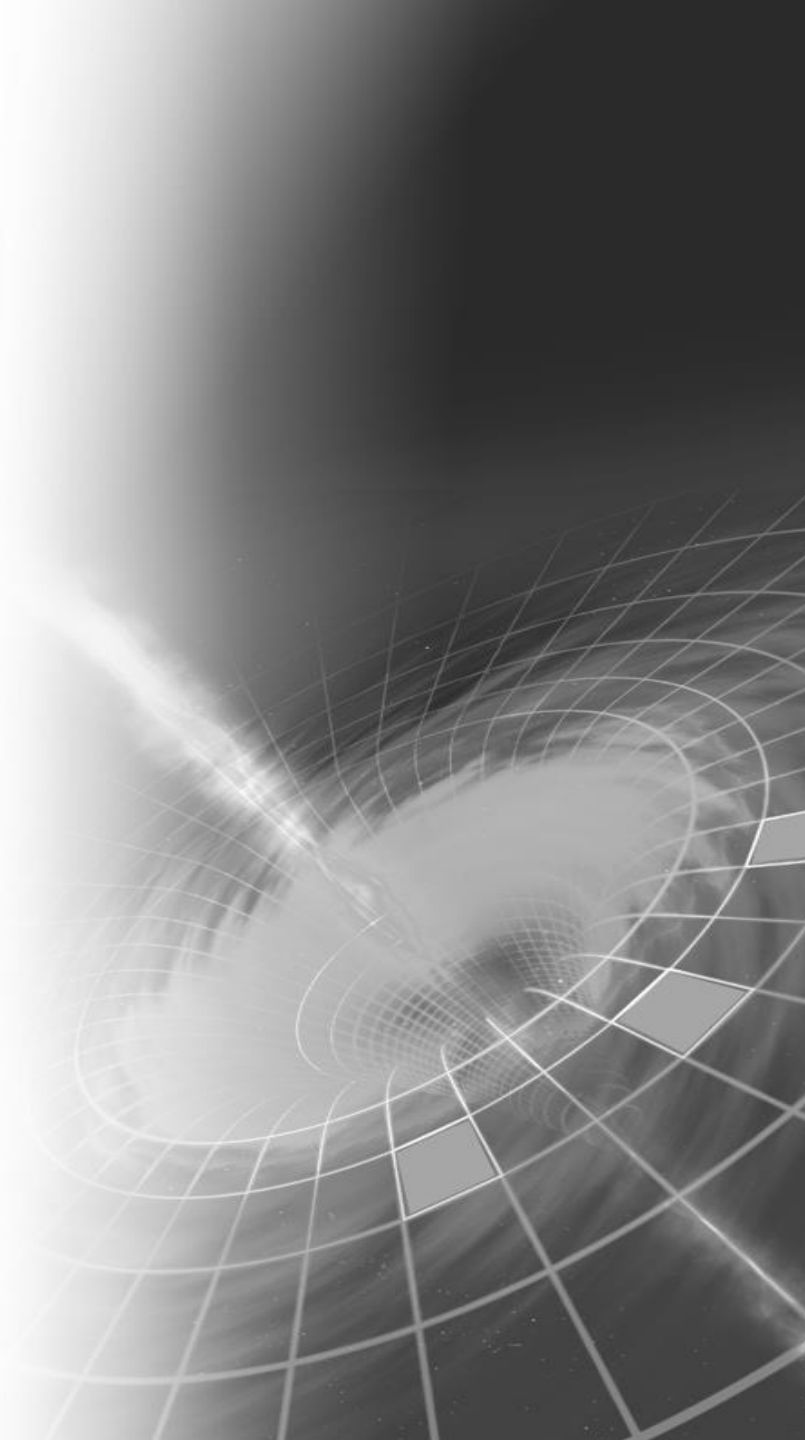
Bachelor-Programm

Compilerbau

im SoSe 2014

Prof. Dr. Joachim Fischer
Dr. Klaus Ahrens
Dipl.-Inf. Ingmar Eveslage

fischer@informatik.hu-berlin.de



Position

- Teil I
  Die Programm
- Teil II
  Methodische
- Teil III
  Entwicklung

- Kapitel 1
  Compilationsprozess
- Kapitel 2
  Formalismen zur Sprachbeschreibung
- Kapitel 3
  Lexikalische Analyse: der Scanner
- Kapitel 4
  Syntaktische Analyse: der Parser
- Kapitel 5
  Parser-Generatoren: Yacc, Bison
- Kapitel 6
  Statische Semantikanalyse &
  Drei-Adress-Codegenerierung
- Kapitel 7
  Laufzeitsysteme

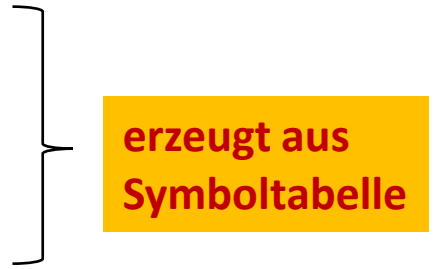
3-Adress-Code (allgemein)

- übliche Zwischensprache, die in vielen Varianten vorkommt
- lässt sich gut in 1- und 2-Adressbefehle transformieren (moderne Maschinen stellen Befehle mit mehr als einer Adresse zur Verfügung, Transformation dadurch noch einfacher)
- weitere Merkmale:
  - nur elementare Datentypen (oft aber Felder)
  - Typisierung oft nur implizit
  - keine zusammengesetzten Ausdrücke
  - sequentielle Ausführung,
  - Sprünge und Prozeduraufruf als Anweisungen
  - benannte Variablen wie in Hochsprache
  - unbeschränkte Anzahl von *Hilfsvariablen* (engl.*temporaries*)

Syntax	Erläuterung
x := y **bop** z x := **uop** y x := y	x Variable (global, lokal, Parameter, Hilfsvariable); y,z Variable oder Konstante; bop binärer Operator; uop unärer Operator
goto L **if** x **cop** y **goto** L	Sprung bzw. bedingter Sprung zur Marke L; cop Vergleichsoperator (nur prozedurlokale Sprünge)
x := a[i] a[i] := y x := &a x := * y x := y	a eindimensionales Feld a globale, lokale Variable oder Parameter; &a Adresse von a; * Dereferenzierungsoperator
param x **call** p **return** y	Befehle für Prozeduraufrufe: Aufruf p(x1,..,xn) wird codiert als: param x1 ... param xn call p return y bewirkt Rücksprung mit optionaler Rückgabe von Ergebnis y

Struktur eines 3-Adress-Code-Programms

1. einer Liste globaler Variablen
2. einer Liste von Prozeduren mit Parametern und lokalen Variablen
3. Auszeichnung einer Hauptprozedur



Dabei ist jede Prozedur

eine Sequenz von 3-Adress-Befehlen als Rumpf

Weitere Restriktionen

- Ann.: 3-Adress-Code enthält keine Marken,
  zu denen es keinen Sprungbefehl gibt

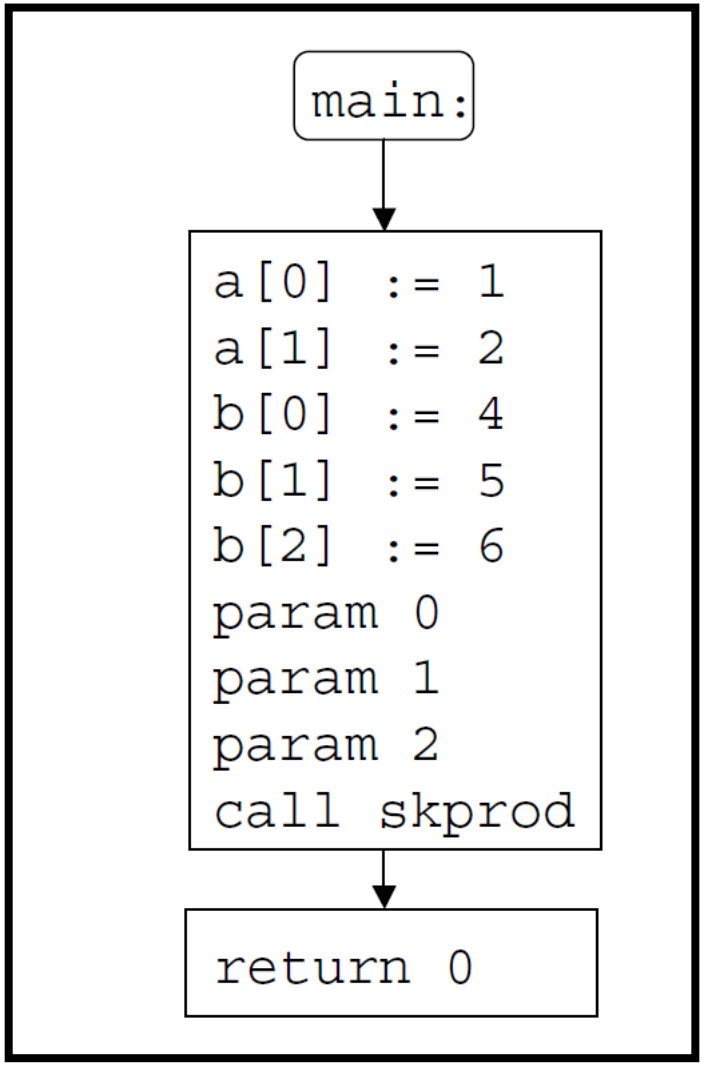
- Eine Sequenz von 3-Adress-Befehlen lässt sich eindeutig in sogenannte Basisblöcke zerlegen

Ein ***Basisblock*** B ist eine maximale Sequenz von Befehlen,
so dass

- Sprungbefehle, Prozeduraufrufe und return-Befehle
  nur am Ende von B vorkommen und zwar maximal ein solcher Befehl
- Marken nur vor dem ersten Befehl eines Basisblocks stehen

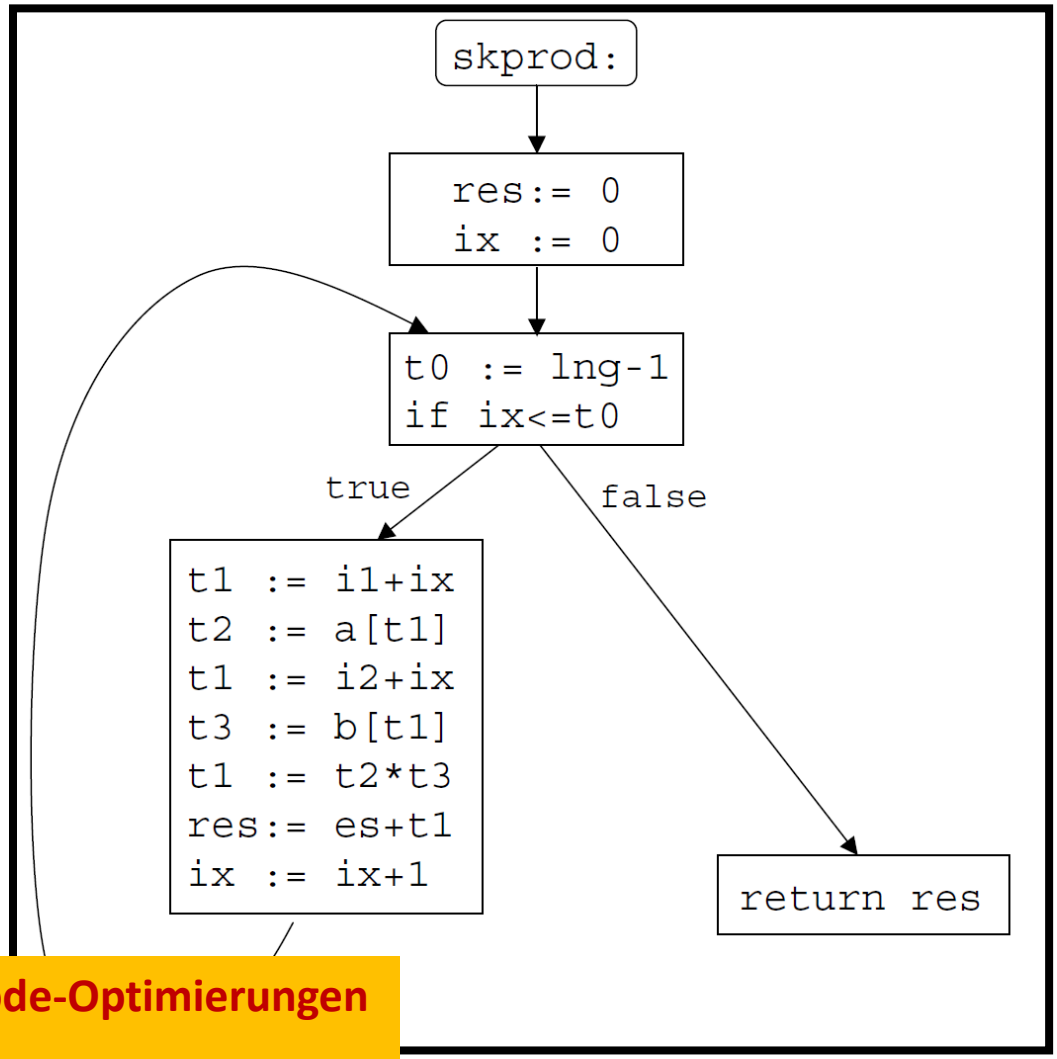
Beispiel

```
int a[2];
int b[7];
int skprod(int i1, int i2, int lng) {
... }
int main( ) {
        a[0] = 1;
        a[1] = 2;
        b[0] = 4;
        b[1] = 5;
        b[2] = 6;
        skprod(0,1,2);
        return 0;
}
```



Beispiel

```
int skprod(int i1, int i2, int lng) {
        int ix,
            res = 0;
        for( ix=0; ix <= lng-1; ix++ ){
                res += a[i1+ix] * b[i2+ix];
        }
        return res;
}
```



Basisblockstrukturierung ist die Grundlage für **Code-Optimierungen**
wird hier nicht weiter vertieft

Programme in 3-Adress-Code

… bestehen aus

- einer Liste globaler Variablen
- einer Liste von Prozeduren mit Parametern und lokalen Variablen
- einer Hauptprozedur (Einstiegspunkt)

aus Symboltabelle

Jede Prozedur besitzt eine Sequenz von 3-Adress-Befehlen

Zwischensprachen werden häufig als Bindeglied zwischen Übersetzer und Laufzeitsystem eingesetzt

Übersetzung von Prozeduraufrufen

… bedeutet mehr als die Generierung von Zwischencode-Befehlen

Drei-Adressbefehle

aus
Prozeduraufruf
p(x1, ..., xn) im Quelltext
entsteht durch Compilierung:

```
param x1
param x2
...
param xn
call p, n
```

