Einführung in die Informatik II

Maschinennahe Programmierung 2: Übersetzung von Konstrukten höherer Programmiersprachen

Prof. Bernd Brügge, Ph.D
Institut für Informatik
Technische Universität München

Sommersemester 2004

15-20. Juli 2004

Überblick über die Vorlesung

- Übersetzung von Konstrukten höherer Programmiersprachen (z.B. Java) in maschinennahe Sprachen (z.B. PMI-Assembler)
  - Ausdrücke
  - Zuweisung
  - While-Schleife
  - Methodenaufruf ("Unterprogrammsprung")
    - insbesondere Rekursion
  - Rekursive Datenstrukturen

Ziele dieser Vorlesung

- Sie verstehen die Problemstellung bei der Übersetzung von höheren Sprachen in maschinennahe Sprachen.
  - Sie können ***manuell*** für einige Quellsprachkonstrukte PMI-Assemblercode erzeugen
    - insbesondere für Zuweisungen und Unterprogrammaufrufe (Methodenaufrufe).
- Sie verstehen, wo die folgenden Konzepte eingesetzt werden:
  - Aufrufbaum
  - Kontrollkeller
  - Aktivierungssegment
- Sie können erklären, wie man rekursive Datenstrukturen (z.B. eine verkettete Liste) auf der Halde anlegt.
- Sie verstehen die Arbeitsweise eines Übersetzers

Übersetzer

- **Definition:** Ein **Übersetzer** ist ein Programm, das ein in einer **Quellsprache** geschriebenes Programm in ein äquivalentes Programm in einer **Zielsprache** übersetzt.
- Quellsprachen:
  - Modellierungssprachen
  - höhere Programmiersprachen, maschinennahe Sprachen
  - Spezialsprachen
- Wir unterscheiden drei Klassen von Übersetzern:
  - **CASE-Werkzeug:** Übersetzt eine Modellierungsprache in eine höhere Programmiersprache.
  - **Compiler:** Übersetzt eine höhere Programmiersprache in eine maschinennahe Sprache.
  - **Assembler:** Übersetzt eine maschinennahe Sprache in Maschinencode, der von einer Rechenanlage ausgeführt werden kann.

Fortschritte in der Übersetzertechnik

- Die ersten Compiler kamen mit dem Entstehen höherer Programmiersprachen (Fortran, Cobol, Algol) in den Fünfziger Jahren. Compiler galten als extrem komplexe und schwer zu schreibende Programme.
  - Der Aufwand für die Implementierung des ersten Fortran-Compilers im Jahr 1957 betrug 18 Personenjahre.
- Seitdem hat man viele Übersetzungs-Probleme gelöst. Beispiel:
  - Entdeckung des Kellers zur Verwaltung von Prozeduraufrufen (F. L. Bauer & K. Samelson, 1958)
- Außerdem wurden Programmiersprachen und Software-Werkzeuge entwickelt, um den Übersetzungsprozess zu vereinfachen. Beispiele:
  - Der erste Pascal-Compiler wurde selbst in Pascal geschrieben (N. Wirth & U. Amman, 1970)
  - Das Unix-Programm **`yacc`** erzeugt aus einer Chomsky-2-Grammatik den Parser eines Compilers für die Syntax-Analyse.
- Heute können Sie einen guten Compiler in einem Semester entwickeln
  => ***Hauptstudium: Übersetzerbau***

Einsatz von Übersetzern

- Es gibt viele Bereiche, in denen wir Übersetzer verwenden.
- **Textverarbeitungssystem:** Erhält als Eingabe eine Zeichenkette mit Kommandos (RTF, MIF, LaTeX usw.) und formatiert sie als Dokument (Word, Framemaker, Adobe Acrobat, Powerpoint, usw).
- **Silicon-Compiler:** Die Variablen der Quellsprache (z.B. VHDL, Verilog) repräsentieren logische Signale in einem Schaltwerk. Daraus wird eine Beschreibung für die Herstellung von Microchips erzeugt.
  - ***Hauptstudium: Vorlesung und Praktikum Rechnerarchitekturen***
- **Anfrage-Interpretierer:** Übersetzt ein Prädikat einer Quellsprache mit relationalen und booleschen Operatoren in ein Kommando einer Ziel-Sprache (z.B. SQL), das dann in einer Datenbank nach Einträgen sucht, die dieses Prädikat erfüllen.
  - ***Hauptstudium: Vorlesung und Praktikum Datenbanksysteme***

Der Übersetzungsprozess

- Der allgemeine Übersetzungsprozess besteht aus 2 Teilen: Analyse und Synthese.
- In der **Analyse** wird das Quellprogramm in seine Bestandteile zerlegt und eine Zwischendarstellung erstellt.
- In der **Synthese** wird aus der Zwischendarstellung das gewünschte Zielprogramm konstruiert.

Der Analyse-Teil eines Compilers

- Der Analyse-Teil eines Übersetzers besteht aus **lexikalischer, syntaktischer und semantischer Analyse**.
- Während der **lexikalischen Analyse** werden die Bestandteile (reservierte Worte, Bezeichner, Operatoren) ermittelt.
- In der s**ynxtaktischen Analyse** werden die im Quellprogramm enthaltenen Operationen bestimmt und in einem Syntax-Baum angeordnet:
  - Im Syntax-Baum stellt jeder Knoten eine Operation dar, die Kinderknoten repräsentieren die Operanden der Operation.
- In der **semantischen Analyse** werden die Typ-Informationen für jeden Operanden ermittelt, Typ-Überprüfungen gemacht, und Operandenanpassungen (type-casting) durchgeführt.

Der Synthese-Teil eines Compilers

- Der Synthese-Teil eines Übersetzers besteht aus Zwischencode-Erzeugung, Code-Optimierung und Code-Erzeugung.
- **Zwischencode-Erzeugung:** Manche Compiler erzeugen nach der Analyse eine Zwischendarstellung des Quellprogramms.
  - Die Zwischendarstellung ist gewissermassen ein Programm für eine gedachte Maschine.
- **Code-Optimierung:** In dieser Phase wird der Zwischencode verbessert, um möglichst effizienten Maschinencode zu erzeugen.
- **Code-Erzeugung:** Ziel ist die Erzeugung von Maschinencode für die Maschine, auf der das Programm ausgeführt werden soll.
  - Jeder im Programm benutzten Variablen wird in dieser Phase Speicherplatz zugeordnet.
  - Die Instruktionen der Zwischendarstellung werden in Maschinenbefehle übersetzt.

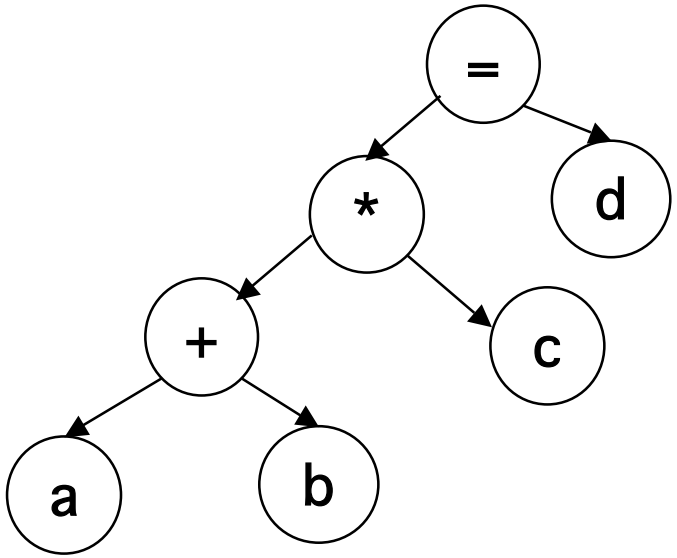
Syntaktische Analyse und Code-Erzeugung

- Im folgenden nehmen wir an, dass
  - die lexikalische Analyse bereits stattgefunden hat (z.B. mit einem endlichen Automaten)
  - alle Bezeichner, Operatoren und Operanden als Elemente gefunden worden sind.
- Wir konzentrieren uns auf die Grundkonzepte der syntaktischen Analyse, Zwischencode-Erzeugung und der Code-Erzeugung:
  - Als Repräsentation für den **Syntax-Baum** nehmen wir den *Kantorowitsch-Baum*
  - Als **Zwischensprache** benutzen wir *Postfix-Notation*
  - Als **Zielsprache** nehmen wir *PMI-Assembler*

Wiederholung: Kantorowitsch-Baum und Postfix-Notation

Zuweisung: `d = (a+b)*c;`

Kantorowitsch-Baum




```
private void postOrder(Node localRoot) {
    if(localRoot != null)   {
      postOrder(localRoot.getLeftChild());
      postOrder(localRoot.getRightChild());
      localRoot.displayNode();
    }
}
```

Die **Postorder-Traversierung** des Kantorowitsch-Baumes ergibt die klammerfreie **Postfix-Notation**

