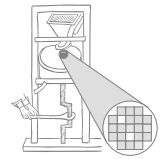
- Code für Registermaschine ist effizient.
- Compilierung ist aber komplexer
  - Verwaltung (Allokation) von Registern
  - Speichere Zwischenergebnisse in Registern
  - Problem: Endlich viele Register! Was, wenn Ausdruck komplizierter (zuviele Zwischenergebnisse)?

Beispiel:

```
d := a*a + 2*a*b - 4*a*c;
```

```
LOAD  R1 a    ; R1: a
MULT  R1 a    ; R1: a*a
LOAD  R2 2    ; R2: 2
MULT  R2 a    ; R2: 2*a
MULT  R2 b    ; R2: 2*a*b
ADD   R1 R2   ; R1: a*a+2*a*b
LOAD  R2      ; R2: 4
MULT  R2 a    ; R2: 4*a
MULT  R2 c    ; R2: 4*a*c
SUB   R1 R2   ; R1: a*a + ...
STORE R1 d    ; store result
```

Speicherverwaltung



Speicher auf der Zielmaschine

Datenspeicher

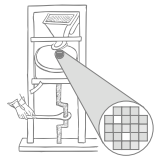
- Beispielsweise: Stack oder Heap
- Adressierbare Elemente: 8/16/32/64b Worte

Programmspeicher

- Variable Instruktionslänge (x86)
- Feste Instruktionslänge (RISC)
- Organisation weniger wichtig für Compiler
- Ausnahmen: Embedded Systems, virtueller Speicher (Linker)

➥Computerarchitektur (von-Neumann vs. Harvard, NUMA, COMA, . . .)

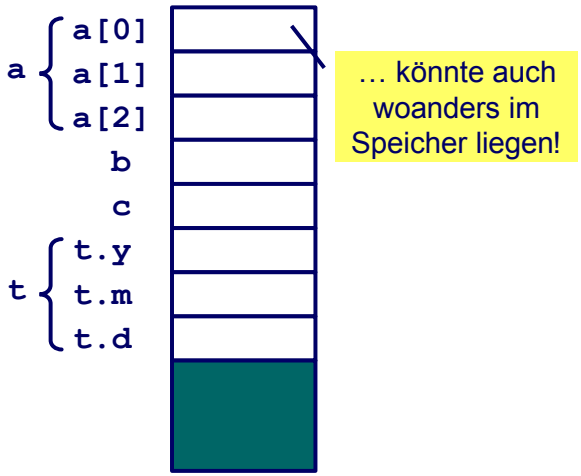
Statische Speicherverwaltung 1



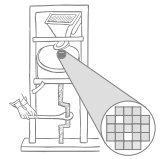
Globale Variablen: Existieren über gesamte Programmlaufzeit

- Compiler kann bereits Speicherbedarf jeder Variable berechnen
- Damit kann jeder Variable passender Speicher zugewiesen (alloziert) werden
- Nun bekannt: Adresse jeder Variable im Speicher

```
let
   type Date = record
      y: Integer,
      m: Integer,
      d: Integer
   end;
   var a: array 3 of Integer;   3
   var b: Boolean;              1
   var c: Char;                 1
   var t: Date                  3
in
   ...
```



Statische Speicherverwaltung 2



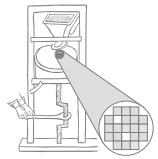
Einfache Vorgehensweise bei Vergabe von Adressen:
Bündige Anreihung

A. Koch

```
let
    var a : Boolean;
    var b : array 3 of Integer;
    var c : Char
in
    ...
```

- address[a] = 0 (relativ zum Beginn des Datenspeichers)
- address[b] = 1
- address[b[0]] = address[b] = 1
- address[b[1]] = address[b] + 1 = 2
- address[b[2]] = address[b] + 2= 3
- address[c] = 4

Verwaltung von Stapelspeicher 1



Lokale Variable v

A. Koch

- Ist im Innern eines Blocks definiert
  - Prozedur, Funktion, Let
- Existiert nur, während der Block aktiv ist
  - Beachte: “Existiert” bedeutet nicht auch “zugreifbar”
- Hat so eine begrenzte Lebensdauer

```
let
   var b: Boolean;
   var c: Char
in
   proc Y() ~
   let var d: Integer
   in  ...

   proc Z() ~
   let var f: Integer
   in ... Y(); ...
in begin
   ... Y(); ...; Z(); ...
end
```

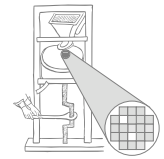
Globale Variablen; Lebensdauer: über das gesamte Programm

Lokale Variablen von Y: solange Prozedur Y aktiv ist

Lokale Variablen von Z: solange Prozedur Z aktiv ist

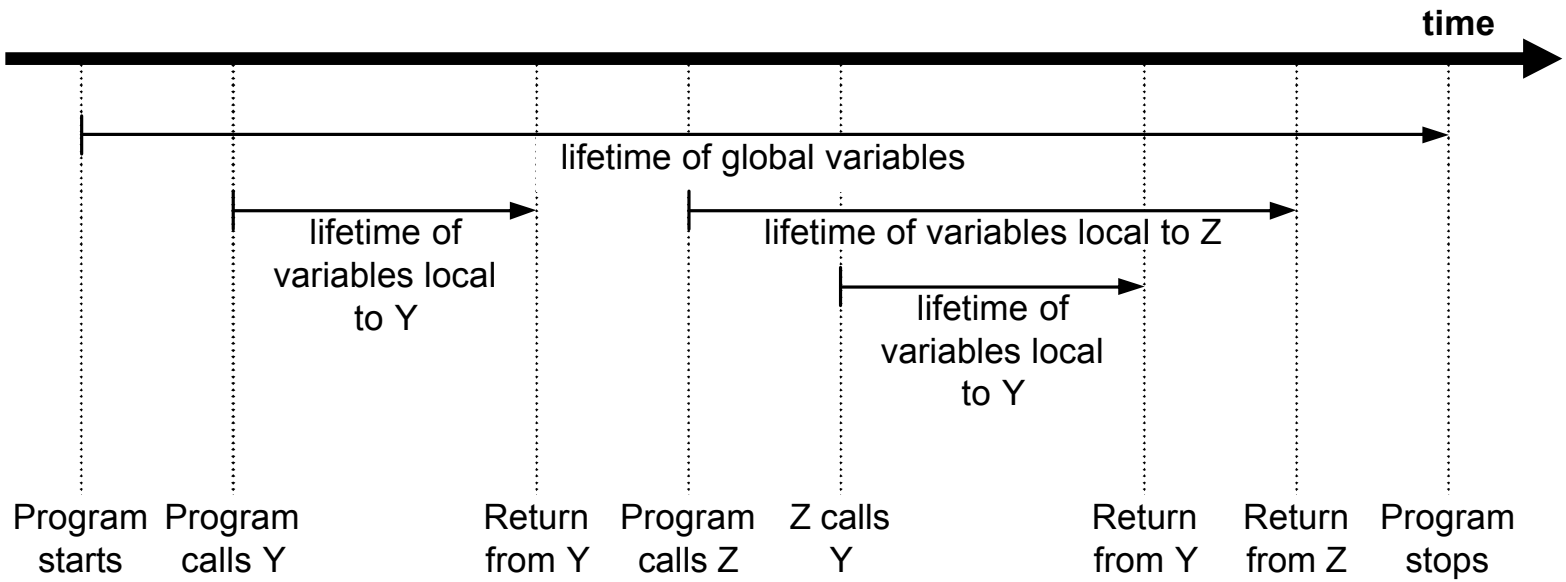
Prozedur kann gleichzeitig mehrfach aktiv sein (Rekursion), dann müssen auch mehrere Kopien der lokalen Variablen existieren.

