



## Nachtrag: Visualisierung von Werten im PMI-Arbeitsspeicher

- ❖ Die Inhalte der einzelnen Speicherzellen (Größe: 1 Byte) des PMI-Arbeitsspeichers werden als vorzeichenlose Zahl in Hexadezimal-Darstellung (zur Basis 16) repräsentiert. (⇒ 2 Ziffern pro Speicherzelle)
- ❖ Führende Nullen werden angegeben

- ❖ Beispiele:

-1 → ffffffff

42 → 000002a

255 → 00000fff

256 → 0000100

Repräsentation des Werts -1 in PMI-Assemblersprache

Repräsentation des Werts -1 in PMI-Maschinencode

## Nachtrag: Adressierung in PMI-Assembler

i: dd 65532 // 0xffffc

- ❖ Unmittelbare Adressierung

push 5

push i

Code/  
Statische  
Daten

0000a 0000ffffc

- ❖ Direkte Adressierung:

push @i

- ❖ Indirekte Adressierung:

push >i

Stapel

0fff0	00000005
0fff4	0000ffffc
0fff8	0000000a
0fffc	00000005

Annahme:  
i = 0000000a  
sp = 00010000

## Übersetzer

- ❖ **Definition:** Ein **Übersetzer** ist ein Programm, das ein in einer **Quellsprache** geschriebenes Programm in ein äquivalentes Programm in einer **Zielsprache** übersetzt.
- ❖ Das Spektrum bei den Quellsprachen reicht von Modellierungssprachen über höhere Programmiersprachen bis zu Spezialsprachen, die in nahezu allen Bereichen entstanden sind, in denen Informatik-Systeme einsetzbar sind.
- ❖ Dementsprechend gibt es viele Übersetzer. Wir unterscheiden drei Klassen von Übersetzern:
  - **CASE-Werkzeug:** Übersetzt eine Modellierungssprache 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 Kontrollkellers 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: Vorlesung und Praktikum Übersetzerbau**

## *Einsatz von Übersetzern*

- ❖ Es gibt viele Bereiche, in denen wir Übersetzer verwenden, um von einer Quellsprache in eine Zielsprache zu übersetzen.
- ❖ **Textverarbeitungssystem:** Erhält als Eingabe eine Zeichenkette mit Kommandos (RTF, MIF, LaTeX usw.) und formatiert sie als Dokument (Word, Framemaker, PDF 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.
  - *VHDL haben Sie schon in TGI kennengelernt.*
  - *Hauptstudium: Vorlesung und Praktikum Rechnerarchitektur*
- ❖ **Anfrage-Interpreterer:** Übersetzt ein Prädikat einer Quellsprache mit relationalen und booleschen Operatoren in ein Kommando einer Zielsprache (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
  - 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 Übersetzers*

- ❖ Der Analyse-Teil eines Übersetzers besteht aus lexikalischer, syntaktischer und semantischer Analyse.
- ❖ Während der **lexikalischen Analyse** wird der Programmtext in seine Bestandteile (reservierte Wörter, Bezeichner, Operatoren) zerlegt.
- ❖ In der **syntaktischen Analyse** werden, ausgehend von den einzelnen Programmbestandteilen, die im Quellprogramm enthaltenen Operationen bestimmt und in einem sog. *Syntaxbaum* angeordnet:
  - Im Syntaxbaum stellt jeder Knoten eine Operation dar, die Kinderknoten repräsentieren die Operanden der Operation.
- ❖ In der **semantischen Analyse** werden die Typ-Informationen für jeden Operanden im Syntaxbaum ermittelt, Typ-Überprüfungen gemacht, und evtl. notwendige Operandenanpassungen (implicit type casts) durchgeführt.

## *Der Synthese-Teil eines Übersetzers*

- ❖ Der Synthese-Teil eines Übersetzers besteht aus Zwischencod- Erzeugung, Code-Optimierung und Code-Erzeugung.
- ❖ **Zwischencod-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 Zwischencod verbessert, um möglichst effizienten Zielcode zu erzeugen.
- ❖ **Code-Erzeugung:** Ziel ist die Erzeugung von Maschinencod für die Maschine, auf der das Programm ausgeführt werden soll.
  - Allen im Programm benutzten Variablen wird in dieser Phase Speicherplatz zugeordnet.
  - Die Instruktionen der Zwischendarstellung werden in Maschinenbefehle übersetzt.