Zusätzlich sind vom Compiler Laufzeitroutinen
mit folg. Leistungen bereitzustellen:

zum Prozedurruf

- Aufbau von Speicherplatz für das **Aktivierungselement** der gerufenen Prozedur
- Sicherung der Verfügbarkeit der Parameter
- Setzen eines Umgebungszeigers für globale Größen
- Retten des Zustandes der rufenden Prozedur (so dass auch die spätere Rückkehr vollzogen werden kann)

zum Return

- Ergebnisbereitstellung für die Umgebung
- Wiederherstellung des **Aktivierungssegmentes** der rufenden Prozedur
- Sprung zur Rückkehradresse

Position

- Teil I
  Die Programm
- Teil II
  Methodische
- Teil III
  Entwicklung

- Kapitel 1
  Compilationsprozess
- Kapitel 2
  Formalismen zur Sprachbeschreibung
- Kapitel 3
  Lexikalische Analyse: der Scanner
- Kapitel 4
  Syntaktische Analyse: der Parser
- Kapitel 5
  Parser-Generatoren: Yacc, Bison
- Kapitel 6
  Statische Semantikanalyse &
  Drei-Adress-Codegenerierung
- Kapitel 7
  Laufzeitsysteme

- 7.1
  Begriffsklärung
- 7.2
  Speicherorganisation
- 7.3
  Stack-Verwaltung

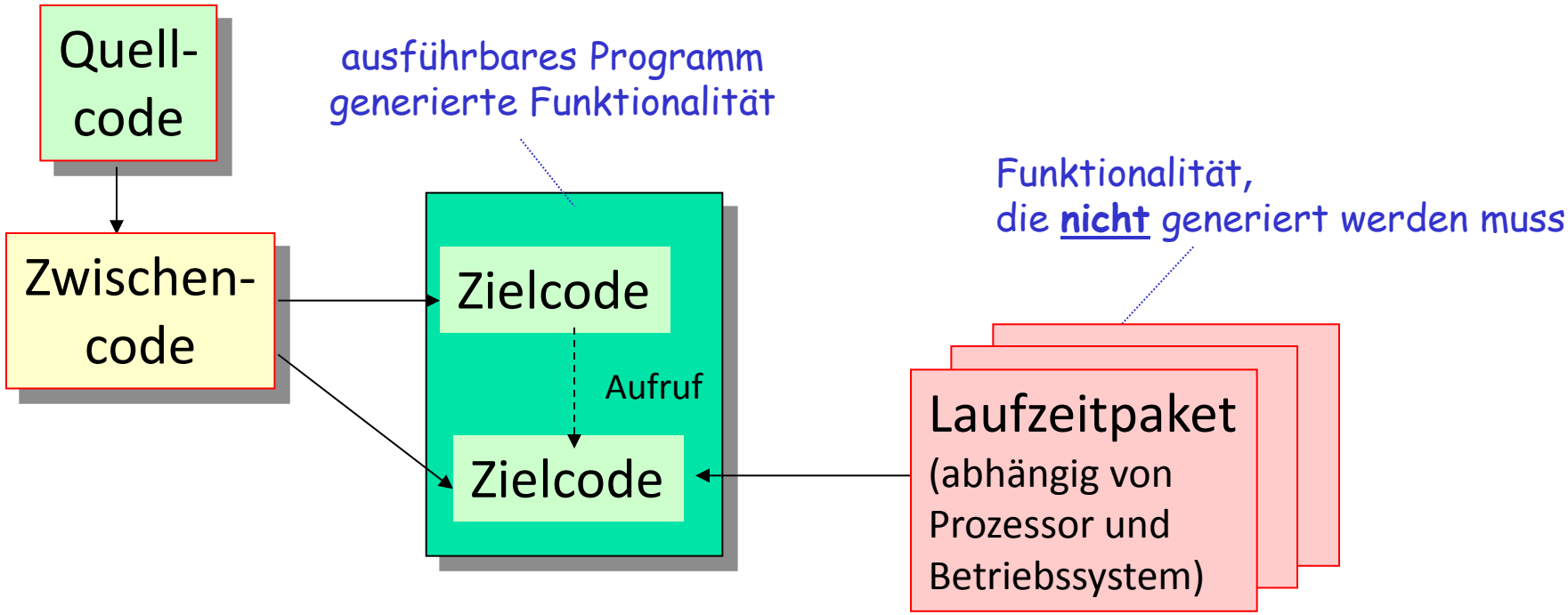
Laufzeitsysteme

- Ein **Laufzeitsystem** ist ein Computerprogramm,
  - das gemeinsam mit dem Anwendungsprogramm (unser Quellprogramm),
    das selbst nicht direkt mit dem Betriebssystem kommunizieren kann,
    ausgeführt wird und dabei
  - die Aufgabe hat
    zwischen Anwendungsprogramm und Betriebssystem zu vermitteln

- Traditionelle Laufzeitsysteme stellen **Dienste** zur Verfügung
  z.B. für Speicherverwaltung, Koroutinenverwaltung, Threadverwaltung, I/O, etc),
  die von den Schnittstellen des Betriebssystems mehr oder weniger stark abstrahieren.

- Programmiersprachen setzen bei dieser Vermittlung verstärkt auf das Konzept der **virtuellen Maschine**.
  - Fortsetzung der Abstraktion einer Laufzeitumgebung, indem auch der Befehlssatz, auf den ein Programm zurückgreifen kann, von der konkreten Maschine abstrahiert wird.

Einordnung des Laufzeitsystems



abhängig
von dynamischer
Semantikdefinition
der Quellsprache

klassische Aufgaben eines Laufzeitsystems

- Speicherplatzzuweisung und Freigabe von Datenobjekten
- dynamische Typtests
- Verkettung von Prozedur-Rufen, Parameterübergabe
- Schnittstelle zum Betriebssystem (Ein- und Ausgabe)
- ...

Laufzeitumgebung

… lässt Anwendungsprogramme plattformunabhängig ablaufen,
d.h. unabhängig von verwendeter Hardware und Betriebssystem

- falls Laufzeitumgebung auf mehreren Plattformen verfügbar, können Programme auf all diesen Plattformen ausgeführt werden

➔ Eine Laufzeitumgebung stellt damit bereits selbst eine „Plattform“ dar

- Zuweilen: portable virtuelle Maschine gehört zur Laufzeitumgebung

Beispiele

1. Java Runtime Environment (Java-Klassenbibliotheken und die Java Virtual Machine zur Ausführung des Java-Bytecodes)

2. .NET-Plattform, Common Language Runtime (CLR) ist Laufzeitumgebung für C#-, Visual Basic, J#-, JScript - und C++ .Net- Programme

Position

- Teil I
  Die Programm
- Teil II
  Methodische
- Teil III
  Entwicklung

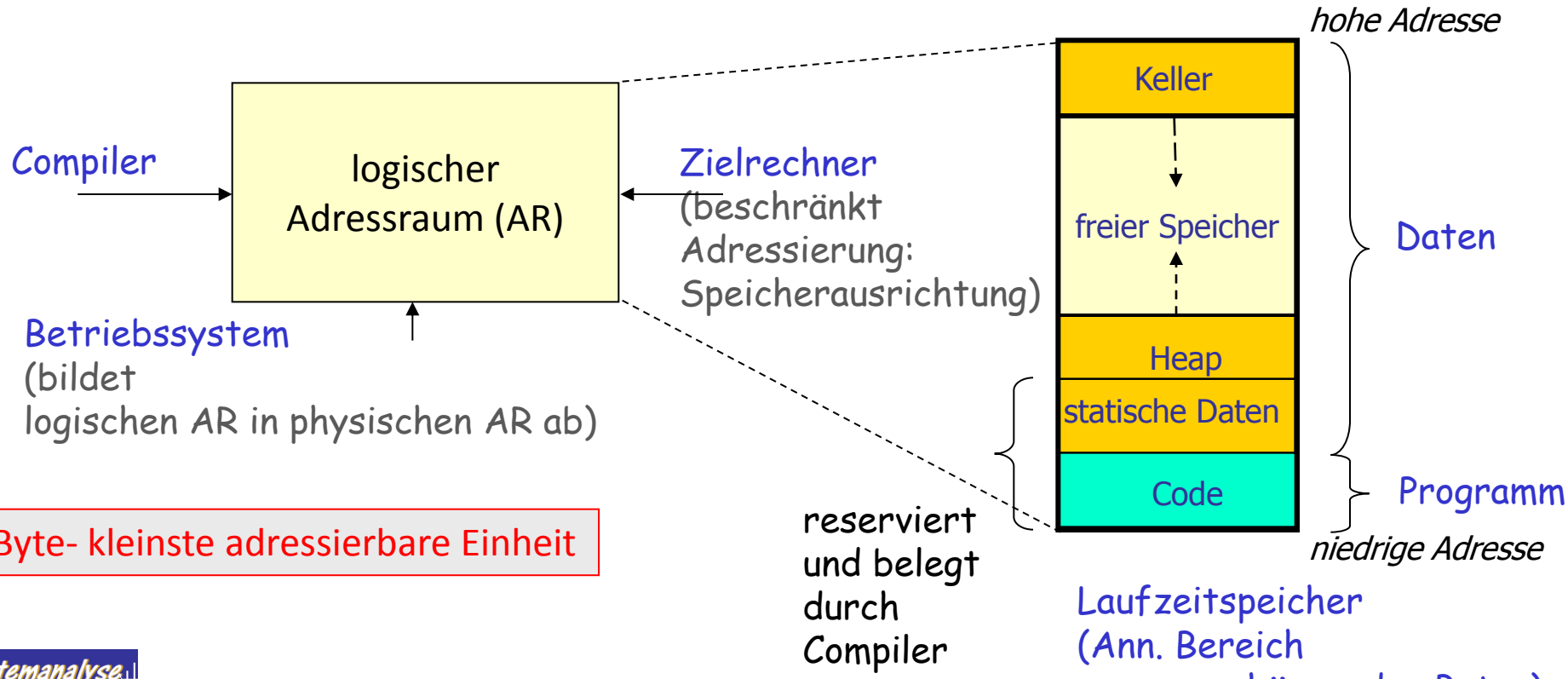
- Kapitel 1
  Compilationsprozess
- Kapitel 2
  Formalismen zur Sprac
- Kapitel 3
  Lexikalische Analyse: d
- Kapitel 4
  Syntaktische Analyse:
- Kapitel 5
  Parser-Generatoren: Ya
- Kapitel 6
  Statische Semantikana
- Kapitel 7
  Laufzeitsysteme
- Kapitel 8
  Ausblick: Codegenerier

- 7.1
  Begriffsklärung
- 7.2
  Speicherorganisation
- 7.3
  Stack
- 7.4
  Zugrif

- Allgemeine Aufgaben eines Laufzeitsystems
- Modelle zur Aufteilung des Hauptspeichers für die Programmausführung
- Laufzeitkeller zur
  a) Ausführungssteuerung (Prozedurkonzept)
  b) Datenverwaltung (Pendant zur Symboltabelle)
- Heap zur Verwaltung dynamischer Daten

Speicheraufbau (elementar)

- Zielprogramm läuft in seinem eigenen logischen Adressraum
- Compiler, Betriebssystem, Zielrechner verwalten den logischen Adressraum des Zielprogramms



Speicherorganisation (1)

Konventionen für die Speicherplatzaufteilung

Compiler realisiert Allokation

1. Speicherplatz für Programmcode
   - feste Länge, deshalb statische Allokation für Compiler(Code-Generator) möglich

2. Speicherplatz für Daten
   (globale Größen, zusätzlich generierte Größen)
   - Daten fester Länge können bereits statisch allokiert werden
     Vorteil: Compiler kann Adressen schon fest in den Code eintragen

   statische Daten

   - Daten variabler Länge müssen dynamisch allokiert werden

3. Steuerungskeller (für Pascal, C, ...)
   - beinhaltet Teil zur Verwaltung von Prozeduraktivierungen (Aktivierungssegmente) inklusive Rücksprungadresse
   - spezielle Maschinenbefehle für Stack-Manipulationen möglich (z.B. Sparc)

4. Heap
   - Allokation und Deallokation von Speicher für dynamische Objekte (C: malloc, free)

   dynamische Daten

Laufzeitsystem realisiert Allokation

Speicherorganisation (2)

- erfolgt in Abhängigkeit der angestrebten **Lebensdauer** verschiedener Klassen von Datenobjekten:

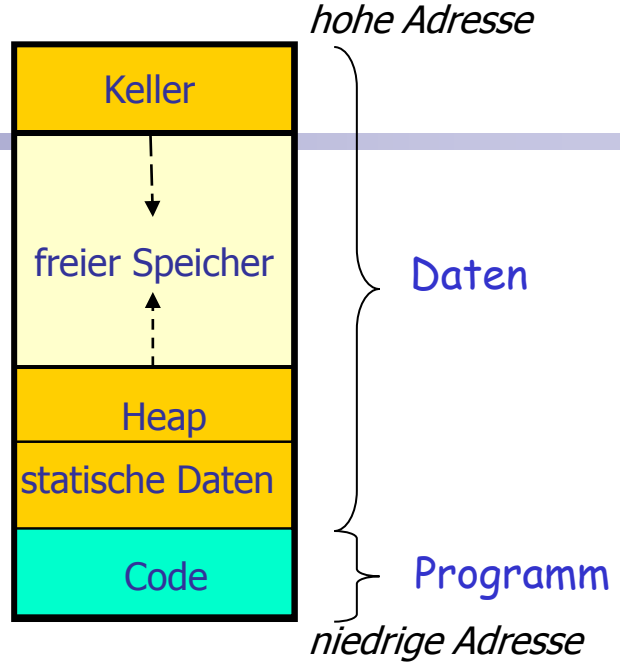
  (1) ein oder mehrere **Kellerspeicher** für Objekte,
  deren Lebensdauer geschachtelten Intervallen entspricht

  (2) eine **Halde** (Heap) für Objekte, deren Lebensdauer nicht einer solchen Disziplin gehorcht

- Kellerspeicher sind in Bereiche (Segmente/ Rahmen/ Frames) unterteilt,
  die eine statisch bestimmte Struktur aufweisen

- zur Organisation wird zusätzlich ein Satz von **Registern**, wie
  - Kellerpegel, Haldenpegel (Zeiger) und
  - Weitere **Rahmenzeiger** eingesetzt

Speicherklassen



jeder Variablen muss eine **Speicherklasse** zugewiesen werden (Basisadresse)

Speicherklasse: wo ist Speicherplatz zu allokieren?