Verwaltung von Stapelspeicher 2

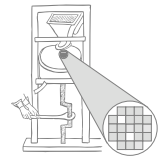


A. Koch

```
let
    ... ! global variables
    proc Y() ~
        let
            ... ! local variables for Y
        in
            ...
    proc Z() ~
        let
            ... ! local variables for Z
        in
            ... Y(); ...
in
    ... Y(); Z(); ...
```



Verwaltung von Stapelspeicher 3

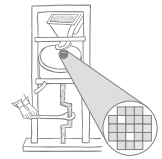


Beobachtungen

- Nur globale Variablen existieren über die gesamte Programmlaufzeit
- Lebenszeiten der lokalen Variablen sind hierarchisch verschachtelt

➥Handhabung via Stack

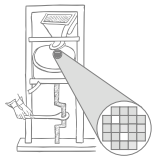
Verwaltung von Stapelspeicher 4



Organisationsstruktur: Stack Frame (Activation Record)

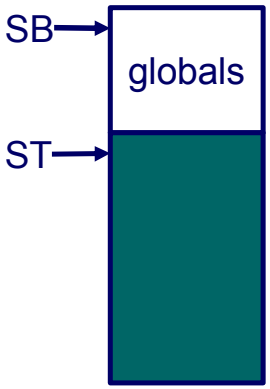
- Jede Prozedur hat einen Stack Frame, enthält
  - Lokale Variablen
  - Verwaltungsdaten
  - Aktuelle Parameter
- Stack Frame wird angelegt bei Prozeduraufruf
- . . . abgebaut (pop) nach Prozedurende

Beispiel Stapelspeicher

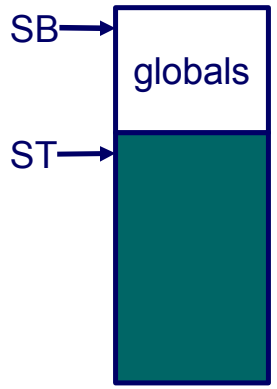


```
let ...
in proc Y() ~
   proc Z() ~ .. Y()
in .. Y();  Z();
```

after start



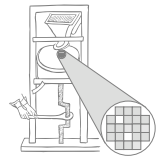
after start



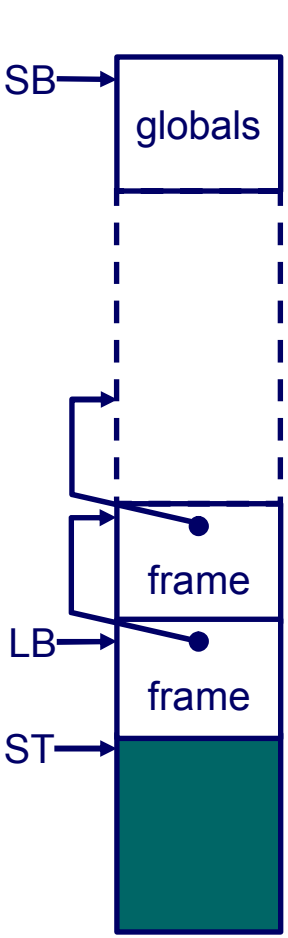
registers

SB	Stack Base
LB	Local Base
ST	Stack Top

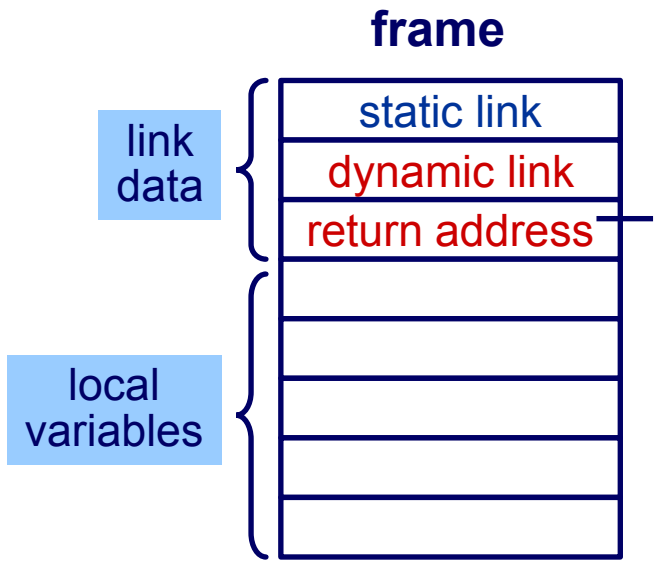
Verwaltung Stapelspeicher 1



A. Koch

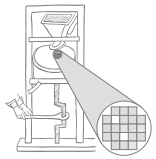


Dynamische Verkettung (dynamic link) ist Verweis auf den vorherigen Stack Frame (angelegt durch aufrufende Prozedur). Entspricht somit dem alten Wert von LB. Wird nach Ende der aktuellen Prozedur wieder hergestellt.



Rücksprungadresse nach Abarbeiten der Prozedur

Ablauf Prozeduraufruf 1

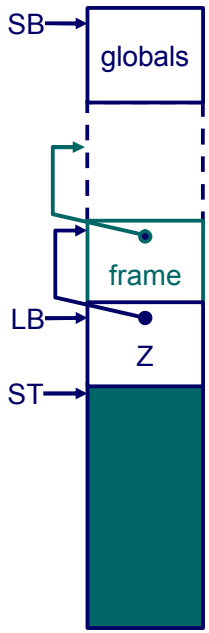


- Aufruf von `Y` aus `Z`

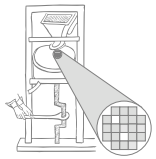
A. Koch

```
let ...
in
   proc Y() ~
   proc Z() ~
      in ... Y(); ...
in ...
```

R



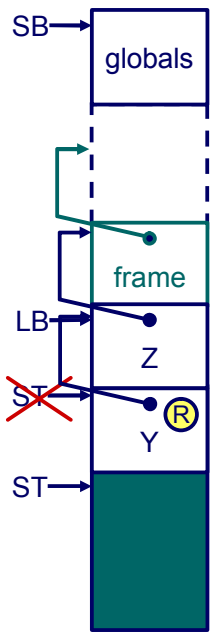
Ablauf Prozeduraufruf 2



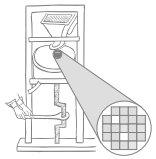
- Lege neuen Stack Frame für `Y` an
- Merke Rücksprungadresse
- Verkette dynamisch zu altem Frame über alten LB-Wert

A. Koch

```
let ...
in
   proc Y() ~
   proc Z() ~
      in ... Y(); ...
in ...
```



Ablauf Prozeduraufruf 3



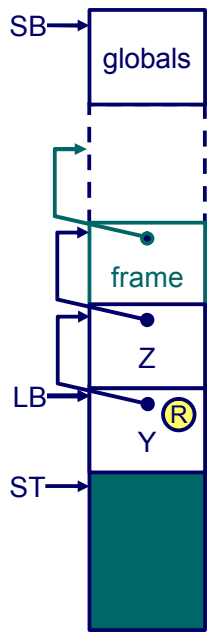
A. Koch

- Markiere neuen Frame als aktuellen durch Umsetzen von LB

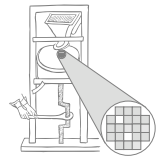
Nach Ende von `Y`

- Setze LB auf alten LB via dynamischer Verkettung zurück
- Setze ST auf alten Wert zurück
- Setze Ausführung bei Rücksprungadresse R fort

```
let ...
in
   proc Y() ~
   proc Z() ~
      in ... Y(); ...
in ...
```



