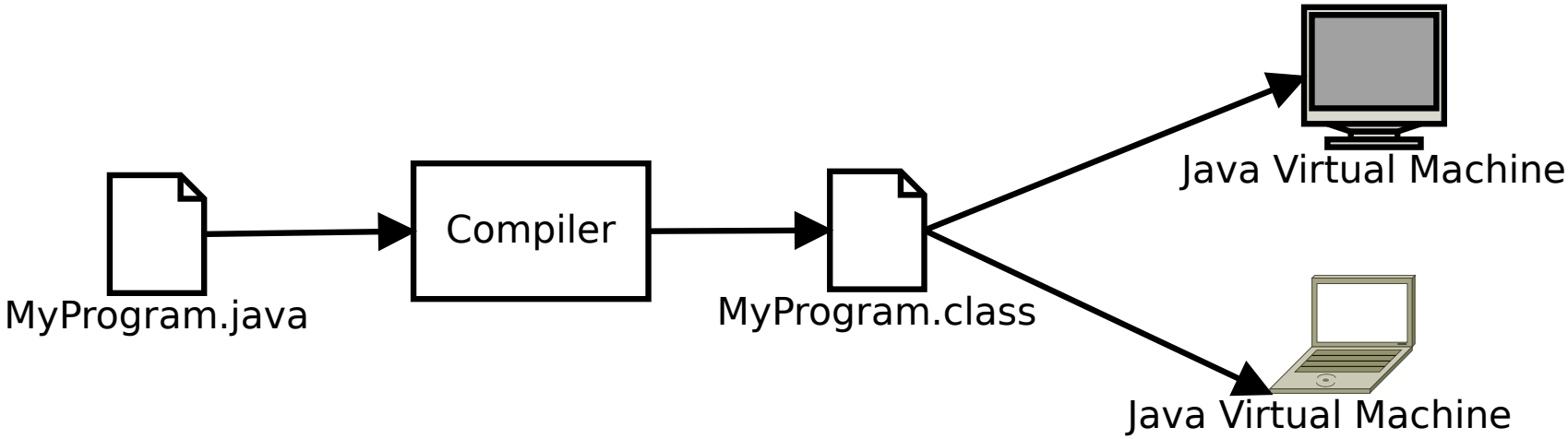
Kapitel 6

Transformation

Kapitel 6: Transformation

Java Technologie



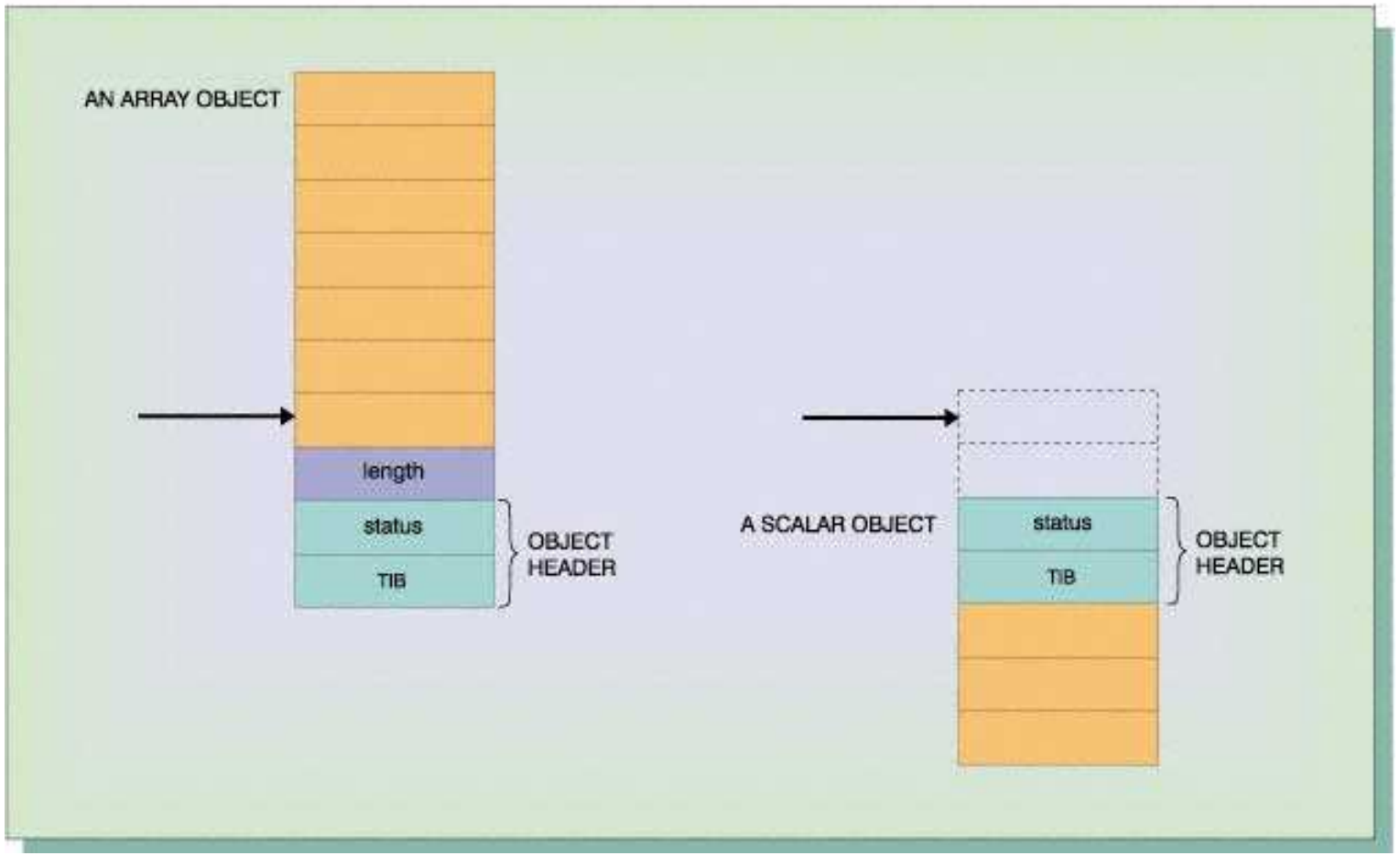
- Portable Zwischensprache: Bytecode
- Als virtuelle Maschine spezifiziert
- Umfangreiche Bibliothek
- Laufzeitsystem
- *The Java Virtual Machine Specification*
  `http://java.sun.com/docs/books/jvms/`

Virtuelle Maschine - Laufzeitsystem

- **Heap**: Speicher für Objektinstanzen. Getypt, automatische Speicherbereinigung (Garbage Collection), gemeinsamer Speicher für alle Threads.
- **Method Area**: Code für Methoden, nur lesbar.
- **Runtime Constant Pool**: Konstante Daten (Literale, Typinformationen, . . .)
- **Threads**: Je Thread:
  - **Program Counter**
  - **JVM Stack**: Activation Records (Stackframes)
  - **Native Method Stack**: Für Laufzeitsystem (meist in C/C++ geschrieben)
  - **Operandenstack**: zur Auswertung von (arithmetischen, logischen, ...) Ausdrücken

Objektlayout (1/2)

Figure 1 Layout of an array object and a scalar object in Jalapeño

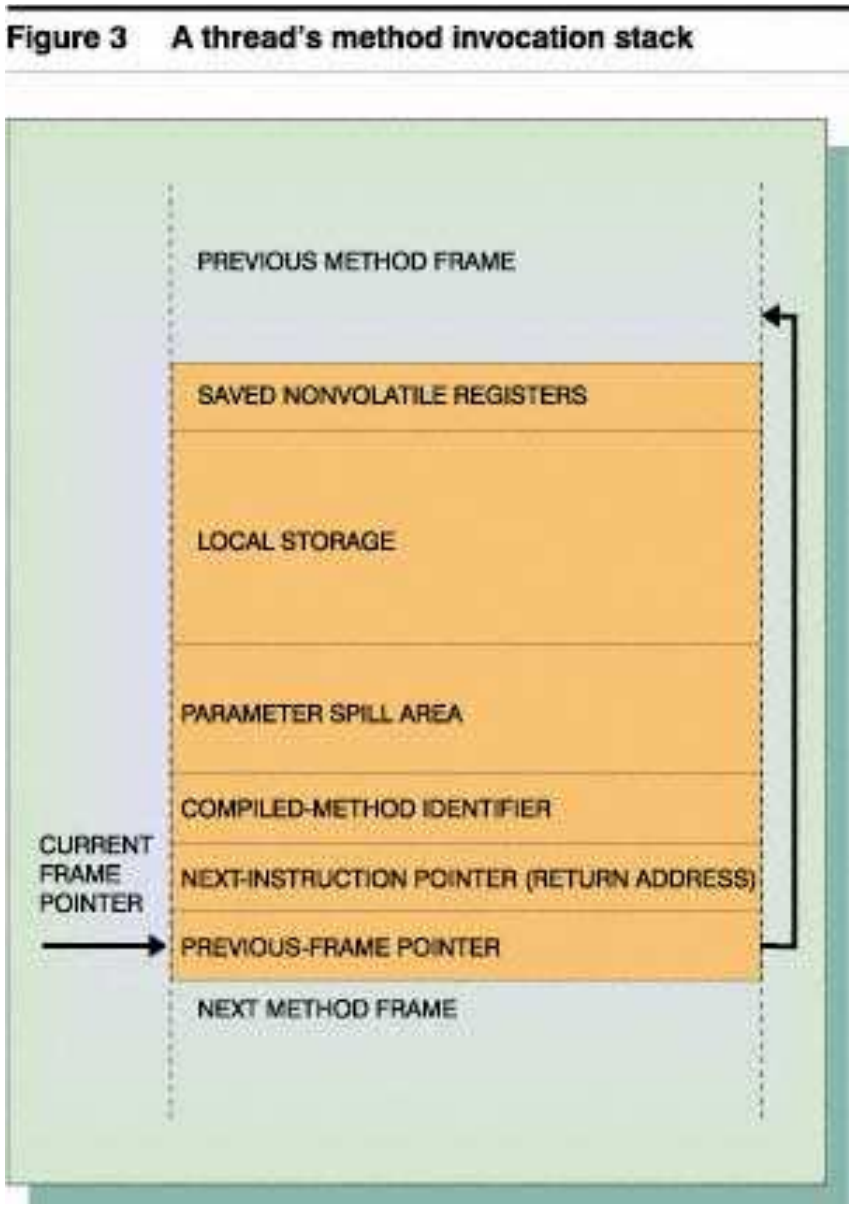


[0]Abbildungen aus: IBM Systems Journal, Vol 39 Nr 1

Objektlayout (2/2)

- ersten 12 Byte: Länge (für Arrays; für Nicht-Arrays nicht belegt)
- Status: Lock-Bits, Hash-Bits, Garbage-Collect-Bits
- TIB: Type Information Block = vptr. JVM enthält in vtable zusätzlich Klassendeskriptor (vgl. Reflection-Interface)
- Nullpointer-Zugriff erzeugt Hardware-Interrupt, da das length-Feld Offset -4 hat
- Typische JVMs opfern Speicher, um Performance zu gewinnen!

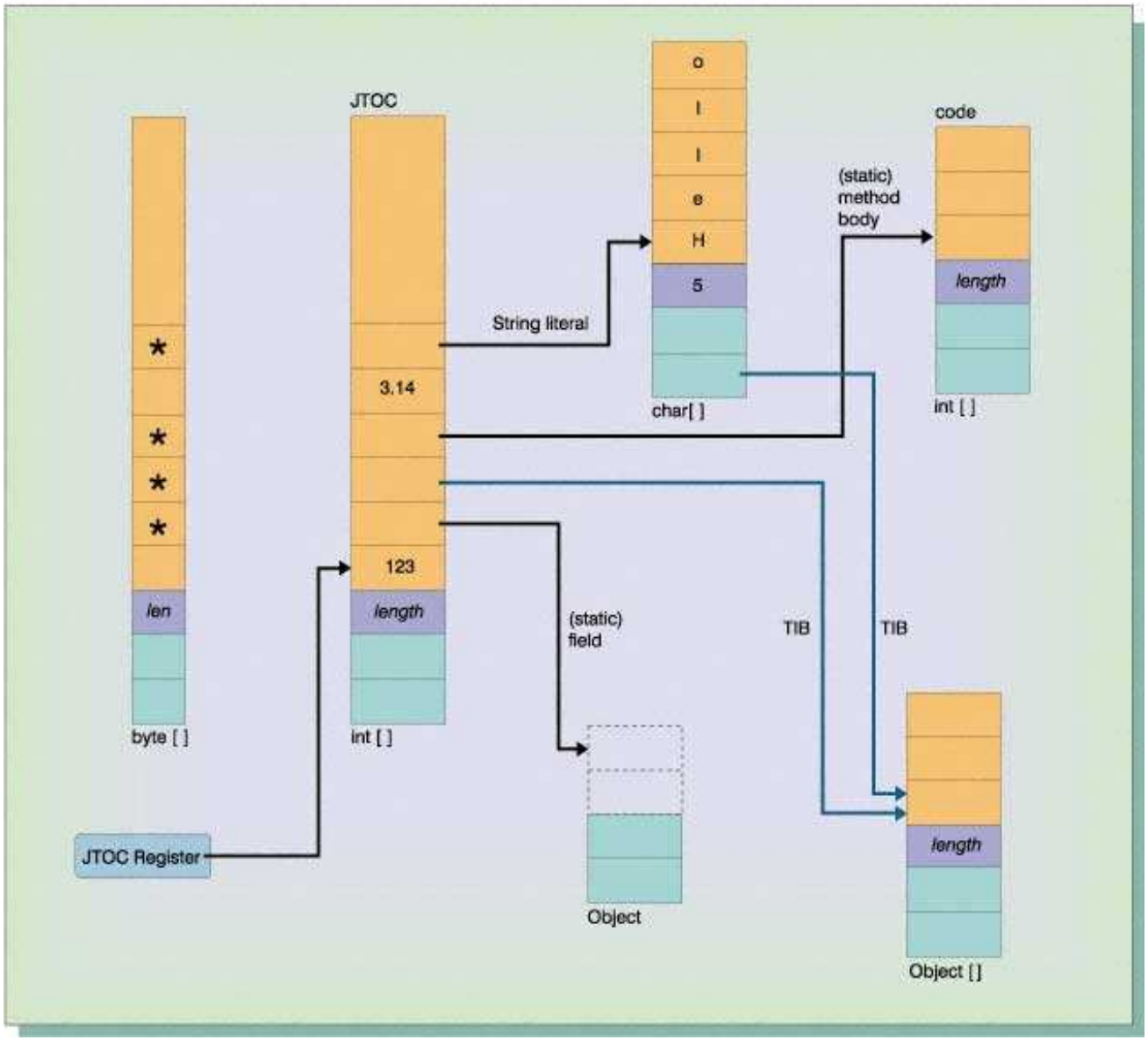
Aufbau des Activation Records



- analog zu C
- JVM-Operandenstack wird in Hardwareregistern + Spillarea realisiert

Globale JTOC

Figure 2 **The Jalepeño Table of Contents and other objects**

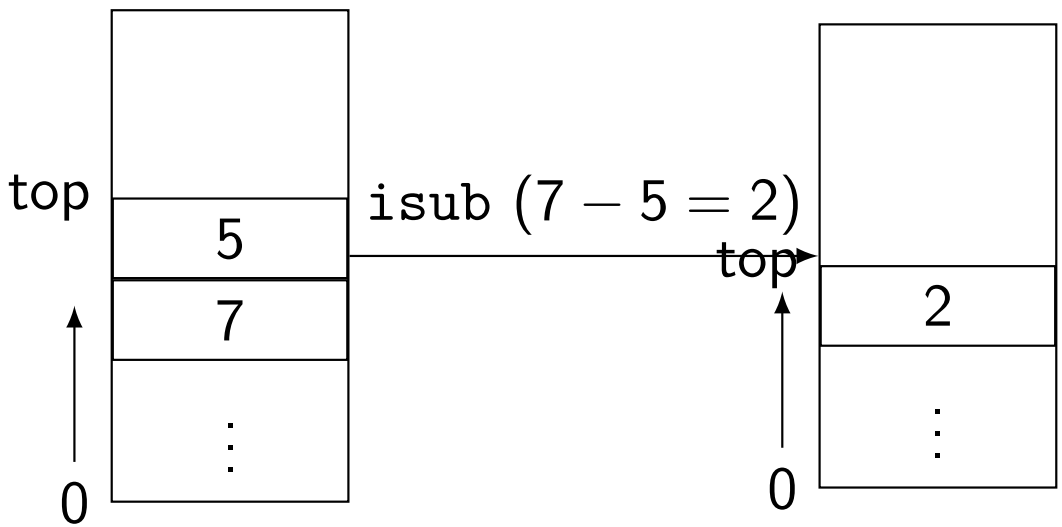


- Array mit Verweisen auf
  - Konstanten
  - Klassen-deskriptoren

Bytecode, Operandenstack

- Stackbasierter Bytecode: Operanden und Rückgabewerte liegen auf Operandenstack.
- Kürzere Befehlscodierung da Operanden und Ziele nicht explizit.
- Maximale Stackgröße pro Methode im `.class`-File angegeben.

Beispiel:



Instruktionen

- Typen bekannt aus Java
- Instruktionen explizit typisiert: `iadd` (`int`), `fadd` (`float`)
- Unäre Operatoren (z.B. Typkonversion) wirken auf Topstack
- Binäre Operatoren verknüpfen die beiden obersten und schreiben Ergebnis wieder auf den Stack.
- Ferner Lade/Speicherinstruktionen (push/pop).
- Alle Bytecodes kommen in verschiedenen typisierten Varianten.

Typisierte Bytecode-Varianten

opcode	*byte*	*short*	*int*	*long*	*float*	*double*	*char*	*reference*
Tipush	*bipush*	*sipush*						
Tconst			*iconst*	*lconst*	*fconst*	*dconst*		*aconst*
Tload			*iload*	*lload*	*fload*	*dload*		*aload*
Tstore			*istore*	*lstore*	*fstore*	*dstore*		*astore*
Tinc			*iinc*					
Taload	*baload*	*saload*	*iaload*	*laload*	*faload*	*daload*	*caload*	*aload*
Tastore	*bastore*	*sastore*	*iastore*	*lastore*	*fastore*	*dastore*	*castore*	*aastore*
Tadd			*iadd*	*ladd*	*fadd*	*dadd*		
Tsub			*isub*	*lsub*	*fsub*	*dsub*		
Tmul			*imul*	*lmul*	*fmul*	*dmul*		
Tdiv			*idiv*	*ldiv*	*fdiv*	*ddiv*		
Trem			*irem*	*lrem*	*frem*	*drem*		
Tneg			*ineg*	*lneg*	*fneg*	*dneg*		
Tshl			*ishl*	*lshl*				
Tshr			*ishr*	*lshr*				
Tushr			iushr	lushr				
Tand			*iand*	*land*				
Tor			*ior*	*lor*				
Txor			*ixor*	*lxor*				
i2T	*i2b*	*i2s*		*i2l*	*i2f*	*i2d*		
l2T			*l2i*		*l2f*	*l2d*		
f2T			*f2i*	*f2l*		*f2d*		
d2T			*d2i*	*d2l*	*d2f*			
Tcmp				*lcmp*				
Tcmpl					fcmpl	dcmpl		
Tcmpg					fcmpg	dcmpg		
if_TcmpOP			*if_icmpOP*					*if_acmpOP*
Treturn			*ireturn*	*lreturn*	*freturn*	*dreturn*		*areturn*

Beispiel: Ausdruck berechnen

```
void calc(int x, int y) {
        int z = 6;
        x = x + y * z;
}
```

Lokale Variablen:

[0]	[1]	[2]	[3]	[4]
this	x	y	z	...
0x?	7	5	?	...