Postfix-Notation

`a b + c * d =`

Übersetzung von Ausdrücken und Zuweisungen in PMI

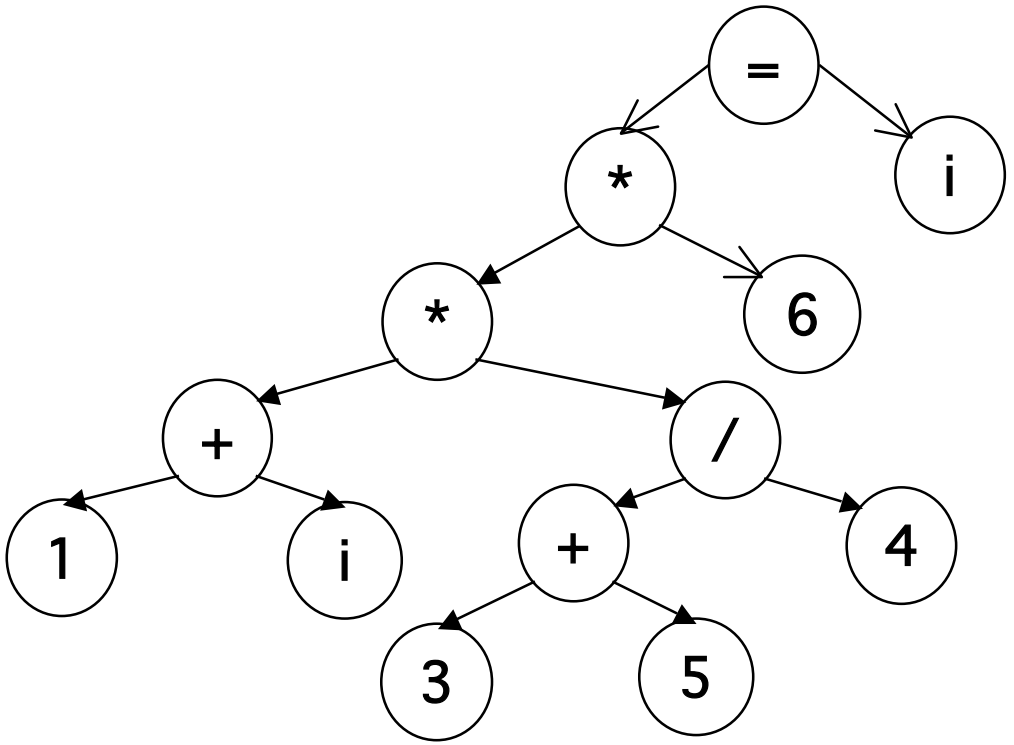
1. **Syntaktische Analyse:** Die Ausgabe der lexikalischen Analyse in *Infix-Notation wird in einen Syntax-Baum* umgewandelt.
2. **Zwischencode- Erzeugung:** Der Syntax-Baum wird in einen *Postfix-Ausdruck* umgewandelt
3. **Code-Erzeugung:** Aus dem Postfix-Ausdruck wird *PMI-Assembler Code nach folgenden Regeln erzeugt*:
   - Jeder Variablen `x` wird Speicherplatz mit dem PMI-Befehl `x: dd 0` zugewiesen.
   - Für eine Variable `x` gefolgt von dem Zuweisungsoperator `=` erzeuge den PMI-Befehl `pop x`
   - Für jede andere Variable `x` erzeuge den PMI-Befehl `push @x`
   - Für eine Konstante `x` erzeuge den PMI-Befehl `push x`
   - Für die Operatoren `+ - /` bzw. `*` erzeuge die PMI-Befehle `add, sub, div` bzw. `mult`

Beispiel

Eingabe (in Infix-Notation):

```
i = ((1 + i) * ((3 + 5) / 4)) * 6;
```

Syntax-Baum:



Zwischencode (in Postfix-notation):

```
1 i + 3 5 + 4 / * 6 * i =
```

Erzeugung des PMI-Assembler Codes

Postfix-Ausdruck: `1 (i) + 3 5 + 4 / * 6 * i =`

Abarbeitung des Ausdrucks

1 i + 3 5 + 4 / * 6 * i =	push 1
1 **i** + 3 5 + 4 / * 6 * i =	push @i
1 i **+** 3 5 + 4 / * 6 * i =	add
1 i + **3** 5 + 4 / * 6 * i =	push 3
1 i + 3 **5** + 4 / * 6 * i =	push 5
1 i + 3 5 **+** 4 / * 6 * i =	add
1 i + 3 5 + **4** / * 6 * i =	push 4
1 i + 3 5 + 4 **/** * 6 * i =	div
1 i + 3 5 + 4 / ***** 6 * i =	mult
1 i + 3 5 + 4 / * **6** * i =	push 6
1 i + 3 5 + 4 / * 6 ***** i =	mult
1 i + 3 5 + 4 / * 6 * **i =**	pop i

Erzeugter PMI-Assembler Code

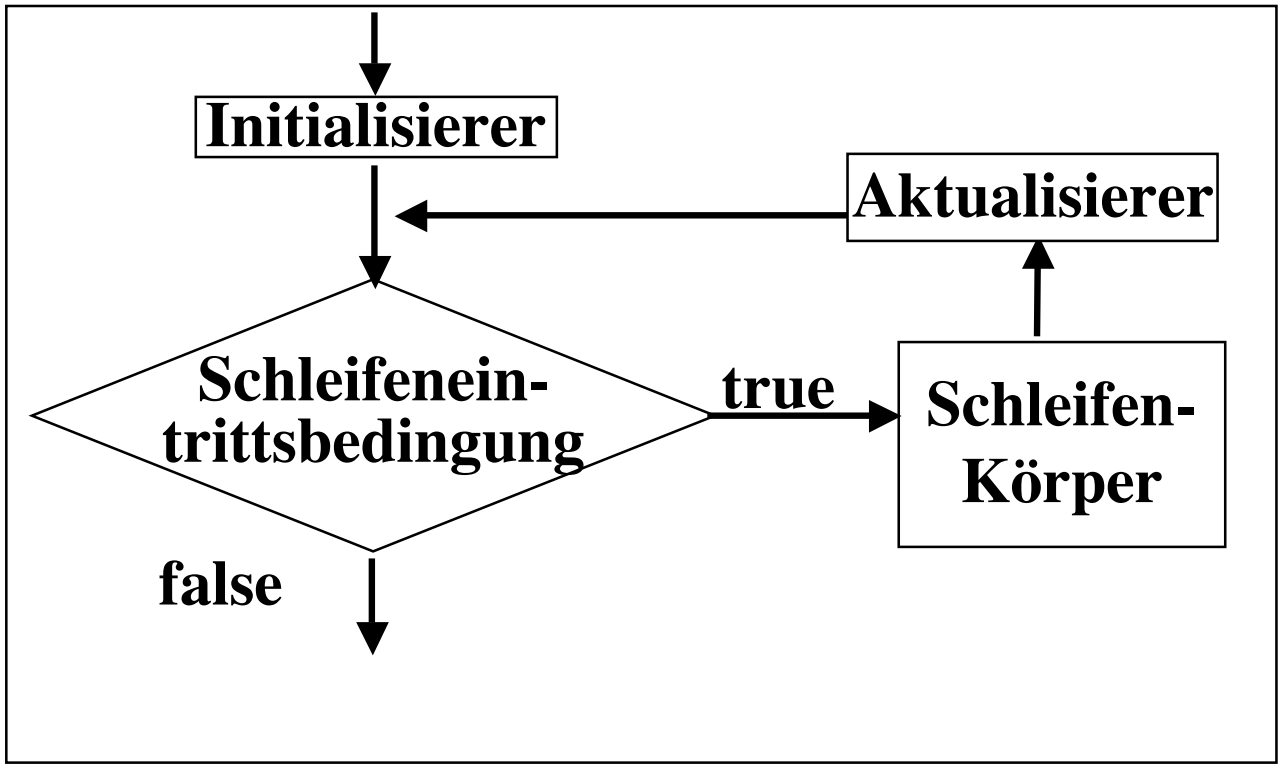
```
i:      dd      0
```

Das vollständige Ziel-Programm (in PMI)

```
push  1
push  @i
add         // 1 i +
push  3
push  5
add         // 1 i + 3 5 +
push  4
div         // 1 i + 3 5 + 4 /
mult        // 1 i + 3 5 + 4 / *
push  6
mult        // 1 i + 3 5 + 4 / * 6 *
pop   i     // i =((1+i)*((3+5)/4))*6;
halt
i:    dd     0
```

Übersetzung von While Strukturen

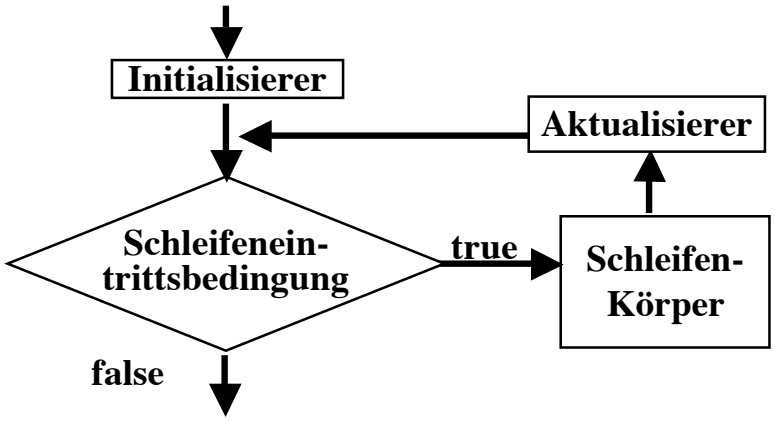
Aus Info I: Terminierende While Struktur



- Im folgenden konzentrieren uns nur auf die Code-Erzeugung.
- Wir nehmen also an, dass die Lexikalische Analyse, Syntaktische Analyse und Zwischencode-Erzeugung schon stattgefunden haben.

Übersetzung von While-Schleifen

- Die Code-Erzeugung für eine While-Schleife besteht aus folgenden Teilen:
  - PMI-Code für Initialisierer
  - PMI-Code für Schleifeneintrittsbedingung
  - PMI-Code für Schleifenkörper
  - PMI-Code für Aktualisierer



Java-Beispiel:

```
i = 10;
while (i > 0) {
  j = j+2; i--;
}
```

PMI Assembler-Code für While-Schleife

Initialisierer

Schleifenkörper

Aktualisierer

Schleifeneintritts-bedingung

```
        push  10     // i = 10;
        pop   i
        jump  test
loop:   push  @j     // j = j+2;
        push  2
        add
        pop   j
        push  @i     // i--;
        push  1
        sub
        pop   i
test:   push  @i     // while (i > 0)
        comp
        del
        jmpn ende    // i < 0 ==> ende
        jmpz ende    // i == 0 ==> ende
        jump   loop  // i > 0 ==> loop

ende:
```

Übersetzung von Methodenaufrufen

- Zunächst einige wichtige Konzepte
  - Aktivierung
  - Aufrufbaum
  - Kontrollkeller
  - Aktivierungssegment
- Wir machen dabei folgende Annahme:
  - Die Ausführung des Programms besteht aus einer Folge von Schritten. Die Kontrolle befindet sich bei jedem Schritt an einen bestimmten Punkt im Programm.
    - Wir betrachten also nur sequentielle, keine nebenläufigen Programme.

Aktivierung

- Als **Aktivierung** einer Methode *m* bezeichnen wir die Ausführung ihres Methodenrumpfes.
  - Die Aktivierung beginnt am Anfang des Rumpfes und führt irgendwann zu dem Punkt direkt hinter dem Methodenaufruf zurück (d.h. wir behandeln keine Ausnahmen).
- Als **Lebenszeit einer Aktivierung** bezeichnen wir die Zeit der Aktivierung einer Methode *m*
  - Dies ist die Zeit für die Ausführung der Aktivierung, inklusive der Zeit für die Ausführung der von *m* aufgerufenen Methoden, der von diesen wiederum aufgerufenen Methoden, usw.
- Falls *m* und *n* Aktivierungen sind, dann sind ihre **Lebenszeiten** entweder **nicht überlappend** oder **geschachtelt**.