Stapelverwaltung auf Maschinenebene



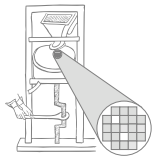
Instruktionen für Speicherzugriff

- LOAD d[reg] Lese Adresse d+reg, lege Inhalt auf Stapel ab
- STORE d[reg] Speichere obersten Stapelwert (TOS) an Adresse d+reg

Zugriff auf Variablen

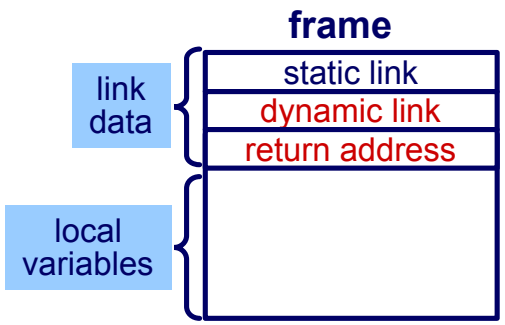
- Globale Variablen immer im Frame beginnend bei SB
  - Also: LOAD d[SB] und STORE d[SB]
- Lokale Variablen immer in Frame beginnend bei LB
- Also: LOAD d[LB] und LOAD d[LB]
- Vorsicht: Hier vereinfacht! ($\rightarrow$ statische Verkettung)

Beispiel Adressierung von Variablen



A. Koch

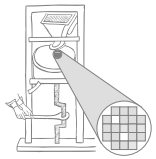
```
let
   var a: array 3 of Char;
   var b: Boolean;
   var c: Char;
in
   proc Y() ~
   let var d: Integer;
       var e: Integer
   in  ...

   proc Z() ~
   let var f: Integer
       var g: Char;
   in
      ... Y(); ...
in begin
   ... Y(); ...; Z(); ...
end
```



Wegen der Verwaltungsdaten (3 Worte) beginnen die lokalen Variablen erst bei Adresse 3 im Stack Frame

var	size	address
a	3	0[SB]
b	1	3[SB]
c	1	4[SB]
d	1	3[LB]
e	1	4[LB]
f	1	3[LB]
g	1	4[LB]

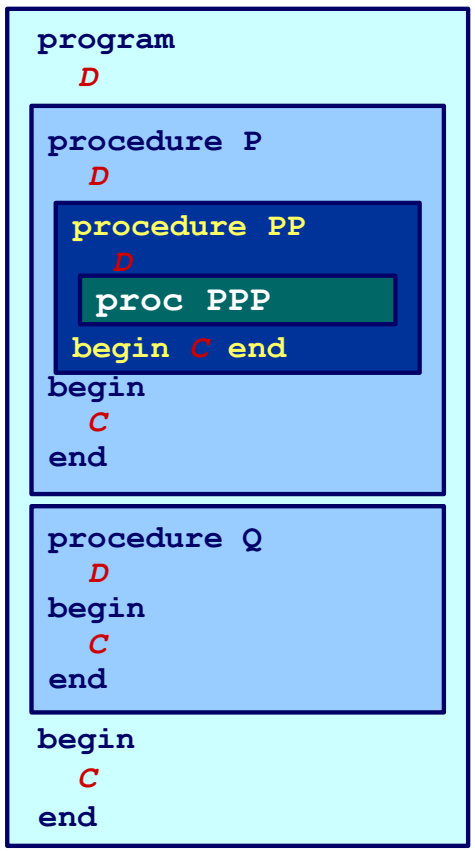
Statische Programmhierarchie



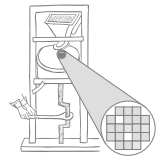
Verschachtelte Blockstruktur

- `PPP` hat Zugriff auf Variablen von `PPP`, `PP`, `P` und die globalen Variablen.
- Problem: Mit d[SB] und d[LB] können wir von `PPP` aus nur lokale Variablen von `PPP` und globale Variablen zugreifen
- Die anderen Variablen aus umschliessenden Prozeduren `PP` und `P` existieren aber noch auf dem Stapel!
- `P` und `PP` wurden vorher aktiviert
- Idee: Irgendwie hochhangeln und an die Daten kommen

A. Koch



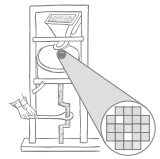
Statische Verkettung



- Verweis auf Frame der im Programmtext umschliessenden Prozedur
- Unterschied dynamische Verkettung
  - Hier Verweis auf Frame der aufrufenden Prozedur
- Dient dem Zugriff auf nicht-lokale Variablen

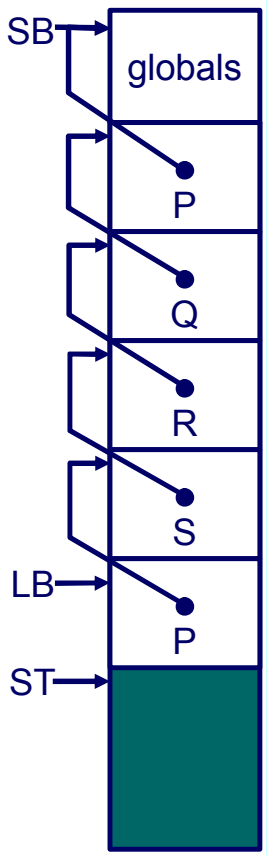
Wird nicht von allen Sprachen unterstützt und ist von zweifelhaftem Nutzen (siehe später).

Beispiel statische Verkettung

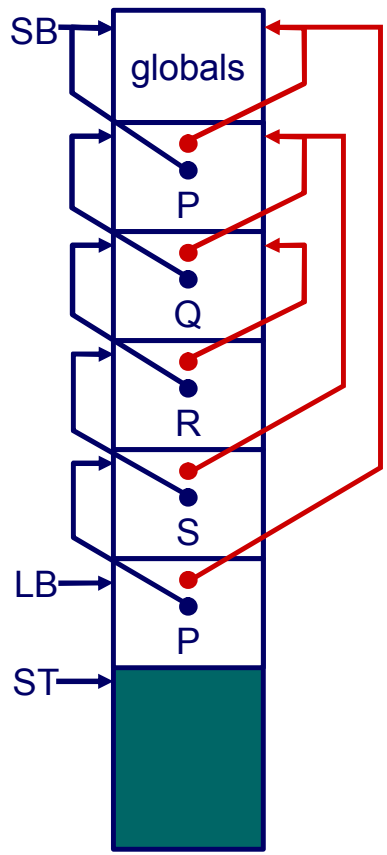


A. Koch

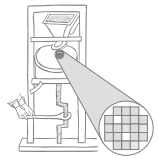
```
let
    ...
    proc P() ~
    let
        ...
        proc S() ~
        let ...
        in ... P(); ...

        proc Q() ~
        let
            ...
            proc R() ~
            let
            in ... S()

        in ... R(); ...

    in ... Q(); ...

in ... P(); ...
```




```
let
    ...
    proc P() ~
    let
        ...
        proc S() ~
        let ...
        in ... P(); ...

        proc Q() ~
        let
            ...
            proc R() ~
            let
            in ... S()

        in ... R(); ...

    in ... Q(); ...

in ... P(); ...
```