Stack:

Befehl:

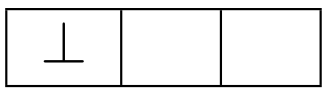
```
// Lade Konstante 1
bipush 6
// Schreibe in z
istore_3
// Lade y
iload_2
// Lade z
iload_3
// y * z
imul
// Lade x
iload_1
// x + (y * z)
iadd
// Speichere x
istore_1
```

Beispiel: Ausdruck berechnen

```
void calc(int x, int y) {
        int z = 6;
        x = x + y * z;
}
```

Lokale Variablen:

[0]	[1]	[2]	[3]	[4]
`this`	`x`	`y`	`z`	...
`0x?`	7	5	?	...

Stack: 

Befehl: 

```
// Lade Konstante 1
bipush 6
// Schreibe in z
istore_3
// Lade y
iload_2
// Lade z
iload_3
// y * z
imul
// Lade x
iload_1
// x + (y * z)
iadd
// Speichere x
istore_1
```

Beispiel: Ausdruck berechnen

```
void calc(int x, int y) {
        int z = 6;
        x = x + y * z;
}
```

Lokale Variablen:

[0]	[1]	[2]	[3]	[4]
`this`	`x`	`y`	`z`	...
`0x?`	7	5	?	...

Stack: | **6** | ⊥ | |

Befehl: `bipush 6`

```
// Lade Konstante 1
bipush 6
// Schreibe in z
istore_3
// Lade y
iload_2
// Lade z
iload_3
// y * z
imul
// Lade x
iload_1
// x + (y * z)
iadd
// Speichere x
istore_1
```

Beispiel: Ausdruck berechnen

```
void calc(int x, int y) {
        int z = 6;
        x = x + y * z;
}
```

Lokale Variablen:

[0]	[1]	[2]	[3]	[4]
`this`	`x`	`y`	`z`	...
`0x?`	7	5	6	...

Stack: | ⊥ | | |

Befehl: `istore_3`

```
// Lade Konstante 1
bipush 6
// Schreibe in z
istore_3
// Lade y
iload_2
// Lade z
iload_3
// y * z
imul
// Lade x
iload_1
// x + (y * z)
iadd
// Speichere x
istore_1
```

Beispiel: Ausdruck berechnen

```
void calc(int x, int y) {
        int z = 6;
        x = x + y * z;
}
```

Lokale Variablen:

[0]	[1]	[2]	[3]	[4]
`this`	`x`	`y`	`z`	...
`0x?`	7	5	6	...

Stack: | **5** | ⊥ | |

Befehl: `iload_2`

```
// Lade Konstante 1
bipush 6
// Schreibe in z
istore_3
// Lade y
iload_2
// Lade z
iload_3
// y * z
imul
// Lade x
iload_1
// x + (y * z)
iadd
// Speichere x
istore_1
```

Beispiel: Ausdruck berechnen

```
void calc(int x, int y) {
        int z = 6;
        x = x + y * z;
}
```

Lokale Variablen:

[0]	[1]	[2]	[3]	[4]
`this`	`x`	`y`	`z`	...
`0x?`	7	5	6	...

Stack: | **6** | 5 | $\perp$ |

Befehl: `iload_3`

```
// Lade Konstante 1
bipush 6
// Schreibe in z
istore_3
// Lade y
iload_2
// Lade z
iload_3
// y * z
imul
// Lade x
iload_1
// x + (y * z)
iadd
// Speichere x
istore_1
```

Beispiel: Ausdruck berechnen

```
void calc(int x, int y) {
        int z = 6;
        x = x + y * z;
}
```

Lokale Variablen:

[0]	[1]	[2]	[3]	[4]
`this`	`x`	`y`	`z`	...
`0x?`	7	5	6	...

Stack: | **30** | ⊥ | |

Befehl: `imul`

```
// Lade Konstante 1
bipush 6
// Schreibe in z
istore_3
// Lade y
iload_2
// Lade z
iload_3
// y * z
imul
// Lade x
iload_1
// x + (y * z)
iadd
// Speichere x
istore_1
```

Beispiel: Ausdruck berechnen

```
void calc(int x, int y) {
        int z = 6;
        x = x + y * z;
}
```

Lokale Variablen:

[0]	[1]	[2]	[3]	[4]
`this`	`x`	`y`	`z`	...
`0x?`	7	5	6	...

Stack: | **7** | 30 | $\perp$ |

Befehl: `iload_1`

```
// Lade Konstante 1
bipush 6
// Schreibe in z
istore_3
// Lade y
iload_2
// Lade z
iload_3
// y * z
imul
// Lade x
iload_1
// x + (y * z)
iadd
// Speichere x
istore_1
```

Beispiel: Ausdruck berechnen

```
void calc(int x, int y) {
        int z = 6;
        x = x + y * z;
}
```

Lokale Variablen:

[0]	[1]	[2]	[3]	[4]
`this`	`x`	`y`	`z`	...
`0x?`	7	5	6	...

Stack: | **37** | ⊥ | |

Befehl: `iadd`

```
// Lade Konstante 1
bipush 6
// Schreibe in z
istore_3
// Lade y
iload_2
// Lade z
iload_3
// y * z
imul
// Lade x
iload_1
// x + (y * z)
iadd
// Speichere x
istore_1
```

Beispiel: Ausdruck berechnen

```
void calc(int x, int y) {
        int z = 6;
        x = x + y * z;
}
```

Lokale Variablen:

[0]	[1]	[2]	[3]	[4]
`this`	`x`	`y`	`z`	...
`0x?`	**37**	5	6	...

Stack: | ⊥ | | |

Befehl: `istore_1`

```
// Lade Konstante 1
bipush 6
// Schreibe in z
istore_3
// Lade y
iload_2
// Lade z
iload_3
// y * z
imul
// Lade x
iload_1
// x + (y * z)
iadd
// Speichere x
istore_1
```

Weitere Bytecodes

- Objekterzeugung, Memberzugriff: new, newarray, anewarray, multianewarray, getfiled, putfiled, getstatic, putstatic
- Arrayzugriff: Taload, Tastore, arraylength
- Typetest: instanceof, ckeckcast
- bedingte Sprünge: ifeq, iflt, ifnull, if_icmpeq, if_acmpeq, ..., tableswitch, lookupswitch
- unbedingte Sprünge: goto, goto_w
- Methodenaufruf: invokevirtual, invokeinterface, invokespecial, invokestatic, Treturn
- Exceptions: athrow, jsrm jsr_w, ret
- Synchronisation: monitorenter, monitorexit

Beispiel: Fibonacci-Berechnung

Java-Code:

```
static void calcSequence() {
  long fiboNum = 1;
  long a = 1;
  long b = 1;

  for (;;) {
    fiboNum = a + b;
    a = b;
    b = fiboNum;
  }
}
```

Bytecode:

```
 0 lconst_1 // Push long constant 1
 1 lstore_0 // Pop long into local vars 0 & 1:
            // long a = 1;
 2 lconst_1 // Push long constant 1
 3 lstore_2 // Pop long into local vars 2 & 3:
            // long b = 1;
 4 lconst_1 // Push long constant 1
 5 lstore 4 // Pop long into local vars 4 & 5:
            // long fiboNum = 1;
 7 lload_0 // Push long from local vars 0 & 1
 8 lload_2 // Push long from local vars 2 & 3
 9 ladd // Pop two longs, add them, push result
10 lstore 4 // Pop long into local vars 4 & 5:
            // fiboNum = a + b;
12 lload_2 // Push long from local vars 2 & 3
13 lstore_0 // Pop long into local vars 0 & 1: a = b;
14 lload 4 // Push long from local vars 4 & 5
16 lstore_2 // Pop long into local vars 2 & 3:
            // b = fiboNum;
17 goto 7 // Jump back to offset 7: for (;;) {}
```

Methodenaufrufe

1. Bezugsobjekt auf den Stack (falls nicht **static**)
2. Parameter auf den Stack
3. **invokevirtual** / **invokestatic** ausführen:
   Folgendes passiert vor / nach dem Aufruf automatisch:
   1. Array für Parameter und lokale Variablen anlegen (Größe ist angegeben)
   2. Returnadresse (Program Counter+1) und alten Framepointer sichern
   3. Neuen Framepointer setzen
   4. **this** Pointer und Parameter vom Stack ins Parameter Array kopieren
   5. Zu Methodenanfang springen und Code ausführen
   6. Returnwert auf den Stack
   7. Alten Framepointer setzen und zur Returnadresse springen
4. Returnwert vom Stack holen und weiterverarbeiten

Beispiel: Methodenaufruf

```
int bar() {
        return foo(42);
}

int foo(int i) {
        return i;
}
```

Konstantenpool

#2	Method	#3.#16
#3	class	#17
#11	Asciz	foo
#12	Asciz	(I)I
#16	NameAndType	#11:#12
#17	Asciz	Test

```
int bar();
        aload_0
        bipush 42
        invokevirtual #2
        ireturn

int foo(int);
        iload_1
        ireturn
```

Deskriptoren

Namen von Klassen, Feldern und Methoden müssen einem festgelegtem Schema entsprechen. (siehe JVMS 4.3)

- Klassennamen: `java.lang.Object` → `Ljava/lang/Object;`
- Typen: `int` → `I`, `void` → `V`, `boolean` → `Z`
- Methoden: `void foo(int, Object)` → `foo(ILjava/lang/Object;)V`
  Deskriptor: (*Parametertypen*) *Rückgabetyp*
  Identifiziert über "*Name* × *Deskriptor*"
- Felder: `boolean b` → `b:Z`
  Identifiziert nur über "*Name*"
- Konstruktoren: Name ist `<init>`, Static Initializer `<clinit>`

Objekt erzeugen & initialisieren

1. Objekt anlegen → Speicher reservieren
2. Objekt initialisieren → Konstruktor aufrufen

Hinweis: Jede Klasse braucht einen Konstruktor (Defaultkonstruktor)!

```
class Test {
  Test foo() {
    return new Test();
  }
}
```

#1	java/lang/Object.<init>()V
#2	Test
#3	Test.<init>()V

```
Test();
        aload_0
        invokespecial #1;
        return

Test foo();
        new #2;
        dup
        invokespecial #3;
        areturn
```

Beispiel: Array anlegen und darauf zugreifen

```
public void arr() {
  int[] array = new int[10];
  array[7] = 42;
}
```

```
bipush 10 // Konstante 10
newarray int // array anlegen vom Typ int
astore_1 // in variable array (var 1) speichern
aload_1 // variable array (var 1) laden
bipush 7 // Konstante 7
bipush 42 // Konstante 42
iastore // Wert (42) auf array index (7)
                    // von array("array") schreiben
return // Aus Funktion zurueckkehren
```

Beispiel: Auf Feld zugreifen

```
class field {
  public field field;
  public void setNull() {
    field = null;
  }
}
```

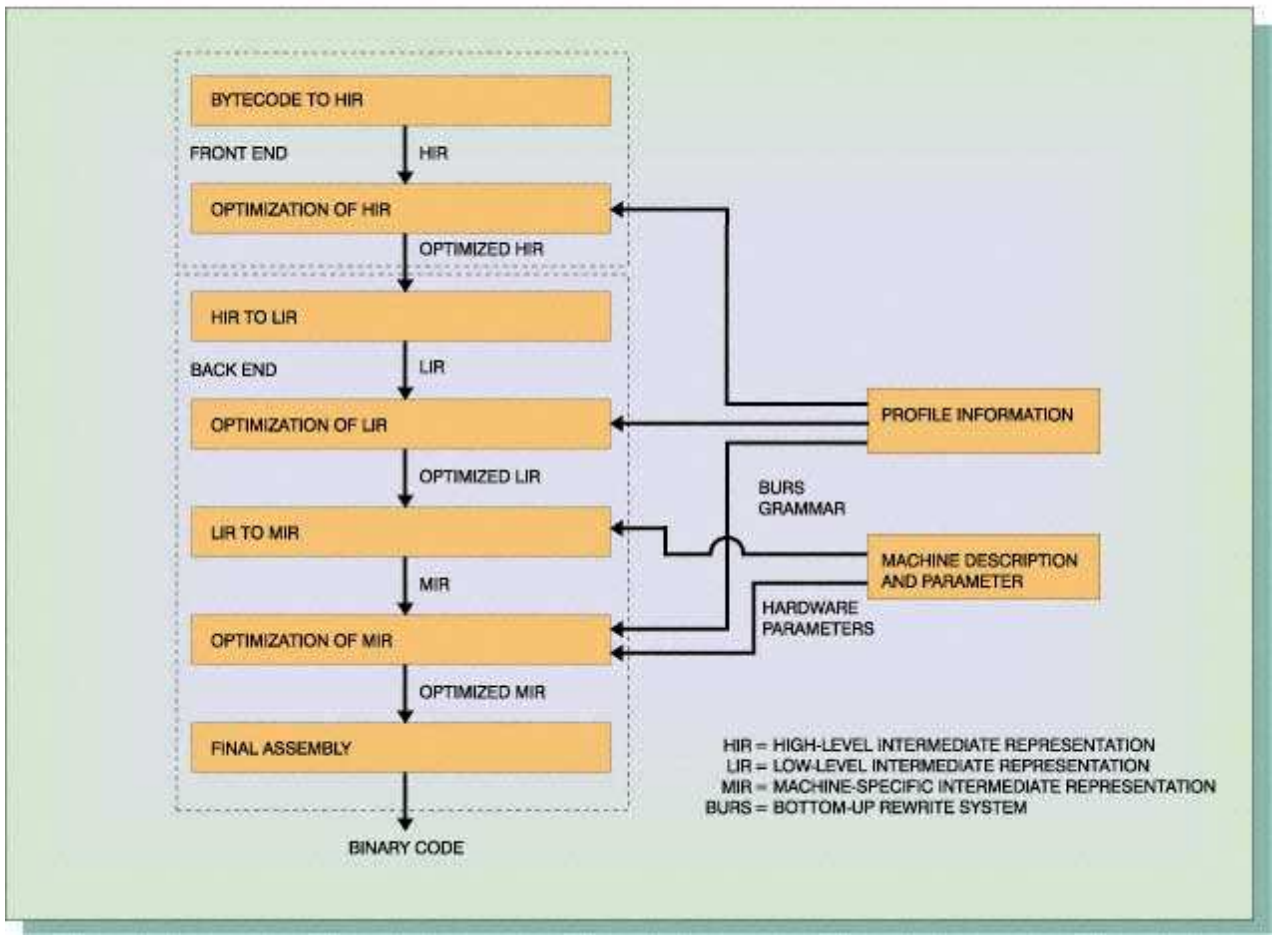
```
aload_0 // Parameter0 (this) auf Stack
aconst_null // null-Referenz auf den Stack
putfield field:Lfield; // Schreibe Wert (null)
  // auf Feld field:Lfield; von Objekt(this)
return // Aus Funktion zurueckkehren
```

Compilervariationen

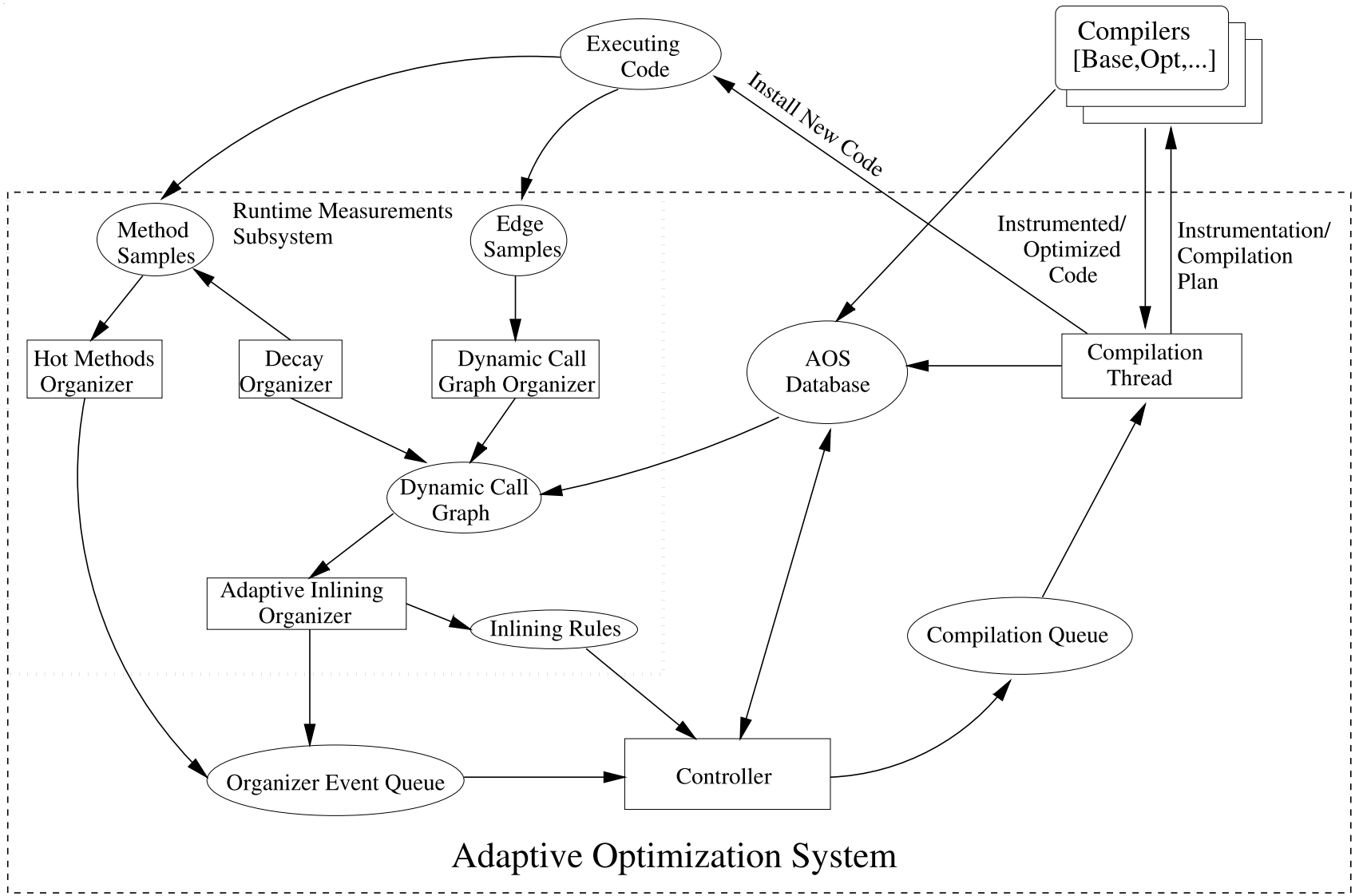
- nur Bytecode-Generierung, JVM in C
- Just-in-time: Maschinencode für Methoden, sobald sie das erstemal aufgerufen werden
- Adaptive Compilation (Jalapeno): JVM größtenteils in Java, Generierung von Maschinencode und Optimierung aufgrund dynamischem Profiling