Aufrufbaum

- **Definition:** Ein **Aufrufbaum (auch Aktivierungsbaum genannt)** ist ein allgemeiner Baum (also kein Binärbaum), der die Folge aller Aktivierungen während der Ausführung eines Programms beschreibt.

Für Aufrufbäume gilt:

Jeder Knoten stellte die Aktivierung einer Methode dar.

Der Knoten für *a i*st genau dann Elternknoten für *b*, wenn der Kontrollfluss von Aktivierung *a* zu *b* verzweigt.

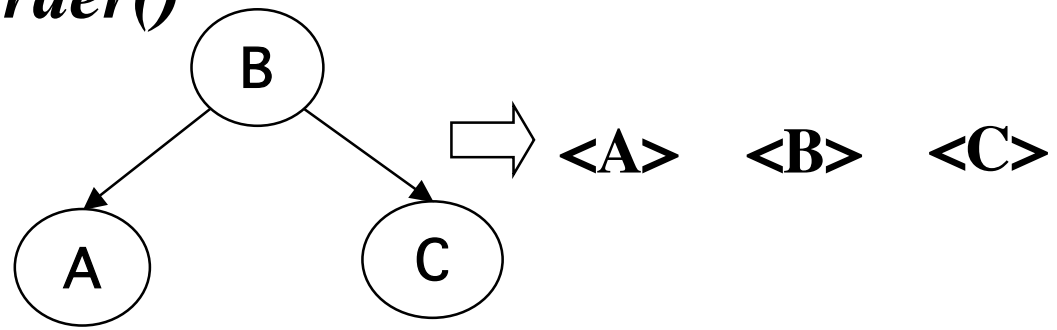
Eine Kante *(a,b)* bedeutet also, dass *b* von *a* aktiviert worden.

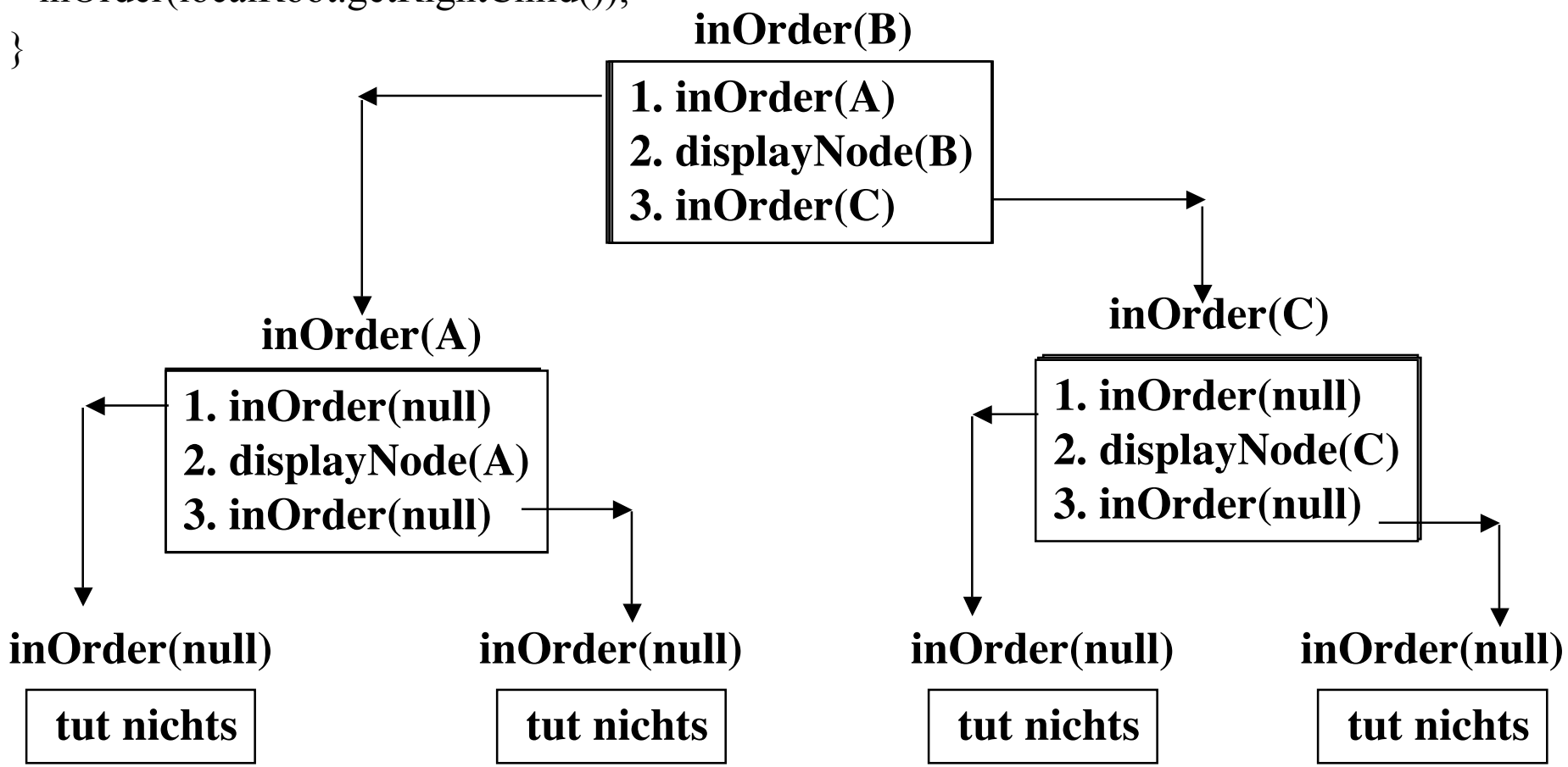
Der Knoten für *b* ist unter dem Knoten für *a*, wenn die Lebenszeit von *b* vor der Lebenszeit von *a* beendet ist.

Wenn der Aufrufbaum für ein gesamtes Java Programm gezeigt wird, dann stellt die Wurzel die Aktivierung der Hauptklasse **main()** dar.

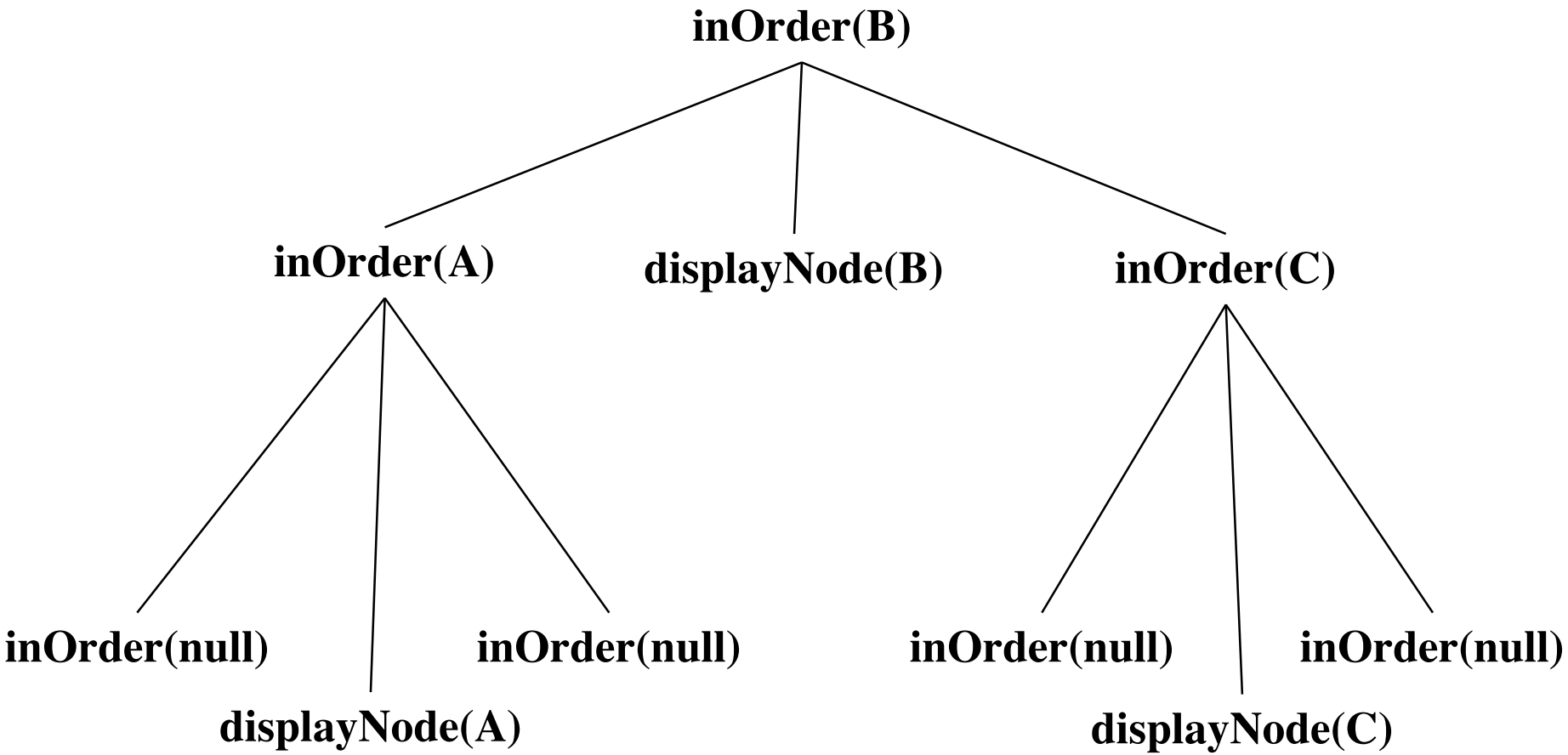
Beispiel: Aufrufbaum für inOrder()

```
private void inOrder(Node localRoot) {
    if(localRoot != null)   {
      inOrder(localRoot.getLeftChild());
      localRoot.displayNode();
      inOrder(localRoot.getRightChild());
    }
}
```