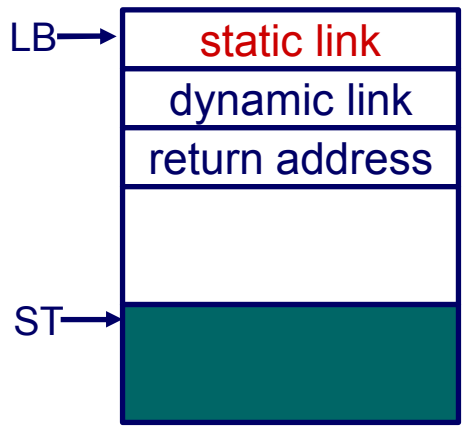
Organisation der statischen Verkettung 1



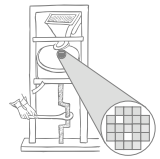
A. Koch

- Statische Verkettung ist hier 1. Wort des Stack Frame
- Wird durch LB referenziert
- Effekt:
  contents(LB) = umschliessende Stack Frame (von `R=Q`)
  contents(contents(LB))= noch weiter aussenliegende Stack Frame (von `R=P`)

```
let proc P()
in let proc Q()
   in let proc R()
...
```



Organisation der statischen Verkettung 2



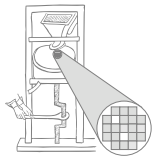
A. Koch

Realisierung durch sogenanntes Display

display registers

SB		Zeigt auf Frame mit globalen Variablen
LB		Zeigt auf oberste Frame R
L1	contents(LB)	Zeigt auf Frame R' umschließend R
L2	contents(L1)	Zeigt auf Frame R'' umschließend R'
L3	contents(L2)	Zeigt auf Frame R''' umschließend R''
L4	contents(L3)	Zeigt auf Frame R'''' umschließend R'''
. . .	...	...

Bestimmung der statischen Verkettung 1



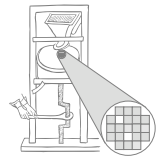
A. Koch

```
let ! level 0
  var a: Integer;
  proc P() ~
  let ! level 1
    var b: Integer;
    proc Q() ~
    let ! level 2
      var c: Integer;
      proc R() ~
      let ! level 3
        var d: Integer;
      in ...
    in ...
  in ...
in ...
```

```
let ! level 0
  var a: Integer;
  proc P() ~
  let ! level 1
    var b: Integer;
    proc Q() ~
    let ! level 2
      var c: Integer;
      proc R() ~
      let ! level 3
        var d: Integer;
      in ...
    in ...
  in ...
in ...
```

In R sind alle Va
d zugreifbar. A
bekannt:
Geltung

Bestimmung der statischen Verkettung 2

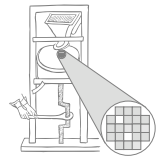


A. Koch

R sei Routine deklariert auf Ebene l, dann gilt für die statische Verkettung (hier SV)

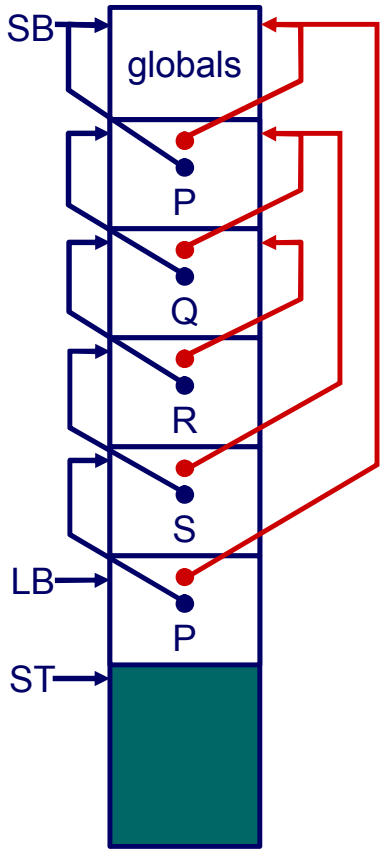
- Wenn $l = 0$ (R ist globale Routine)
  SV=SB $\rightarrow$ R sieht statisch nur globale Variablen
- Wenn $l > 0$ (R ist eingeschachtelt deklariert)
  - SV=LB vor Aufruf
    $\rightarrow$ wenn Aufruf von R aus Ebene l erfolgt
  - SV=L1 vor Aufruf
    $\rightarrow$ wenn Aufruf von R aus Ebene $l + 1$ erfolgt
  - SV=L2 vor Aufruf
    $\rightarrow$ wenn Aufruf von R aus Ebene $l + 2$ erfolgt
  - ... (bis L7 in TAM)

Beispiel: Bestimmung statische Verkettung

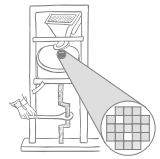


A. Koch

```
let
    ...
    proc P() ~
    let
        ...
        proc S() ~
        let ...
        in ... P(); ...

        proc Q() ~
        let
            ...
            proc R() ~
            let
            in ... S()

        in ... R(); ...

    in ... Q(); ...

in ... P(); ...
```



Anlegen von SV an Aufrufstelle



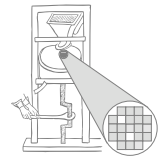
Wie SV für aufgerufene Routine setzen?
Nur Aufrufer kennt seine Ebene!

➥In Triangle/TAM: Parameter für `CALL`-Instruktion

Beispiel:
`S()` deklariert auf $l = 1$, Aufruf auf $l = 3$
$\longrightarrow$ `L2` verwenden

```
CALL (L2) S
```

Nicht-lokale Variablen



- Kompliziertere Compilierung
- Auch Laufzeitoverhead durch statische Verkettung
  - Komplizierterer Funktionsaufruf
  - Erhöhter Speicherbedarf

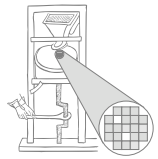
Lohnt sich das ganze überhaupt?

Beispiel Pascal

Art des Zugriffs	Relativer Anteil
Global	49%
Lokal	49%
Nicht-Lokal	2%

➥Nein, überflüssiger Aufwand!

Routinen 1

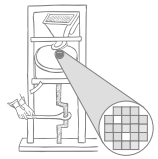


- Routinen sind Assembler-Äquivalent von Prozeduren und Funktionen einer Hochsprache (HLL)
- Wichtige Maschineninstruktionen

CALL r	Lege nächste Programmzeigeradresse auf Stapel und springe auf Adresse r
RETURN	Nehme einen Wert vom Stapel und springe dorthin

➥Basismechanismus für Routinenaufruf

Routinen 2



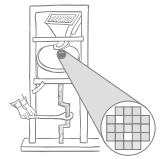
Weitere Aspekte bei der Abbildung von HLL-Mechanismen

- Aufruf einer Routine und Übergabe von Parametern
- Rückkehr von einer Routine und Rückgabe eines Ergebnisses
- Realisierung von statischen Verkettungen etc.

➥In Form eines Protokolls definieren (maschinenabhängig)

Oft vom Betriebssystem in Form eines Application Binary Interface (ABI) vorgegeben.