- **statische Variablen** (in C static)
  - werden zur Übersetzungszeit allokiert
  - Speicheradressen werden bereits in den Code eingesetzt
  - eingeschränkt für Objekte fester Größe
  - Zugriff wird mit einem Namensschema kontrolliert

Achtung:
Der Linker (der einzelne Module zu einem Programm zusammenführt) muss Duplikate der statischen Variablen handhaben können (z.B.: Modul-Name/Variablen-Name)

- **globale Variablen** (für ganzes Programm zugreifbar)
  - fast identisch zur statischen Variablen
  - Namensschema garantiert universellen Zugriff

Gültigkeitsbereich von Deklarationen

- ein Bezeichner wird **lokal** zur Prozedur genannt,
  falls er zum Gültigkeitsbereich (also zu deren Deklaration) gehört
  andernfalls: **nichtlokal**

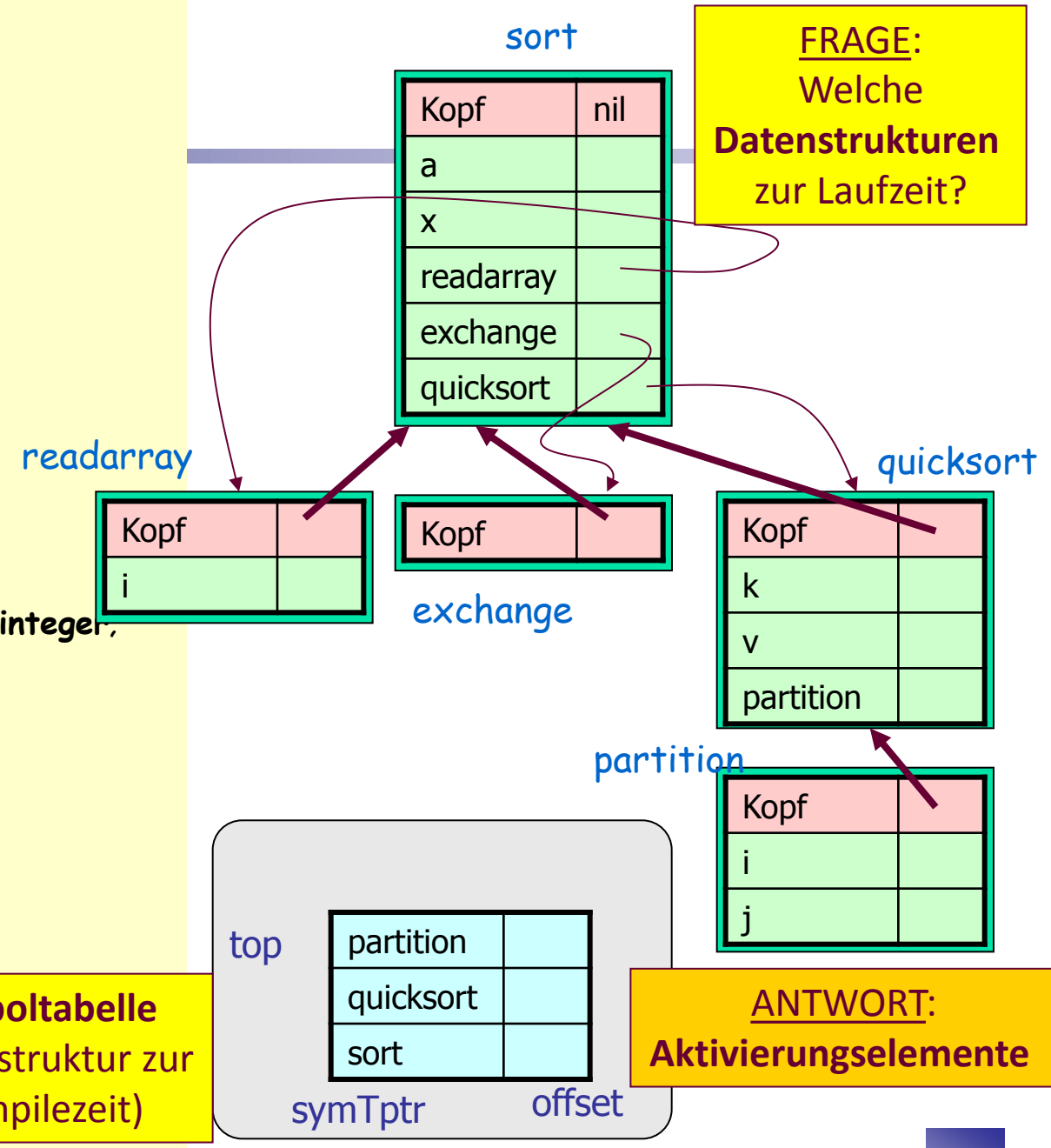
- zur Übersetzungszeit kann die **Symboltabelle** benutzt werden, um die gültige Deklaration eines Bezeichners zu finden

- zur Laufzeit muss ein entsprechender Ersatz geschaffen werden:

statisches Konzept	dynamisches Konzept
Definition einer Prozedur	Aktivierung einer Prozedur
Deklaration eines Bezeichners	Bindung dieses Bezeichners
Gültigkeitsbereich einer Deklaration	Lebenszeit der Bindung

Laufzeitkeller
zur Verwaltung der Daten entsprechend der deklarierten Gültigkeitsbereiche

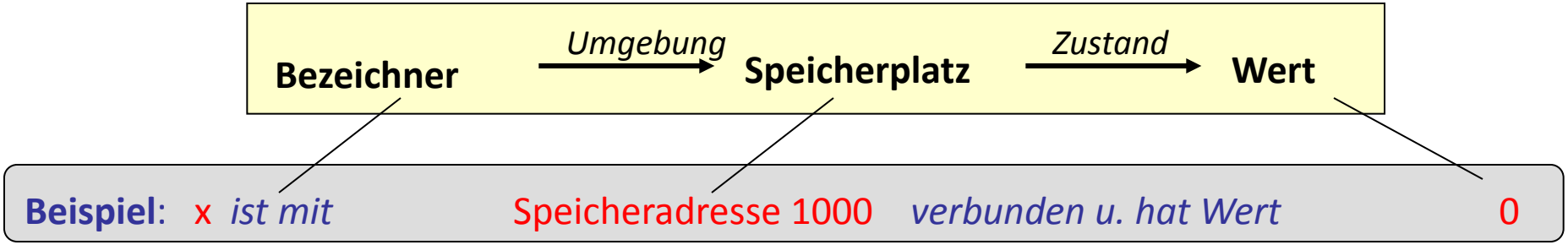
```
program sort (input, output);
    var a: array [0..10] of integer;
        x: integer;

    procedure readarray;
        var i: integer;
    begin … a … end {readarray};

    procedure exchange (i, j: integer);
    begin
        x:= a[i]; a[i]:= a[j]; a[j]:= x
    end {exchange };

    procedure qicksort (m, n: integer);
        var k,v: integer;

        function partition (y, z: integer): integer;
            var i, j: integer;
        begin
            …a…
            …v…
            …exchange(i,j) …
        end {partition };

    begin
        …
    end {quicksort };
begin
…
end {sort }.
```



Bindung von Bezeichnern

Problemstellung

- Bindung eines Bezeichners **x** an einen Speicherplatz **s**



- Bindung findet zur Laufzeit statt

statisches Konzept	dynamisches Konzept
Definition einer Prozedur	Aktivierung einer Prozedur
Deklaration eines Bezeichners	Bindung dieses Bezeichners
Gültigkeitsbereich einer Deklaration	Lebenszeit der Bindung

Konzept zur Bindung von Bezeichnern

… beeinflusst Antworten auf wichtige Fragen

- Dürfen Prozeduren rekursiv sein?
- Darf eine Prozedur auf nicht-lokale Namen zugreifen?
- Welche Art der Parameterübergabe ist zugelassen?
- Dürfen Funktionen als Parameter übergeben werden?
- Darf eine Funktion als Ergebnis zurückgegeben werden?
- Ist eine dynamische Speicherallokation erlaubt?
- Muss Speicher explizit/implizit freigegeben werden?

Position

- Teil I
  Die Programm
- Teil II
  Methodische
- Teil III
  Entwicklung

- Kapitel 1
  Compilationsprozess
- Kapitel 2
  Formalismen zur Sprac
- Kapitel 3
  Lexikalische Analyse: d
- Kapitel 4
  Syntaktische Analyse:
- Kapitel 5
  Parser-Generatoren: Ya
- Kapitel 6
  Statische Semantikana
- Kapitel 7
  Laufzeitsysteme
- Kapitel 8
  Ausblick: Codegenerier

- 7.1
  Begriffsklärung
- 7.2
  Speicherorganisation
- 7.3
  Stack-Verwaltung
- 7.4
  Zug

- Aktivierungsgraphen (Prozeduren, Koroutinen)
- Aktivierungsbäume
- Laufzeitkeller und Aktivierungssegmente
- Rahmen- und Kellerzeiger
- Statische und dynamische Links
- Organisation der Operationen über den Laufzeitkeller
- Behandlung von lokalen Prozedurdaten variabler Länge

Annahme der Quellsprache

- … sie sei hier der Einfachheit halber **prozedural** (Pascal, Fortran, Lisp, ...)

- Prozedurkonzept, eine »Einbindungs«-Konvention

  - stellt sicher, dass Prozeduren
    - **vor ihrer eigentlichen Ausführung** zur Laufzeit eine korrekte Laufzeitumgebung erhalten und
    - **nach Beendigung** eine korrekte Laufzeitumgebung für die aufrufende Prozedur wiederherstellen

  - der entsprechende Einbindungscode wird
    - zur Laufzeit ausgeführt,
    - kann aber bereits zur Übersetzungszeit generiert werden

- weitere Anforderungen haben Sprachen mit einem zusätzlichen Koroutinen/Prozesskonzept (Ada, SDL/UML, ...)

Annahmen zur Programmausführung

Grundprinzip: sequentieller Kontrollfluss (schrittweise Abarbeitung)

- gilt generell für Programme/Prozeduren der betrachteten Sprachfamilien
- Prozedurausführung wird als **Aktivierung der Prozedur** bezeichnet
  - beginnt am Anfang des Prozedurkörpers und
    führt bei Beendigung hinter den Prozeduraufruf zurück

    ➔ ist von **begrenzter Lebensdauer**
  - Schachtelung von Lebenszeiten ist beobachtbar
    (insbesondere bei rekursiven Prozeduren)
- Aktivierungsbäume/Aktivierungsgraphen zur Kontrollflussdarstellung

Aktivierungsgraphen /-diagramme

aktive Klassen

passive Klassen

Eine Member-Funktion ist als Koroutine angelegt
~ beschreibt Lebenslauf eines Objektes

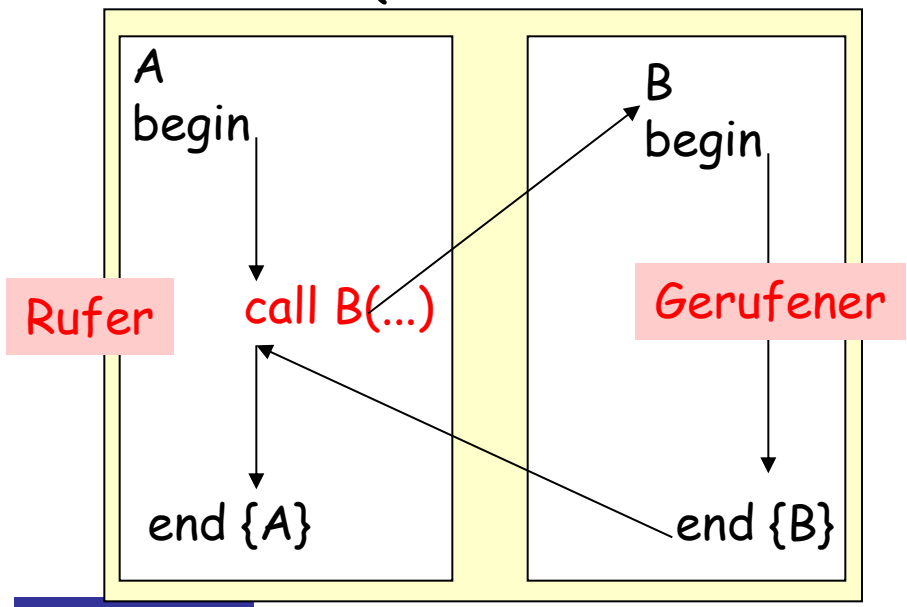
Vergleich: Prozedur - Koroutine

{ Algol, Pascal, C, ...}

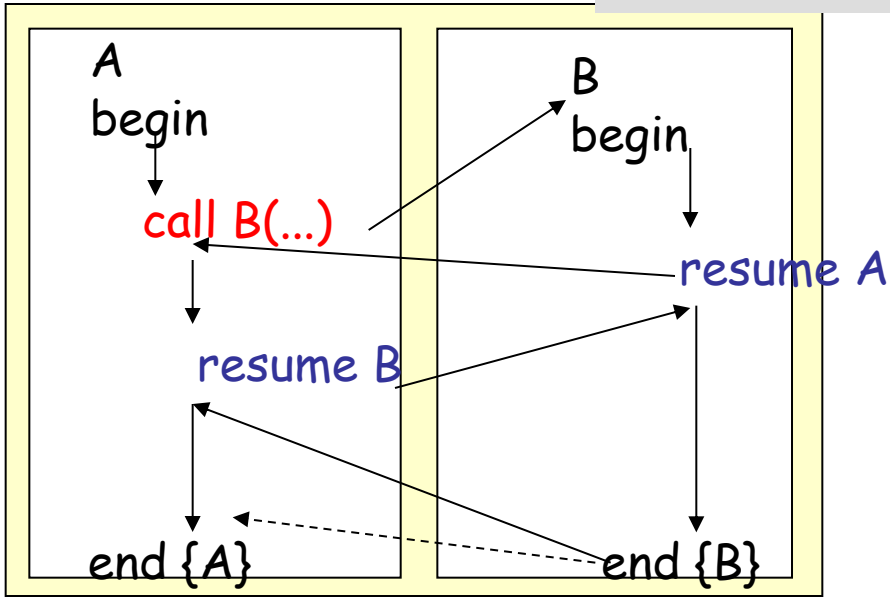
{ Simula, Modula, SLX, UML, ...}

fast alle Koroutinenkörper häufig als Endlosschleifen angelegt

Prozeduren (ohne Ausnahmefehler)