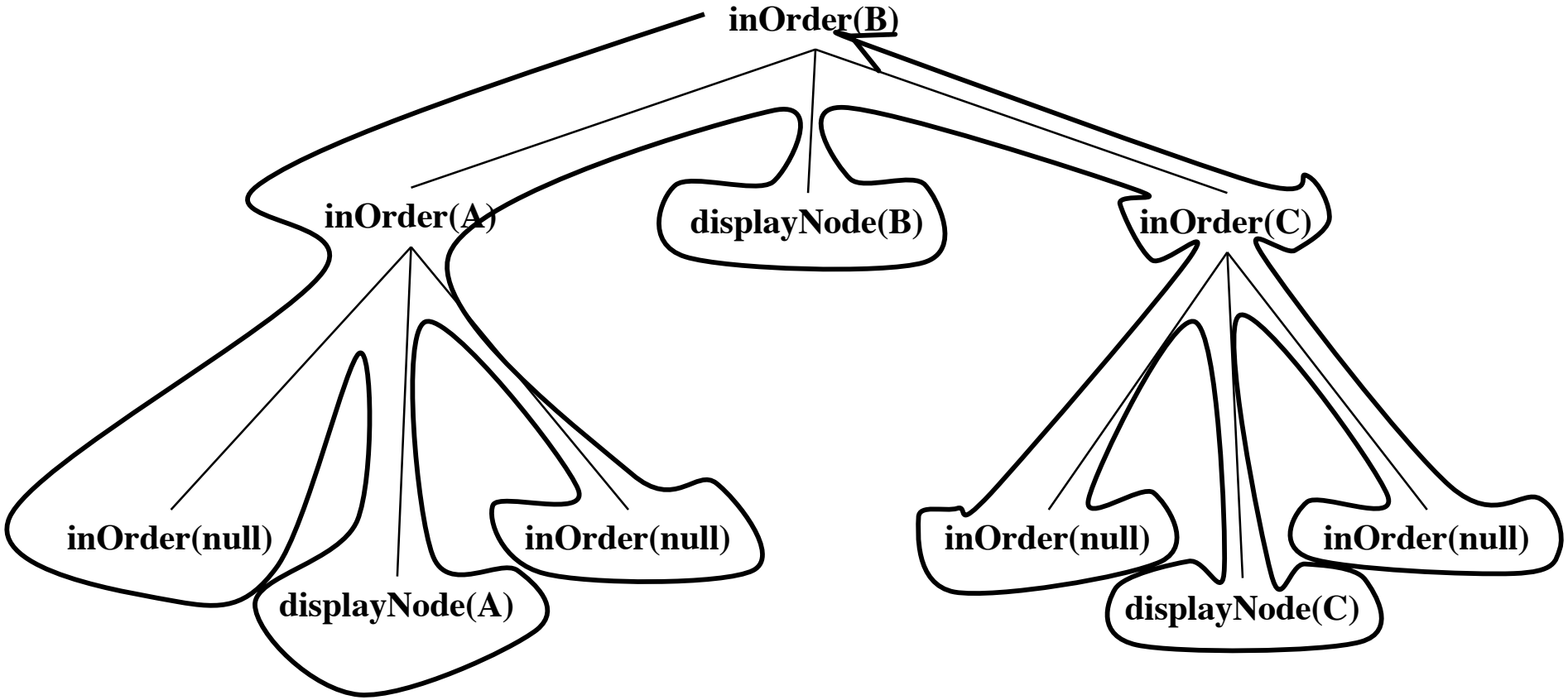


Aufrufbaum für das InOrder-Beispiel



Kontrollfluss

- **Definition:** Der **Kontrollfluss** eines Programms entspricht einer Vorordnungs-Traversierung des Aufrufbaums, der an der Wurzel beginnt. Vorordnungs-Traversierung wird auch Tiefendurchlauf (engl. *depth-first*) genannt).

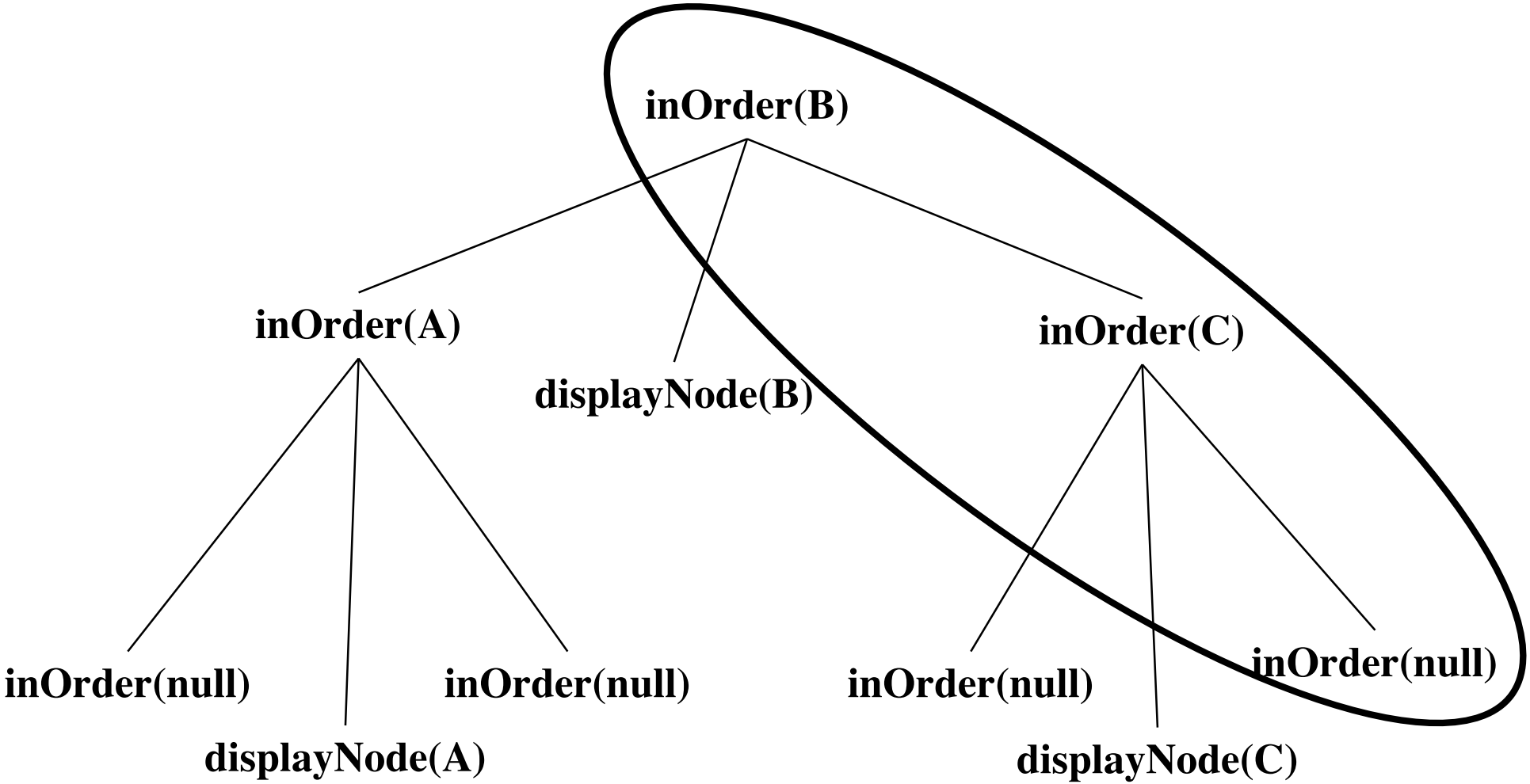


Kontrollkeller

- **Definition Kontrollkeller:** Enthält die aktuellen Aktivierungen während der Ausführung eines Programms. Ein Kontrollkeller wird mit den folgenden Operationen manipuliert:
  - `Push()`: Legt den Knoten für eine Aktivierung auf den Kontrollkeller, wenn die Aktivierung beginnt.
  - `Pop()`: Nimmt den Knoten für eine Aktivierung vom Keller, wenn die Aktivierung endet.
- Der Aufrufbaum beschreibt ***alle*** Aktivierungen, die im Verlauf eines Programms ausgeführt werden. Der Kontrollkeller beschreibt dagegen nur die ***aktuellen*** Aktivierungen.
  - Der Aufrufbaum enthält also die gesamte Geschichte der Ausführung des Programs.
  - Der Kontrollkeller enthält die Knoten des Aufrufbaumes entlang eines Pfades bis zur Wurzel.

Beispiel

- An einem bestimmten Punkt während der Exekution enthält der Kontrollkeller `inOrder(B), inOrder(C), inOrder(null).`



Speicherverwaltung mit dem Kontrollkeller

- Bei der Implementierung von Übersetzern für höhere Programmiersprachen übernimmt der Kontrollkeller die Verwaltung des Stapels im Arbeitsspeicher.
  - Der Stapel wird im Übersetzerbau auch oft als **Laufzeitstapel** bezeichnet.
- Die Knoten des Kontrollkellers werden durch sogenannte **Aktivierungssegmente** realisiert.
- **Definition Aktivierungssegment:** Die Menge aller Elemente, die zur Verwaltung der Ausführung einer Aktivierung benötigt wird.

Elemente eines Aktivierungssegmentes

Rückgabewert: Wird von der aufgerufenen Methode benutzt, um der aufrufenden Methode einen Wert zurückzugeben.

Aktuelle Parameter: Dieses Feld wird von der aufrufenden Methode benutzt, um die Parameter an die aufgerufene Methode zu übergeben.

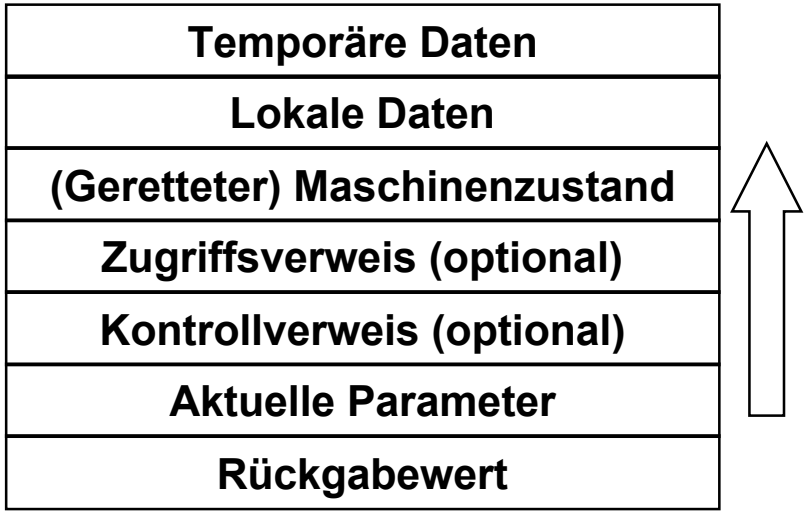
Kontrollverweis (optional)**:** Zeiger auf Aktivierungssegment der aufrufenden Methode.