Grobaufbau des Jalapeno-Compilers: (Bytecode nach Maschinencode)

Figure 4 Jalapeño's optimizing compiler



Struktur der Compileroptimierung (Hotspot-Technologie)

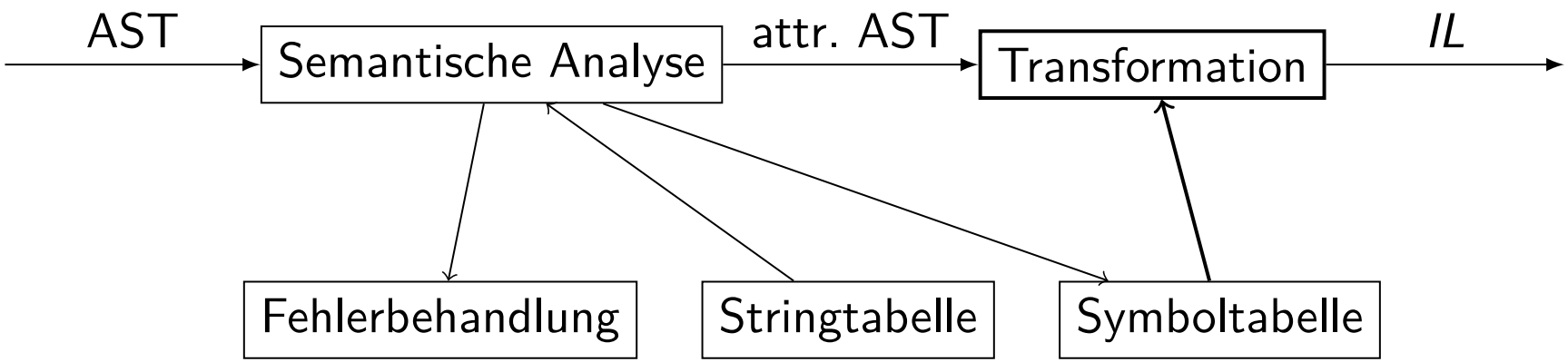


Struktur der Compileroptimierung (Hotspot-Technologie)

- Es wird sowohl Häufigkeit von Methodenausführungen als auch Zahl der Aufrufe $A.f() \rightarrow B.g()$ gemessen
- Falls Schwellwert überschritten: Maschinencode; für Kanten im dynamischen Call Graph: Inlining
- Schwellwerte sind heuristisch adaptiv; alte Werte „verfaulen“; Datenbank mit alten Messwerten

Kapitel 6: Transformation

Eingliederung in den Compiler



Zerlegung der Synthesephase

1. **Abbildung**, d.h. **Transformation/Optimierung**: Code für abstrakte Zielmaschine *ZM* (ohne Ressourcenbeschränkung) herstellen und optimieren, Repräsentation als Zwischensprache *IL*
2. **Codeerzeugung**: Transformation *IL*$\rightarrow$ symbolischer Maschinencode; unter Beachtung von Ressourcenbeschränkungen
3. **Assemblieren/Binden**: symbolische Adressen auflösen, fehlende Teile ergänzen, binär codieren

Abstraktion der Zwischensprache

Problem Abstraktionsniveau:

- Compiler vs. Laufzeitsystem
- Portabilität des Compilers vs. Effizienz der übersetzten Programme

Beispiele:

- E/A-Routinen gewöhnlich im Laufzeitsystem
- Indexrechnung wird vollständig übersetzt
- Prozeduraufrufe werden gewöhnlich auf parameterlose Prozedurrufe reduziert
- Speicherzuteilung und Speicherbereinigung gewöhnlich im Laufzeitsystem
- Ausnahmebehandlung mit Unterstützung des Laufzeitsystems

2 Klassen von Zwischensprachen

1. Code für **Kellermaschine mit Heap**, z.B. Pascal-P, ..., JVM, CLR (.net)
   - Ablaufsteuerung mit (bedingten) Sprüngen aufgelöst
   - Datentypen und Operationen auf Daten entsprechen weitgehend der *QM*, zusätzlich Umfang und Ausrichtung im Speicher berücksichtigen
2. Code für **RISC-Maschine mit unbeschränkter Registerzahl** und (stückweise) linearem Speicher
   - Ablaufsteuerung mit (bedingten) Sprüngen aufgelöst
   - Datentypen entsprechen Zielmaschine einschl. Umfang und Ausrichtung im Speicher
   - Operationen entsprechen Zielmaschine (Laufzeitsystem berücksichtigen!)
   - **aber** noch keine konkreten Befehle, keine Adressierungsmodi
   - Vorteil: fast alle Prozessoren auf dieser Ebene gleich

Kellermaschinencode gut für (Software-)Interpretation, schlecht für explizite Codeerzeugung, RISC-Maschine: umgekehrt

3 Unterklassen

Im Fall „Code für RISC-Maschine mit unbeschränkter Registerzahl“ drei Darstellungsformen:

1. **keine explizite Darstellung**: *IL* erscheint nur implizit bei direkter Codeerzeugung aus AST: höchstens lokale Optimierung, z.B. Einpaßübersetzung
2. **Tripel-/Quadrupelform:** Befehle haben schematisch die Form $t_1 := t_2 \ \tau \ t_3$ oder $m : t_1 := t_2 \ \tau \ t_3$ analog auch für Sprünge
3. **SSA-Form** (Einmalzuweisungen, static single assignment): wie Tripelform, aber jedes t_i kann nur einmal zugewiesen werden (gut für Optimierung)

Programmstruktur der *IL*

Gesamtprogramm eingeteilt in Prozeduren,
Prozeduren unterteilt in Grundblöcke oder erweiterte Grundblöcke

- Grundblock: Befehlsfolge maximaler Länge mit: wenn ein Befehl ausgeführt wird, dann alle genau einmal, also
  - Grundblock beginnt mit einer Sprungmarke,
  - enthält keine weiteren Sprungmarken
  - endet mit (bedingten) Sprüngen, enthält sonst keine weiteren Sprünge
  - Unterprogrammaufrufe zählen nicht als Sprünge!
- Erweiterter Grundblock: wie Grundblock, aber kann mehrere bedingte Sprünge enthalten: ein Eingang, mehrere Ausgänge

Aufgaben der Transformationsphase

Definition der abstrakten Zielmaschine (Speicherlayout, Befehlssatz, Laufzeitsystem), dann:

- Typabbildung
- Operatorabbildung
- Ablaufabbildung

Kapitel 6: Transformation

Einfache Datentypen

- Aufzählungstyp (`enum`)
- Referenz (!)
- `bool`, `char`, `int`, `unsigned`, `float`, ...

Aufgabe: Quellsprachentypen auf adressierbaren Speicherbereich mit Ausrichtung abbilden, Minimalgröße gewöhnlich ein Byte

Zusammengesetzte Datentypen

- Arrays: unterscheide
  - Array statisch fester Länge (statische R.)
  - Arraylänge bei Vereinbarung fest (dynamische R.)
  - Arraylänge durch Zuweisung änderbar (flexible R.)
  - Strings (`array[*](char)`)
- Records
- Vereinigungstypen, einschl. Records mit Varianten
- gepackte zusammengesetzte Typen, einschl. BCD-Zahlen
- OO-Objekte, Activation Record wie Records behandeln

Typklassifikation

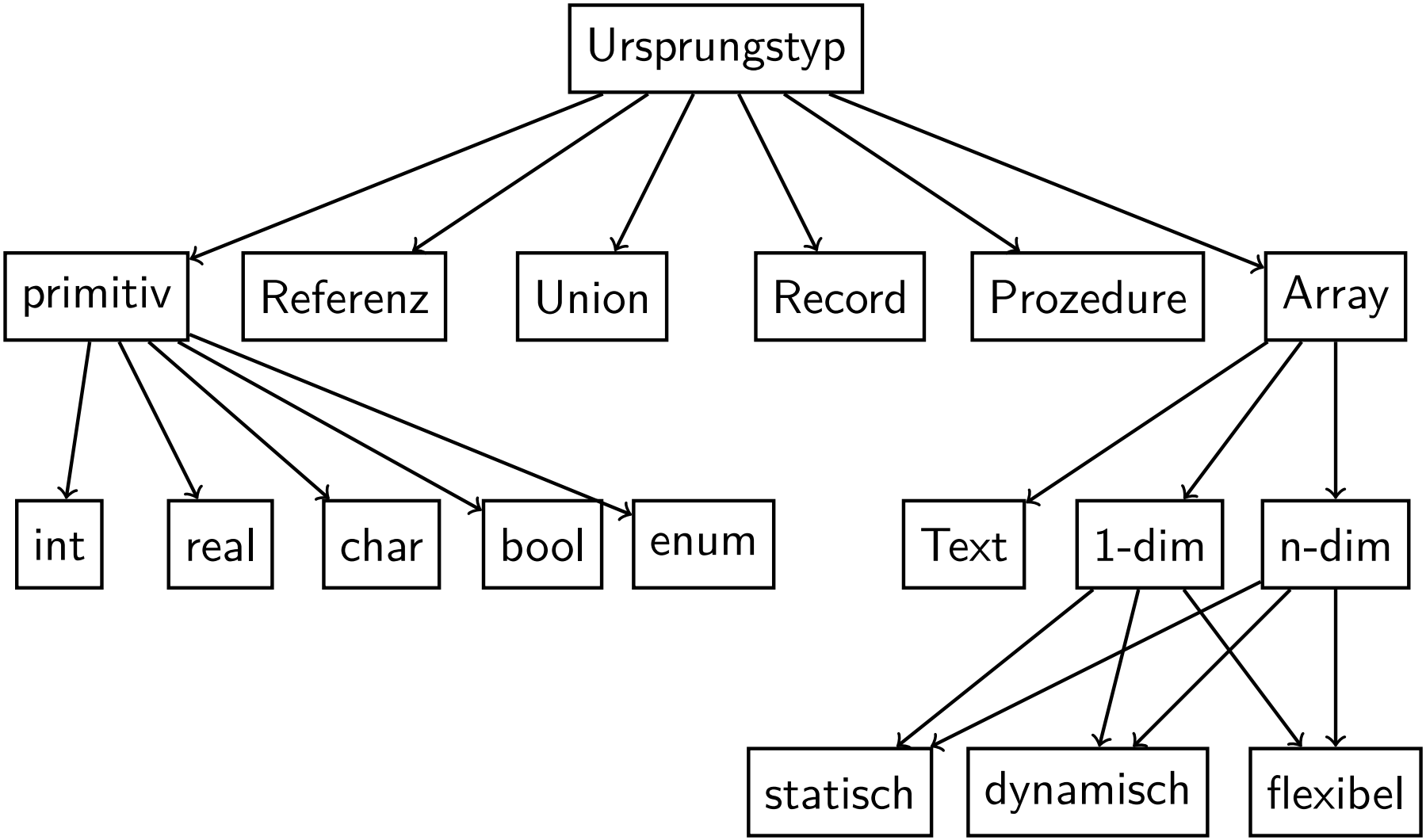


Abbildung einfacher Datentypen

Unterscheide Abbildung auf 1,8,16,32,64 Bit
(80-Bit Gleitkommazahlen?), Ausrichtung
Aufzählungstyp: Werte durch ganze Zahlen codieren

1. bei `bool`
   Festlegung der Codierung für true, false
   - 0 false, 1 true
   - 0 false, $\neq 0$ true
     C Interpretation von `int` in if-Anweisungen
   - 0 true, $\neq 0$ false
     C exit code für Programmterminierung

   vermeide Abbildung auf 1 Bit (auch `bool` mindestens 8 Bit)
2. bei `char`
   Festlegung der Codierung: ASCII, ISO 8859-*, EBCDIC, UTF-8, UTF-16, ...

Allgemeines zur Typcodierung

- bei allen Typen: auf Kompatibilität mit Betriebssystem achten, wegen Systemaufrufen daher gewöhnlich die Konventionen des C-Compilers nutzen, mit dem das BS geschrieben ist
- bei Ausrichtung auf die Geschwindigkeit der Speicherlogik achten
- big/little endian beachten (erstes Byte höchst-/geringst-wertig)

Abbildung von Arrays

Festlegung zeilenweise/spaltenweise Speicherung
Zerlegung in Deskriptor und Datensatz

- Deskriptor enthält alle Info für
  - Speicherabbildungsfunktion
  - Test der Grenzen
- Deskriptor hat feste Länge
- Deskriptor und Datensatz getrennt im Speicher (außer eventuell bei statischen Arrays)
  - Abbildung also auf *zwei* Speicherobjekte
  - adr(a[0,...,0]) heißt virtuelle Anfangsadresse

Array-Adressierung

- eindimensionales Array $a[u_1..o_1]$: $adr(a[i]) = adr(a[0]) + d * i$
- zweidimensionales Array $a[u_1..o_1, u_2..o_2]$: klassische zeilenorientierte Speicherung:

$a[u_1, u_2] \ldots a[u_1, o_2]$	$a[u_1 + 1, u_2] \ldots a[u_1 + 1, o_2]$	$\cdots$	$a[o_1, u_2] \ldots a[o_1, o_2]$

  $adr(a[i, j]) = adr(a[0, 0]) + d * (i * (o_2 - u_2 + 1) + j)$
- dreidimensionales Array $a[u_1..o_1, u_2..o_2, u_3..o_3]$: Sei $l_i = o_i - u_i + 1$
  $adr(a[i_1, i_2, i_3]) = adr(a[0, 0, 0]) + d * (i_1 * l_2 * l_3 + i_2 * l_3 + i_3)$
- Allgemeine Speicherabbildungsfunktion:

$$adr(a[i_1, \ldots, i_n]) = adr(a[0, \ldots, 0]) + d * \left(\sum_{\nu=1}^{n} \left(i_\nu * \prod_{\mu=\nu+1}^{n} l_\mu \right) \right)$$

Implementierung der Array-Adressierung

$$adr(a[i_1, \ldots, i_n]) = adr(a[0, \ldots, 0]) + d * \left(\sum_{\nu=1}^{n} \left(i_\nu * \prod_{\mu=\nu+1}^{n} l_\mu \right) \right)$$

effiziente Auswertung mit Hornerschema:

adr = i_1;
for k = 2 to n **do** {
    adr = adr $*$ l_k;
    adr = adr + i_k;
}

Zwischencode durch „Ausrollen“ dieser Schleife

Beispiel (n=3, symbolischer Tripelcode):

load r, i1
mul r, l2
add r, i2
mul r, l3
add r, i3
mul r, d
add r, adr(a[0,0,0])

Strings

Eigentlich eindimensionale Array von Zeichen
Sonderbehandlung:

- C Konvention: Abschluß mit \0
- Sonst: Deskriptor wie bei Arrays (speichert Länge)
- Wegen Betriebssystemrufen (C Funktionen) oft beides
- Ausrichtung wie Zeichen

Problem Unicode:

- Bei UTF-8: Länge erforderlich, da nicht aus Anzahl Bytes herleitbar
- Bei UTF-16: Länge = Anzahl Bytes / 2

Referenzen

- Wie elementare Typen behandeln
- Länge der Referenzen definiert maximale Größe des Adressraums
- 16-bit, 32-bit, 64-bit Referenzen?

Records

- Records heissen auch struct
- Folge (oder Menge?) von Elementen
- Ausrichtung des Records ist maximale Ausrichtung der Elemente
- Länge ist Summe der Länge der Elemente plus Verschnitt wegen Ausrichtung
- variante Records (variant record) wie Vereinigungstypen (union) behandeln

Objekte

Allgemein: Objekte wie Records behandeln
Objektlayout, Subobjekte, vptr, vtable, ...
Vgl. Vorlesung Fortgeschrittene Objekt Orientierung

Speicherausrichtung (Alignment)

- Ausgerichtetes Datenelement: Adresse Vielfaches der Größe
- Zugriff auf nicht-ausgerichtete Element langsam (x86) oder unmöglich (ARM)
- Die align Funktion berechnet nächste ausgerichtete Adresse:

  if (offset % alignment == 0)
  then offset
  else ((offset / alignment)+1) $*$ alignment

  Oder kürzer:

  (offset+alignment−1) / alignment $*$ alignment

  Variante falls alignment $= 2^n$ (C,Java,...):

  (offset+alignment−1) & ∼(alignment−1)

- Gemeinsames Alignment mehrerer Elemente: kgV
  Variante falls alignment $= 2^n$: max

Speicherabbildung von Activation Records

Berechnen der Offsets für alle Deklarationen in einer Prozedur:

```
class ActivationRecord {
  void compute() {
    unsigned offset = 0;
    for (Declaration d : this.getDeclarations()) {
      unsigned alignment = d.getType().getAlignment();
      offset = align(offset, alignment);
      d.setOffset(offset);
      offset += d.getType().getByteSize();
    }
  }
```

Verallgemeinerung für beliebige Records

Für verschachtelte Records muss size und alignment für das aktuelle Record mitberechnet werden.

```
class Record extends Type {
  void compute() {
    unsigned offset = 0;
    for (Declaration d : this.getDeclarations()) {
      unsigned alignment = d.getType().getAlignment();
      offset = align(offset, alignment);
      d.setOffset(offset);
      offset += d.getType().getByteSize();
      this.alignment = kgV(alignment, this.alignment);
    }
    this.size = offset;
  }
```

Vereinigungstypen (union)

- Speicherlayout der Vereinigungsvarianten wird linear übereinandergelegt (überlagert)
- Gesamtlänge ergibt sich durch längste Variante
- Für dynamische Typsicherheit muss Variantenart (Record Discriminator) mit abgespeichert werden

Speicherabbildung für Vereinigungstypen

```
class Union extends Type {
  void compute() {
    for (Declaration d : this.getDeclarations()) {
      d.setOffset(0);
      unsigned alignment = d.getType().getAlignment();
      unsigned size = d.getType().getByteSize();
      this.alignment = kgV(this.alignment, alignment);
      this.size = max(this.size, size);
    }
  }
```

Berechnung Relativadressen in Records und Unions

rule Type → 'int' .
attribution Type.size := 4

rule Type → 'double' .
attribution Type.size := 8

rule Field → Type Symbol .
attribution Field.size := Type.size

rule Type → 'record' '{' RFields '}' .
attribution Type.size := RFields.size

rule RFields → ϵ .
attribution RFields.size := 0

rule RFields → RFields Field .
attribution
  Field.offset := RFields[2].size
  RFields[1].size :=
    RFields[2].size + Field.size

rule Type → 'union' '{' UFields '}' .
attribution Type.size := UFields.size

rule UFields → ϵ .
attribution UFields.size := 0

rule UFields → UFields Field .
attribution
  Field.offset := 0
  UFields[1].size :=
    max(UFields[2].size, Field.size)

Erweiterung: Alignment

rule Type → 'int' .
attribution
  Type.size := 4
  Type.align := 4

rule Type → 'short' .
attribution
  Type.size := 2
  Type.align := 2

rule Type → 'double' .
attribution
  Type.size := 8
  Type.align := 8

rule Type → 'long double' .
attribution
  Type.size := 12
  Type.align := 4

rule Type → 'record' '{' RFields '}' .
attribution
  Type.size := RFields.size
  Type.align := RFields.align

rule RFields → ϵ .
attribution
  RFields.size := 0
  RFields.align := 0

rule RFields → RFields Field .
attribution
  Field.offset :=
    padded(RFields[2].size, Field.align)
  RFields[1].size :=
    Field.offset + Field.size
  RFields[1].align :=
    kgV(RFields[2].align, Field.align)

rule Field → Type Symbol .
attribution
  Field.size := Type.size
  Field.align := Type.align

Kapitel 6: Transformation

Abbildung der Ablaufsteuerung

kann als Quell-Quelltransformation beschrieben werden (unter Einsatz von Sprüngen), ist aber tatsächlich ein Erzeugungsverfahren für Einzelbefehle und Grundblöcke der Zwischensprache.