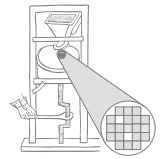
Routinenprotokoll 1



- Auch calling conventions genannt
- Für Stack-Maschinen häufig
  - Aufrufer legt Parameter auf Stapel (Reihenfolge?)
  - Routine wird aufgerufen und benutzt Parameterwerte
  - Aufgerufene Routine nimmt Parameter vom Stapel und ersetzt sie durch Rückgabewert

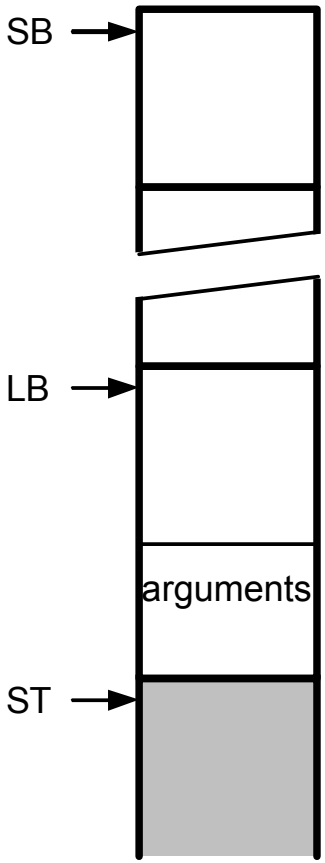
➥Beliebig viele Parameter übergebbar

Routinenprotokoll 2

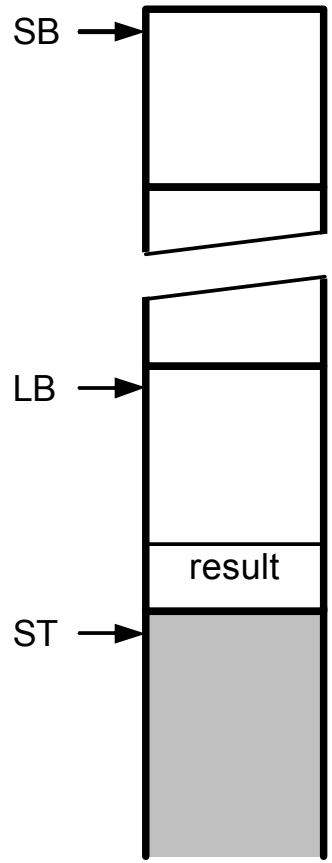


A. Koch

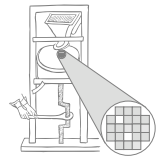
(1) Just before the call:



(2) Just after return:



Routinenprotokoll 3

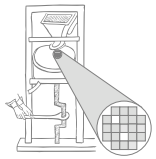


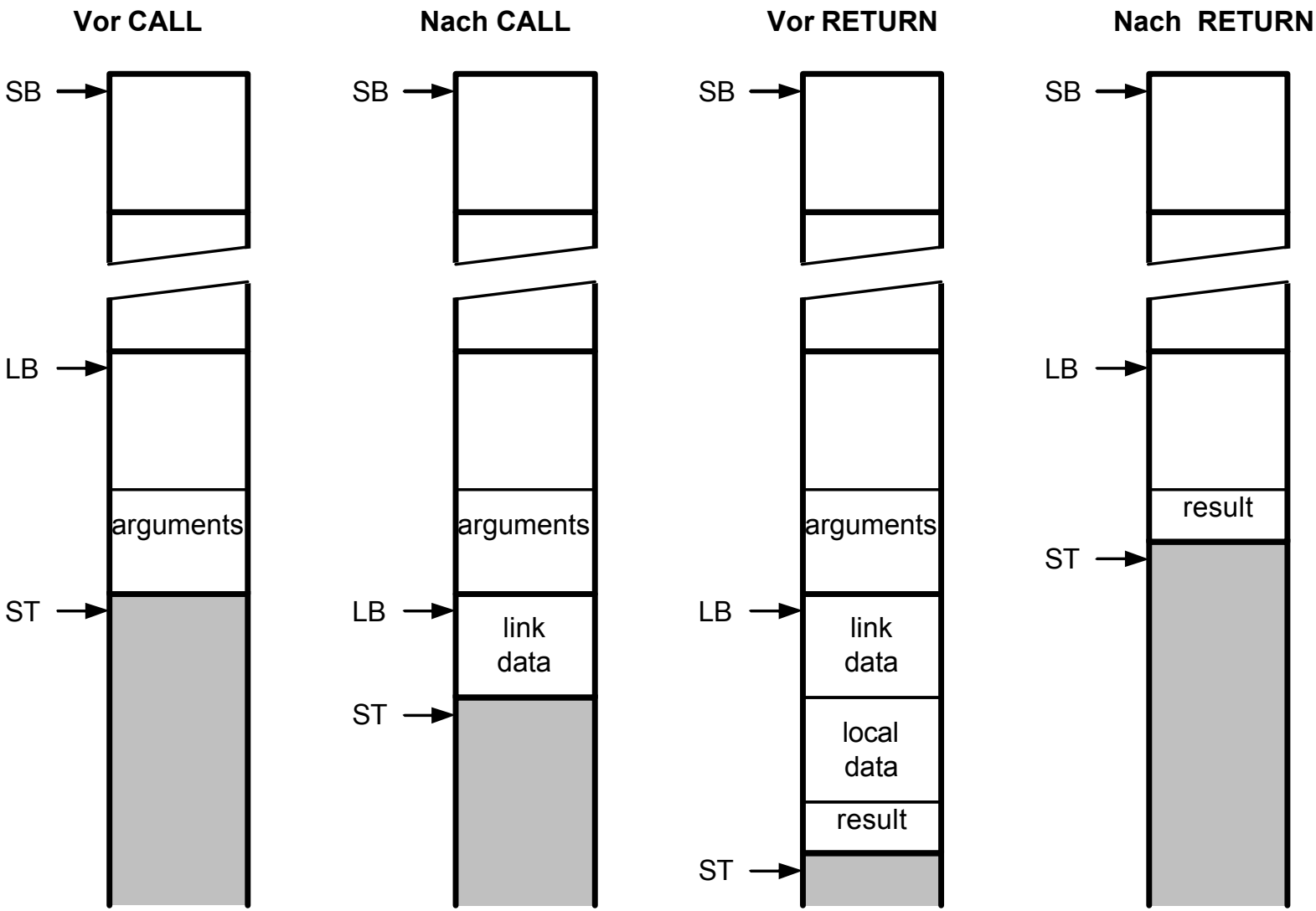
Relevante TAM Instruktionen

CALL (reg) addr ruft Routine an Adresse addr auf, verwendet den Wert in reg als statische Verkettung bei der Anlage eines neuen Frame

RETURN (n) d Sichert n Worte als Ergebnis vom Stack, entfernt den aktuellen Frame und d Parameter, setzt Ausführung nach Aufrufstelle fort, legt Ergebnis oben auf dem Stack ab