Koroutinen



bleiben bei Aktivierungselementen von Prozeduren

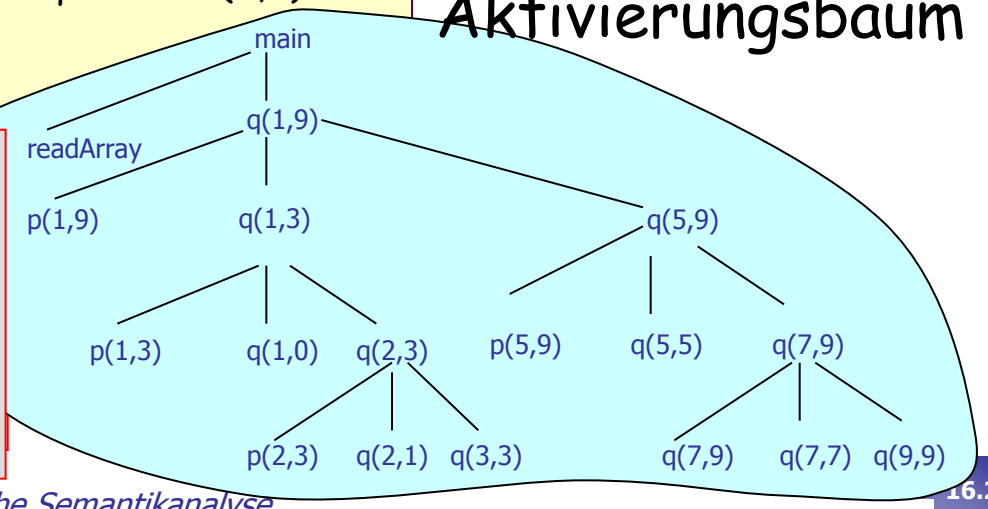
Beispiel in C: Einlesen u. Sortieren von integer-Zahlen

```
int a[11];

void readArray() { /* liest 9 Integer */
      int i : integer;
      ... a[i] ...
}

int partition (int m, n) {
/* wählt v: a[m..p-1] <v und a[p..n] >=v
   liefert p */
...
}

void quicksort (m, n : integer);
      int i;
      if (n > m) then {
            i := partition(m,n);
            quicksort(m, i-1);
            quicksort (i+1, n);
      }
}
```

möglicher Aktivierungsablauf

```
enter main()
      enter readArray()
      leave readArray()
      enter quicksort (1,9)
            enter partition (1,9)
            leave partition (1,9)
            enter quicksort (1,3)
            ...
            leave quicksort (1,3)
            enter quicksort (5,9)
            ...
            leave quicksort (5,9)
leave main()
```

Aktivierungsbaum



Merkmale

- jeder Knoten ist eine Aktivierung einer Prozedur
- Wurzel entspricht der Aktivierung des Hauptprogramms
- Knoten b ist Nachfolger von a, wenn a nach b verzweigt
- Knoten a steht links von b, wenn a in seiner Lebenszeit vor b liegt

➔ **Kontrollfluss entspricht dem Prinzip "Depth-First"**

Aktivierungsbäume

- verschachtelte Prozeduraufrufe sind typische Anwendungsfälle
- Fallunterscheidung für den Lebenslauf einer Prozedur q: (Ann.: p ruft q)

1. Die Aktivierung q endet normal (Rückkehr hinter dem Aufrufpunkt in p)
2. Die Aktivierung von q (oder einer von q gerufenen Prozedur) bricht ab
   (dann endet p mit q gleichzeitig)
3. Die Aktivierung von q liefert eine Programmausnahme,
   die nicht von q gefangen wird (q bricht ab)

   Die Ausnahme könnte von p gefangen werden.
   Wird die Ausnahme nicht von p bearbeitet, wird p mit q zusammen abgebrochen

Prozeduraufruf und Aktivierungssegment

Annahme:

- jeder Prozeduraufruf
  wird mit einem **Aktivierungssegment** zur Laufzeit assoziiert

  Aktivierungssegmente stellen die **Laufzeitumgebung** von Prozeduren dar

- Aktivierungssegmente werden **dynamisch** allokiert und
  (in einer abstrakten Betrachtung) den Knoten des Aktivierungsbaumes zugeordnet

- aktuelle **Parameterwerte** werden bei Aufruf im **Aktivierungssegment** gespeichert

- neben Statusinformationen werden lokale Variablen im **Aktivierungssegment** gespeichert

- **Aktivierungssegmente** können dynamisch erweitert werden
  (lokale Varibalen dynamischer Länge)

➔ alle **Aktivierungssegmente** werden auf einem so genannten **Laufzeitkeller** verwaltet

verbreiteter Lösungsansatz

Nutzung eines Laufzeitkellers für beliebige Aufrufsequenztiefe

Prinzipielle Verwaltung des Laufzeitkellerspeichers

Wann sind welche Aktionen von wem
bei einem Prozeduraufruf auszuführen?

Kandidaten zur Ausführung:

- Code der rufenden Prozedur
- Code der gerufenen Prozedur

- Aktivierungssegmente

  werden mit jedem Aufruf erzeugt und auf dem Keller gespeichert

  durch Realisierung eines sogenannten **Prolog-** und **Epilog-Teils** des gerufenen Prozedur-Codes

Achtung: es gibt keine allgemeine Lösung:
- Aufteilung und Realisierung der Aktionen sowie
- das Layout der Aktivierungselemente

können sogar bei Compilern für dieselbe Sprache stark variieren

Allg. Anforderungen an das Konzept Aktivierungssegment

Unterstützung bei

- der Übergabe wichtiger Daten zwischen Rufer und Gerufenen,
- flexibel und platzsparend erfolgen muss

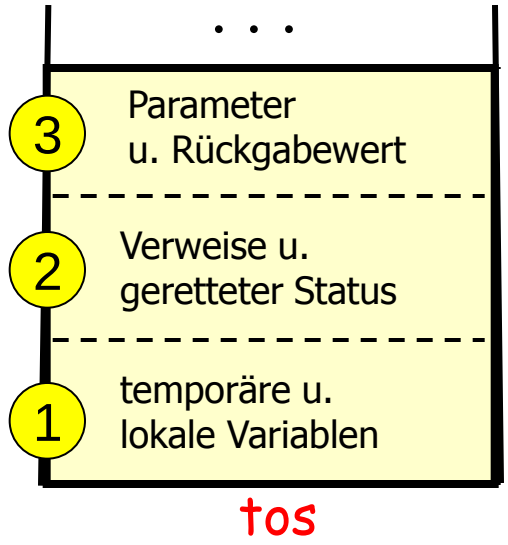
Aufbau von Aktivierungssegmenten - erste Näherung -

1. Informationsgruppe
   - **temporäre Daten**
     Werte von Ausdrücken (falls nicht schon in Registern)
   - **lokale Daten** der Prozedur (falls nicht auch in speziellen Registern)

2. Informationsgruppe
   - **Status** geretteter Abarbeitungszustand
     Informationen, die bei Rückkehr wieder benötigt werden
   - **Zugriffsverweis** (static link)
     optionaler Verweis auf (nichtlokale) Daten anderer Aktivierungssegmente)
   - **Steuerungsverweis** (dynamic link)
     optionaler Verweis zum Aktivierungssegment der gerufenen Prozedur

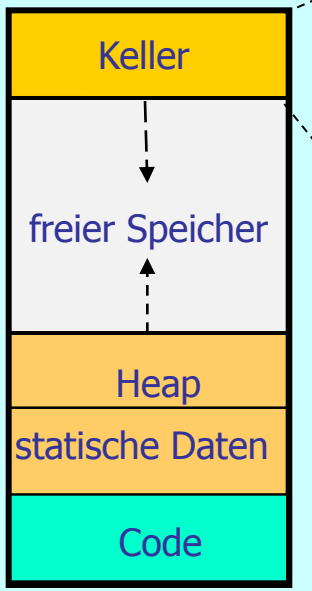
3. Informationsgruppe
   - aktuelle **Parameter**
   - **Rückgabewert**

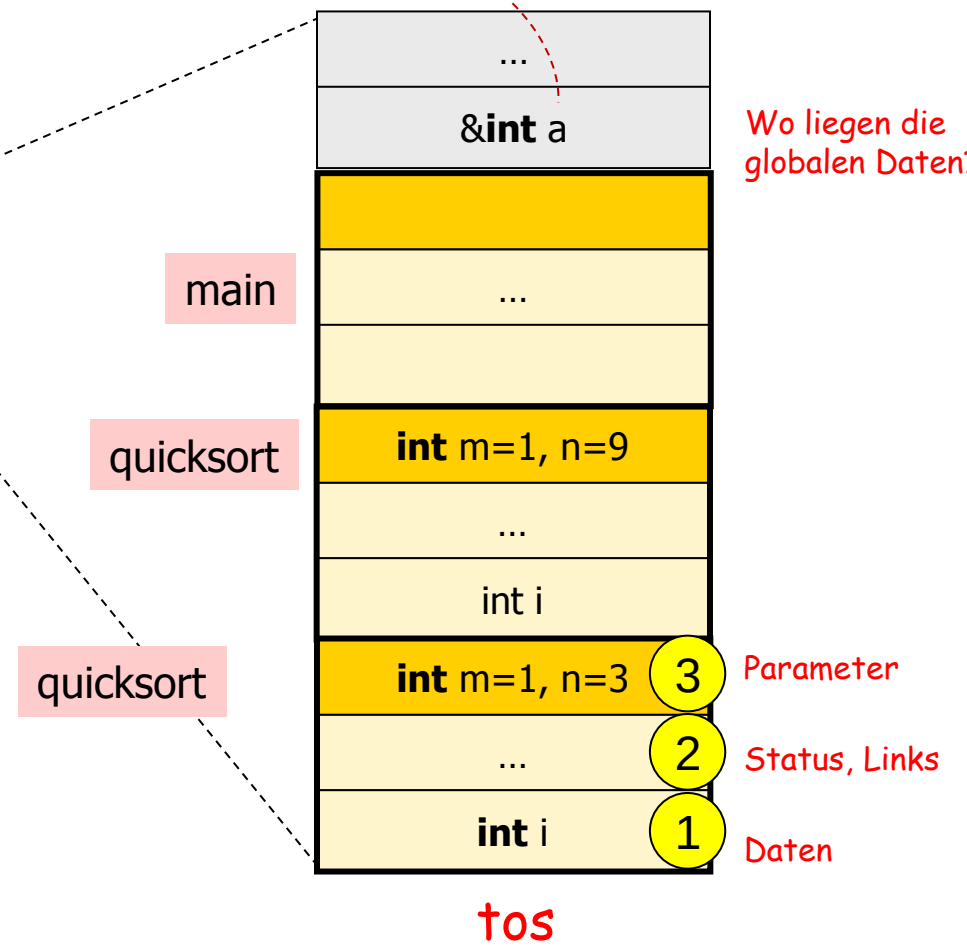
Felder werden gemeinsam von rufender und gerufener Prozedur benutzt



Beispiel Laufzeitkeller

```
int a[11];

void readArray() { /* liest 9 Integer */
        int i;
        ... a[i]= ...
}

int partition (int m, n) {
/* wählt v: a[m..p-1] <v und a[p..n] >=v
   liefert p */
...
}

void quicksort (int m, n );
        int i;
        if (n > m) then {
                i := partition(m,n);
                quicksort(m, i-1);
                quicksort (i+1, n);
    }
}

main() {
        readArray();
        a[0]= -9999;
        a[11]= 9999;
        quicksort(1,9);
}
```

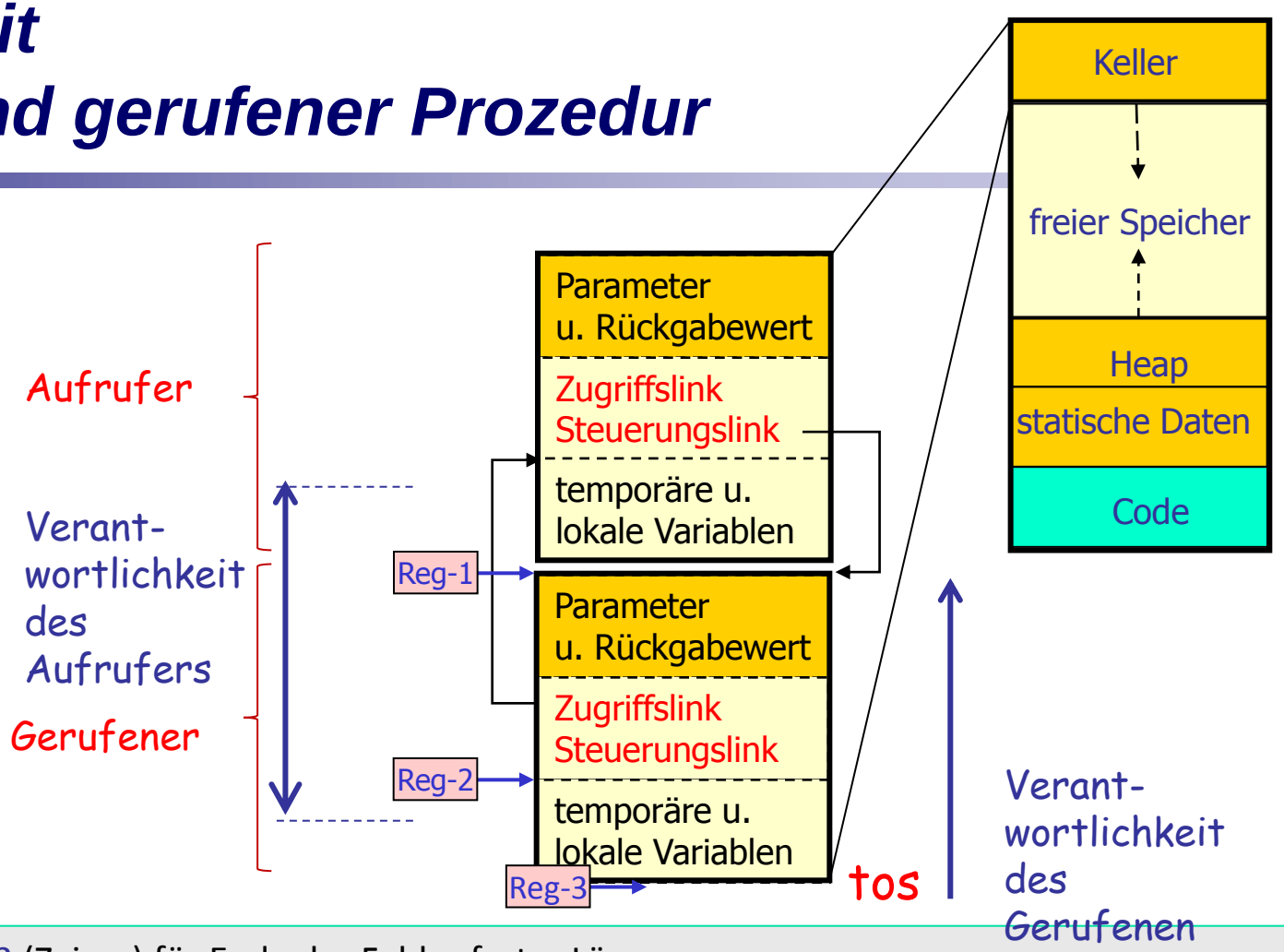




Zusammenarbeit von rufender und gerufener Prozedur

verschiedene Zeiger:

- Reg-1: Segmentzeiger zeigt den Beginn des momentan »aktiven« Segmentes an
- Reg-2: Basis-Segmentzeiger zeigt das Ende des momentan »aktiven« Basis-Segments an (Anfang des Datenbereichs)
- Reg-3: Kellerzeiger zeigt jetzt den momentanen tos an

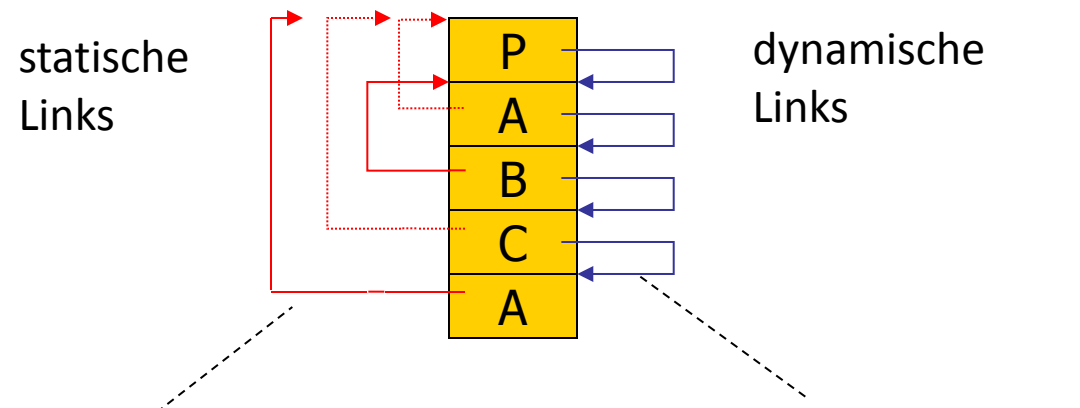


Reg-3 (Zeiger) für Ende der Felder fester Länge
- Zugriff auf Daten fester Länge: Reg-2 + offset (kennt bereits Compiler)
- Daten variabler Länge liegen tatsächlich über Reg-3-Zeigerwert,
  Zugriff auf Daten variabler Länge: Reg-3 + offset (muss zur Laufzeit berechnet werden)

Reg-2, Reg-3 –werden vom **Aufrufer** vor Aktivierung des Gerufenen (!) gesetzt
Aufrufer kann Adressen tatsächlich bestimmen (nur feste offsets!)