Einzelfälle:

- Sprung
- bedingte Anweisung
- Fallunterscheidung
- Typ-Fallunterscheidung
- Schleife
- Zählschleife
- Prozeduraufruf
- Ausnahmebehandlung

Sprunganweisung

goto M;		JMP M
...	$\Rightarrow$	...
M:		M:

beendet Grundblock

Bedingte Anweisung

if B then S else S' end;	$\Rightarrow$	B JMP ZERO Then S JMP Ende Then: S' Ende:

Beachte: Sprünge mit erfüllter Bedingung oft schneller!

3 Grundblöcke: B; JMP ZERO Then, S; JMP Ende, S'

Fallunterscheidung

Einfache Übersetzung: Kaskade von bedingten Anweisungen

```
case expr
when x1 S1;
when x2 S2;
...
default Sd;
```

$\Rightarrow$

```
e:=expr
if e = x1 then S1 else
if e = x2 then S2 else
if ...
else Sd
end;
end;
end;
```

$2n + 1$ Grundblöcke, beginnend mit

e:=expr; e = x_1; JMP NOTEQUAL MS”

Fallunterscheidung mit Sprungtabelle

Abbildung von x_1 x_2 ... in die ganzen Zahlen muss eindeutig sein.

$\Rightarrow$

JUMP IND Sprungtabelle + expr
Sprungtabelle:
M_1, Sonst, Sonst, Sonst, M_2, Sonst, ...
M_1:
S_1
JMP Ende
M_2:
S_2
JMP Ende
...
Sonst:
S_d
Ende:

Problem bei großen Lücken in der Tabelle
n+3 Grundblöcke, einschl. Sprungtabelle (Sonderfall)

Typ-Fallunterscheidung

Fallunterscheidung über dem Eintrag, der den dynamischen Typ eines Objekts kennzeichnet. Bsp. Ada, Sather(-K)
Implementierung polymorpher Aufrufe/Objektzugriffe erzeugt Typ-Fallunterscheidung implizit
Behandlung wie gewöhnliche Fallunterscheidung
Vorsicht mit Sprachen, bei denen die Typkennung nicht gespeichert wird - sie sind nicht typsicher! Z.B. Variante Records in Pascal, erzeugte Variante wird nicht gemerkt.

Anfangsgesteuerte Schleife

while B loop S end; $\Rightarrow$

Anfang: B JMP ZERO Ende S JMP Anfang Ende:	*oder*	JMP Anfang weiter: S Anfang: B JMP NONZERO weiter

2 Grundblöcke, Fassungen unterscheiden sich in Anzahl ausgeführter Sprungbefehle (Anzahl Sprünge im Code gleich), Anordnung rechts günstiger, wenn Sprünge mit erfüllter Bedingung schneller sind

aber: Grundblöcke beliebig im Code plazierbar, dann beide Fassungen äquivalent

Wiederholung: Attributierte Grammatik für **while**-Anweisungen

rule statement → 'while' '(' **condition** ')' statement .
attribution
  l1 := new_label();
  l2 := new_label();
  statement[2].next = l1;
  condition.false = statement[1].next
  condition.true = l2;
  statement[1].code = label || l1 || **condition**.code || label || l2 || statement[2].code

Bemerkungen:

- statement.next ist das Label des nächsten Statements
- **condition**.false ist das Sprungziel bei falscher Bedingung
- **condition**.true ist das Sprungziel bei wahrer Bedingung

Wiederholung: Als semantische Aktionen

$$
\begin{array}{lll}
S \rightarrow & \textbf{while (} & \{\ \text{L1} = \text{new\_label}();\ \text{L2} = \text{new\_label}(); \\
 & & \text{C.false} = \text{S.next};\ \text{C.true} = \text{L2}; \\
 & & \text{print("label", L1);}\ \} \\
 & C\ \textbf{)} & \{\ \text{S1.next} = \text{L1};\ \text{print("label", L2);}\ \} \\
 & S1 &
\end{array}
$$

Wiederholung: Rekursiver Abstieg mit direkter Codeerzeugung

statement → **while (** *condition* **)** *statement*

```
void parse_statement(label next) {
    if (token == T_while) {
        next_token();
        if (token == '(') next_token(); else error(...);
        label L1 = new_label();
        label L2 = new_label();
        print("label", L1);
        /* parse and print condition. Jump to first arg if true,
           jump to 2nd arg if false */
        parse_condition(L2, next);
        if (token == ')') next_token(); else error(...);
        print("label", L2);
        parse_statement(L1);
    } else {
        /* other statements */
    }
}
```

Endgesteuerte Schleife

loop S until B end; $\Rightarrow$

Anfang:
S
B
JMP ZERO Anfang

1 Grundblock

Zentralgesteuerte Schleife

loop
S_0;
exit when B_0;
S_1;
exit when B_1;
S_2
...
end;

$\Rightarrow$

Anfang:
S_0
B_0
JMP NON ZERO Ende
S_1
B_1
JMP NON ZERO Ende
S_2
...
JMP Anfang
Ende:

$n + 1$ Grundblöcke

Zählschleife

for i := a step s until e do S
Annahme: Schrittweite s (samt Vorzeichen) statisch bekannt

Standardübersetzung (entspricht C, C++, ...):

```
i = a;
while (i <= e) { // bei s < 0: i >= e
  S;
  i = i+s;
}
```

Zählschleife „richtiger“

_o ist der letzte Wert $\leq$e, für den die Schleife ausgeführt wird, bei s$=1$ sinnvoll, da keine Div- oder Mod-Operation und e$=$_o.

```
if (a <= e) {
  i = a;
  _x = a % s;
  _y = e % s;
  _o = (_y >= _x) ? e-(_y-_x) : e-(_y+s-_x);
  while (true) {
    S;
    if (i == _o) break; else i = i+s;
  }
}
```

Funktioniert immer, auch bei e$=$maxint ! Aber Vorsicht, wenn a$\leq$e in Vergleich auf 0 übersetzt werden muss

Beispiel

c=0;	s1: ST c 0
	s2: LD x
if x > 0 {	s3: GT 0
	s4: JMP FALSE u1
a=2;	t1: ST a 2
	t2: LD a
b=a*x+1;	t3: LD x
	t4: MUL t2 t3
	t5: ADD t4 1
	t6: ST b t5
a=2*x;	t7: LD x
	t8: MUL 2 t7
	t9: ST a t8
c=a+1+b;	t10: LD a
	t11: ADD t10 1
	t12: LD b
	t13: ADD t11 t12
	t14: ST c t13
}	t15: JMP u1
x=c;	u1: LD c
	u2: ST x u1

Kapitel 6: Transformation

Abbildung der Operatoren

Prinzip: jede Maschinenoperation hat nur ein Ergebnis

- Arithmetische Operationen: 1 zu 1 Zuordnung entsprechend Speicherabbildung
  Vorsicht: Bei manchen Sprachen Prüfung auf Überlauf bei ganzzahliger Multiplikation erforderlich! meist sehr aufwendig!
- Maschinenoperationen mit mehreren Ergebnissen:
  - Operation mit anschließender Projektion, z.B. divmod
  - Operationen, die zusätzlich Bedingungsanzeige setzen: Zusätzlicher cmp-Befehl, falls Bedingung benötigt
- Logische Operationen und Relationen: Unterscheide, ob Ergebnis zur Verzweigung benutzt oder abgespeichert wird
- Speicher-Zugriffe: Zugriffspfad explizit mit Adressarithmetik codieren, Basisadressen sind Operanden, woher bekannt? (Konventionen festlegen)
  Achtung: Indexrechnung ganzzahlig, Adressen vorzeichenlos!
- Typanpassungen: Dereferenzieren, deprozedurieren, Anpassung `int`→`float` usw. explizit, Uminterpretation Bitmuster implizit

Umgekehrte polnische Notation

Umgekehrte polnische Notation (UPN)

Schreibweise für Ausdrücke, bei der zuerst die Operanden und dann die auszuführende Operation angegeben wird.

Beispiel:

$$\begin{aligned} 7 * 4 \quad &\text{in UPN:} \quad 7\ 4\ * \\ 2 * (2 + 3) \quad &\text{in UPN:} \quad 2\ 2\ 3\ +\ * \\ 5 + y + 3 * 5 \quad &\text{in UPN:} \quad 5\ y\ +\ 3\ 5\ *\ + \end{aligned}$$

Vorteile:

- Eindeutig, auch ohne Präzedenzen und Klammern
- Natürliche Darstellung für Stackmaschinen

UPN zu Bytecode

Ausdruck: $a - e * (c + d)$

UPN: $a\ e\ c\ d\ +\ *\ -$

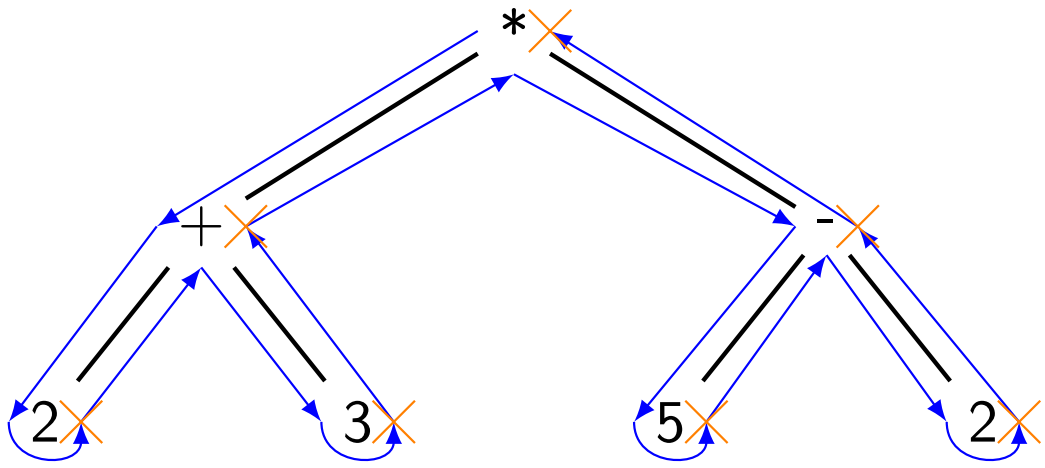
Bytecode:

```
iload a
iload e
iload c
iload d
iadd
imul
isub
```

Typanalyse/Operatoridentifikation beachten!

Erzeugung von UPN

- Gegeben: Berechnungsformel als Baum
- Postfixordnung bei Tiefensuche erzeugt UPN.
- Postfixordnung: Ausgabe beim Verlassen eines Knoten (also nachdem Kinder besucht sind)



Tiefensuche

Ausgabe

Ergebnis: $2\ 3\ +\ 5\ 2\ -\ *$

Ordnung nach Ershov

- Ershov-Prinzip: Der grössere Unterbaum zuerst
- Führt auf Stackmaschinen zu minimaler Stackgrösse

Ausdruck: $a - e * (c + d)$

Ershov	Stackgröße	UPN	Stackgröße
`iload c`	1	`iload a`	1
`iload d`	2	`iload e`	2
`iadd`	1	`iload c`	3
`iload e`	2	`iload d`	4
`imul`	1	`iaddd`	3
`iload a`	2	`imul`	2
`swap`	2	`isup`	1
`isub`	1		

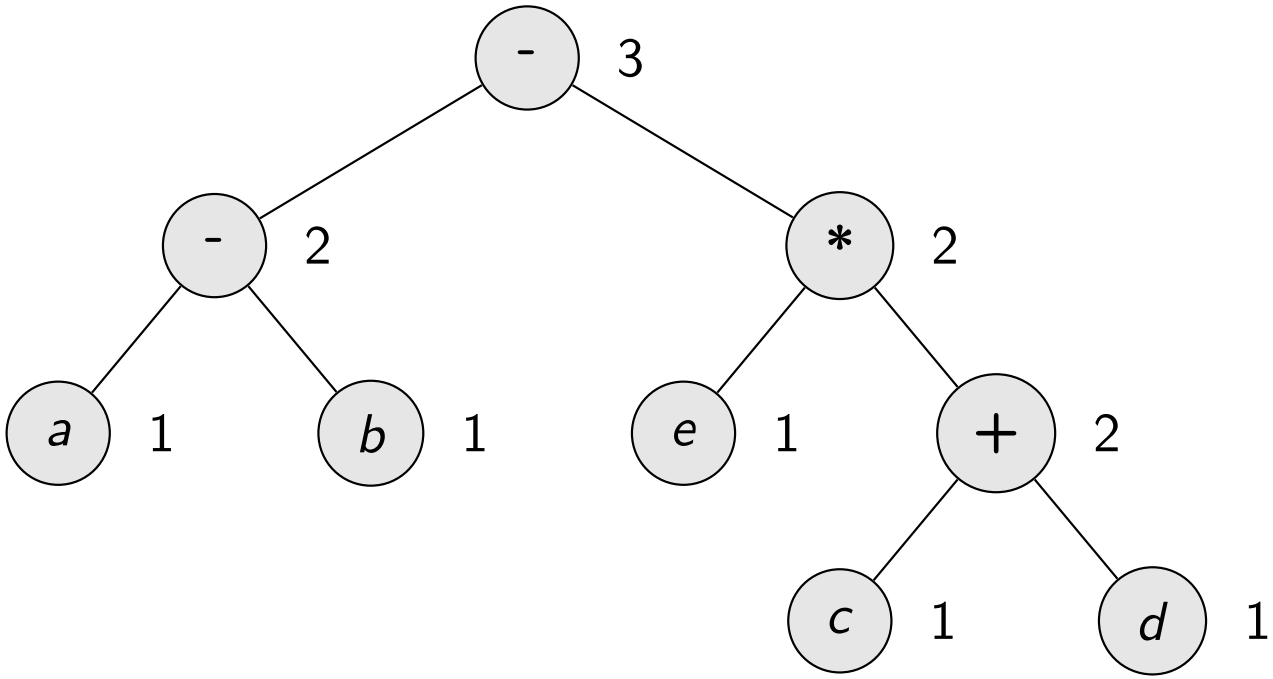
Ershov-Zahlen

Ershov-Zahlen geben die Zahl der Register/Stackhöhe an, die zur Auswertung eines Ausdrucks benötigt werden.
Markieren eines Ausdrucksbaums:

1. Kennzeichne alle Blätter mit 1.
2. Bei 2 Kindern:
   - gleiche Kennzeichnung der Kinder: übernimm Kennzeichnung plus 1
   - sonst: nimm größte Kennzeichnung der Kinder
3. Allgemein: Für absteigend sortierte Markierungen der Kinder $M_1, \ldots, M_n$:

$$\max(M_1, M_2 + 1, \ldots, M_n + (n-1))$$

Baum mit Ershov-Zahlen



Ausdruck: $(a - b) - e * (c + d)$

Abbildung von Zuweisungen

Beachte: Zuweisung ist keine „normale“ binäre Operation, da linker Teil nicht klassisch ausgewertet wird

Ausdruck: $a = a + 1$

Bytecode	Ausdruckteil
`iload a`	a
`iconst_1`	1
`iadd`	$+$
`istore a`	$a =$

Adresse `a` ist `Basisregister(+Offset)`

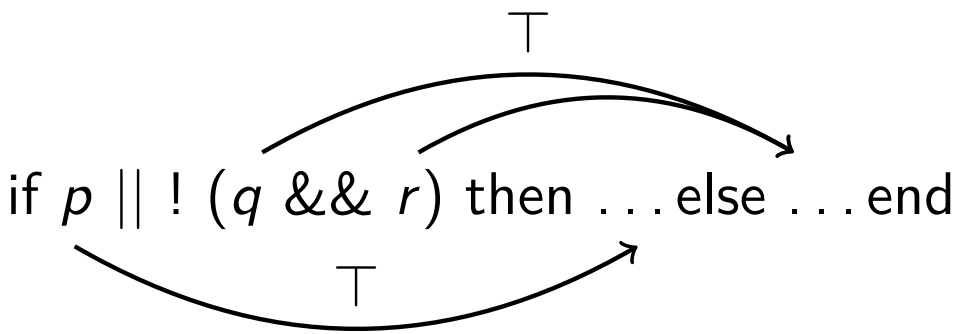
- Informationen aus Typabbildung berechenbar
- kann durch Optimierung verändert werden, daher vorläufig nur symbolisch

Kurzauswertung

Die Operationen **&&** und || werten ihre Argumente faul aus.
Beispiel: `foo() == 2 && bar() < 10`
Wertet linke Seite foo() $==$ 2 aus, bar() $<$ 10 wird nur ausgewertet, wenn linke Seite wahr ist.

Vorgehen bei Codeerzeugung:

1. Erzeuge Label *l*, das angesprungen wird, falls rechte Seite ausgewertet werden muss.
2. Code für rechte Seite hinter *l* platzieren.