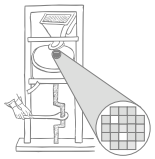
Routinenprotokoll 4





A. Koch

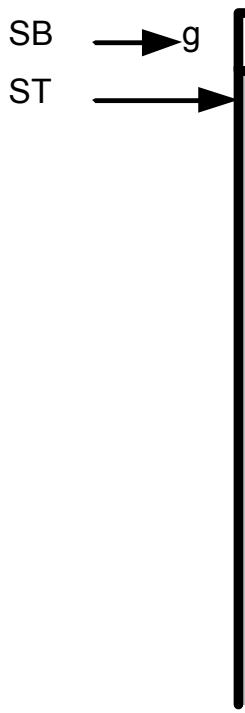
Routinenprotokoll 5



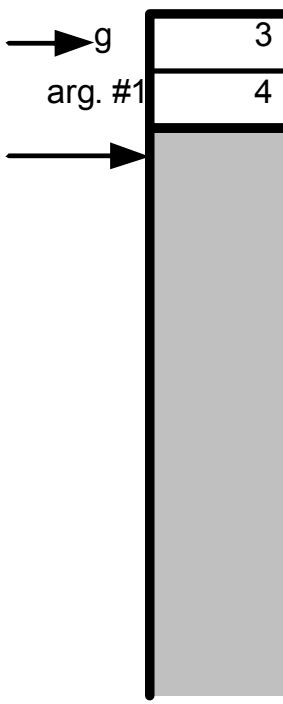
A. Koch

```
let var g: Integer;
   func F(m: Integer, n: Integer)
     : Integer ~ m*n ;
   proc W(i:Integer) ~
      let const s ~ i*i
      in begin
         putint(F(i,s));
         putint(F(s,s))
      end
in begin
   getint(var g);
   W(g+1)
end
```

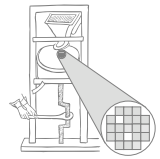
(1) Just after reading g:



(2) Just before cal



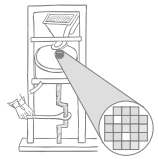
Routineprotokoll 6



Parameter (Argumente) zum Datenaustausch zwischen Aufrufer und Routine

- Aktuelle Parameter verwendet von Aufrufer bei Aufruf der Prozedur
- Formale Parameter innerhalb der Prozedur verwenden
  - Verhalten sich innerhalb der Prozedur wie lokale Variablen
- Eins-zu-eins Zuordnung von aktuellen und formalen Parametern

Übergabe von Werten

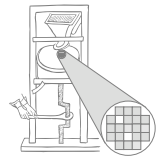


- Lege Wert der aktuellen Parameter auf Stack ab
- Liest Inhalte aus Variablen
- Effekt: Übergebe eine Kopie der Variable
- Zuweisungen innerhalb der Prozedur nicht im Aufrufer sichtbar

A. Koch

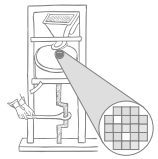
```
let
    proc sum(i:Integer, j:Integer) ~ begin
        i := i+j;
        putint(i);
    end
    var x: Integer
in begin
    x := 23; sum(x, 27)
end
```

Übergabe von Referenzen 1



- In Triangle durch Schlüsselwort `var`
  - Bei Deklaration und Aufruf der Prozedur!
- Übergebe die Variable selbst
  - Nicht nur ihren aktuellen Wert!
  - Änderungen werden auch außerhalb der aufgerufenen Prozedur sichtbar

Übergabe von Referenzen 2

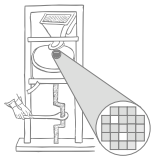


Wie implementieren?

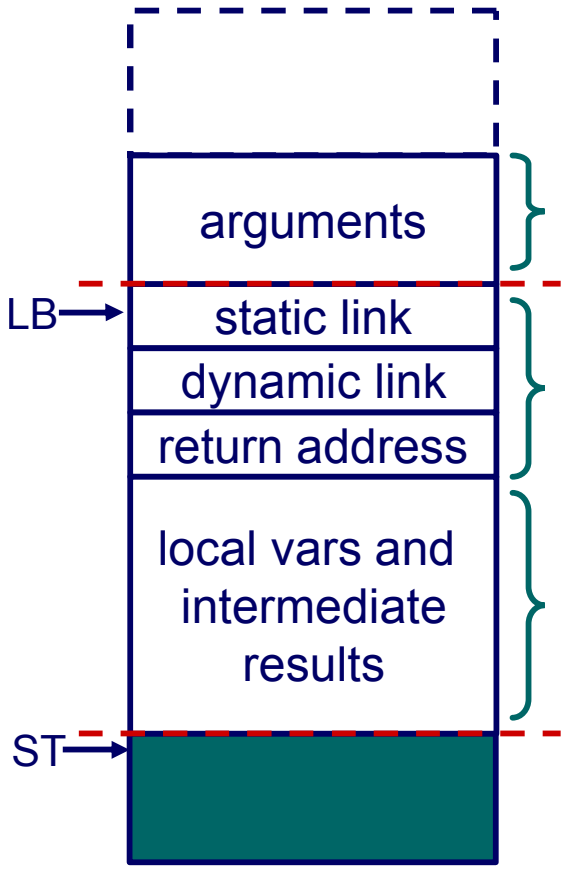
- Übergebe Adresse der Variable (als Zeiger)
- Aufgerufene Routine benutzt dann Indirektion um Wert abzurufen (dereferenziert Zeiger)

```
let proc S(var n:Integer, i:Integer) ~ n:=n+i;
    var today: record
        y:integer, m:Integer, d:Integer
    end
in begin
    b := {y~2003, m ~ 4, d ~ 10};
    S(var b.m, 6)
end
```

Erweiterung des Stack Frame



A. Koch



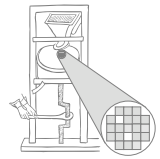
Parameter werden relativ zu LB mit negativen Offsets adressiert

Parameter für aktuelle Prozedur, dort abgelegt durch Aufrufer

Verkettungsdaten

Lokale Daten; Bereich wächst und schrumpft während der Ausführung

Implementierung der Aufrufkonventionen 1



A. Koch

```
let var g: Integer;
    func F(m: Integer, n: Integer)
      : Integer ~ m*n ;
    proc W(i:Integer) ~
       let const s ~ i*i
       in begin
          putint(F(i,s));
          putint(F(s,s))
       end
in begin
    getint(var g);
    W(g+1)
end
```

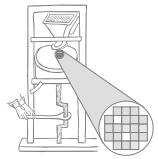
`g` ist var-Parameter

`g+1` ist Wert-Parameter

```
PUSH        1
LOADA       0[SB]
CALL        getint
LOAD        0[SB]
CALL        succ
CALL(SB)    W
POP         1
HALT
```

- *expand globals to make space for g*
- *push the address of g*
- *read an integer into g*
- *push the value of g*
- *add 1*
- *call W (using SB as the static link)*
- *remove globals*
- *end the program*

Implementierung der Aufrufkonventionen 2



A. Koch

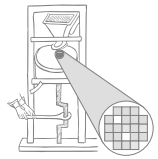
```
func F(m: Integer, n: Integer)
   : Integer ~ m*n ;
proc W(i:Integer) ~
   let const s ~ i*i
   in begin
      putint(F(i,s));
      putint(F(s,s))
   end
```

Parameter liegen direkt unter dem aktuellen Frame.

```
W:  LOAD        -1[LB]
    LOAD        -1[LB]
    CALL        mult
    LOAD        -1[LB]
    LOAD        3[LB]
    CALL(SB)    F
    CALL        putint
    LOAD        3[LB]
    LOAD        3[LB]
    CALL(SB)    F
    CALL        putint
    RETURN(0)   1

F:  LOAD        -2[LB]
    LOAD        -1[LB]
    CALL        mult
    RETURN(1)   2
```

- *push the value of i*
- *push the value of i*
- *multiply, the result will be the value of s*
- *push the value of i*
- *push the value of s*
- *call F (using SB as static link)*
- *write the value returned*
- *push the value of s*
- *push the value of s*
- *call F (using SB as static link)*
- *write the value returned*
- *return, replacing the 1-word argument by a 0-word result*
- *push the value of m*
- *push the value of n*
- *multiply*
- *return, replacing the 2-word argument pair by a 1-word result*