Verweise: statische und dynamische Links

```
program P (input, output)
  procedure A()
      procedure B()
          begin {B}
                    C(); ...
          end {B};
    begin {A}
        B(); ...
    end {A}

  procedure C()
    begin
        A(); ...
    end {C};

begin {P}
  A();
  ...
end {P}.
```



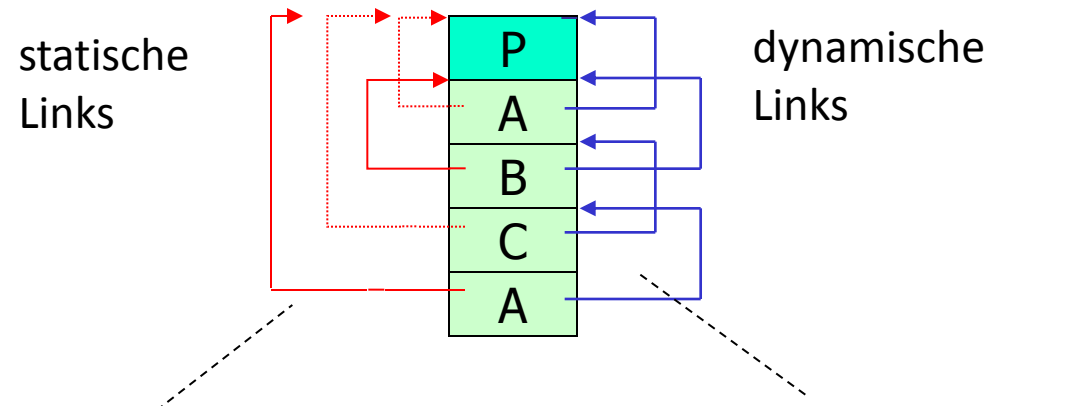
Zugriffe auf nichtlokale Variablen

Aufrufkette (Aufrufer kann Aktivierungssegment des Gerufenen bequem erzeugen und entfernen)

Pascal (als Beispiel-Sprache) erlaubt verschachtelte Prozeduren

Alternative: rückwärtsverkette dynamische Links

```
program P (input, output)
  procedure A()
      procedure B()
          begin {B}
                    C(); ...
          end {B};
    begin {A}
        B(); ...
    end {A}

  procedure C()
    begin
        A(); ...
    end {C};

begin {P}
  A();
  ...
end {P}.
```



Organisation der Stack-Operationen (Überblick)

Aspekte einer Prozeduraktivierung: **p** ruft **q**

- bei Eintritt in **p**: **aktualisiere/erweitere** Aktivierungssegment von **p**
- bei Aufruf von **q**, **sichere** Aktivierungssegment von **p**

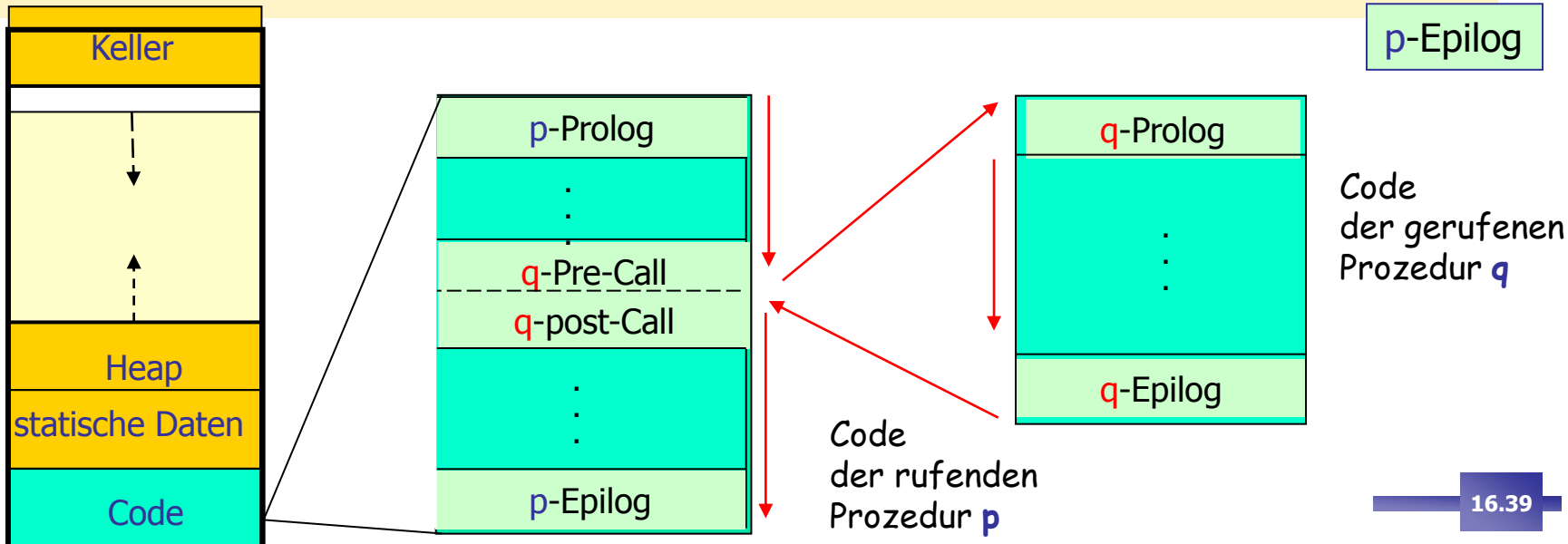
  einrichten eines Aktivierungssegments für **q** mit Parameterübergabe
- **Sprung** nach **q** **(q**-Prolog**, q, q**-Epilog**)**
- bei Rückkehr aus **q**, Return-Übernahme, **lösche** Aktivierungssegment von **q**
- vor Austritt aus **p**, Bereitstellung der **Return-Werte** im Aktivierungssegment von **p**

p-Prolog

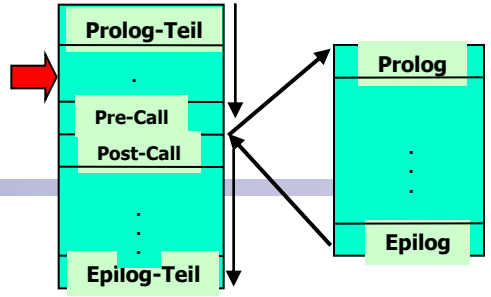
q-Pre-Call

q-Post-Call

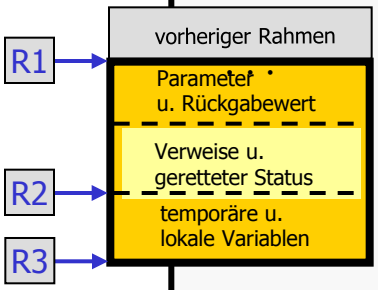
p-Epilog



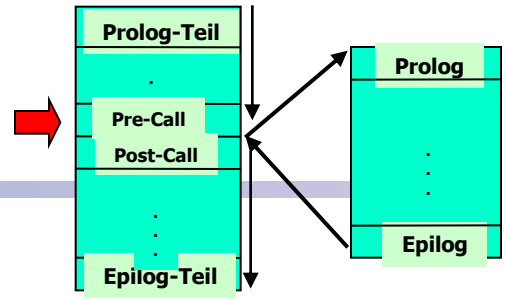
Aktivierung von Prozeduren (1)



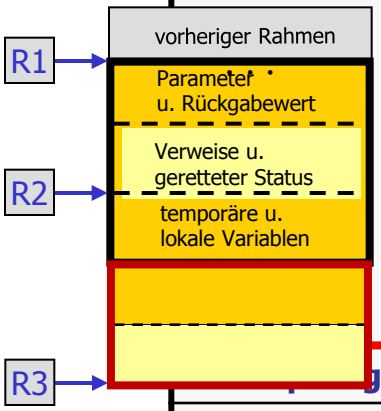
- Ausgangssituation: Steuerung im Aufrufer -



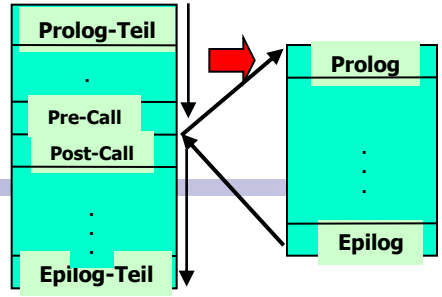
	rufende Prozedur	gerufene Prozedur
Aufruf	**Pre-Call**	**Prolog**
Rücksprung	**Post-Call**	**Epilog**



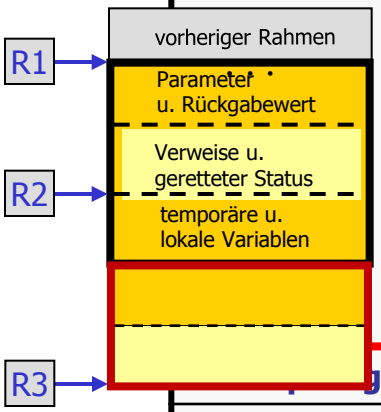
- Pre-Call-Aktionen: Steuerung im Aufrufer -



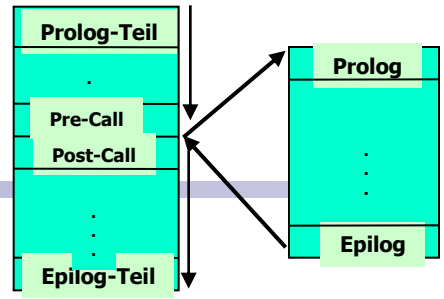
	rufende Prozedur	gerufene Prozedur
Aufruf	**Pre-Call**	**Prolog**
	1. Allokiere Basis-Segment 2. Berechne und speichere Parameter 3. Springe zur gerufenen Prozedur	
	Post-Call	**Epilog**



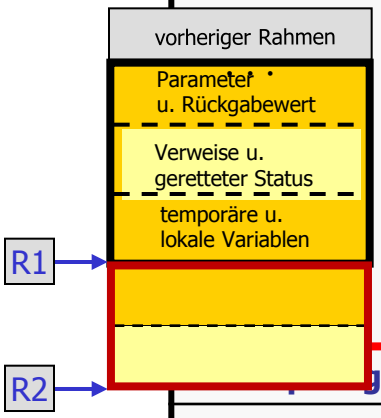
- Prolog-Aktionen: Steuerung im Aufgerufenen -



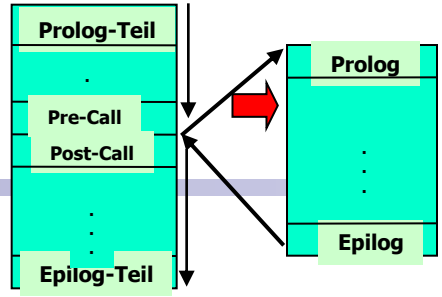
	rufende Prozedur	gerufene Prozedur
Aufruf	**Pre-Call**	**Prolog**
	1. Allokiere Basis-Segment 2. Berechne und speichere Parameter 3. Springe zur gerufenen Prozedur	1. Speichere Zustand, Rücksprungadresse 2. Speichere alte Segment u. Basis-Segmentzeiger 3. Setze neuen Segment- u. Basis-Segmentzeiger 4. Speichere statische Links /* nicht in C */
	Post-Call	**Epilog**



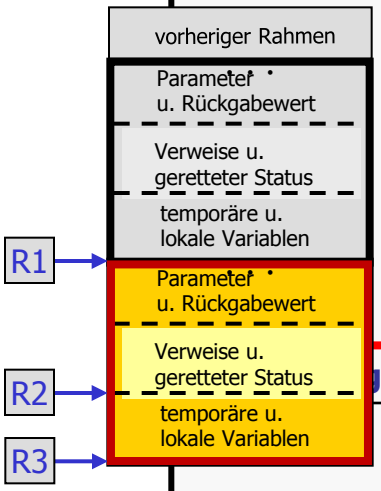
geteilte Verantwortung zwischen rufender und gerufener Prozedur



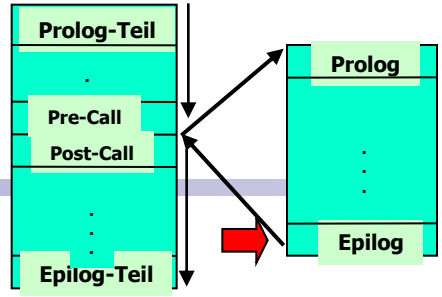
	rufende Prozedur	gerufene Prozedur
Aufruf	**Pre-Call**	**Prolog**
	1. Allokiere Basis-Segment 2. Berechne und speichere Parameter 3. Springe zur gerufenen Prozedur	1. Speichere Zustand, Rücksprungadresse 2. Speichere alte Segment u. Basis-Segmentzeiger 3. Setze neuen Segment- u. Basis-Segmentzeiger 4. Speichere statische Links /* nicht in C */
	Post-Call	**Epilog**