Zugriffsverweis (optional): Zeiger auf Aktivierungssegment der statisch übergeordneten Methode (für nichtlokale Zugriffe)

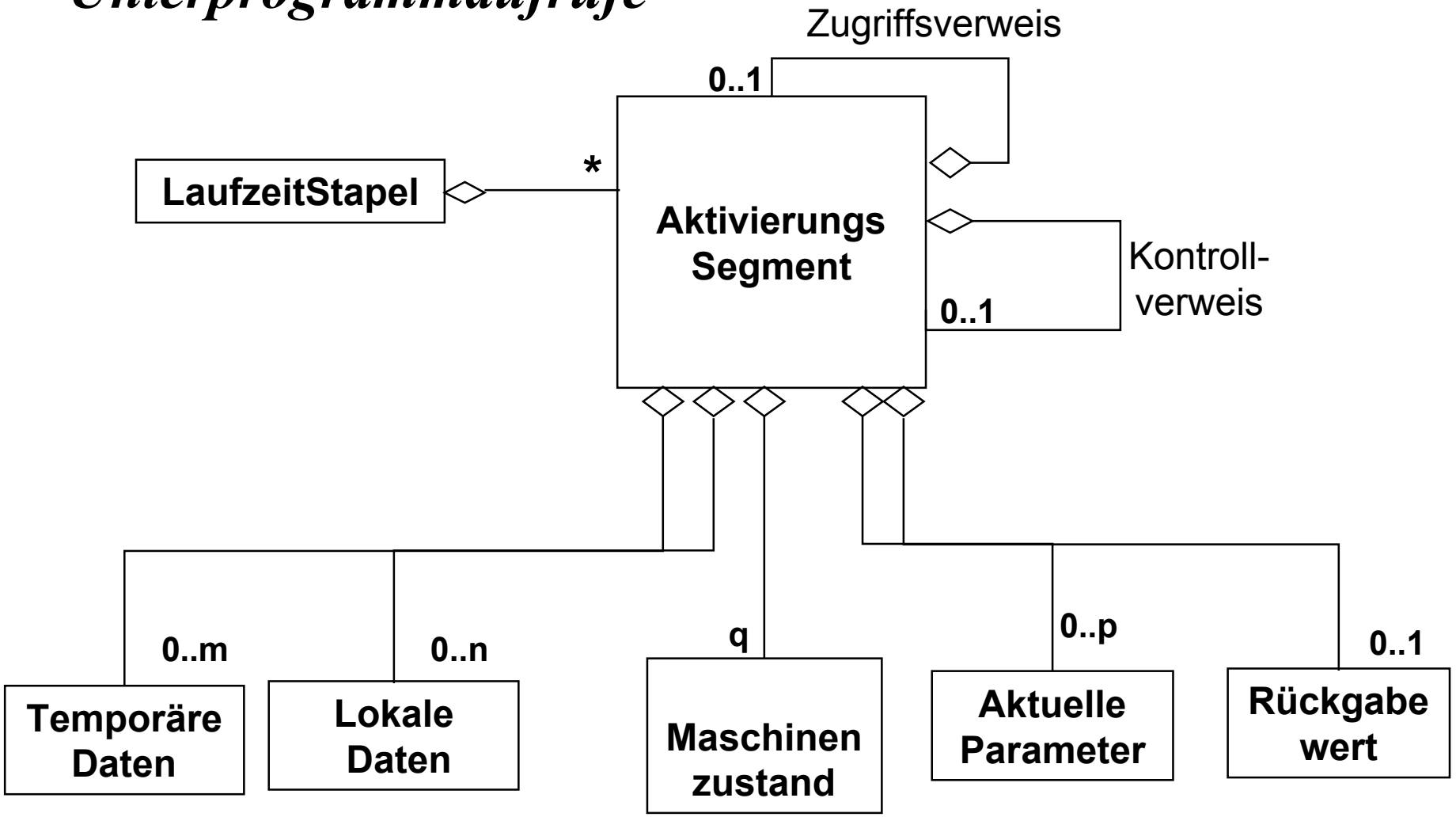
Maschinenzustand: Zustand der Maschine, bevor die Methode aufgerufen wurde. Bei PMI die Adresse des nächsten Befehls, d.h. die Rücksprungadresse (siehe jsr-Befehl).

Lokale Daten: Zum Speichern lokaler Attribute der aufgerufenen Methode.

Temporäre Daten: Treten beim Berechnen von Ausdrücken in der aufgerufenen Methode auf.