Kurzauswertung

```
type marken = record
  ja,nein:symb_adresse;
  nachf:Boolean end;

rule bed_anw ::= 'if' ausdruck 'then' anw
                                'else' anw 'end' .
attribution
  ausdruck.loc := neue_adresse;
  bed_anw.then_loc := neue_adresse;
  bed_anw.else_loc := neue_adresse;
  ausdruck.ziel :=
    neue_marken(bed_anw.then_loc, bed_anw.else_loc, true);
```

neue_adresse	generiere neues Sprungziel für Zielcode
neue_marken	neuer Record des Typs marken
loc	Startadresse von Ausdruck/Anweisung
ziel	Record der Sprungziele
nachf	gibt an, welches Sprungziel unmittelbar folgt (Sprungbefehl nicht nötig, erweiterter Grundblock)

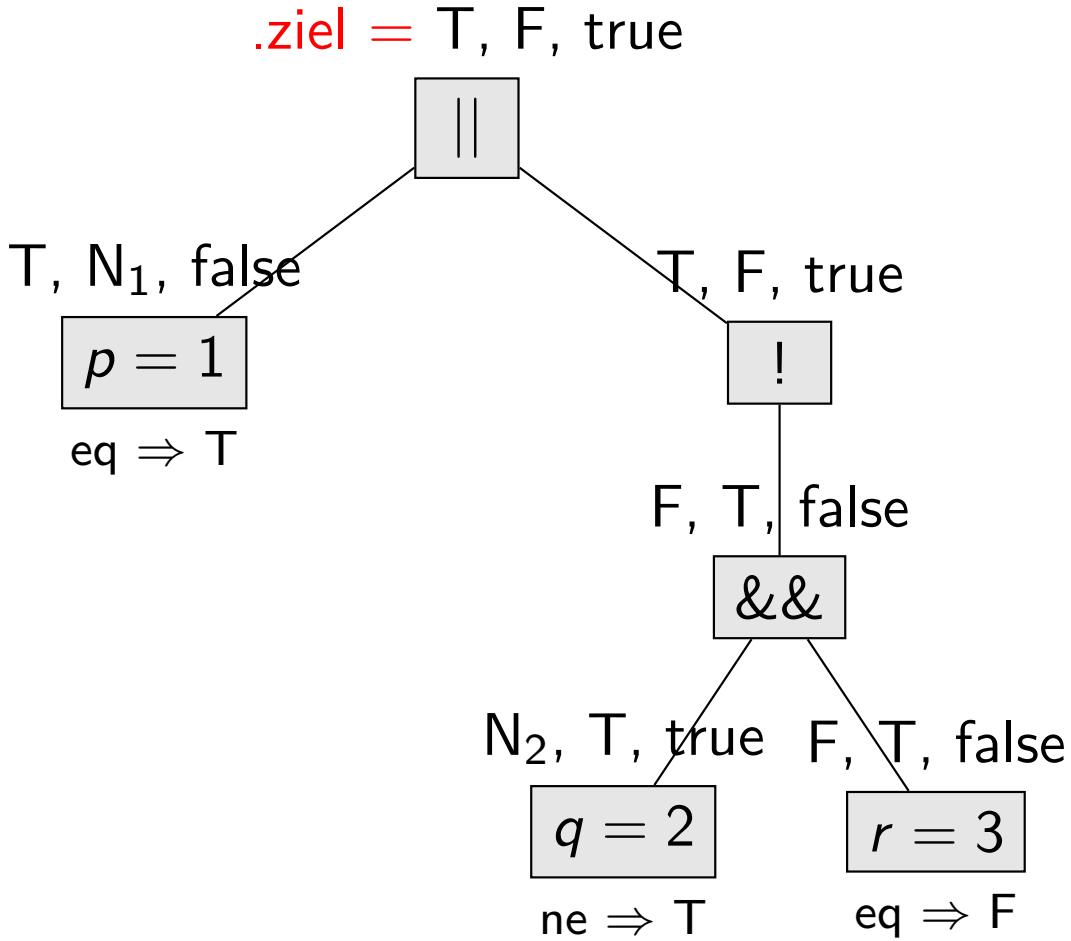
Kurzauswertung

```
rule ausdruck := ausdruck operator ausdruck .
attribution
  ausdruck[2].loc := ausdruck[1].loc;
  ausdruck[3].loc := neue adresse;
  ausdruck[2].ziel :=
    if operator.operator = 'or'
    then neue_marken(ausdruck[1].ziel.ja,ausdruck[3].loc,false)
    else neue_marken(ausdruck[3].loc,ausdruck[1].ziel.nein,true)
    end;
  ausdruck[3].ziel := ausdruck[1].ziel;

rule ausdruck := 'not' ausdruck .
attribution
  ausdruck[2].loc := ausdruck[1].loc;
  ausdruck[2].ziel :=
    neue_marken(ausdruck[1].ziel.nein,ausdruck[1].ziel.ja,
    not ausdruck[1].ziel.nachf)
```

Beispiel Kurzauswertung

Ausdruck: if $p = 1$ || ! ($q = 2$ && $r = 3$) then T else F end; E



Beispiel Kurzauswertung

Ausdruck: if $p = 1$ || ! $(q = 2$ && $r = 3)$ then T else F end; E

```
        iload p
        iconst_1
        if_icmpeq T    (eq ⇒ T)
N1:     iload q
        iconst_2
        if_icmpne T    (ne ⇒ T)
N2:     iload r
        iconst_3
        if_icmpeq F    (eq ⇒ F)
T:      ...
        goto E
F:      ...
E:
```

Kapitel 6: Transformation

Speicherorganisation und Prozeduraufruf

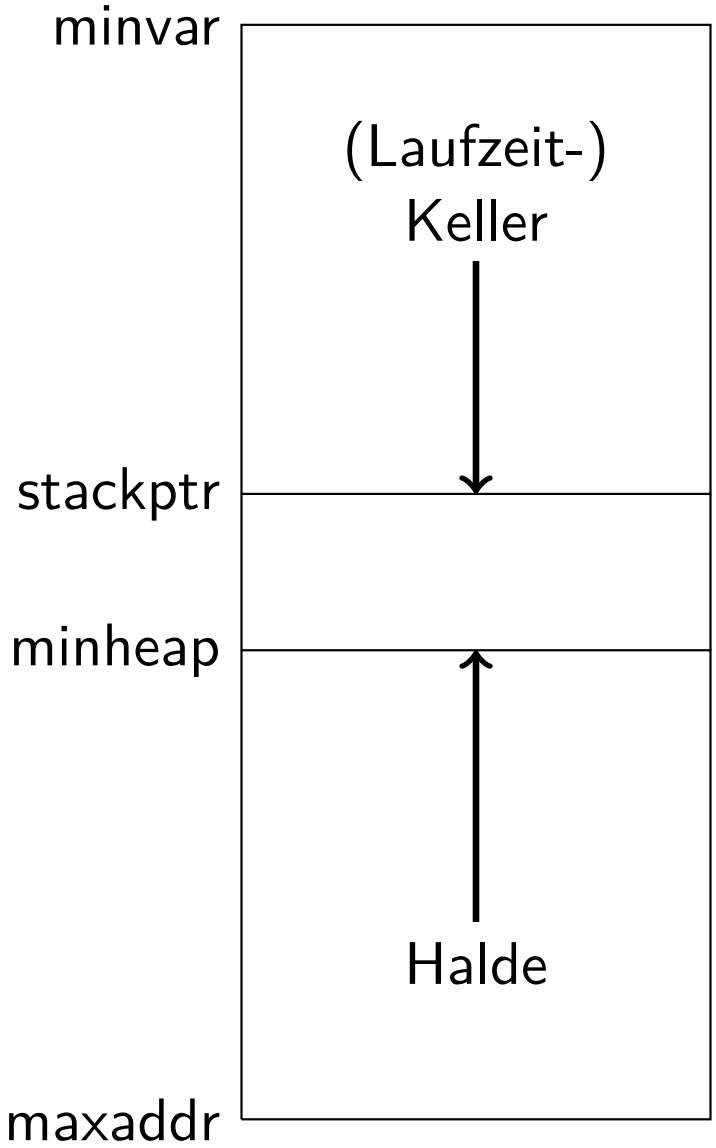
Aufgaben:

- alle Programmvariablen allozieren
- statische/dynamische/anonyme Allozierung unterscheiden
- Vorbereitung auf dynamische Prozedurschachtelung einschl. Rekursion

Verfahren:

- Unterscheide (Laufzeit-)Keller für dyn. Variable und Halde für Variable mit unbekannter Lebensdauer (anonyme Objekte)
- Keller eingeteilt in Activation Records unterster Activation Record für statische Variable
- Activation Record enthält Prozedurparameter, lokale Variable, Rücksprungadresse, Verweis stat./dyn. Vorgänger, sonstige organisatorische Info, Hilfsvariable für Zwischenergebnisse
- Activation Record besteht aus statischem Teil (Länge dem Compiler bekannt) und dynamischem Teil (für dynamische Arrays) Erweiterung für multithreaded Programme: mehrere Keller, Kaktuskeller

Speicherorganisation



Einteilung in zwei Speicherbereiche

Anordnung hardwareabhängig

Garantiere Invariante:
minheap $>$ stackptr

Basisadressen

minvar	Basis statischer Variablen (Beginn Keller)
baseptr	Basis lokaler Variablen eines Unterprogramms (Beginn UP-Activation Record)
stackptr	Kellerpegel

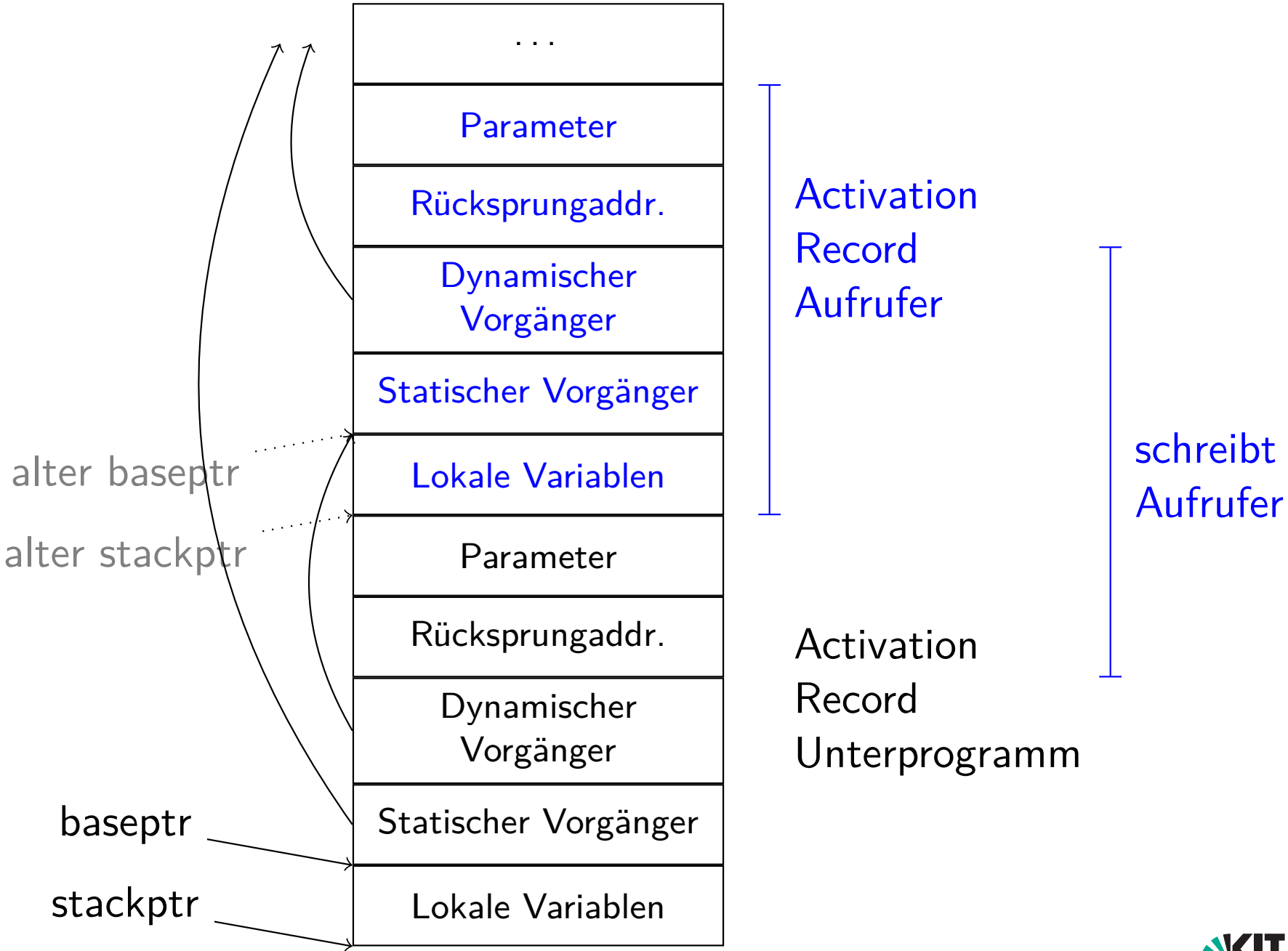
Adressen statischer Variable v mit Rel.Adr. r_v:

minvar $+ r_v$

Adressen lokaler Variable v mit Rel.Adr. r_v:

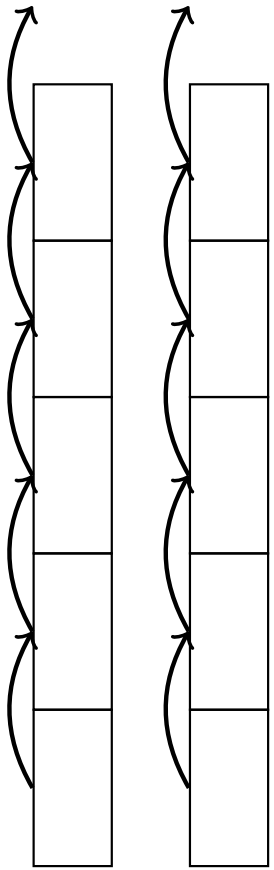
baseptr $+ r_v$

Laufzeitkeller

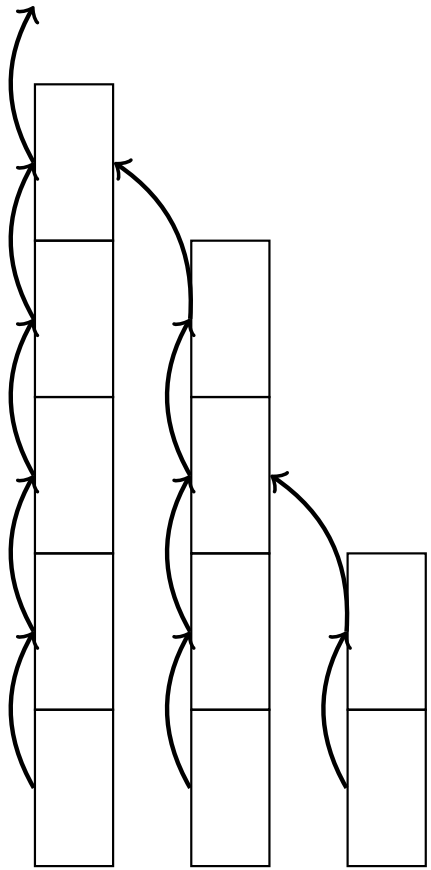


Mehrere Keller

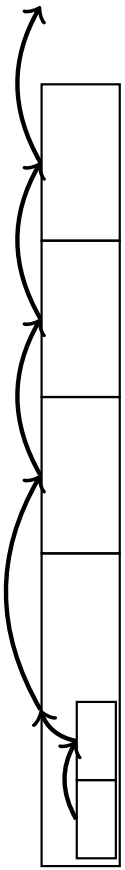
Disjunkte Keller



Kaktuskeller
(auf der Halde)



Keller rekursiv
(max. Kellergröße?)

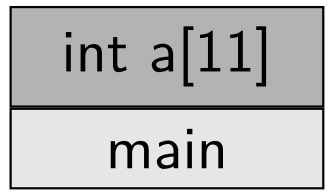


Beispiel: Quicksort

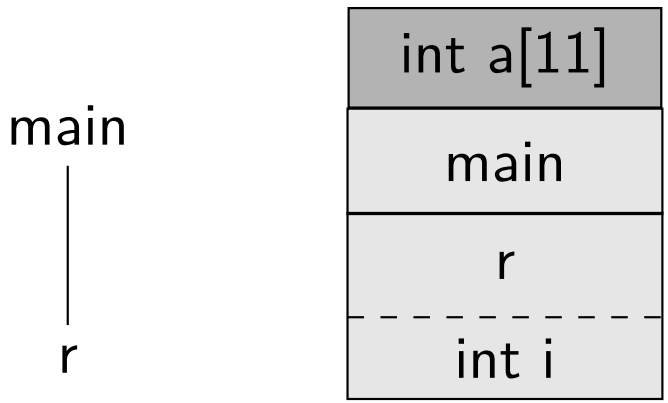
```
int a[11];
void r(void) { /* ... read integers into a[1] to a[9] ... */
    int i;
}
int p(int m, int n) {
    /* choose pivot element p, partition array into
     * { x | x < p }, { x | x >= p}; return position of p ... */
}
void q(int m, int n) {
    int i;
    if (n > m) {
        i = p(m, n); /* partition array */
        q(m, i-1); /* sort left part */
        q(i+1, n); /* sort right part */
    }
}
int main(void) {
    r(); a[0] = -INT_MIN; a[10] = INT_MAX;
    q(1,9);
}
```

Stack mit Activation Records

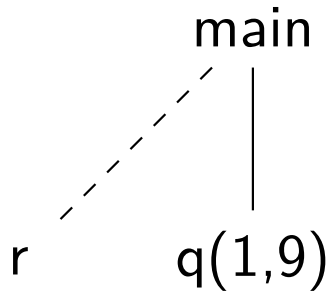
main

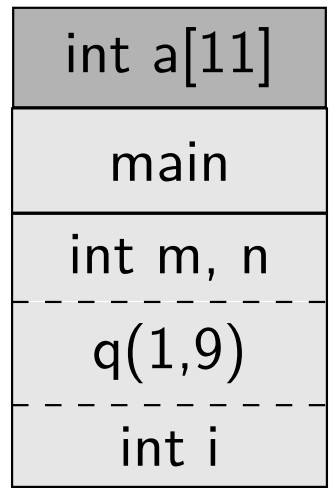


Stack mit Activation Records

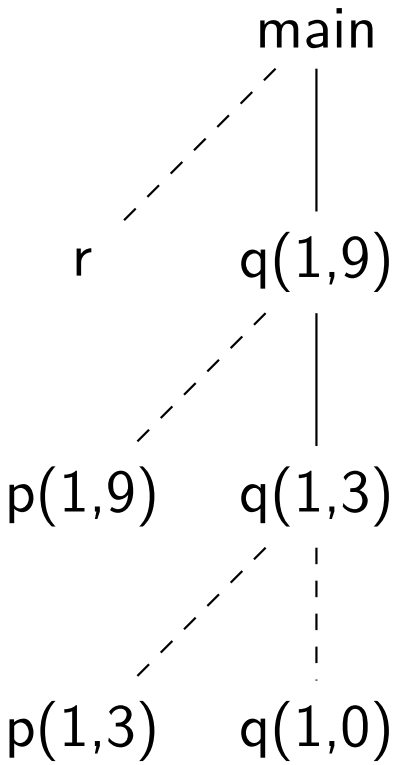


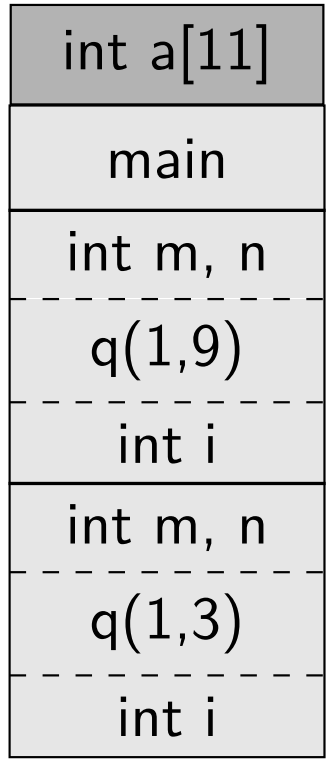
Stack mit Activation Records





Stack mit Activation Records





Statische Variablenbindung bei geschachtelten Prozeduren

Viele Sprachen bieten geschachtelte Prozeduren.