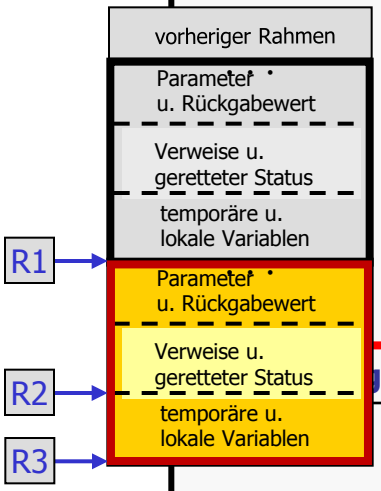
geteilte Verantwortung zwischen rufender und gerufener Prozedur



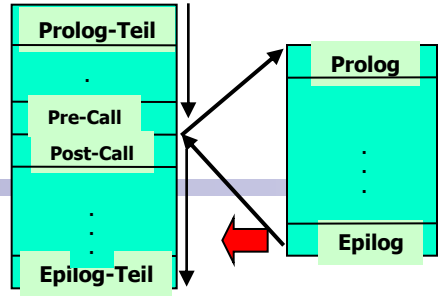
	rufende Prozedur	gerufene Prozedur
Aufruf	**Pre-Call**	**Prolog**
	1. Allokiere Basis-Segment 2. Berechne und speichere Parameter 3. Springe zur gerufenen Prozedur	1. Speichere Zustand, Rücksprungadresse 2. Speichere alte Segment u. Basis-Segmentzeiger 3. Setze neuen Segment- u. Basis-Segmentzeiger 4. Speichere statische Links /* nicht in C */ 5. Erweitere Basis-Segment um lokale Daten (Größe ~ Typ) 6. Initialisiere lokale Daten /* nicht in C */ 7. Springe zum eigentlichen Code der Prozedur
	Post-Call	**Epilog**



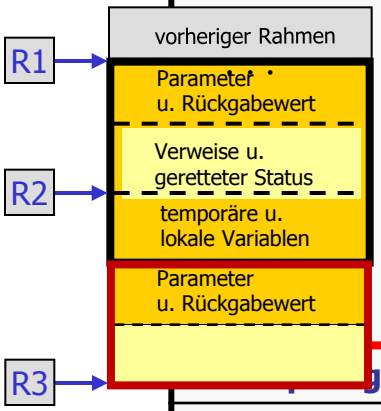
geteilte Verantwortung zwischen rufender und gerufener Prozedur



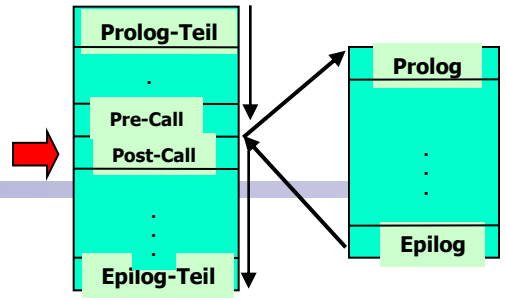
	rufende Prozedur	gerufene Prozedur
Aufruf	**Pre-Call**	**Prolog**
	1. Allokiere Basis-Segment 2. Berechne und speichere Parameter 3. Springe zur gerufenen Prozedur	1. Speichere Zustand, Rücksprungadresse 2. Speichere alte Segment u. Basis-Segmentzeiger 3. Setze neuen Segment- u. Basis-Segmentzeiger 4. Speichere statische Links /* nicht in C */ 5. Erweitere Basis-Segment um lokale Daten (Größe ~ Typ) 6. Initialisiere lokale Daten /* nicht in C */ 7. Springe zum eigentlichen Code der Prozedur
	Post-Call	**Epilog**
		1. Speichere Rückgabewerte im aktuellen Segment 2. Stelle alten Zustand wieder her 3. Reduziere Segment auf Basis-Segment 4. Stelle Segmentzeiger der aufrufenden Prozedur wieder her 5. Springe zur Rücksprungadresse



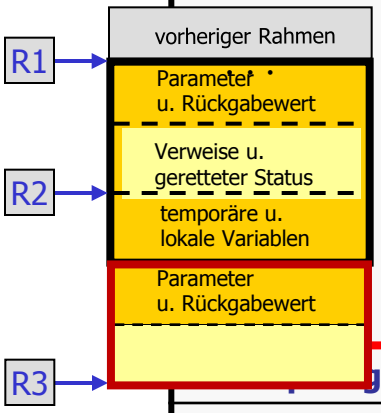
geteilte Verantwortung zwischen rufender und gerufener Prozedur



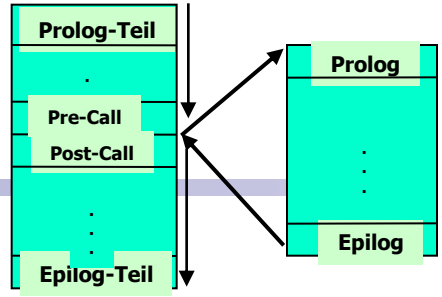
	rufende Prozedur	gerufene Prozedur
Aufruf	**Pre-Call**	**Prolog**
	1. Allokiere Basis-Segment 2. Berechne und speichere Parameter 3. Springe zur gerufenen Prozedur	1. Speichere Zustand, Rücksprungadresse 2. Speichere alte Segment u. Basis-Segmentzeiger 3. Setze neuen Segment- u. Basis-Segmentzeiger 4. Speichere statische Links /* nicht in C */ 5. Erweitere Basisrahmen um lokale Daten (Größe ~ Typ) 6. Initialisiere lokale Daten /* nicht in C */ 7. Springe zum eigentlichen Code der Prozedur
	Post-Call	**Epilog**
		1. Speichere Rückgabewerte im aktuellen Rahmen 2. Stelle alten Zustand wieder her 3. Reduziere Rahmen auf Basisrahmen 4. Stelle Rahmenzeiger der aufrufenden Prozedur wieder her 5. Springe zur Rücksprungadresse



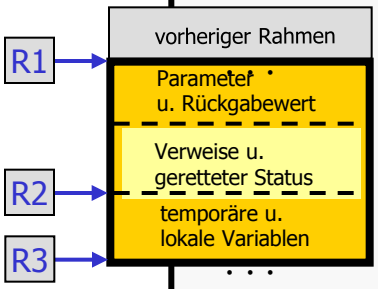
geteilte Verantwortung zwischen rufender und gerufener Prozedur



	rufende Prozedur	gerufene Prozedur
Aufruf	**Pre-Call**	**Prolog**
	1. Allokiere Basis-Segment 2. Berechne und speichere Parameter 3. Springe zur gerufenen Prozedur	1. Speichere Zustand, Rücksprungadresse 2. Speichere alte Segment u. Basis-Segmentzeiger 3. Setze neuen Segment- u. Basis-Segmentzeiger 4. Speichere statische Links /* nicht in C */ 5. Erweitere Basis-Segment um lokale Daten (Größe ~ Typ) 6. Initialisiere lokale Daten /* nicht in C */ 7. Springe zum eigentlichen Code der Prozedur
	Post-Call	**Epilog**
	1. Kopiere Ergebniswerte 2. Deallokiere Basis-Segment	1. Speichere Rückgabewerte im aktuellen Segment 2. Stelle alten Zustand wieder her 3. Reduziere Segment auf Basis-Segment 4. Stelle Segmentzeiger der aufrufenden Prozedur wieder her 5. Springe zur Rücksprungadresse



geteilte Verantwortung zwischen rufender und gerufener Prozedur



	rufende Prozedur	gerufene Prozedur
Aufruf	**Pre-Call**	**Prolog**
	1. Allokiere Basis-Segment 2. Berechne und speichere Parameter 3. Springe zur gerufenen Prozedur	1. Speichere Zustand, Rücksprungadresse 2. Speichere alte Segment u. Basis-Segmentzeiger 3. Setze neuen Segment- u. Basis-Segmentzeiger 4. Speichere statische Links /* nicht in C */ 5. Erweitere Basis-Segment um lokale Daten (Größe ~ Typ) 6. Initialisiere lokale Daten /* nicht in C */ 7. Springe zum eigentlichen Code der Prozedur
Rücksprung	**Post-Call**	**Epilog**
	1. Kopiere Ergebniswerte 2. Deallokiere Basis-Segment	1. Speichere Rückgabewerte im aktuellen Segment 2. Stelle alten Zustand wieder her 3. Reduziere Segment auf Basis-Segment 4. Stelle Segmentzeiger der aufrufenden Prozedur wieder her 5. Springe zur Rücksprungadresse

FERTIG

Daten mit variabler Länge auf dem Stack

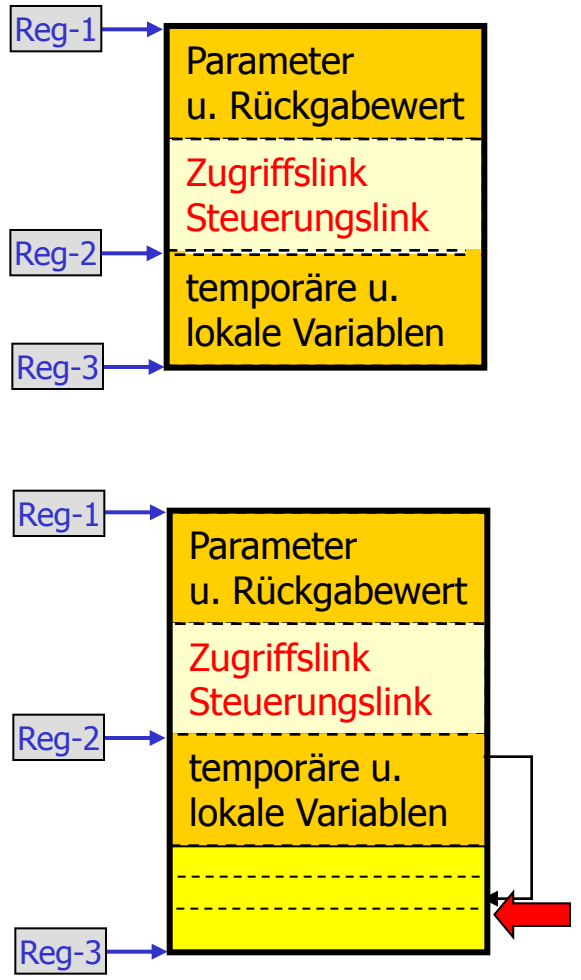
lokale Variablen einer Prozedur

werden auf dem Keller im Aktivierungssegment gespeichert, falls

- die Größe zur Compilezeit fest steht
- die Lebenszeit mit der der Prozedur übereinstimmt (der Wert also nicht darüber hinaus aufgehoben werden muss)

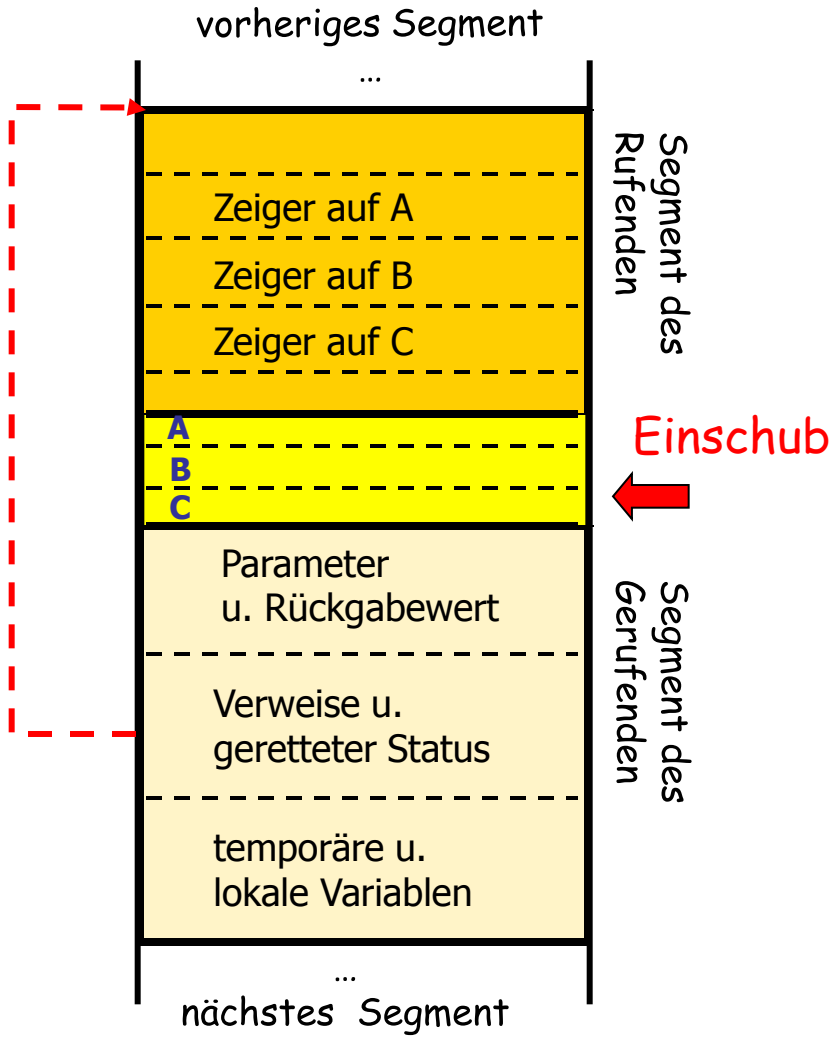
müssen auf dem Keller speziell behandelt werden, wenn sie dynamisch allokiert wurden (Einschub)

- Call-by-reference, Zeiger (Pointer) erzeugen Werte mit lokaler Lebenszeit
- (meist) explizite Allokation
- explizite oder implizite Deallokation



Daten mit variabler Länge auf dem Stack

- eine **rufende** Prozedur p habe 3 lokale Felder A, B, C
- Speicherbereich für die Felder selbst ist **nicht** Bestandteil des Aktivierungssegmentes von p,
  dafür aber Zeiger (die relativen Adressen dieser Zeiger sind bereits zur Compile-Zeit bekannt!)
  ➔Zielcode kann deshalb auf Feldelemente zugreifen
- das Aktivierungselement der **gerufenen** Prozedur liegt hinter dem Segment für die lokalen Felder (Einschub-Segment)