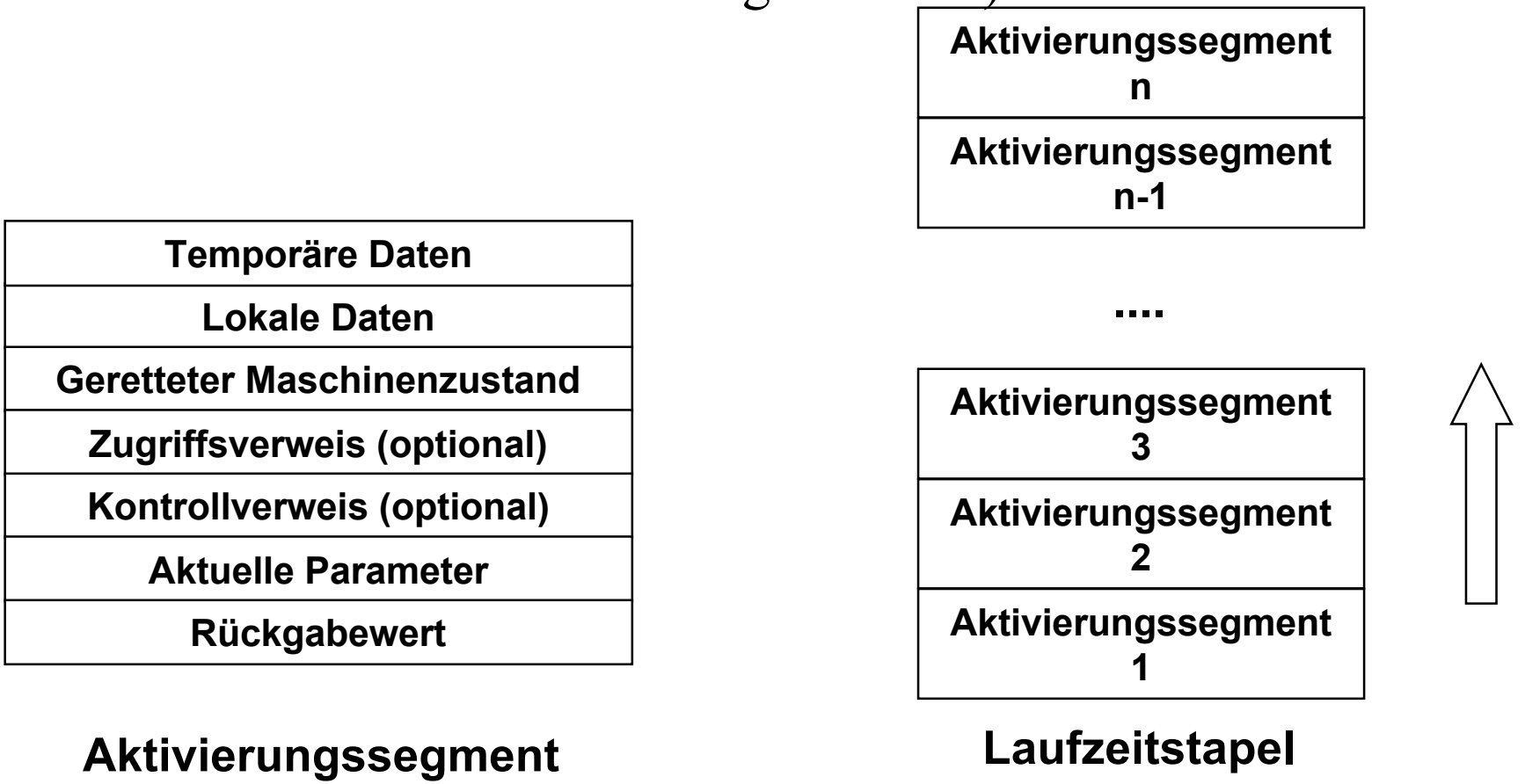


Modellierung der Speicherverwaltung für Unterprogrammaufrufe

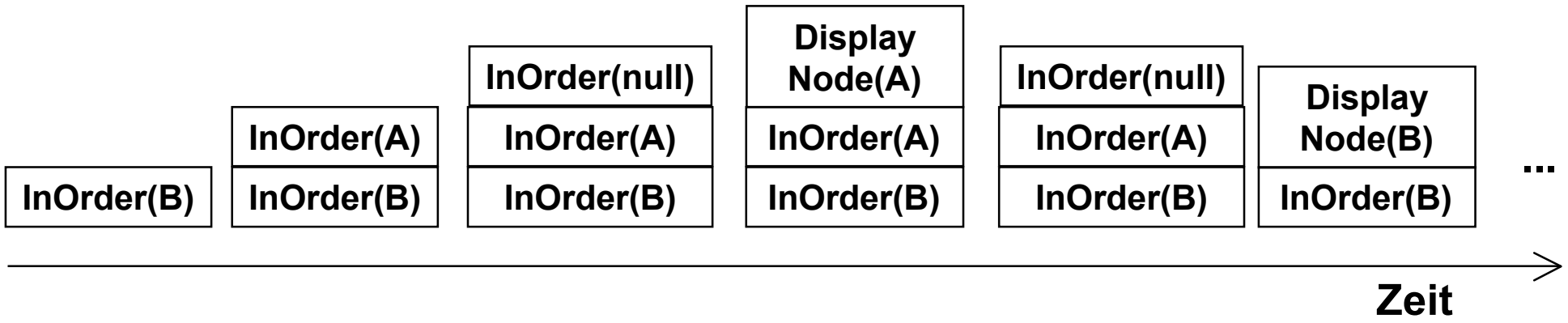


Darstellungen für Laufzeitstapel und Aktivierungssegment

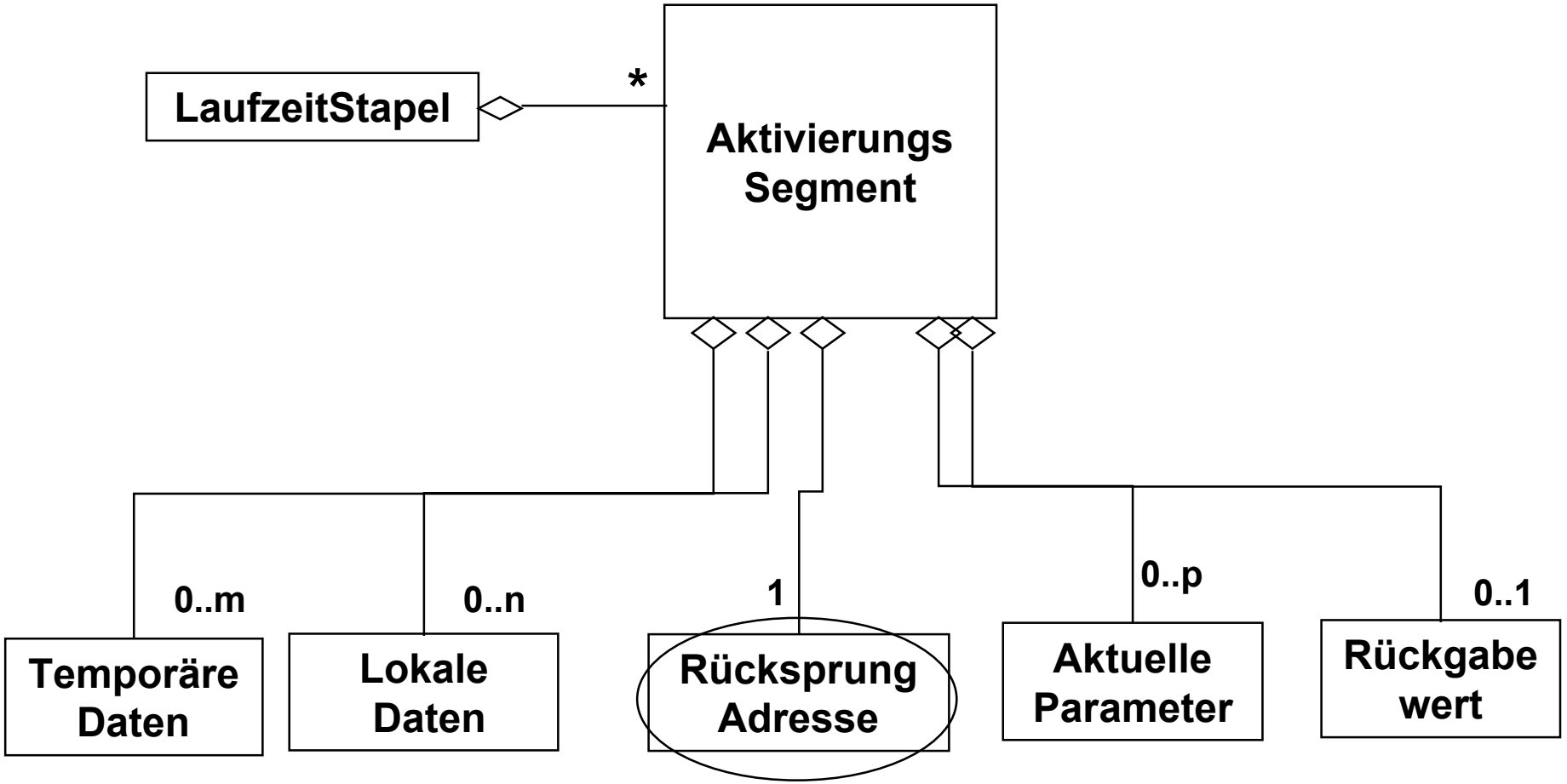
- Im Übersetzerbau werden Laufzeitstapel und Aktivierungssegmente im allgemeinen nicht in UML gezeichnet, sondern als kontinuierliche Blöcke von Speicherzellen (Insbesondere werden also keine Assoziationen gezeichnet).



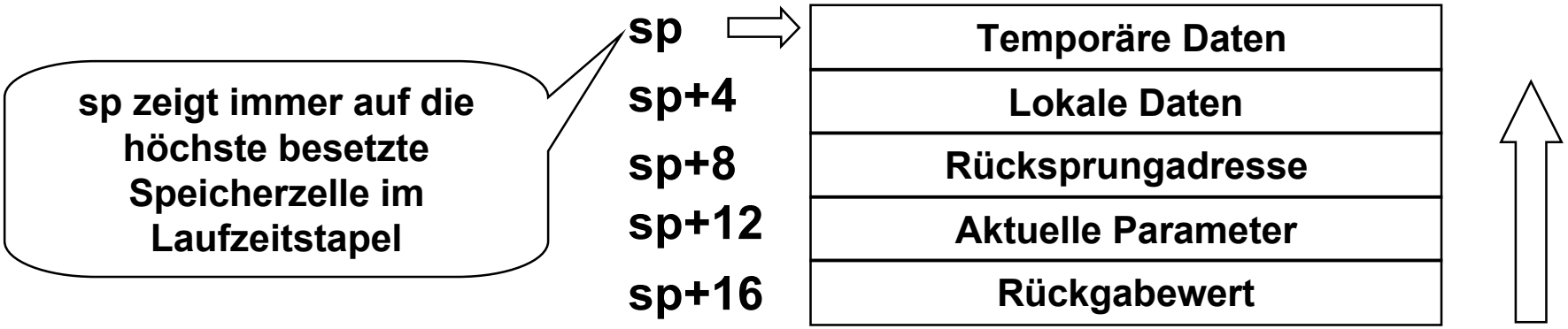
„Füllung“ des Laufzeitstapels bei der Ausführung von `InOrder()`



Modellierung des Stapelaufbaus bei Unterprogramm-aufrufen in PMI



Aufbau eines Aktivierungssegmentes für PMI



Übersetzung von Methodenaufrufen

Grundidee: Aufbau des Aktivierungssegmentes durch Arbeitsteilung zwischen aufrufendem Teil und aufgerufenem Teil.

1. Aufrufender Teil:

a) Platzreservierung für Rückgabewert

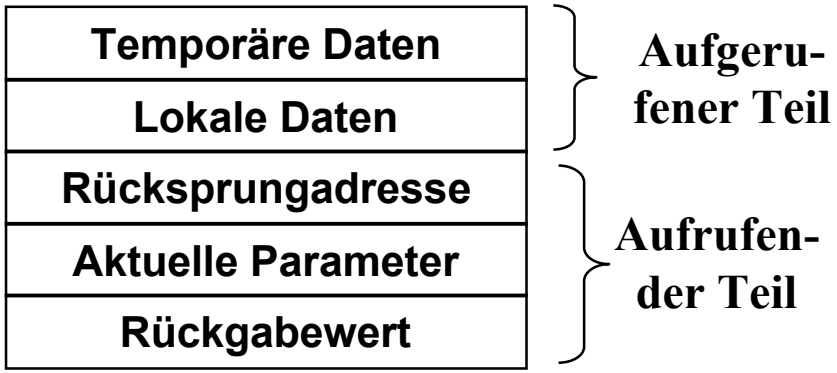
b) Kopieren der aktuellen Parameter

c) Sprung zum Unterprogramm (jsr)

2. Aufgerufener Teil:

a) Belegung der lokalen Daten

b) Ausführung des Unterprogramms
Berechnung von temporären Daten



Rücksprung über Rücksprungadresse

Beispiel: Fakultätsfunktion

Deklaration der Methode („Unterprogramm“):

```
public int fakultaet(int i) {
    int n = i;
    if (n == 0) return 1;
    else return n * fakultaet(n-1);
}
```

Aufruf im Hauptprogramm:

```
.....
i = 5;
j = fakultaet(i);
```

PMI-Code für Fakultätsfunktion

```
        push    5
        pop     i           // i = 5;
// Hauptprogramm:
        push    0           // Platz für Rückgabewert
        push    @i          // Akt. Parameter int i
        jsr     fakultaet
        del                 // Parameter löschen
        pop     j           // j = fakultaet(i);
        halt
```

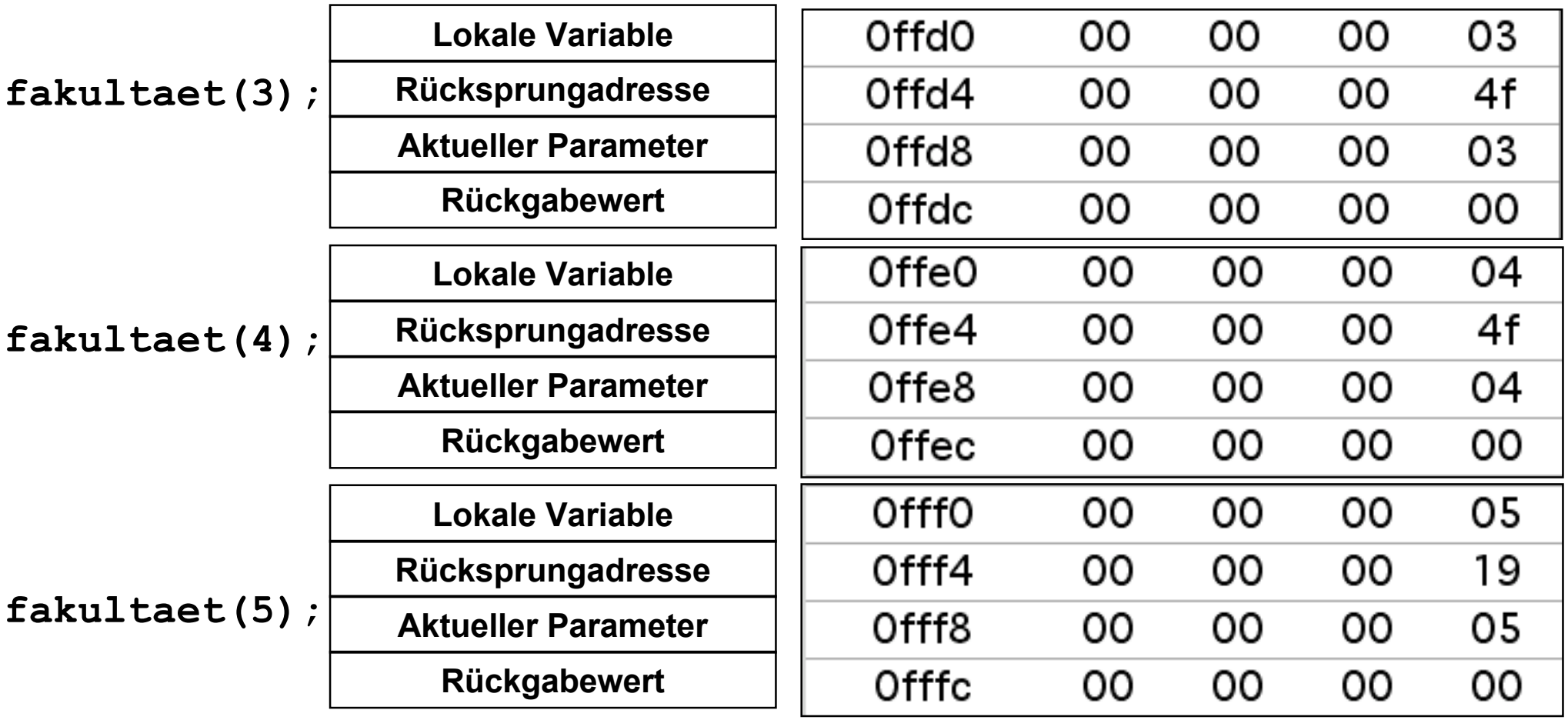
Aufrufender Teil

```
fakultaet:push  @sp+4       // int n = i;
test:   comp                // if (n ==0) return 1;
        jmpz    ende
        jump    rekursion
ende:   push    1
        jump    ergebnis
```