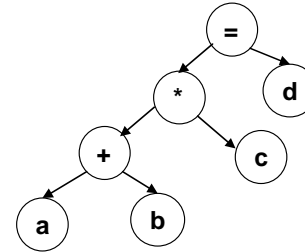
### Beispiel für den Übersetzungsvorgang: Übersetzung von Ausdrücken

- ❖ Im folgenden nehmen wir an, dass die lexikalische Analyse bereits stattgefunden hat (z.B. mit einem endlichen Automaten) und der zu übersetzende Ausdruck in Infix-Notation zur Verfügung steht. Alle verwendeten Bezeichner, Operatoren und Operanden seien gültige Programmelemente.
- ❖ Wir konzentrieren uns auf die Grund-Konzepte der Syntax-Analyse, Zwischencode-Erzeugung und der Code-Erzeugung.
  - Als Repräsentation für den Syntaxbaum nehmen wir den Kantorowitsch-Baum aus Info I.
  - Als Zwischensprache benutzen wir Postfix-Notation
  - Als Zielsprache nehmen wir PMI-Assembler.

### Aus Info I: 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

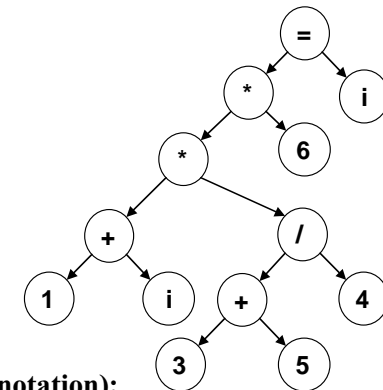
- 1. Syntaktische Analyse:** Die Ausgabe der lexikalischen Analyse in *Infix-Notation* wird in einen *Syntaxbaum* umgewandelt.
- 2. Zwischencode-Erzeugung:** Der *Syntaxbaum* wird in einen *Postfix-Ausdruck* umgewandelt
- 3. Code-Erzeugung:** Aus dem *Postfix-Ausdruck* wird *PMI-Assembler-code* nach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

### Syntax-Analyse und Erzeugung von Zwischencode

Eingabe (in Infix-Notation):

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

Syntax-Baum:



Zwischencode (in Postfix-notation):

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



## PMI-Assemblercode für while-Schleife

Initialisierer	<pre> push 10 // i = 10; pop i push 0 // j = 0; pop j jump test         </pre>	
Schleifenkörper	<pre> loop: push @j // j = j+2;       push 2       add       pop j         </pre>	
Aktualisierer	<pre> push @i // i--; push 1 sub pop i         </pre>	Die Schleifeneintritts- Bedingung wird oft <i>hinter</i> den Schleifenkörper gesetzt (historische Ursachen)
Schleifeneintritts- bedingung	<pre> test: push @i // while (i &gt; 0)       comp       del       jmpn ende // i &lt; 0 ==&gt; ende       jmpz ende // i == 0 ==&gt; ende       jump loop // i &gt; 0 ==&gt; loop         </pre>	
	<pre> ende: halt  i: dd 0 // int i; j: dd 0 // int j;         </pre>	

## 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

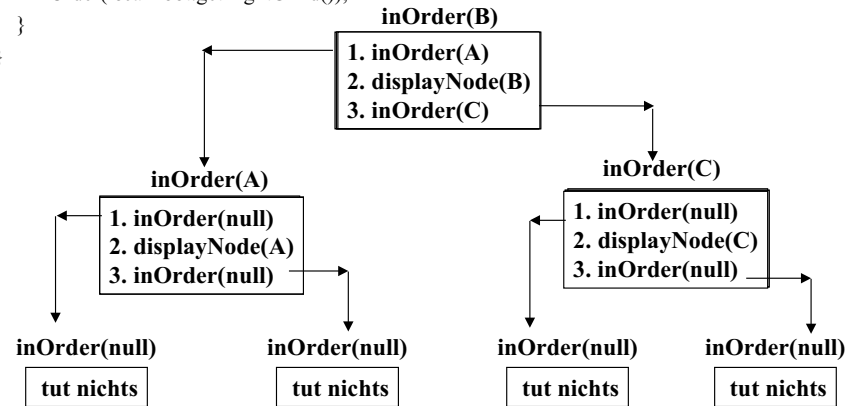
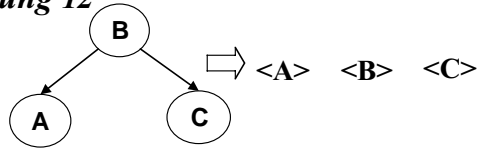
- ❖ Als **Aktivierung**  $A_m$  einer Methode  $m$  bezeichnen wir eine vollständige Ausführung des Methodenrumpfs von  $m$  in Folge eines Aufrufs von  $m$ .
  - Die Aktivierung beginnt am Anfang des Rumpfs und führt irgendwann zu dem Punkt direkt hinter dem Methodenaufruf zurück (d.h. wir behandeln keine Ausnahmen).
- ❖ Als **Lebenszeit einer Aktivierung** bezeichnen wir den Zeitraum, der für die Ausführung des Methodenrumpfs von  $m$  benötigt wird.
  - Dies ist die Zeit für die Ausführung der Methode  $m$ , inklusive der Zeit für die Ausführung der von  $m$  aufgerufenen Methoden, der von diesen wiederum aufgerufenen Methoden usw.
- ❖ Falls  $A_m$  und  $A_n$  zwei verschiedene Aktivierungen sind, dann sind ihre **Lebenszeiten** entweder **nicht überlappend** oder **geschachtelt**.
- ❖ Statt von der **Aktivierung** kann man übrigens auch von der **Inkarnation** einer Methode sprechen.