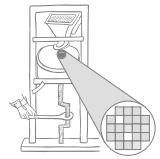
Sonderfall: Prozeduren/Funktionen als Parameter 1



- In Triangle, C, Modula, . . . , möglich
- Beispiel: Vergleichsfunktion an Sortierprozedur übergeben

```
let
   func twice(func doit(Integer x): Integer, i: Integer): Integer ~
       doit(doit(i));
   func double(Integer d) ~ d*2;
   var x: Integer
in begin
   x := twice(double, 10);
end
```

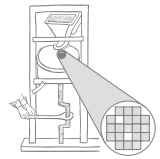
Sonderfall: Prozeduren/Funktionen als Parameter 2



Implementierung

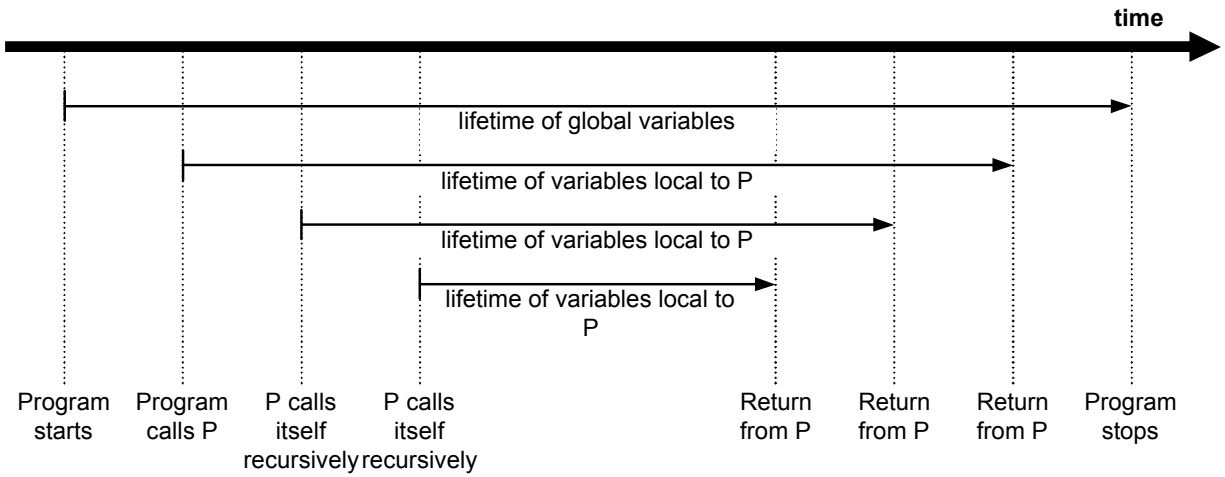
- Repräsentiere Funktion durch Paar (Startadresse, statische Verkettung)
- Sogenannte closure or Funktionsdeskriptor
- Aufruf dann über Closure
- TAM: Lege Closure auf Stack, dann `CALLI` zum Aufruf

Rekursion 1: Lebensdauern der Variablen

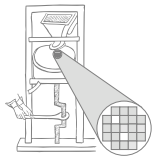


A. Koch

```
let
  proc P (i :  Integer, b: Integer) ~
    let
      const d ~ chr(i//b + ord('0'))
    in
      if i < b then
        put(d)
      else
        begin
          P(i / b, b);
          put(d)
        end;
  var n: Integer
in
  begin
    getint(var n);
    P(n, 8);
  end
```

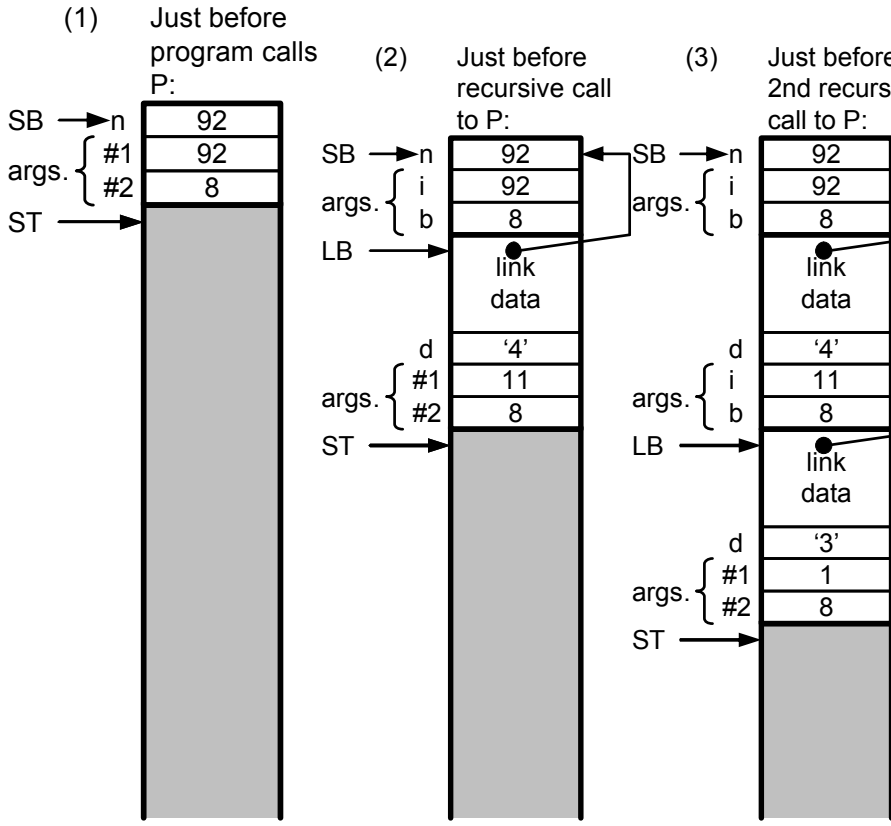


Rekursion 2

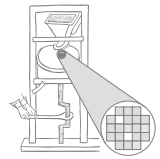


A. Koch

```
let
  proc P (i :  Integer, b: Integer) ~
    let
      const d ~ chr(i//b + ord('0'))
    in
      if i < b then
        put(d)
      else
        begin
          P(i / b, b);
          put(d)
        end;
  var n: Integer
in
  begin
    getint(var n);
    P(n, 8);
  end
```



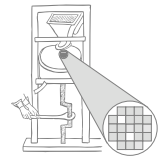
Anderer Ansatz der Speicherverwaltung



- Bisher Lebenszeit von Variablen gebunden an Geltungsbereiche
  - Auch verschachtelt (statische Verkettung)
- Reicht aber nicht immer!
- Häufig: Lebenszeiten unabhängig von Geltungsbereichen
- Beispiel: Datenstrukturen wie Listen, Bäume, etc.
  - Struktur lebt unabhängig von Prozeduren/Funktionen

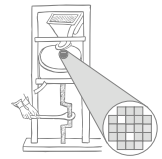
➥Braucht anderes Speicherverfahren als Stack

Speicherung auf Heap



- Auch Halde oder Haufen genannt
- . . . wir bleiben bei Heap
- Vorteil: Beliebige Lebenszeiten realisierbar
- Nachteil: Explizite Verwaltung durch Programm erforderlich
  - Pascal, C, C++
- Gilt nicht immer: Teilweise Automatisierung möglich
  - Java, Lisp, Smalltalk