Aufgerufener Teil

```
rekursion: push  0          // Rückgabewert
        push    @sp+4       // Akt.Parameter
        push    1
        sub                 // n - 1
        jsr     fakultaet
        del                 // Parameter löschen
        push    @sp+4
        mult
ergebnis: pop   sp+16       // Ergebniswert
        del                 // Lokale Daten löschen
        ret
i:      dd      0           // Die Variable i
j:      dd      0           // Die Variable j
```

PMI-Stapel mit den ersten 3 Aktivierungssegmenten



Übersetzung rekursiver Datenstrukturen

- Rekursive Datenstrukturen wie Listen, Bäume usw. sind nicht statisch definiert, sondern werden dynamisch zur Laufzeit aufgebaut.
- Wir können rekursive Datenstrukturen in PMI deshalb nicht im Speicherbereich für Code/statische Daten speichern.

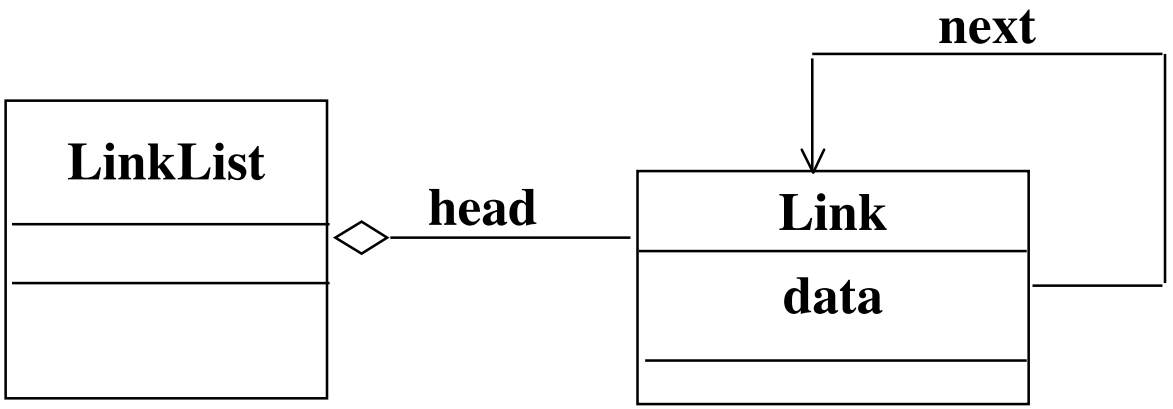
Grundidee: Wir speichern dynamisch erzeugte Daten auf der ***Halde*.** Ein Daten-Element einer rekursiven Datenstruktur entspricht also einem Speicherbereich auf der Halde.

Allgemein: die ***Halde*** wird zur Speicherung nicht-lokaler Daten verwendet, die dynamisch während des Programmablaufs "erzeugt" werden.

Repräsentation von Referenzen in PMI

- Eine Referenz dient zur eindeutigen Identifizierung bzw. Lokalisierung eines Objektes
- **Konzept:** Wir verwenden die ***Adresse*** des Daten-Elementes einer rekursiven Datenstruktur, d.h. seine Speicherbereichs-Adresse auf der Halde, als Referenz auf dieses Element.
- Referenzen auf andere Daten-Elemente werden zusammen mit dem Daten-Element auf der Halde gespeichert

Beispiel: Verkettete Liste



- Grundbaustein der verketteten Liste ist das Listenelement (`link`). Ein Listenelement enthält zwei Attribute:
  - Applikationsspezifische Daten-Elemente (`data`)
  - Eine Referenz auf das nächste Listenelement (`next`)

Repräsentation von Listen auf der Halde

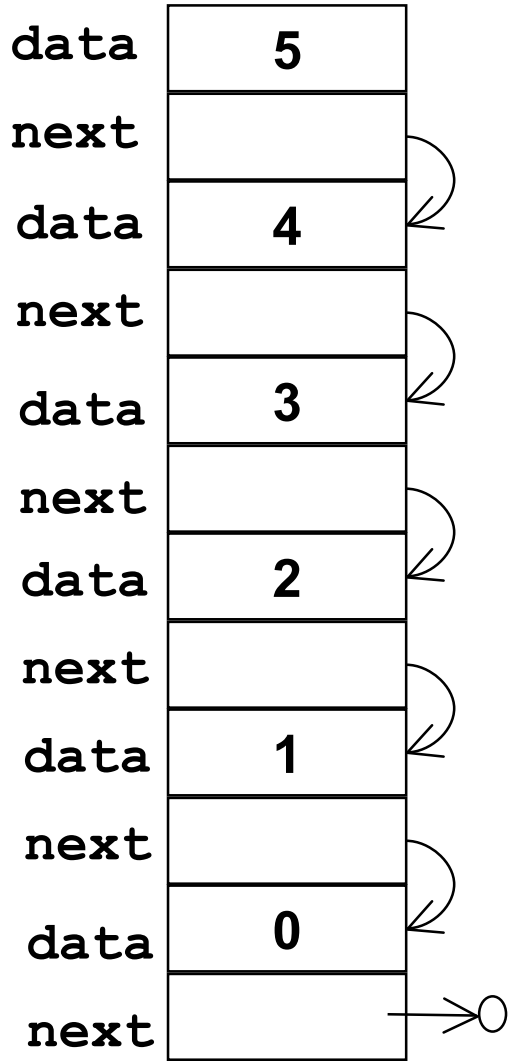
1) Repräsentation jedes Listenelementes
   - `data` (4 bytes): die ganze Zahl
   - `next` (4 bytes): Adresse des nächsten Listenelementes

2) Das letzte Element der Liste kennzeichnen wir dadurch, dass wir sein `next` Feld auf `Null` setzen.

Berechnung der Nachfolgeradresse eines Listenelementes:

- Ausgehend von der Adresse des Listenelementes greifen wir auf das Feld **`next`** zu.
- Mit indirekter Adressierung auf **`next`** holen wir uns von dort die Nachfolgeradresse.

Allgemein: Indirekte Adressierung wird benötigt, um auf Daten, die auf der Halde abgelegt sind, zugreifen zu können.



Java-Beispiel: Erzeugung einer Liste mit 5 ganzen Zahlen

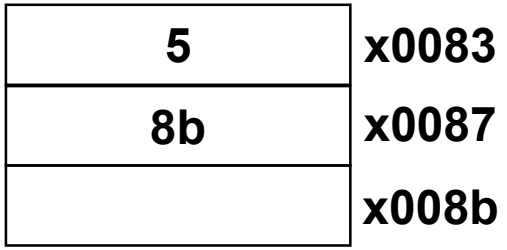
```
...
int i = 5;
while (i > 0)
    { int address = sucheEnde();
      einfügeElement(address, i);
      i--; }
...
public int sucheEnde() {
  int a = hp;
  while (memory[a+4] != 0)  {
      a = memory[i+4]; }
  return a;
}
...
public einfügeElement(int a, int v) {
  memory[a] = v;
  int next = a+8;
  memory[a+4] = next;
}
```