## Aufrufbaum

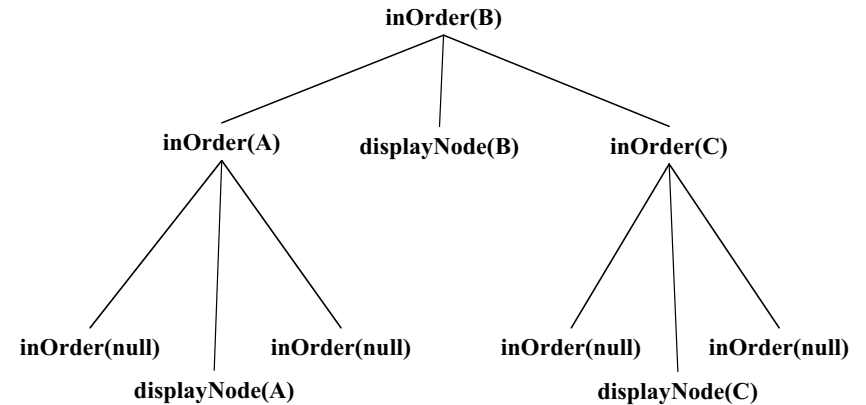
- ❖ **Definition:** Ein **Aufrufbaum** (auch **Aktivierungsbaum** genannt) ist ein allgemeiner Baum (also kein Binärbaum), der die Folge aller Aktivierungen während der Ausführung eines Programms beschreibt.
- ❖ Für Aufrufbäume gilt:
  - Jeder Knoten stellt die Aktivierung einer Methode dar.
  - Der Knoten für  $a$  ist genau dann Elternknoten für  $b$ , wenn der Kontrollfluss von Aktivierung  $a$  zu  $b$  verzweigt.
    - Eine Kante  $(a,b)$  bedeutet also, dass  $b$  von  $a$  aufgerufen wird.
    - Der Knoten für  $b$  ist ein Unterknoten des Knotens 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 Methode **main()** der Hauptprogramm-Klasse dar.