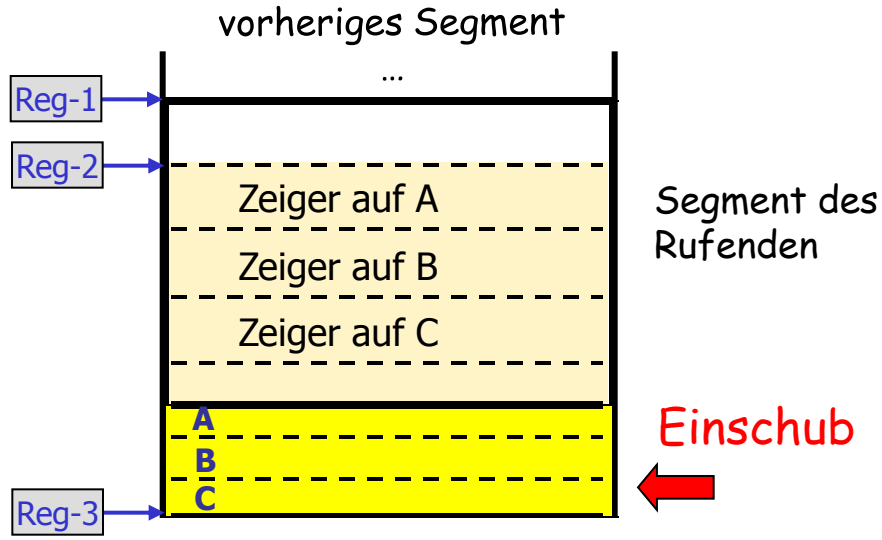
Daten mit variabler Länge auf dem Stack

Zugriff auf Daten
immer mittels zweier Zeiger
(der Gerufene sei noch nicht aktiviert)

- tos (Reg-3) Ende des Einschubs,
  potentieller Anfang eines neuen Rahmens

- Basis-Segmentzeiger (Reg-2)
  Anfang der temp. Variablen
  (mit fester Länge)

  Positionen für die Adressen von A, B, C werden aus Reg-2 (Basisadresse) und dem Compiler bekannten Offset (Symboltabelle) gebildet



Vorteil
- A, B, C werden mit Beendigung der Prozedur speichermäßig entsorgt (kein gc notwendig)

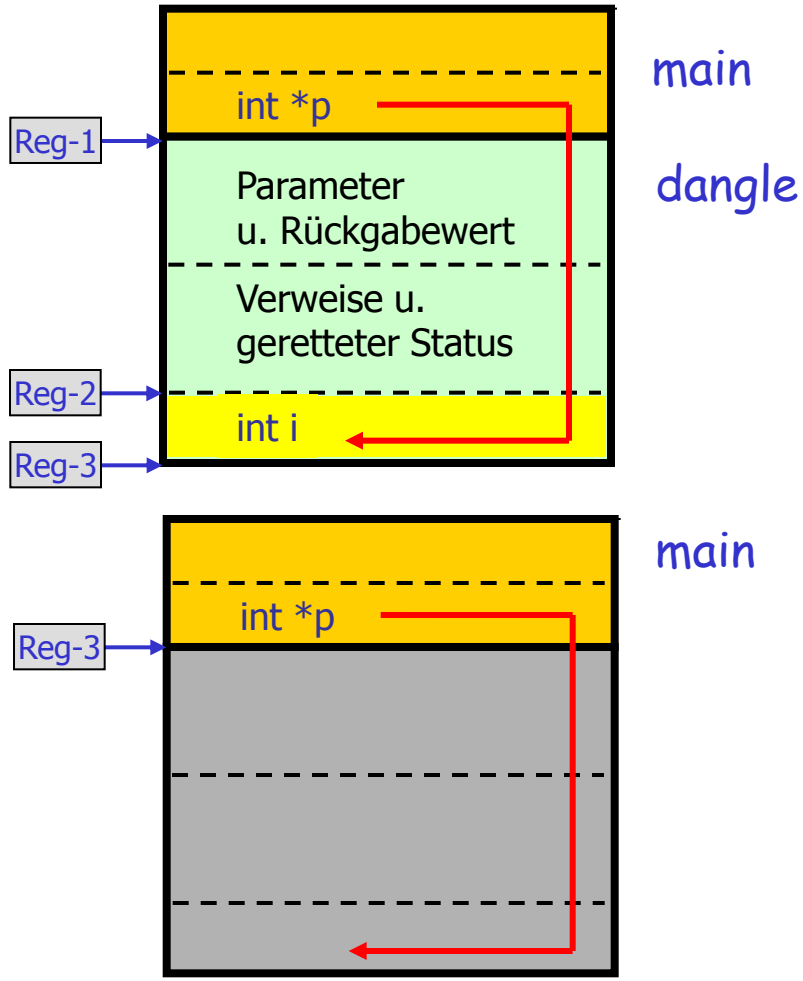
Nachteil
- hängende Referenzen müssen verhindert werden

Beispiel: Entstehung hängender Referenzen

- ...sind Referenzen auf einen Speicherplatz, der bereits freigegeben ist
- Verwendung solcher Referenzen ist Ursache mysteriöser Fehler!!!

```
int main () {
    int *p;
    p= dangle();
}
int *dangle() {
    int i = 23;
    return &i;
}
```

- p zeigt nach Verlassen von dangle auf einen solchen Speicherplatz



Grenzen der Kellerzuweisungsstrategie

Kellerzuweisungsstrategie

für Aktivierungssegmente kann **nicht** benutzt werden, wenn

1. Werte lokaler Namen erhalten bleiben müssen
2. eine gerufene Prozedur die aufrufende Prozedur überlebt (z.B. bei Koroutinen)

dann Benutzung des Heap-Speicher

Allgemeines Problem der Heap-Zuweisungsstrategie

Heap besteht (nach einiger Zeit) aus Bereichen, die frei oder in Benutzung sind.

Position

- Teil I
  Die Programm
- Teil II
  Methodische
- Teil III
  Entwicklung

- Kapitel 1
  Compilationsprozess
- Kapitel 2
  Formalismen zur Sprac
- Kapitel 3
  Lexikalische Analyse: d
- Kapitel 4
  Syntaktische Analyse:
- Kapitel 5
  Parser-Generatoren: Ya
- Kapitel 6
  Statische Semantikana
- Kapitel 7
  Laufzeitsysteme
- Kapitel 8
  Ausblick: Codegenerier

- 7.1
  Begriffsklärung
- 7.2
  Speicherorganisation
- 7.3
  Stack-Verwaltung
- 7.4
  Zugriff auf nichtlokale Daten
- 7.5
  Heap-Verwaltung
- 7.6
  Garbage Collection

Zugriff auf nichtlokale Daten

besonderes Problem, falls Sprache eine Prozedurverschachtelung erlaubt

Wie werden nichtlokale Daten zur Laufzeit gefunden?

- Code muss diesen (evtl. kaskadierten) Zugriff beinhalten/realisieren

echt globale Variablen

- Namenskonvention gibt die Adresse des Speicherplatzes an
- Initialisierung (falls gefordert) muss gewährleistet sein

a) lexikalische (oder statische) Bindung:
Zugriff auf Variablen der statischen Umgebung

- lexikalisch: Gültigkeitsbereiche werden anhand der Definition im Programm (zur Übersetzungszeit) bestimmt
- realisiert durch (level, offset)-Paare

Zugriff wertet level-Info aus um beim Kellerabstieg das richtige Aktivierungselement zu treffen

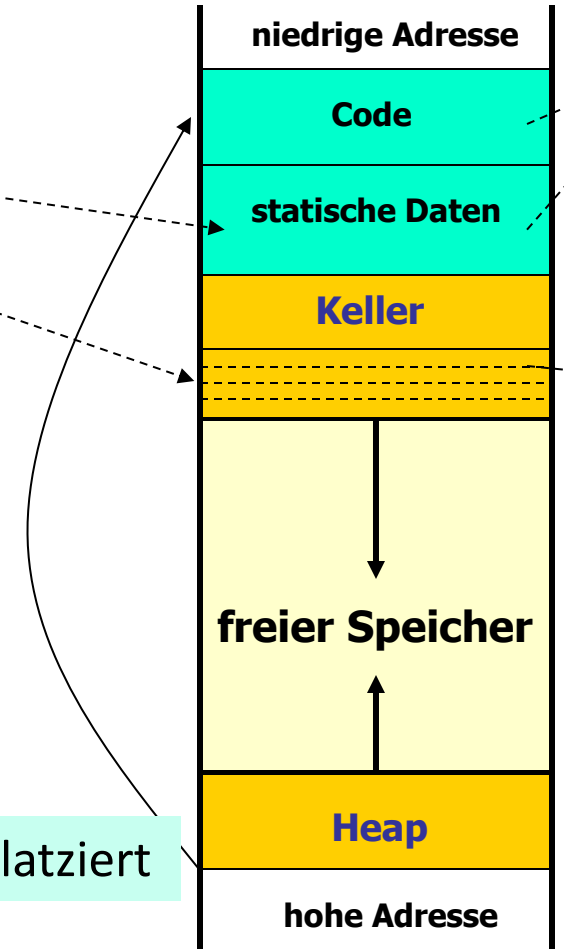
b) dynamische Bindung

- z.B. Lisp: anwendbare Deklaration wird zur Laufzeit ermittelt:
Zugriff auf Variablen des Rufers

Datenzugriff ohne Prozedurschachtelung

C-Sprachfamilie

- globale Größe
- Prozedur-lokale Größen
- verschachtelte Böcke
  (Realisierung mit ähnlichen Aktivierungssegmenten)



Adressen zur Compile-Zeit bekannt

nur Offsets, die zur Compile-Zeit bekannt

Basisadressen ändern sich durch Generierung und Entfernung von Aktivierungs-Elementen auf dem Laufzeitkeller

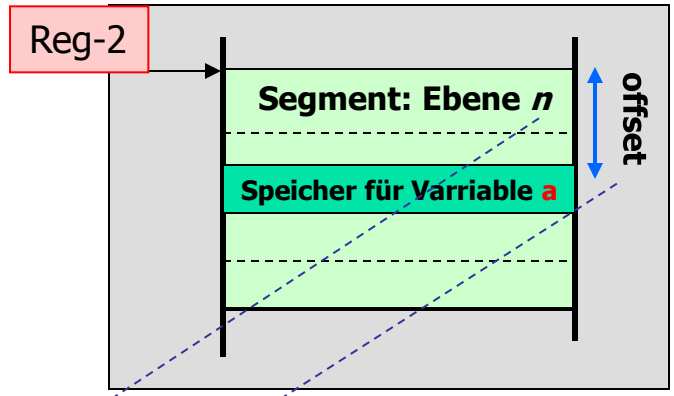
Basisadresse ist der jeweilige Inhalt vom Basis-Segmentzeiger (Reg-2) bzw. im Segment hinterlegt

Code wird durchaus unterschiedlich platziert

Datenzugriff in verschachtelten Prozeduren/Blöcken

zur Übersetzungszeit können bestimmt werden

- Deklarationsniveau eines Bezeichners
  und
- das Offset für den Speicherplatz eines beliebig erzeugtes Aktivierungssegmentes der entsprechenden Prozedur



➔ jeder Bezeichner (z.B. a) ist assoziiert mit einem (level, offset)-Paar

Behandlung von Blockstrukturen (z.B. in C)

- nach gleichem Prinzip: Aktivierungssegmente je aufgerufene Block-Instanz

Beispiel: Level-Angaben quicksort

```
program sort (input, output) ; { level 1 }
    var a  : array [0..10] of integer;
        x : integer;

    procedure readarray; { level 2 }
        var i : integer;
        begin ..... a[i] .... end {readarray} ;

    function partition (y, z : integer): integer;
                                              { level 2 }
        var i, j, x, v : integer;
        begin .... end {partition} ;
```

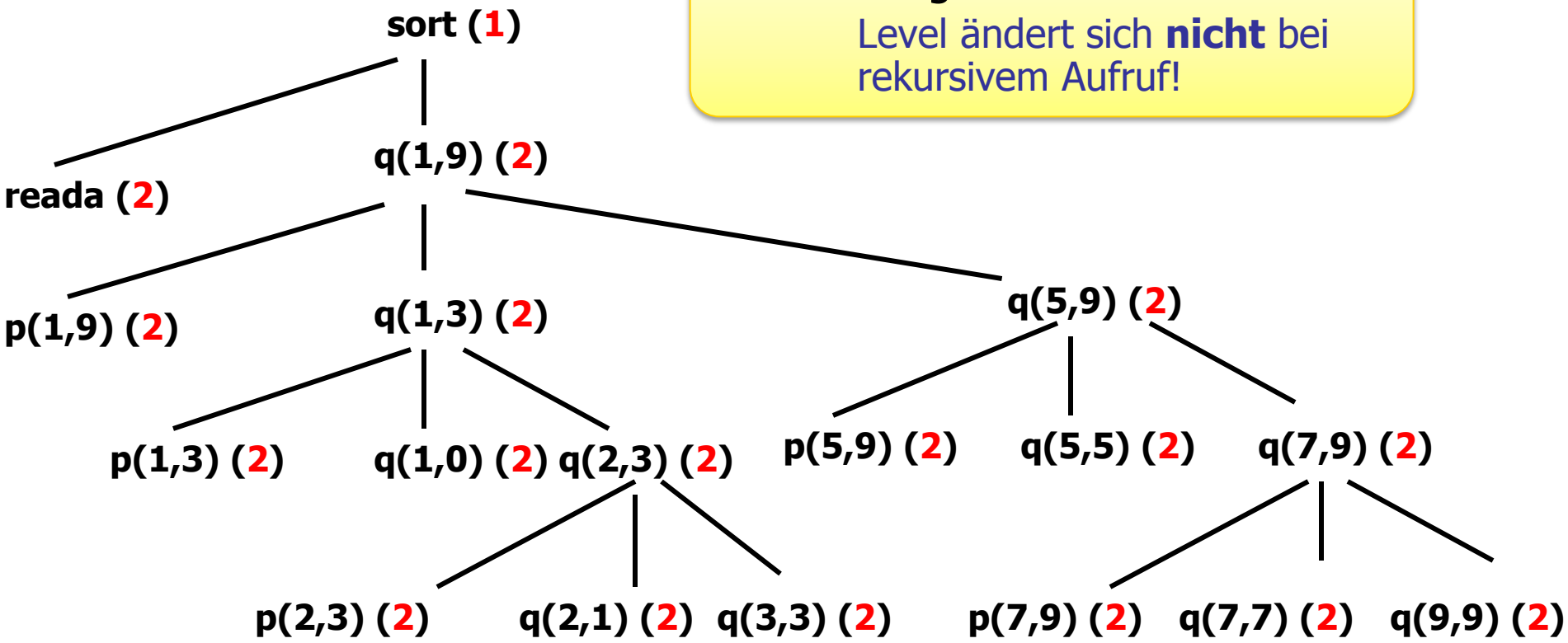
```
procedure quicksort (m, n : integer ); { level 2 }
        var i : integer;
        begin {quicksort}
            if (n > m) then begin
                i := partition(m,n);
                quicksort(m, i-1);
                quicksort (i+1;n);
            end
        end {quicksort };

begin {sort }
    readarray;
    quicksort(1,9)
end {sort } .
```

Statische Level-Zuordnung durch Compiler in der Symboltabelle

Beispiel: Level-Angaben quicksort (Forts.)