Beispiel (Pascal):

```
procedure p();
  var x: integer;
  procedure q();
  begin
    writeln(x);
   end;

  procedure r();
  var x: integer;
  begin
    x := 17;
    q();
  end:
begin
  x :=42;
  r()
end;
```

Prinzip der statischen Variablenbindung:

- für nichtlokale Variablen gelten die Deklarationen in der **textuellen** Umgebung der Verwendung, und **nicht** die Deklarationen in der dynamischen Vorgängerprozedur

$\Rightarrow$ x in q ist x aus p und nicht x aus r

$\Rightarrow$ Ausgabe 42 und nicht 17

Vorteile von statischer Variablenbindung:

- Programme lesbarer
- Symboltabelle als Stack organisierbar
- effiziente Adressierung mit Activation Records auch bei geschachtelten Prozeduren

Skizze eines Programms mit geschachtelten Prozeduren

```
typedef int (*function_pointer)(int parameter);
void a(int x) {
    int b(function_pointer f) {
        /* ... */ int res = f(x); /* ... */
        return res;
    }
    void c(int y) {
        int d(int z) {
            /* ... */ int res = x*y + z; /* ... */
            return res;
        }
        /* ... */
        b(d);
        /* ... */
    }

    c(1);
}
```

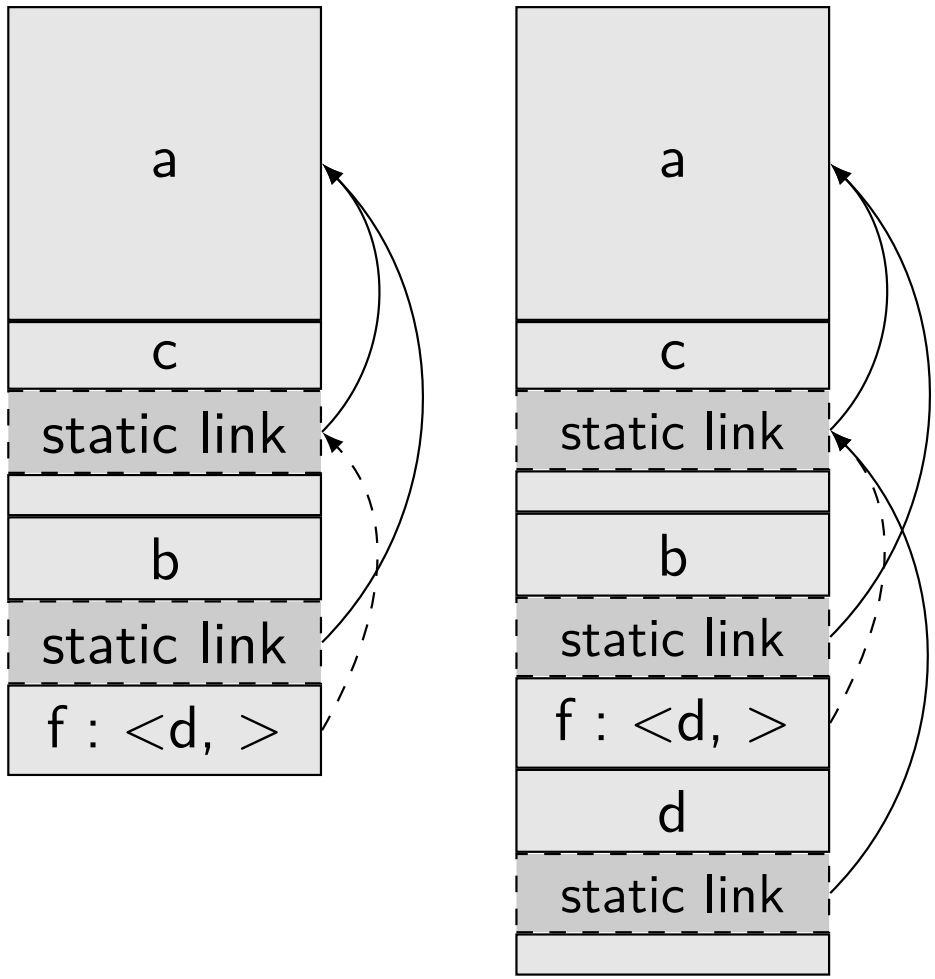
Prozedurparameter / Prozedurvariablen

Prozeduren als Parameter werden als Closure implementiert.

Definition Closure:
Paar aus <Funktion, statischer Umgebung>; erlaubt korrekten Zugriff auf nichtlokale Variablen.

Implementierung:
Paar <Einsprungadresse, Link zur statischen Umgebug>

Activation Record Stack mit Static Links



Displays

Problem: bei Zugriffen auf äußere Variablen aus tief verschachtelten Prozeduren müssen viele Static Links verfolgt werden.

Abhilfe: Displays (Dijkstra [1]).
Displays sind ein Hilfsarray d. Dieses enthält die Adressen der geschachtelten Activation Records. 1 Eintrag pro statische Tiefe:

$$
\begin{aligned}
d_i &= \mathrm{adr}(\mathrm{AR}_i) \\
AR_i &= \text{Activation Record des letzten Aufrufs der statischen Tiefe } i
\end{aligned}
$$

Vorteil: Schneller Zugriff auf nichtlokale Variablen.

Displays (1/2)

Implementierung:

- Bei Aufruf einer Funktion der Tiefe i:
  $d[i]$ sichern; $d[i]$ neu setzen auf $\mathrm{adr}(\mathrm{AR})$; Beim Rücksprung $d[i]$ wiederherstellen.
- Die Einträge des Displays werden in Registerbank organisiert.
  Der Rest mit static Link.
  Bei n Registern:

$$\left.\begin{array}{l} d[0] \\ d[1] \\ \vdots \\ d[n-1] \end{array}\right\} \text{in Registern}$$

$$\left.\begin{array}{l} d[n] \\ d[n+1] \\ \vdots \end{array}\right\} \text{static Links}$$

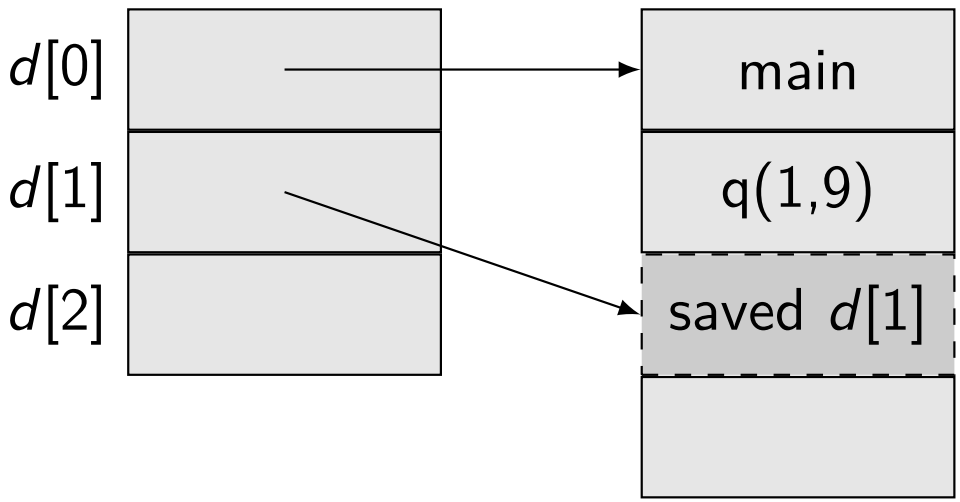
Displays (2/2)

- Falls nur wenige Register verfügbar (z.B. n=4):
  - verwende Static Link und zusätzlich d[0] für globale Variablen
  - d[3] (bzw. Rahmenzeiger) für lokale Variablen (im aktuellen AR)
  - d[1/2] für statischen (Vor)Vorgänger der aktuellen Funktion

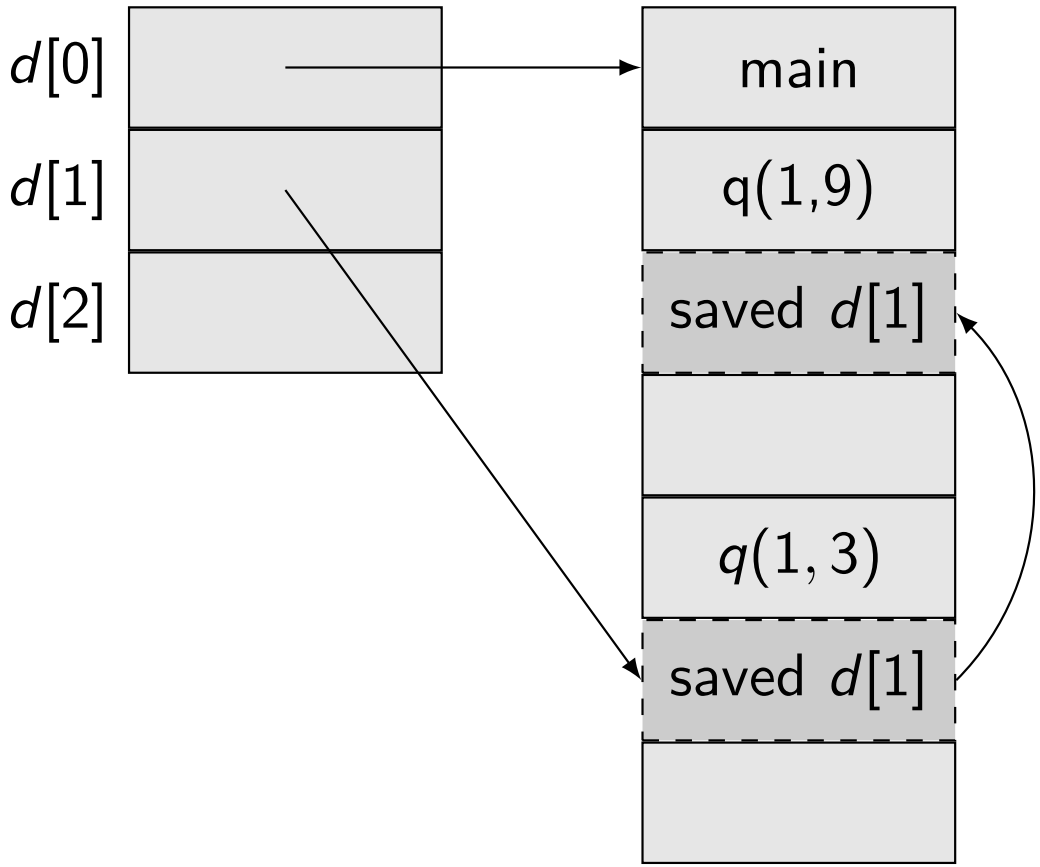
Quicksort mit verschachtelten Prozeduren

```
int main(void) {
    int a[11] = { INT_MIN,6,5,3,4,2,5,19,9,9,INT_MAX };
    void q(int l, int r) {
        void e(int e1, int e2) { /* exchange 2 elements */
            int t = a[e1]; a[e1] = a[e2]; a[e2] = t;
        }
        void p(int l, int r) {
            /* choose pivot element and partition */
            /* ... */ e(x, y); /* ... */
        }
        if (l > r)
            return;
        int i = p(m, n); /* partition array */
        q(l, i-1);
        q(i+1, r);
    }
    return 0;
}
```