Irgendwo in einer Methode

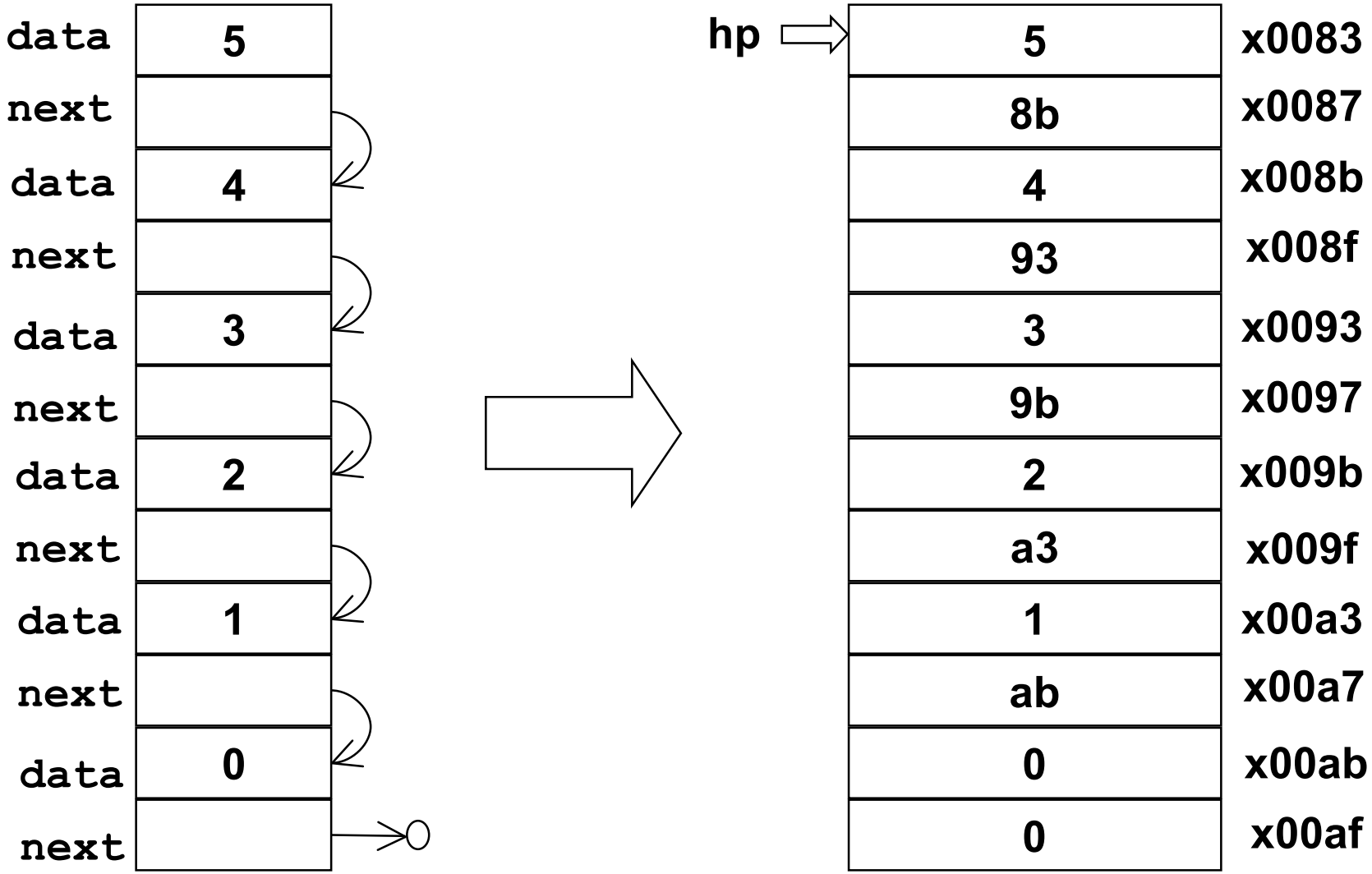
Listenende erreicht?

Zur Vereinfachung nehmen wir an, dass das PMI-Register `hp` und der PMI Arbeitspeicher `memory` direkt zugreifbar sind

Beispiel mit a=x0083 und v = 5:



Speicherung der Liste auf der Halde



PMI-Implementation der verketteten Liste

```
// Java-Code: int i = 5;
// while (i > 0) { int address = sucheEnde();einfügeElement(address, i); i--; }
// PMI-Hauptprogramm
main:     push    5                 // int i = 5;
test:     comp                      // (i > 0) überprüfen
          jmpz    ende
schleife: push    0                 // temporäre Variable: int address;
          jsr     sucheEnde         // address = sucheEnde();
          push    @sp+4             // i als 2. Argument übergeben
          jsr     neuesElement      // einfügeElement(address, i);
          del                       // Aufrufargument i löschen
          del                       // Aufrufargument address löschen
          push    1
          sub                       // i--;
          jump    test              // Ende der Schleife
ende:     del                       // i vom Stack entfernen
          halt                      // Ende Hauptprogramm
```

PMI-Implementation von `sucheEnde()`

```
//Java-Code:  int a = hp; while (mem[a+4] != 0) { a = mem[i+4]; } return a;
// PMI-Unterprogramm sucheEnde
sucheEnde:       push    hp              // lokale Variable: int a = hp;
testListenende:  push    @sp             // temporäre Variable a+4 berechnen
                 push    4
                 add
                 push    >sp             // Nachfolger-Adresse mem[a+4] holen
                 comp                    // (mem[a+4] != 0) überprüfen
                 jmpz    endeGefunden
nachfolger:      pop     sp+8            // a = mem[a+4];
                 del                     // temporäre Variable a+4 löschen
                 jump    testListenende  // Ende des Schleifenrumpfs
endeGefunden:    del                     // mem[a+4] löschen
                 del                     // temporäre Variable a+4 löschen
                 pop     sp+8            // return a;
                 ret
```

PMI-Implementation von `einfügeElement()`

```
// Java-Code: mem[a] = v; int next = a+8; mem[a+4] = next;
// PMI-Unterprogramm einfügeElement
// Erster Parameter:      Einfüge-Adresse a
// Zweiter Parameter:     Elementwert v
einfügeElement: push    @sp+4   // Wert v des neuen Elements holen
        pop     @sp+12          // mem[a] = v;
        push    @sp+8           // temporäre Variable: a+4
        push    4
        add
        push    @sp+12          // lokale Variable: int next = a+8;
        push    8
        add
        pop     @sp+4           // mem[a+4] = next;
        del                     // temporäre Variable a+4 löschen
        ret
```

Übersetzung von Klassen und Objekten

- Objekte werden wie Daten-Elemente dynamisch auf der Halde erzeugt
- Unterschied zu Daten-Elementen:
  - Objekte sind Instanzen einer Klasse, d.h zusätzlich zu Daten-Elementen haben Objekte zugeordnete Methoden.

- Fragen:
  - Wie setzt man Vererbung um?
  - Wie realisiert man Sichtbarkeit?
  - Wie implementiert man Polymorphismus?

Übersetzung von Polymorphismus

- **Frage**: Wie können wir einen Methodenaufruf an den richtigen Methodenrumpf binden?
- **Lösung**: Jede Klasse verwaltet eine Tabelle mit den Adressen aller Methodenrümpfe (Unterprogramme), die auf Instanzen dieser Klasse ausführbar sind.
  - Jede Instanz instanziiert ihre eigene Tabelle
  - Bindung: für jeden Methodenaufruf wird aus der instanz-spezifischen Tabelle des Objektes mit indirekter Adressierung die Adresse des entsprechenden Methodenrumpfs ausgewählt.
- Für Interessierte: PMI-Beispielprogramm `Klasse.pmi`
- ***Hauptstudiumsvorlesung: Übersetzung objektorientierter Sprachen***

Das Halteproblem

- Wir haben jetzt einige Grundkonzepte für die Übersetzung von Java in PMI kennengelernt.
- Obwohl wir nicht alle Konzepte besprochen haben, können wir annehmen, dass es prinzipiell möglich ist einen Compiler zu schreiben, der Java in PMI übersetzt.
  - Dann können wir natürlich die PMI-Maschine selbst als Programm in den Arbeitspeicher der PMI-Maschine laden.
  - Dann haben wir ein Programm, das sich selbst lesen kann (Selbsteinsicht).
- Alan Turing benutzt diesen Trick der Selbsteinsicht, um zu beweisen:
  - Es gibt unendlich viele Probleme, die man nicht mit einer Rechenanlage lösen kann.
  - Das prominenteste Problem bezeichnet man als das Halteproblem.
- **Halteproblem:** Gibt es einen Algorithmus, der entscheiden kann, ob ein beliebiges Programm terminiert oder nicht?

Das Halteproblem

- Wenn es einen Algorithmus geben würde, der das Halteproblem löst, dann könnte man ihn benutzen, um unendliche Schleifen bereits während der Compilationsphase zu entdecken. Das wäre sehr nützlich!
- Indirekter Beweis, dass das Halteproblem unlösbar ist:
  1. Nehmen wir an, wir haben ein Programm *P*, welches das Halteproblem löst. Ausserdem nehmen wir an, P hat eine boolesche Variable `Terminiert` hat, die es folgendermassen setzt:
     - wenn *P* ein terminierendes Programm *Q* als Eingabe bekommt, setzt es `Terminiert` auf `true`.
     - wenn P ein nicht-terminierendes Programm *Q'* als Eingabe bekommt, setzt es `Terminiert` auf `false`.
  2. Wir erstellen jetzt eine neue Version *P'*, die mit *P* identisch ist, abgesehen von einer Änderung:
     - Da, wo *P* `Terminiert` auf `true` oder `false` setzt, enthält *P'* die while-Schleife

```
while (Terminiert == true) do {};
```

Das Halteproblem (2)

3. P' hat folgendes Verhalten:
   - Wenn P' ein terminierendes Programm als Eingabe bekommt, dann führt P' eine unendliche Schleife aus, terminiert also nicht.
   - Wenn P' ein nicht-terminierendes Programm als Eingabe bekommt, dann führt P' diese unendliche Schleife nicht aus, terminiert also.

- **Frage**: Was passiert, wenn P' sich selbst, d.h. P' als Eingabe bekommt?
- Die Antwort erzeugt einen Widerspruch:
  - Wenn P' ein terminierendes Programm ist, dann terminiert es nicht, wenn es P' als Eingabe hat.
  - Wenn P' ein nicht-terminierendes Programm ist, dann terminiert es, wenn es P' als Eingabe hat.
- Unsere Annahme, dass P das Halte-Problem löst, führt also zu einem Widerspruch führt,
- Wir müssen deshalb annehmen, dass es kein Programm gibt, dass das Halteproblem lösen kann. Das Halteproblem ist also unlösbar.

Zusammenfassung

- Ein Compiler besteht aus einem **Analyseteil** (z.B. lexikalische, syntaktische und semantische Analyse) und **Syntheseteil** (z.B. Zwischencode-erzeugung, Optimierung, Code-Erzeugung).
  - In der syntaktischen Analyse wird der Syntax-Baum hergestellt. Während der Codeerzeugung wird maschinen-naher Code für die Zielmaschine erzeugt.
- Übersetzung von
  - Java-Ausdrücken, Zuweisungen, While-Schleifen
  - Operationsaufrufen
  - rekursiven Datenstrukturen
- **Realisierung von Unterprogrammaufrufen**
  - Konzepte Aufrufbaum, Kontrollkeller und Aktivierungssegment.
- Der Laufzeitstapel und die Halde werden zur Speicherung von Daten benutzt, die erst zur Laufzeit erzeugt werden.
  - **Laufzeitstapel**: Verwaltet die Aktivierungssegmente der aufgerufenen Operationen.
  - **Halde**: Verwaltet nicht-lokale Daten von rekursiven Datenstrukturen und Objekten (als Instanzen von Klassen).