Ausführungsbaum

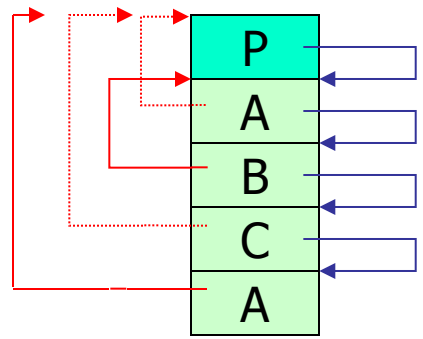
Level ändert sich **nicht** bei rekursivem Aufruf!



Dynamische Level-Zuordnung
in den Aktivierungssegmenten während der Ausführung

Offenes Problem beim Datenzugriff

```
program P (input, output)
  procedure A()
      procedure B()
        begin {B}
                C(); ...
        end {B};
    begin {A}
        B(); ...
    end {A}

  procedure C()
    begin
        A(); ...
    end {C};

begin {P}
  A();
  ...
end {P}.
```



Unterscheidung von Zugriffen auf

- globale Variablen (im oberstes Segment)
- lokale Variablen (im aktuellen Segment)
- nichtlokale Variablen (in einem Zwischensegment)

Methode:
Steigen zur Adressermittlung entlang der statischen Links ab,
Frage: aber wie weit? **Gesucht** ist richtiges Zwischensegment!

Access-Link-Methode *(Zugriff auf nichtlokale Daten)*

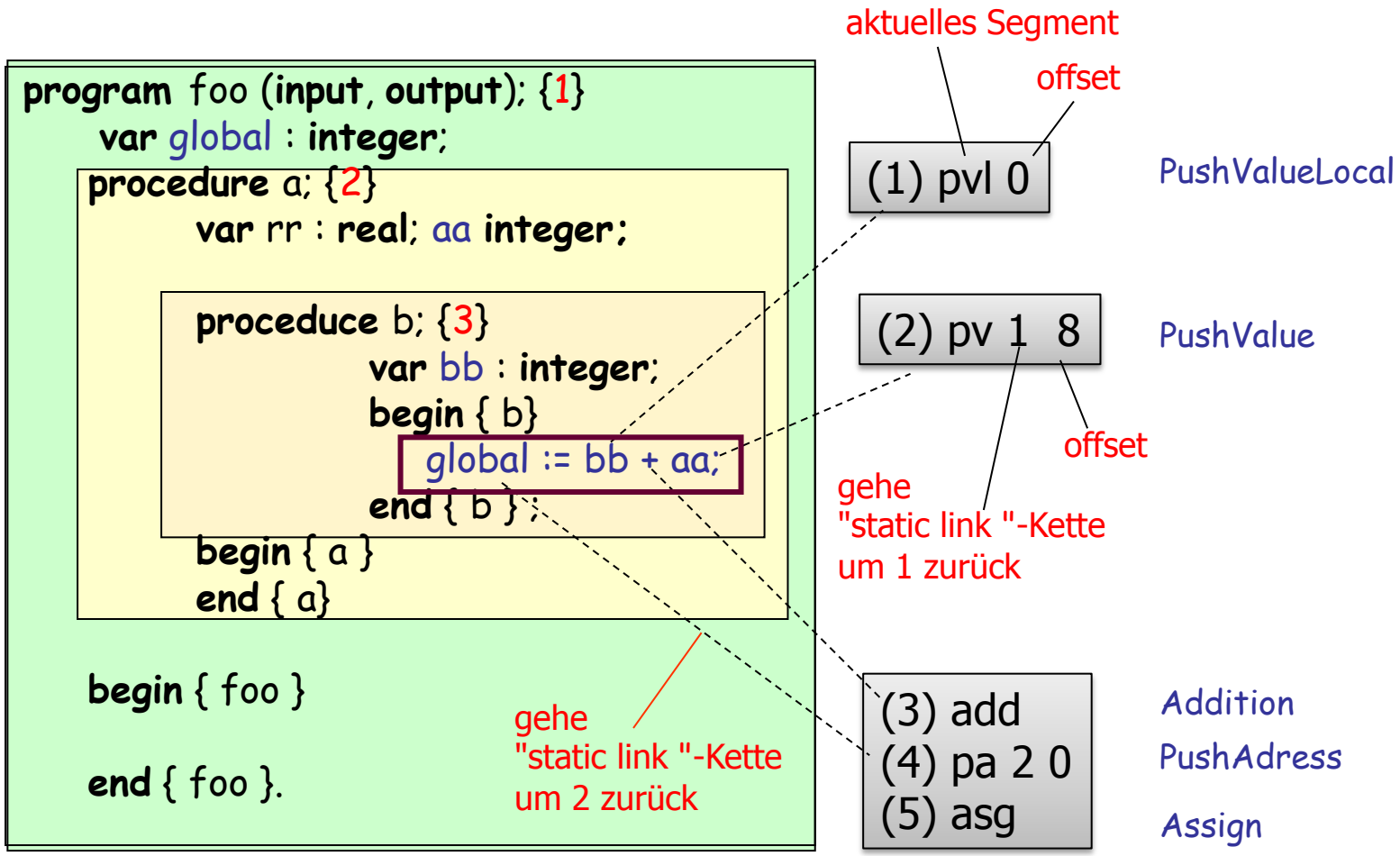
Adressbestimmung einer Variable a

muß im generierten (Zwischen-)Code bei jedem Variablenzugriff als Befehlsfolge generiert werden

- bestimme das Level der aktiven Prozedur (das sei k)
  das ist das Anwendungslevel V_{apply} des Bezeichners a, also
  $V_{apply}(a) := k$
- bestimme das Deklarationslevel V_{decl} und das offset des Bezeichners a aus der Symboltabelle
- bestimme diff
  $diff := V_{apply}(a) - V_{decl}(a) >= 0$
- gehe diff Schritte die Kette der statischen Links zurück,
  man erhält bfp des Aktivierungselements (Basisadresse) für Variable a

Vor.: die statischen Links sind durch das jeweilige Pre-Call-Programm der Rufer gesetzt

Zugriff auf nichtlokale Daten: *Access-Link-Methode (P-Code-Generierung)*



P-Code *(Code einer virtuellen Maschinensprache für Pascal)*

Crashkurs P-Code – Ladebefehle

`PC c`	*push constant*	Legt `c` (Integer) auf den Stack
`PCF c`	*push constant float*	Legt `c` (Float) auf den Stack
`PA d r`	*push address*	Lädt die Adresse auf den Stack
`PAL r`	*push address local*	Lädt die lokale Adresse auf den Stack
`PAG r`	*push address global*	Lädt die glob. Adresse auf den Stack
`PV d r`	*push value*	Lädt den Inhalt der Speicherstelle
`PVL r`	*push value local*	Lädt den Inhalt aus dem lokalen DS
`PVG r`	*push value global*	Lädt den Inhalt aus dem globalen DS
`PVI d r`	*push value indirect*	Lädt indirekt über Adresse (`d`, `r`)
`PVLI r`	*push value local ind.*	Lädt indirekt über Adresse `r`

Crashkurs P-Code – Arithmetik/Logik

`ADD`	`ADDF`	Addition
`MUL`	`MULF`	Multiplikation
`SUB`	`SUBF`	Subtraktion
`DIV`	`DIVF`	Division
`MIN`	`MINF`	Negation („unäres Minus“)
`EQU`	`EQUF`	Gleichheit
`NEQ`	`NEQF`	Ungleichheit
`LEQ`	`LEQF`	Kleiner-oder-gleich
`GEQ`	`GEQF`	Größer-oder-gleich
`LSS`	`LSSF`	Kleiner
`GRT`	`GRTF`	Größer
`AND`		Logische Konjunktion
`OR`		Logische Disjunktion
`NOT`		Logische Negation

Crashkurs P-Code – Prozeduraufruf

`MST d`	*mark stack*	Vorbereitung eines Prozeduraufrufs mit Niveaudifferenz `d`
`CAL a`	*call*	Aufrufen der Prozedur
`ENT c`	*enter*	Reservierung von Speicher für lokale Variablen
`RET`	*return*	Beendigung der Prozedur

Crashkurs P-Code – weitere Befehle

`ASG`	*assign*	Speichern des obersten Elements
`JMP a`	*jump*	Unbedingter Sprung nach `a`
`FJP a`	*false jump*	Bedingter Sprung nach `a`
`ITR`	*integer to real*	konvertiert das oberste Stackelement
`RTI`	*real to integer*	konvertiert das oberste Stackelement
`READ`	*read*	liest Gleitkommazahl in Adresse vom Stack
`WRI`	*write*	gibt oberstes Stackelement aus

Zugriff auf nichtlokale Daten *bei dynamischer Bindung*

besondere Herausforderung:

- Überführung eines Namens der Symboltabelle auf eine Speicheradresse erfolgt erst zur Laufzeit
  - Speicheradresse ergibt sich aus aktueller Aufrufhierarchie

    (Referenz kann sich bei unterschiedlichen Aufrufen ändern)

- Typüberprüfung zur Laufzeit
  - Typverträglichkeit muss bei jeder Referenzanwendung sichergestellt werden

```
program dynamic (input, output);
        var r, z : integer;

    procedure show;
        begin
                write (r); write (z); writeln
        end { show };

    procedure small;
            var r: integer;
        begin
                r := 2; show
        end { small };

    begin { dynamic}
            r := 20; z := 50;
            show; small;

    end { dynamic} .
```

Pascal mit lexikalischer/statischer Bindung

20 50

20 50

Simula, Ada mit dynamischer Bindung

20 50

2 50

Dynamische Bindung (Forts.)

Realisierung

- Aktivierungssegment muss für lokale Variablen auch Namen mitführen
- zur Laufzeit: dynamische Überprüfung für Bindung
- Anmerkung (Preis für höhere Flexibilität)
  - Semantik des Programms nur noch schwer zu verstehen
  - große Belastung zur Laufzeit

Fazit: Speicherorganisation

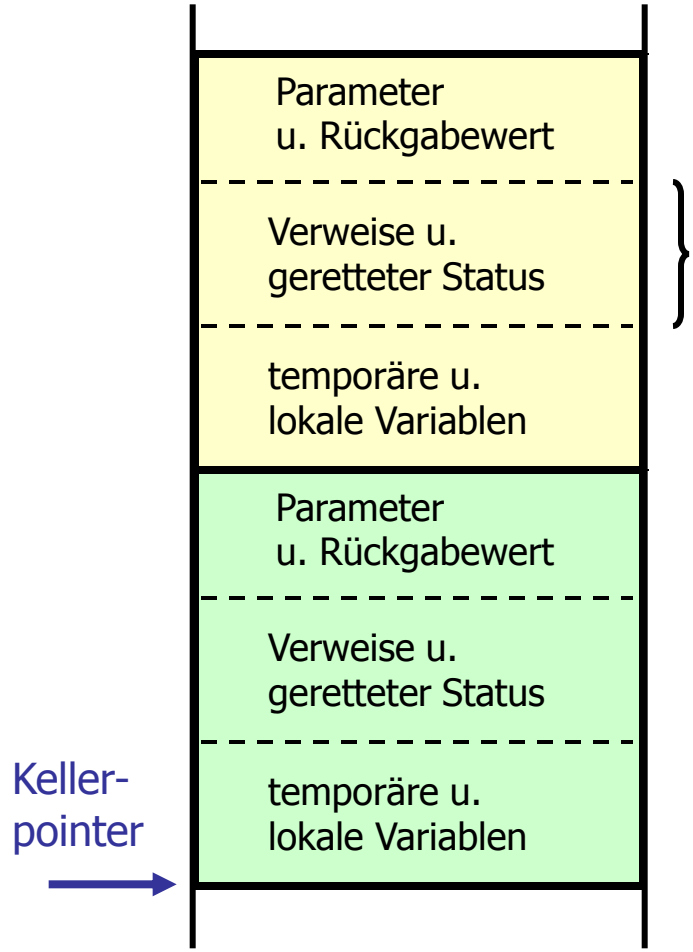
... eines Maschinenprogramms zur Laufzeit
(organisiert durch das Laufzeitsystem)

- Unterteilung des Speichers erfolgt zur besseren Verwaltung in Segmentform
- Segmente werden dynamisch (zur Laufzeit) angelegt und gelöscht
- die Codegröße eines Programms (Assembler oder Maschinencode) ist konstant
- dabei werden Prozeduraufrufe als Calls mit konstantem Argument bei der Compilation erzeugt (Sprungadresse ist fest)
- Vereinheitlichung (Hauptprogramm, Prozedur, Funktion)
- Größe und Anzahl benötigten Datenspeichers für die Segmente ist veränderlich, sie lässt sich nicht zur Compilezeit bestimmen (Rekursivität)

➔ es ist ein dynamisches Speicherverwaltungsmodell erforderlich

- beim Aufruf einer Prozedur wird ein eigenes Datensegment (Aktivierungssegment) erzeugt,
- beim Verlassen wird es wieder zerstört

Fazit: Adressierungsunterstützung



Segment-Bereich enthält u.a.:

- Registerwerte (viele Maschinen: 32 Register)
  Befehlszähler
  Basisadressregister, ...
- Rücksprungadresse (zum Rufer)
- dynamischer Link (für Aufrufverkettung)
  Verweis auf Segment der gerufenen Prozedur
  (einfaches Löschen des Datensegments nach Rückkehr der gerufenen Prozedur möglich)
- statischer Link (für textuelle Verschachtelung)
  Verweis auf Segment der umgebenden Prozedur im Quelltext
  (einfacher Zugriff auf nichtlokale Variablen, möglicherweise über mehrere Stufen)

Fazit: Adressierung

... von Variablen/Parametern innerhalb eines Datensegments

- **Prinzip**: Adressangabe immer relativ zum Segmentanfang (BasisFramePointer)

absolute Adresse:= BasisFramePointer

als Basisregister zur Laufzeit

+

Verschiebung (Offset) im Datensegment

zur Übersetzungszeit

... von Variablen anderer Datensegmente

- **Prinzip**: ebenfalls immer relativ zu deren Segmentanfängen (FP)

über statische Links

Position

- Teil I
  Die Programmiersprache C
- Teil II
  Methodische Grundlagen des Compilerbaus
- Teil III
  Entwicklung eines einfachen Transcompilers
- Teil IV
  Klausur

- Kapitel 1
  Compilationsprozess
- Kapitel 2
  Formalismen zur Sprachbeschreibung
- Kapitel 3
  Lexikalische Analyse: der Scanner
- Kapitel 4
  Syntaktische Analyse: der Parser
- Kapitel 5
  Parser-Generatoren: Yacc, Bison
- Kapitel 6
  Statische Semantikanalyse
- Kapitel 7
  Laufzeitsysteme