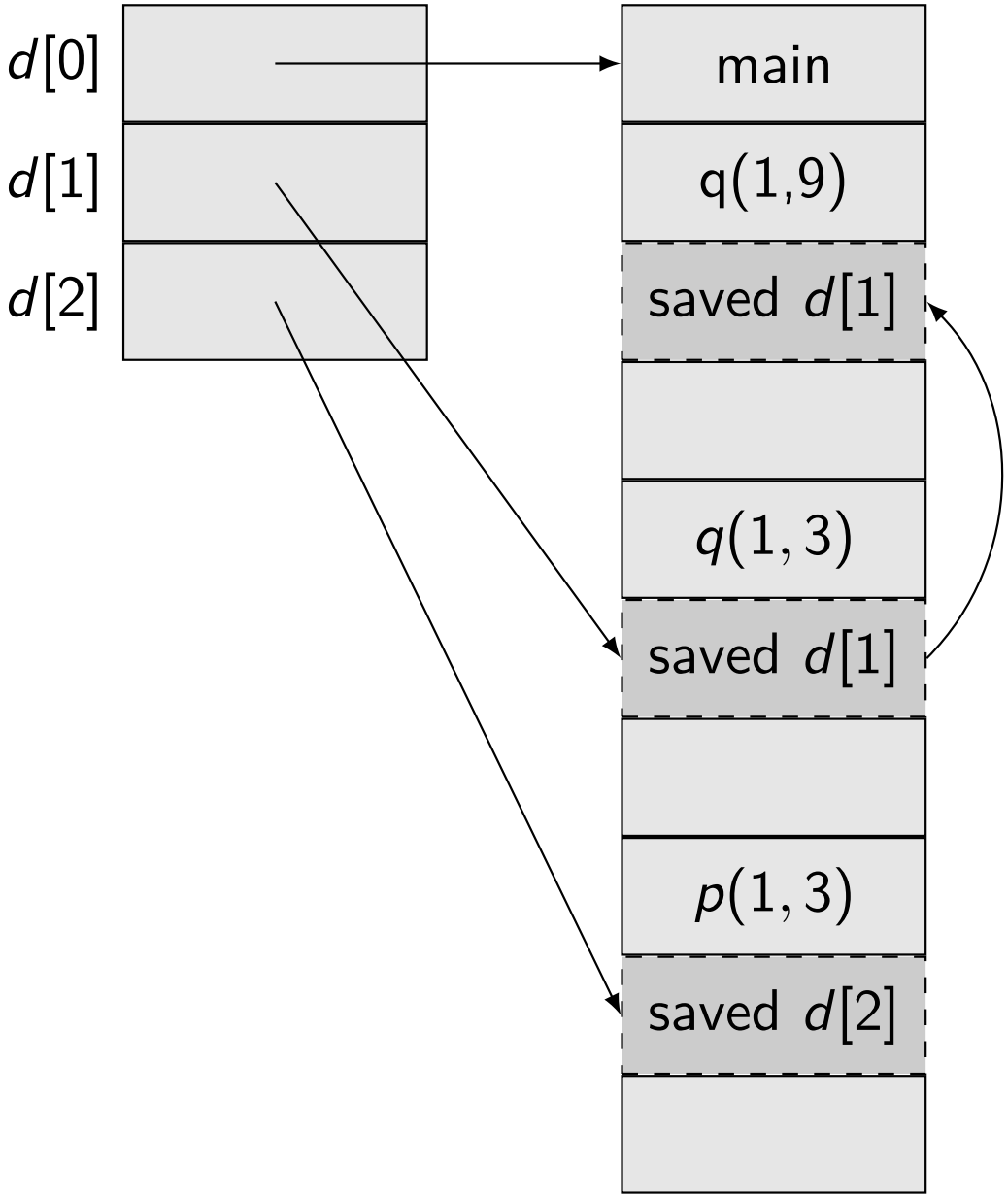
Displays



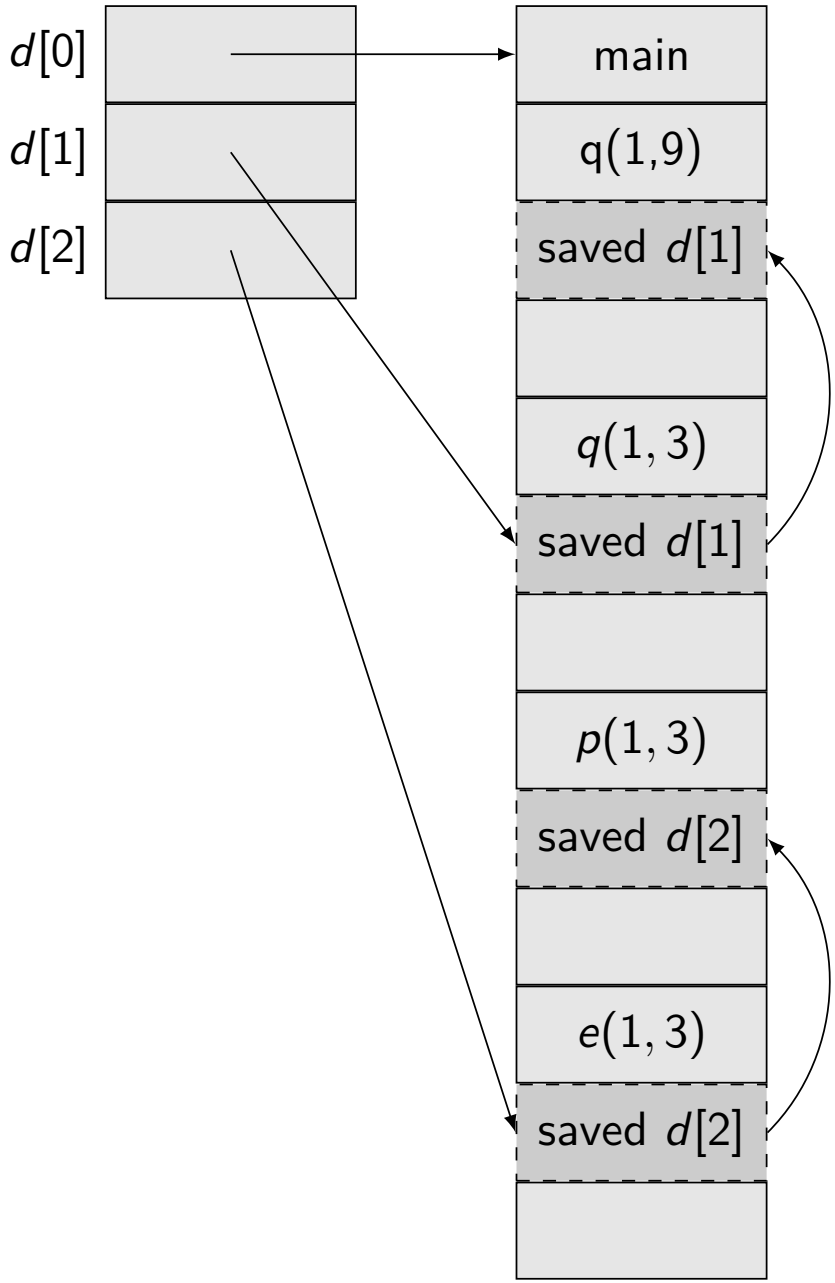
Displays



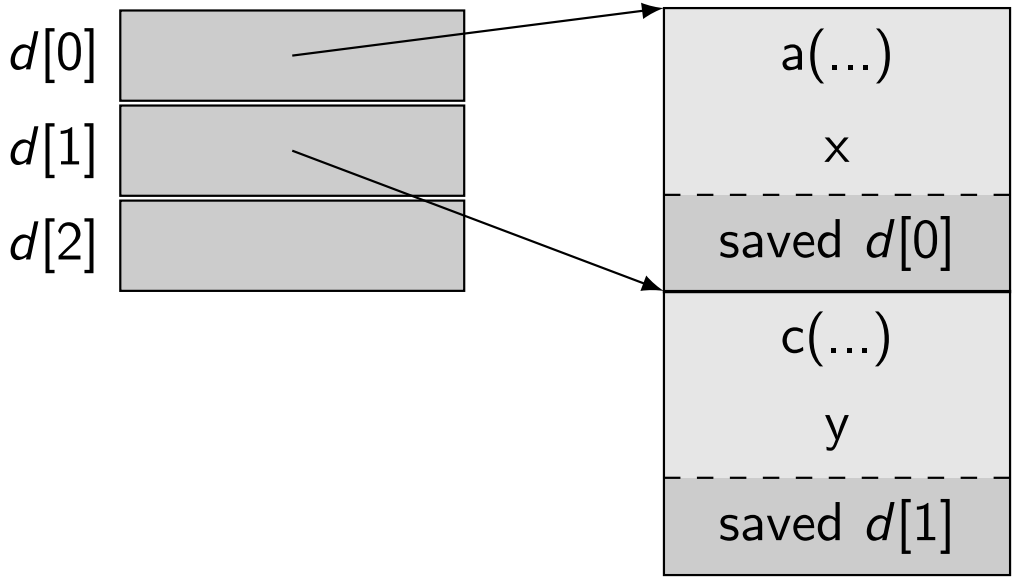
Displays



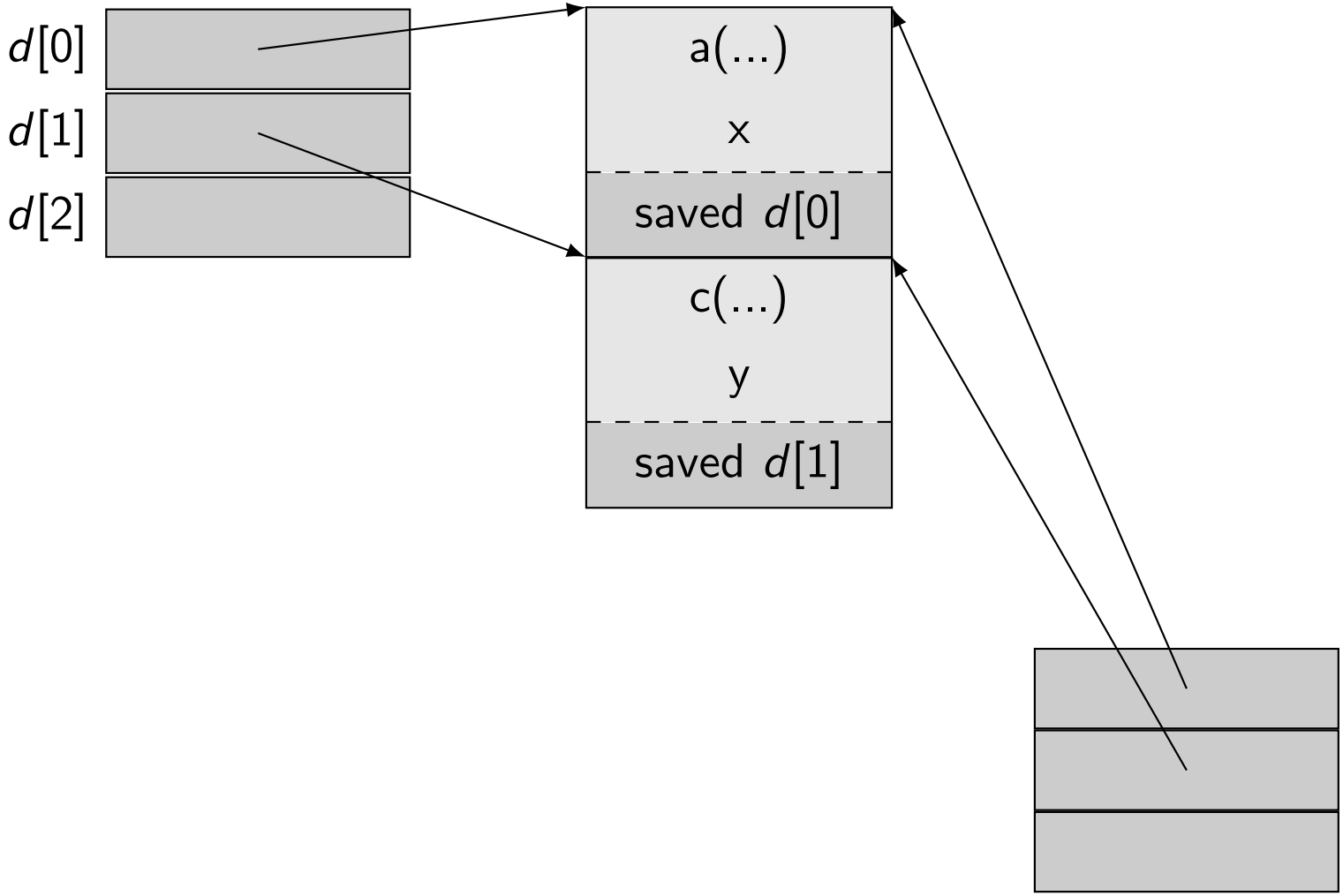
Displays


d[0]
d[1]
d[2]
main
q(1,9)
saved d[1]
q(1, 3)
saved d[1]
p(1, 3)
saved d[2]
e(1, 3)
saved d[2]

Prozedurparameter mit Displays

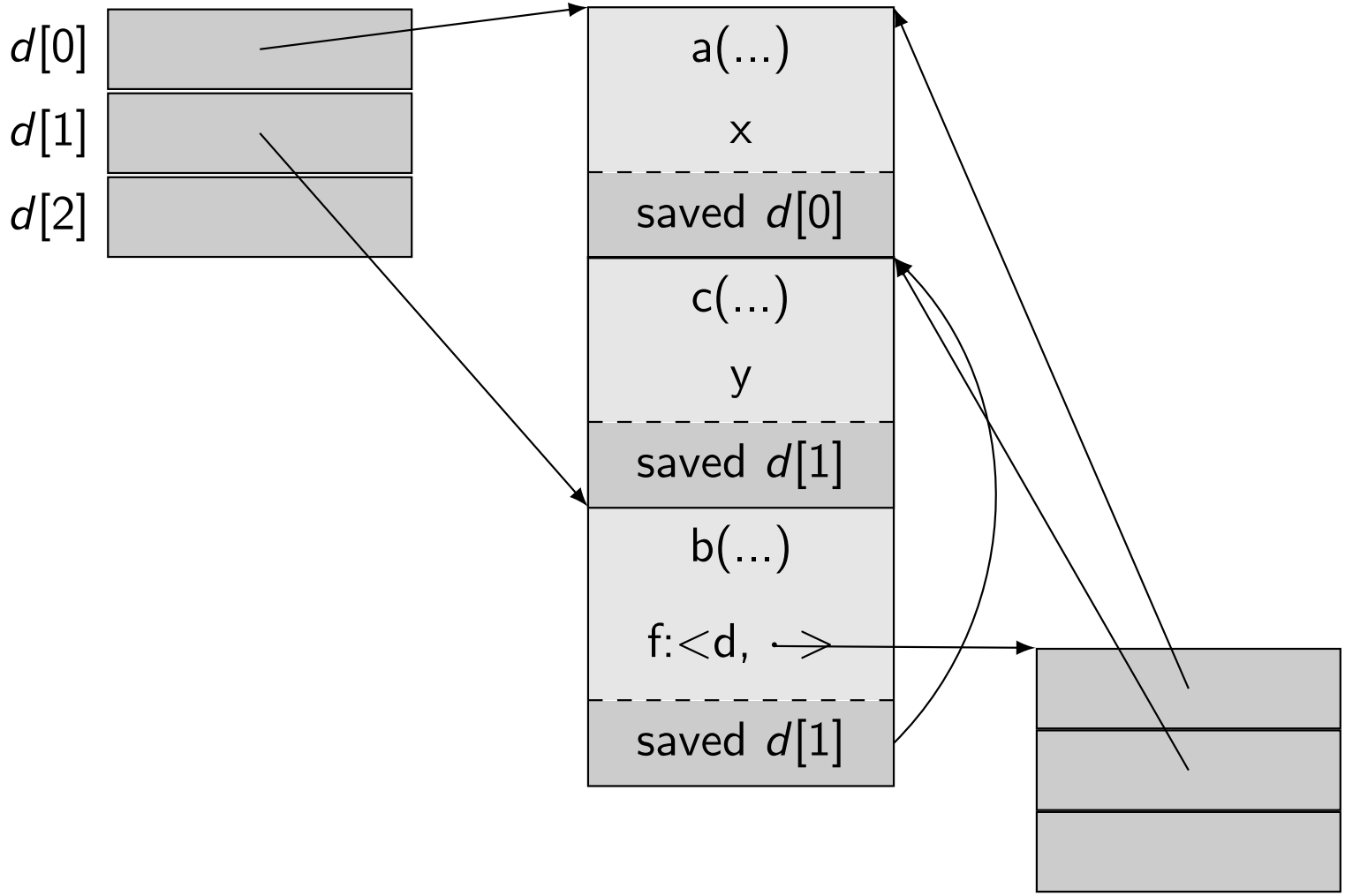


Prozedurparameter mit Displays



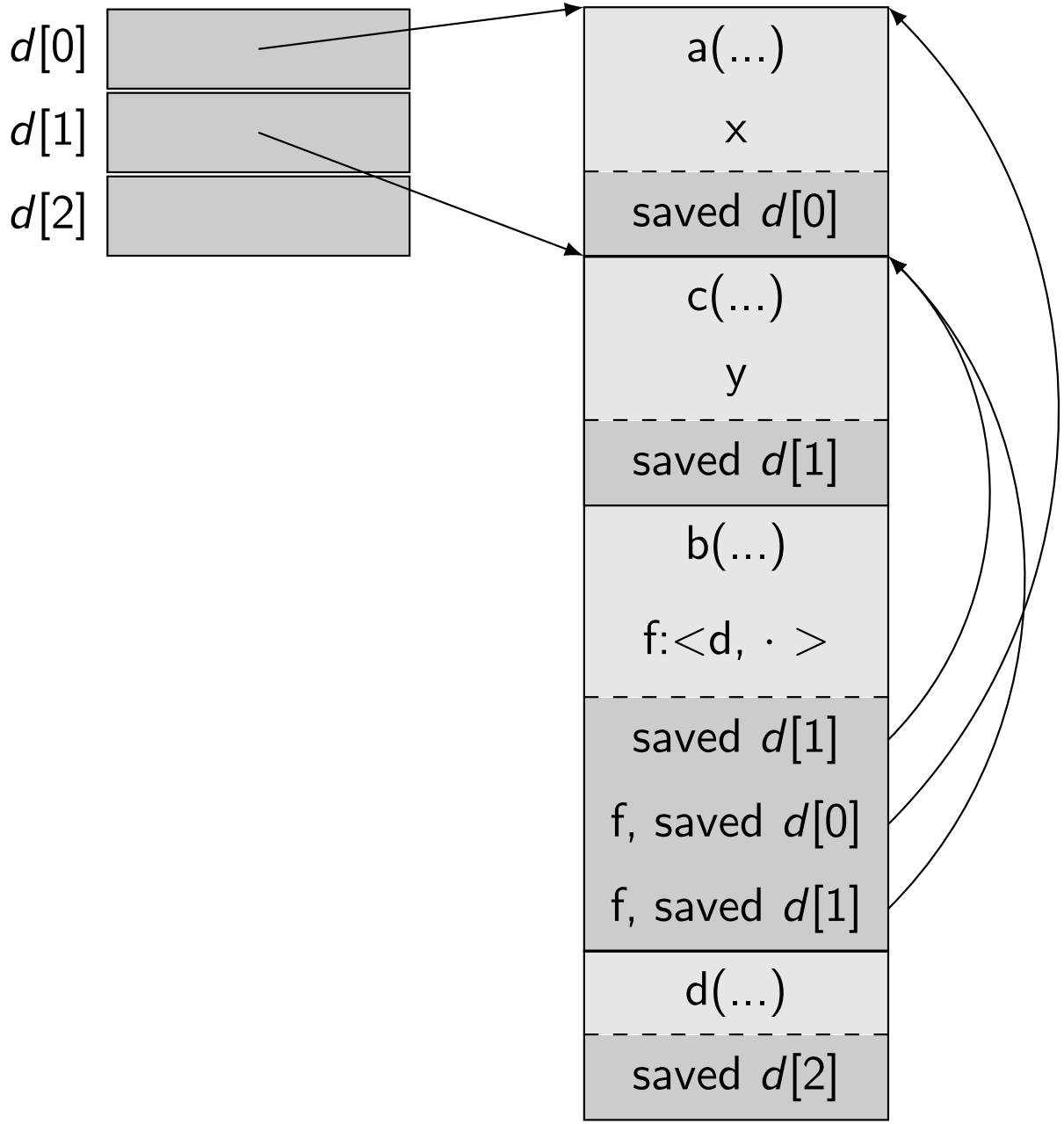
Derzeitiges Display wird zur Umgebung für Prozedurparameter d

Prozedurparameter mit Displays



Derzeitiges Display wird zur Umgebung für Prozedurparameter d

Prozedurparameter mit Displays



Literatur


[Dijkstra, 1960] E. W. Dijkstra.
Recursive Programming.
Numerische Mathematik 2, 312–318, 1960.

Dijkstra's Irrtum

Notwendige Annahme:

Eine benötige, nichtlokale Variablen liegt im *neuesten* AR einer Prozedur.

In der Literatur wird diese Eigenschaft von Programmen als Most-Recent Korrektheit bezeichnet.

Diese Annahme ist in vielen Programmiersprachen (z.B. Pascal) im Allgemeinen falsch.

Gegenbeispiel

```
int n = 1; int k = 0;
void p(void (f*)(), int &j) {
  int i=0;
  void q() {
    n++; if (n == 4) { q(); }
    n++; if (n == 7) { j++; }
    i++;
  }
  n++;
  if (n == 2) { p(q,i); }
     else     { j++; }
  if (n == 3) { f(); }
  i++; printf("%d, ", i);
}
void empty() {}
p(empty,k);
printf("%d", k);
```

Ausgabe:

Pascal-Semantik:
`1, 4, 1`

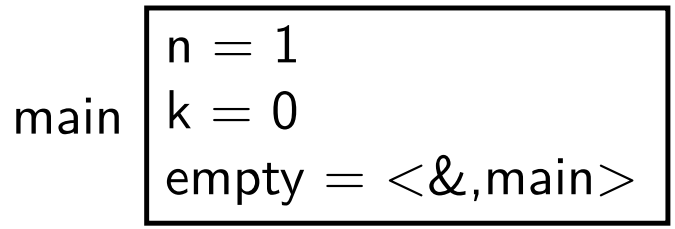
Most-Recent-Semantik:
`3, 3, 0`

Naives Kopieren:
(`int j` statt call-by-reference)
`5, 1, 0`

Ablauf des Gegenbeispiels

```
int n = 1; int k = 0;
void p(void (f*)(), int &j) {
  int i=0;
  void q() {
    n++; if (n == 4) { q(); }
    n++; if (n == 7) { j++; }
    i++;
  }
  n++;
  if (n == 2) { p(q,i); }
     else     { j++; }
  if (n == 3) { f(); }
  i++; printf("%d, ", i);
}
void empty() {}
p(empty,k);
printf("%d", k);
```

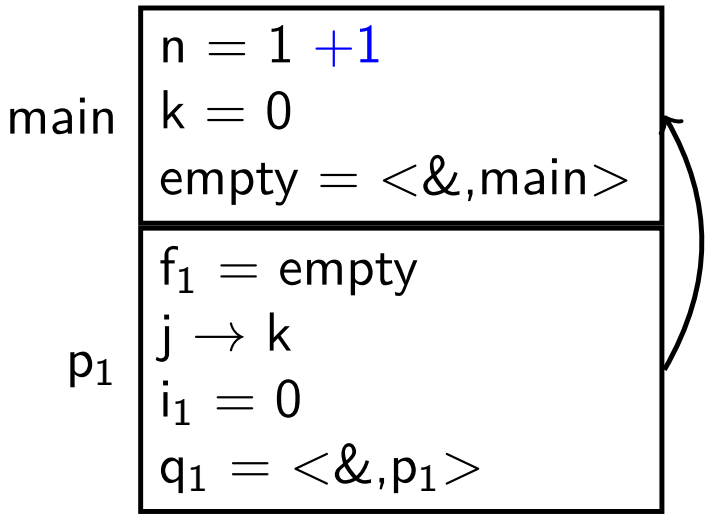
Ausgabe: 1, 4, 1



Ablauf des Gegenbeispiels

```
int n = 1; int k = 0;
void p(void (f*)(), int &j) {
  int i=0;
  void q() {
    n++; if (n == 4) { q(); }
    n++; if (n == 7) { j++; }
    i++;
  }
  n++;
  if (n == 2) { p(q,i); }
     else     { j++; }
  if (n == 3) { f(); }
  i++; printf("%d, ", i);
}
void empty() {}
p(empty,k);
printf("%d", k);
```

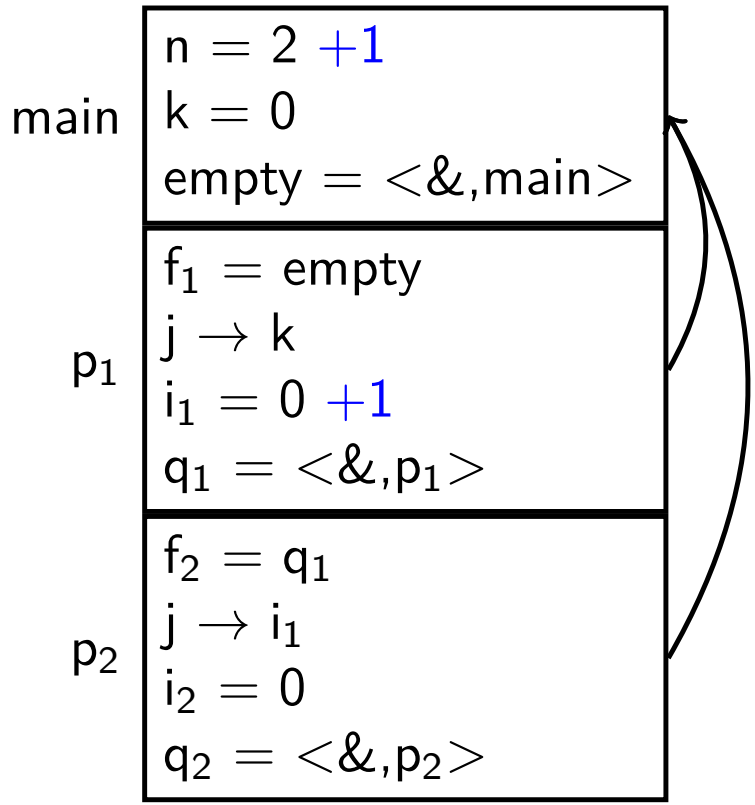
Ausgabe: 1, 4, 1



Ablauf des Gegenbeispiels

```
int n = 1; int k = 0;
void p(void (f*)(), int &j) {
  int i=0;
  void q() {
    n++; if (n == 4) { q(); }
    n++; if (n == 7) { j++; }
    i++;
  }
  n++;
  if (n == 2) { p(q,i); }
     else     { j++; }
  if (n == 3) { f(); }
  i++; printf("%d, ", i);
}
void empty() {}
p(empty,k);
printf("%d", k);
```