Heap-Verwaltung

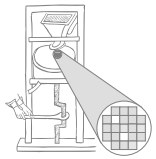


- Heap in der Regel im selben Speicher wie Stack
- Verhalten

  Stack wächst und schrumpft bei Blockeintritt/-austritt

  Heap wächst bei Anlegen neuer Variablen, schrumpft (?) bei Freigabe
- Idee: Heap und Stack an unterschiedlichen Enden des Adressraums beginnen
  - Wachsen aufeinander zu
  - Bei Zusammentreffen: Out-of-memory
- Normalerweise: Stack oben, Heap unten
- TAM: Stack unten, Heap oben

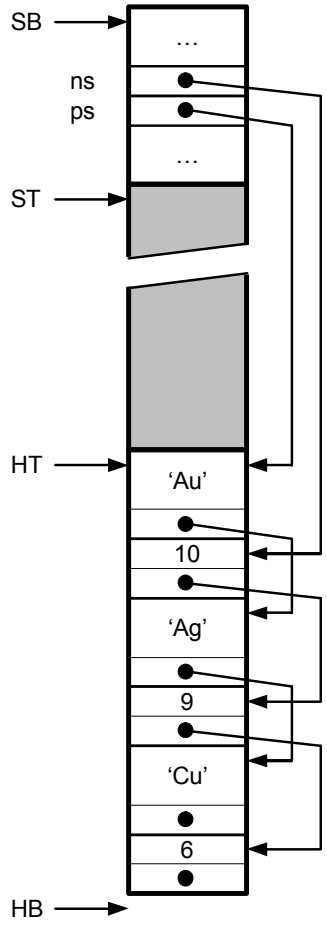
Beispiel: Heap 1



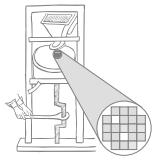
A. Koch

- Einfacher Fall: Nur neue Heap-Variablen anlegen.
- Beispiel hier:

```
var ns:IntList;
ps:SymList;
```

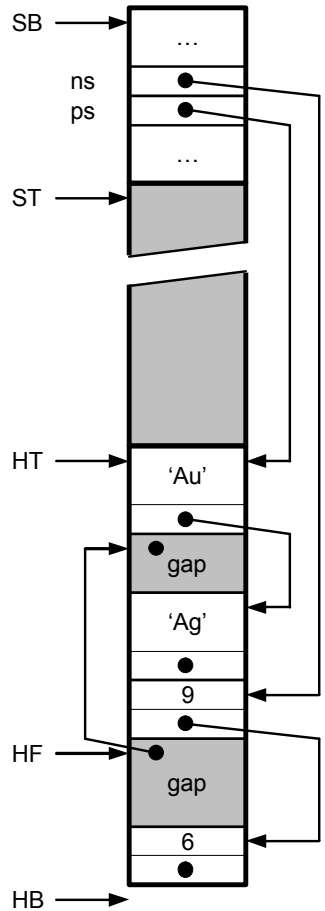


Beispiel: Heap 2

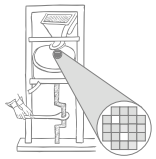


A. Koch

- Problem: Freigeben von Variablen
  - `IntList`: 10
  - `SymList`: 'Cu'
- Vorgehen hier: freien Platz merken (`HF` Liste)

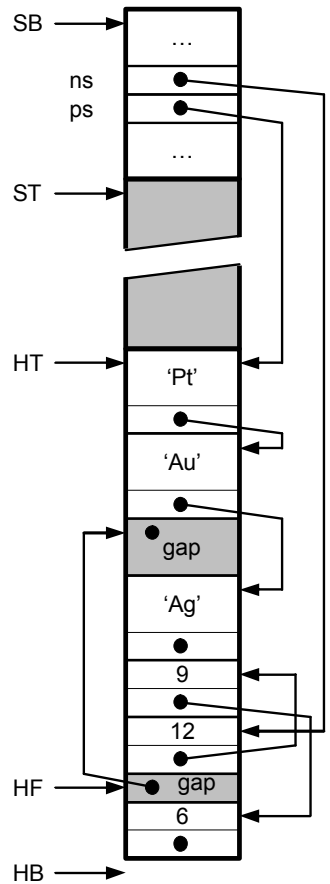


Beispiel: Heap 3

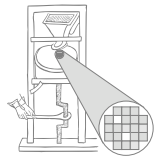


A. Koch

- Neue Heap-Variablen anlegen
  - `IntList`: 12
  - `SymList`: 'Pt'
- Freien Platz bevorzugt benutzen
- Ersten freien Platz verwenden
- Problem: Jetzt viele kleine Löcher in Heap (Fragmentierung)
- Heap wächst weiter



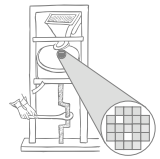
Heapverwaltung 1



Viele Ansätze zur Speicherzuteilung, ein Ansatz:

1. Finde genau passenden freien Speicherblock in `HF` und benutze ihn
2. Finde größeren freien Speicherblock in `HF` und benutze ihn teilweise
3. Vergrößere Heap in Richtung Stack um benötigten Platz
4. Falls nicht möglich: out-of-memory

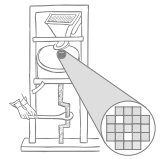
Heapverwaltung 2



Fragmentierung bekämpfen

- Verwende immer kleinsten passenden freien Speicherblock (immer sinnvoll?)
- Verschmelze benachbarte freie Speicherblöcke
- Kompaktiere Heap
  - Alles zusammenschieben
  - Problem: Alle Zeiger im Programm müssen aktualisiert werden
  - Teillösung: Doppelte Indirektion über Handles
    - Realisiert als Zeiger-auf-Zeiger
    - Programm operiert mit Handles, werden nicht beeinflusst
    - Zeiger in Handles werden durch Kompaktierung aktualisiert

Teilautomatische Speicherverwaltung 1

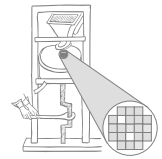


Idee: Automatisiere Freigabe von nicht mehr benutztem Speicher

A. Koch

- Garbage Collection
- In Java, Lisp, Smalltalk, ...
- Viele verschiedene Ansätze
- Ganz einfach: Mark-and-sweep
  1. Kennzeiche alle Elemente auf Heap als nicht erreichbar
  2. Gehen nun alle Variablen durch (auf Heap und auf Stack!)
  3. Falls Zeiger: Markiere referenzierten Heap-Block als erreichbar
  4. Trage alle unerreichbaren Speicherblöcke in `HF`-Liste ein

Teilautomatische Speicherverwaltung 2

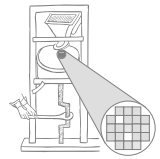


Probleme bei einfachem Mark-and-Sweep

- “Falls Zeiger”: Wie erkennen?
  - Zeiger besonders kennzeichnen
  - oder Buch über alle angelegten Zeiger führen
- Heap-Blöcke müssen ihre Größe kennen
- Was, wenn Zeiger mitten in Heap-Block hinein?

➥Kompliziert, nicht Compiler-spezifisch

Zusammenfassung



- Darstellung von Daten auf Maschinenebene
  - Primitive Typen
  - Zusammengesetzte Typen
- Triangle Abstract Machine
- Auswertung von Ausdrücken
  - Stack-Maschine, Register-Maschine
- Speicherverwaltung
  - Globale, lokale, nicht-lokale Variablen
- Aufrufkonventionen
  - Parameter- und Ergebnisübergabe
- Langlebige Daten
  - Auf Heap
  - Verwaltungstechniken