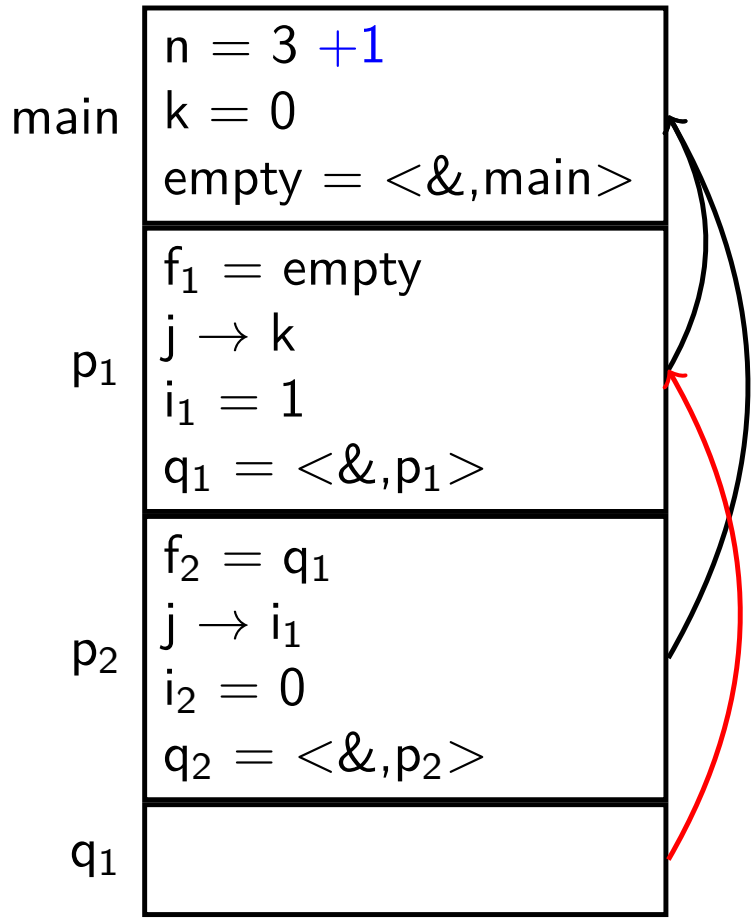
Ausgabe: 1, 4, 1



Ablauf des Gegenbeispiels

```
int n = 1; int k = 0;
void p(void (f*)(), int &j) {
  int i=0;
  void q() {
    n++; if (n == 4) { q(); }
    n++; if (n == 7) { j++; }
    i++;
  }
  n++;
  if (n == 2) { p(q,i); }
     else     { j++; }
  if (n == 3) { f(); }
  i++; printf("%d, ", i);
}
void empty() {}
p(empty,k);
printf("%d", k);
```

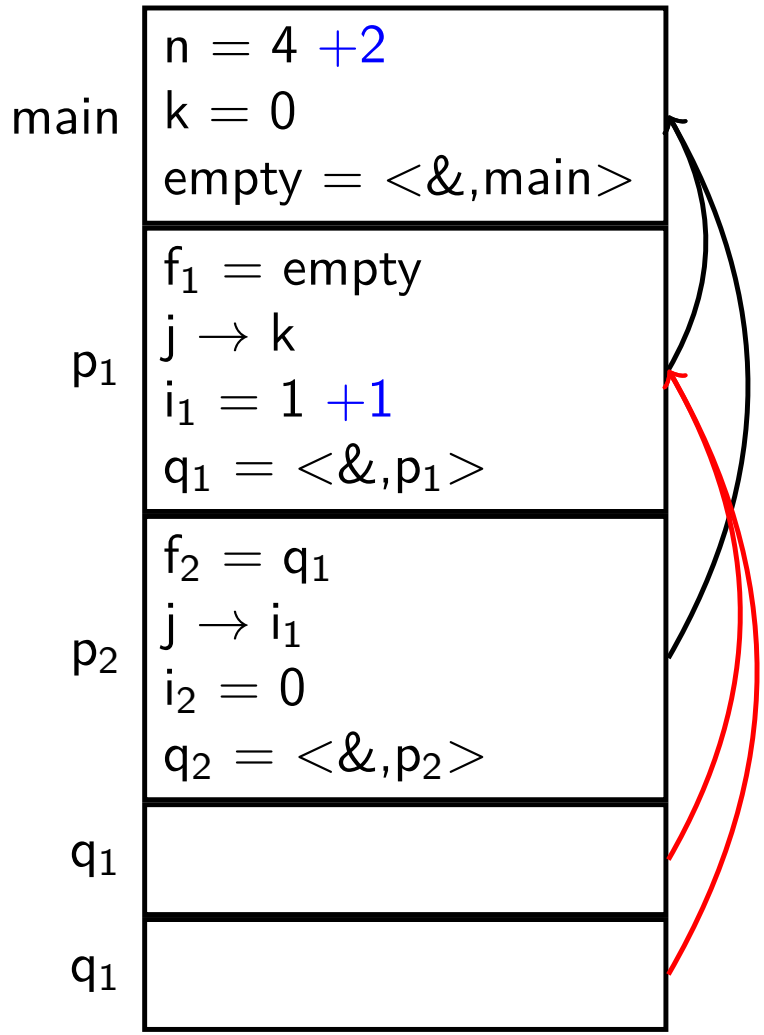
Ausgabe: 1, 4, 1



Ablauf des Gegenbeispiels

```
int n = 1; int k = 0;
void p(void (f*)(), int &j) {
  int i=0;
  void q() {
    n++; if (n == 4) { q(); }
    n++; if (n == 7) { j++; }
    i++;
  }
  n++;
  if (n == 2) { p(q,i); }
     else     { j++; }
  if (n == 3) { f(); }
  i++; printf("%d, ", i);
}
void empty() {}
p(empty,k);
printf("%d", k);
```

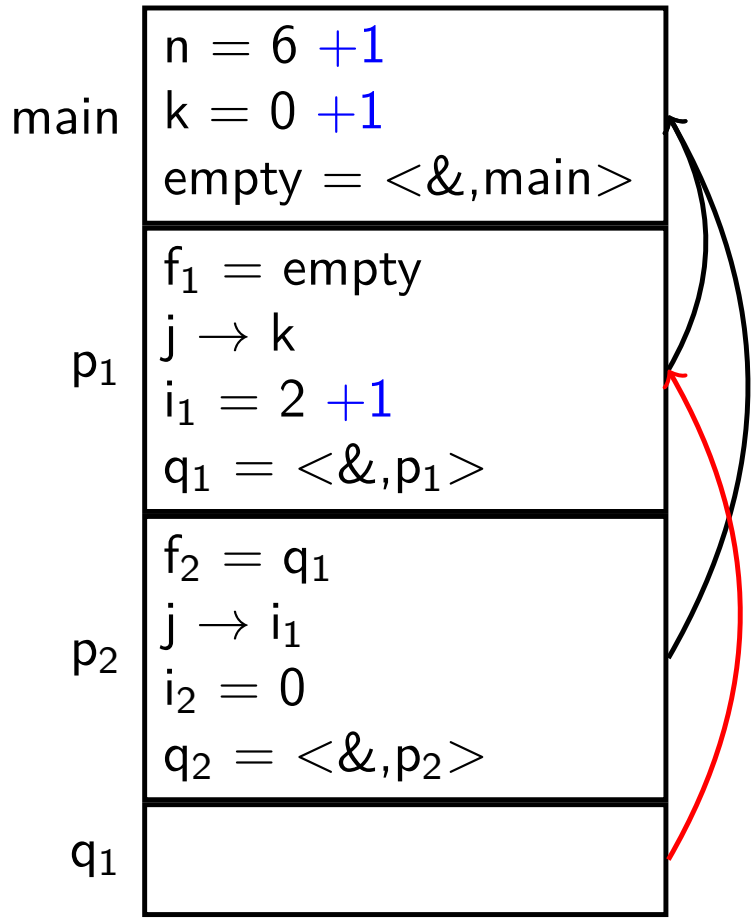
Ausgabe: 1, 4, 1



Ablauf des Gegenbeispiels

```
int n = 1; int k = 0;
void p(void (f*)(), int &j) {
  int i=0;
  void q() {
    n++; if (n == 4) { q(); }
    n++; if (n == 7) { j++; }
    i++;
  }
  n++;
  if (n == 2) { p(q,i); }
     else     { j++; }
  if (n == 3) { f(); }
  i++; printf("%d, ", i);
}
void empty() {}
p(empty,k);
printf("%d", k);
```

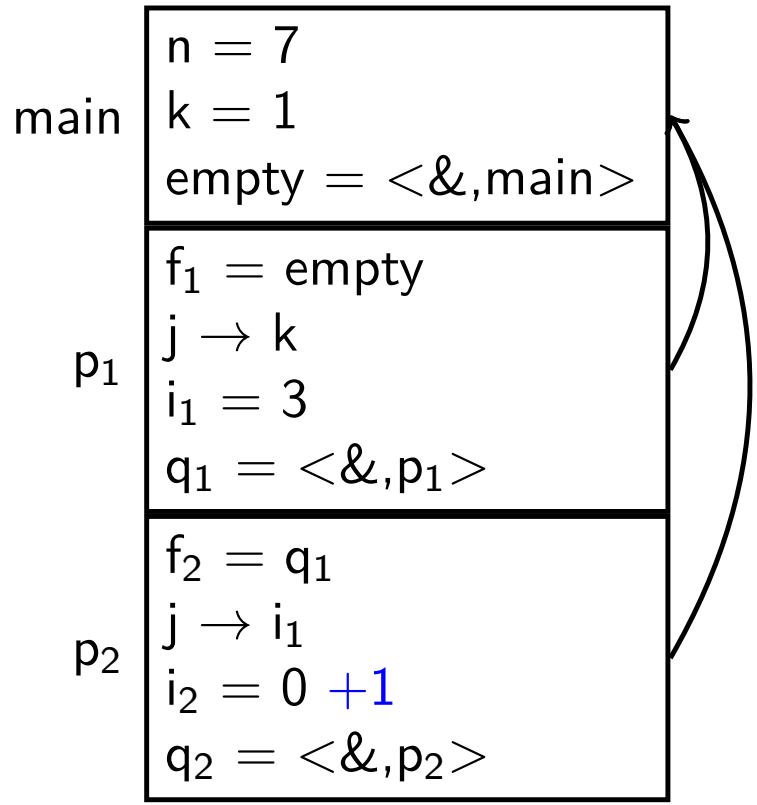
Ausgabe: 1, 4, 1



Ablauf des Gegenbeispiels

```
int n = 1; int k = 0;
void p(void (f*)(), int &j) {
  int i=0;
  void q() {
    n++; if (n == 4) { q(); }
    n++; if (n == 7) { j++; }
    i++;
  }
  n++;
  if (n == 2) { p(q,i); }
     else     { j++; }
  if (n == 3) { f(); }
  i++; printf("%d, ", i);
}
void empty() {}
p(empty,k);
printf("%d", k);
```

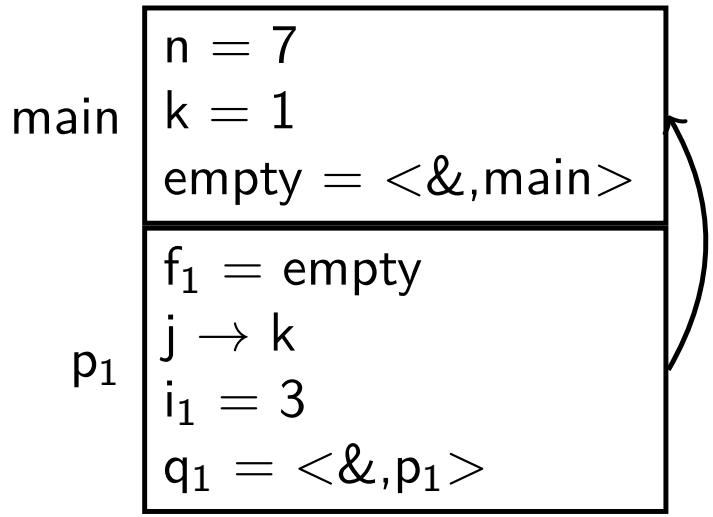
Ausgabe: 1, 4, 1



Ablauf des Gegenbeispiels

```
int n = 1; int k = 0;
void p(void (f*)(), int &j) {
  int i=0;
  void q() {
    n++; if (n == 4) { q(); }
    n++; if (n == 7) { j++; }
    i++;
  }
  n++;
  if (n == 2) { p(q,i); }
     else     { j++; }
  if (n == 3) { f(); }
  i++; printf("%d, ", i);
}
void empty() {}
p(empty,k);
printf("%d", k);
```

Ausgabe: 1, 4, 1



Ablauf des Gegenbeispiels

```
int n = 1; int k = 0;
void p(void (f*)(), int &j) {
  int i=0;
  void q() {
    n++; if (n == 4) { q(); }
    n++; if (n == 7) { j++; }
    i++;
  }
  n++;
  if (n == 2) { p(q,i); }
     else     { j++; }
  if (n == 3) { f(); }
  i++; printf("%d, ", i);
}
void empty() {}
p(empty,k);
printf("%d", k);
```

Ausgabe: 1, 4, 1

main

n = 7
k = 1
empty = <&,main>

Ablauf des Gegenbeispiels

Ausgabe: `1, 4, 1`

```
int n = 1; int k = 0;
void p(void (f*)(), int &j) {
  int i=0;
  void q() {
    n++; if (n == 4) { q(); }
    n++; if (n == 7) { j++; }
    i++;
  }
  n++;
  if (n == 2) { p(q,i); }
     else     { j++; }
  if (n == 3) { f(); }
  i++; printf("%d, ", i);
}
void empty() {}
p(empty,k);
printf("%d", k);
```

Die Lehre aus Dijkstra's Irrtum

Closures auch bei Verwendung von Displays nötig

- Prozedurparameter a benötigt das Display vom Zeitpunkt des Funktionsaufrufes f(a)
- Aufrufer muss derzeitiges Display als Umgebung übergeben

$\Rightarrow$ Bei Verwendung von Closures bekommt man die richtige Ausgabe (1,4,1) für das Gegenbeispiel.

Prozeduraufruf Aufgaben

1. Zustand sichern
2. Neue Prozedurschachtel generieren
3. Kellerpegel setzen
4. Statischen und dynamischen Vorgänger eintragen
5. Parameterübergabe
6. Unterprogrammsprung
7. Rücksprungadresse sichern (bei Rekursion/geschachteltem Aufruf)
8. Prozedurrumpf ausführen
9. Rücksprung
10. Ergebnisrückgabe
11. Kellerpegel zurücksetzen, Zustand wiederherstellen

Reihenfolge teilweise veränderbar

Zustand sichern

- Alle Register
- Auswahl bestimmter Register
- Spezialfall Registerfenster (SPARC, Itanium)

Probleme:

- Register sichern sehr zeitaufwendig
  (n bzw. $2n$ Speicherzugriffe)
- bei automatischer Speicherbereinigung:
  enthält die Sicherung Referenzen?

Geschachtelte Prozeduren

Prozedur p innerhalb einer Prozedur p' deklarierbar (z.B. in Pascal, Modula)

Prozedur p' ist statischer Vorgänger sv von Prozedur p

Lokale Variablen v' von p' auch in p gültig

Addressierung:

$$\begin{aligned} \text{adresse}(v) &= \text{speicher}[\text{speicher}[\text{baseptr}] + r_{sv}\ ] + r_v \\ &= <<\text{baseptr}> + r_{sv}> + r_v \end{aligned}$$

Dynamischer und statischer Vorgänger

Dynamischer Vorgänger: Verweis auf Activation Record Aufrufer
Statischer Vorgänger: Verweis auf Activation Record der statisch umfassenden Prozedur

- Statischer Vorgänger unnötig, wenn alle Prozeduren auf Schachtelungstiefe 1, z.B. C oder viele OO-Sprachen
- aber Vorsicht bei inneren Klassen in Java
- Vorgänger eintragen, um bei Rückkehr und bei Zugriff auf globale Größen den richtigen Activation Record zu finden

Laufzeitkeller unnötig für Sprachen ohne Rekursion, z.B. ursprüngliches Cobol oder Fortran (nur statische Variable)
Implementierung:

- Ausprogrammieren
- Spezialbefehl, z.B. auf MC 680x0 für dynamische Vorgänger

Rücksprungadresse sichern

Rücksprungadresse in den Keller:

- Spezialbefehl (680x0, IA-32)
- in speziellem Register (RISC), von dort abholen

Bei spezieller Anordnung der Elelemente im Activation Record:
Ausprogrammieren nötig, da Spezialbefehle diese Anordnung nicht berücksichtigen

Parameter–, Ergebnisübergabe

- Wertaufruf
- Wert-Ergebnis-Aufruf
- Referenzaufruf
- Namensaufruf (wie gebundene Prozedur behandeln)
- Ergebnisaufruf

Alternativen

- Aufrufendes Programm oder Unterprogramm berechnet Parameter
- Aufrufendes Programm oder Unterprogramm speichert Resultat
- Spezialfall: Parameter oder Resultat im Register bei Ergebnisaufruf

Wert-/Ergebnisaufruf

Parameter ist lokale Variable

- bei Wertaufruf vom Aufrufer initialisiert
- bei Ergebnisaufruf nach Prozedurende Wert vom Aufrufer abgeholt (Prozedur kann Ergebnis selbst abspeichern, wenn sie zusätzlich die Adresse des Ergebnisparameters kennt)

Funktionsergebnisse wie Ergebnisparameter behandeln

Aus Effizienzgründen bei geringer Parameterzahl: Argument und/oder Ergebnis in Register übergeben (unabhängig von Systemaufrufkonvention; nicht möglich bei gebundenen Prozeduren oder polymorphem Aufruf!)

Daher: zuerst neuen Activation Record anlegen, dann Argumente berechnen

Bei Wertübergabe von Arrays wird der dynamische Teil des neuen Activation Record um das Array verlängert

Referenzaufruf

Parameter ist lokale Variable, wird vom Aufrufer mit Adresse des Arguments initialisiert

- wenn Argument ein Ausdruck (keine Variable) ist: Argument an neue Hilfsvariable beim Aufrufer zuweisen, Adresse Hilfsvariable übergeben
- Schutz vor unzulässiger Zuweisung an den Ausdruck, z.B. in Fortran

alle Zugriffe auf den Parameter in der Prozedur haben eine zusätzliche Indirektionsstufe: der Wert des Parameters (Adresse) kann explizit weder gelesen noch überschrieben werden
Referenzaufruf nur möglich, wenn Aufrufer und Aufgerufener im gleichen Adressraum. Im verteilten System simuliert durch Übergabe einer Lese- und einer Schreibprozedur (get, set).

Vorsichtsmaßnahmen

Zuerst Activation Record einrichten, dann Argumente berechnen

- Argumentberechnung kann zu weiteren Aufrufen führen!
- (Kollision mit der Halde vermeiden)

Activation Record des Aufgerufenen erst nach Abholen der Ergebnisse streichen

- (Kollision mit der Halde vermeiden)

Rückhaltestrategie

Annahme: eine gebundene Prozedur p besitzt gebundene Parameter aus der Prozedur p', in der gebunden wird (also aus der Activation Record von p'). p werde erst nach Verlassen von p' aufgerufen. Bei

- Wertaufruf: kein Problem
- Ergebnisaufruf, Referenzaufruf: Activation Record von p' muss erhalten bleiben, um auf den Parameter zugreifen zu können: Rückhaltestrategie (retention strategy)
- Namensaufruf oder gebundener Parameter ist lokale Prozedur von p': ebenfalls Activation Record zurückhalten.

Implementierung: Activation Record wie beim Kaktuskeller auf die Halde.

Notwendig z.B. in Simula 67, funktionalen Sprachen, bei Strömen in Sather-K. Die Vermeidung der Situation führt zu Beschränkungen in vielen anderen Sprachen.

Polymorpher Unterprogrammaufruf

- Wenn alle Untertypen zur Übersetzungszeit bekannt: typecase auf ersten Parameter.
- Bei getrennter Übersetzung oder dynamischem Nachladen: Eintrag in Haschtabelle mit Typeintrag liefert richtige Unterprogrammadresse.
  Je eine Haschtabelle für jeden Vererbungsbaum, Tabelle wird vom Binder konstruiert.
  Also: alle polymorphen Aufrufe mit zusätzlicher Indirektionsstufe belastet.

Trick: Merke Typeintrag und Adresse des jeweils zuletzt aufgerufenen Unterprogramms; teste, ob nächster Aufgerufene denselben Typeintrag besitzt und springe dann direkt, sonst über Haschtabelle