### Beispiel aus Info I - Vorlesung 12

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

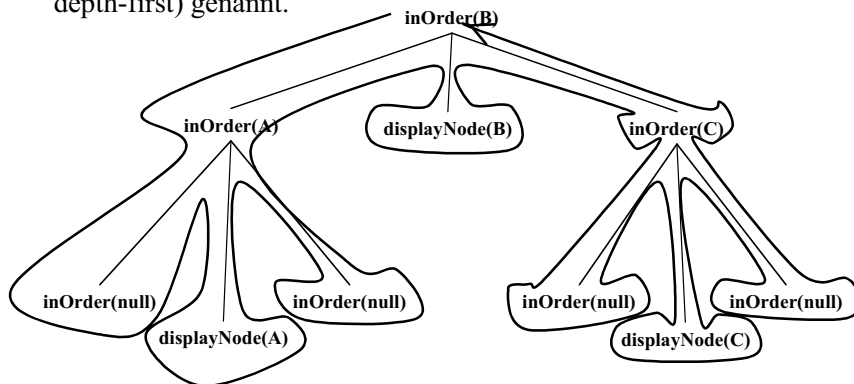


### Aufrufbaum für das inOrder ()-Beispiel



### 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.



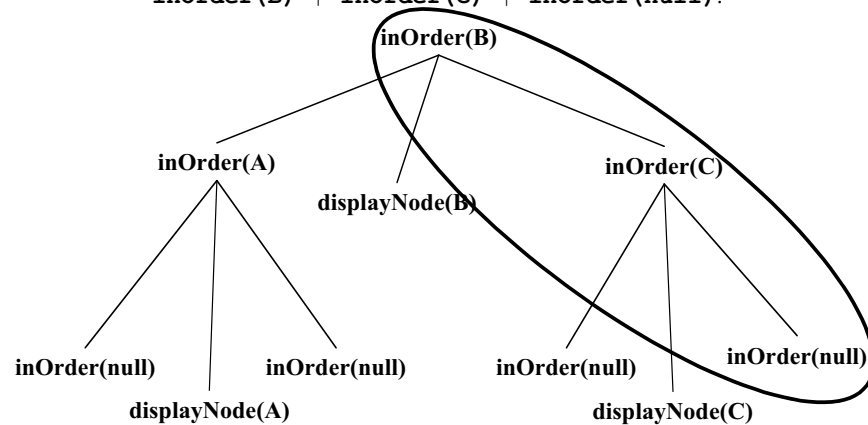
### Kontrollkeller

- ❖ **Definition Kontrollkeller:** Enthält die aktuellen Aktivierungen während der Ausführung eines Programms.
- ❖ **Keller-Prinzip:** Ein Kontrollkeller arbeitet nach dem Keller-Prinzip, d.h. er kann nur mit den folgenden Operationen manipuliert werden:
  - **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** auftretenden Aktivierungen im Verlauf der Ausführung eines Programms. Er enthält also die gesamte Geschichte der Ausführung des Programms.
- ❖ Der Kontrollkeller beschreibt dagegen immer nur die **aktuellen**, d.h. noch nicht beendeten Aktivierungen. Er enthält genau die Knoten, die im Aufrufbaum auf dem Pfad von der jüngsten Aktivierung zur Wurzel liegen.

## Aufrufbaum und Kontrollkeller: Beispiel

- An einem bestimmten Punkt während der Ausführung enthält der Kontrollkeller die Aktivierungen

`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 einer Aktivierung benötigt werden.

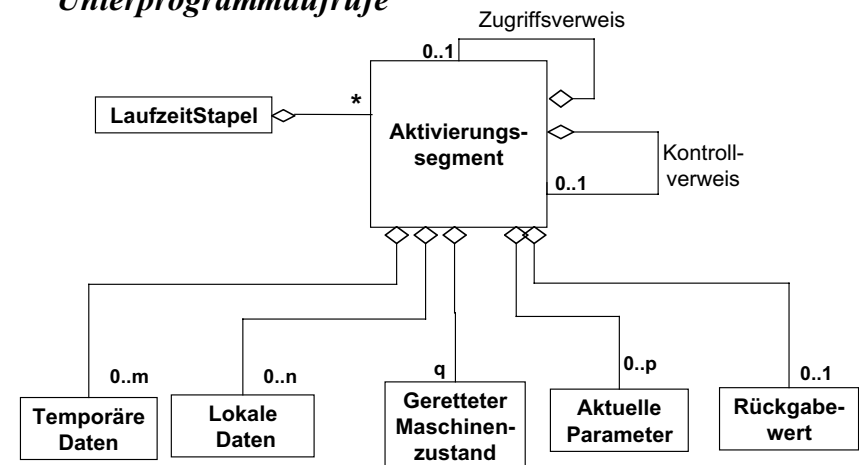
## Elemente eines Aktivierungssegments

- Rückgabewert:** Wird von der aufgerufenen Methode benutzt, um der aufrufenden Methode einen Wert (als Ergebnis) zurückzugeben.
- Aktuelle Parameter:** Dieser Bereich wird von der aufrufenden Methode benutzt, um 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:** Zwischenergebnisse beim Berechnen von Ausdrücken in der aufgerufenen Methode.

Temporäre Daten
Lokale Daten
Geretteter Maschinenzustand
Zugriffsverweis (optional)
Kontrollverweis (optional)
Aktuelle Parameter
Rückgabewert

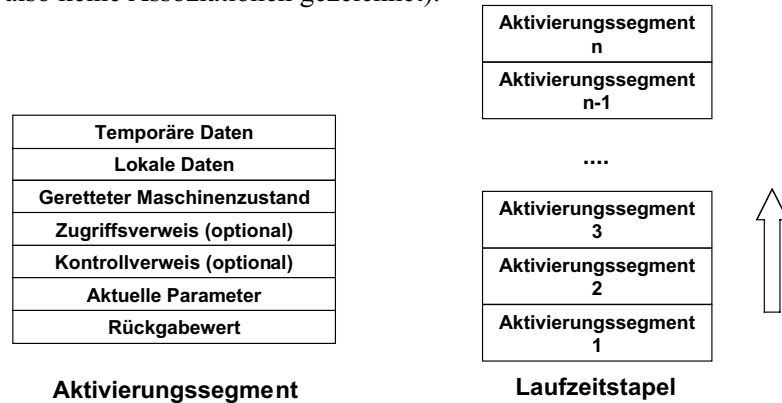


## Modellierung der Speicherverwaltung für Unterprogrammaufrufe

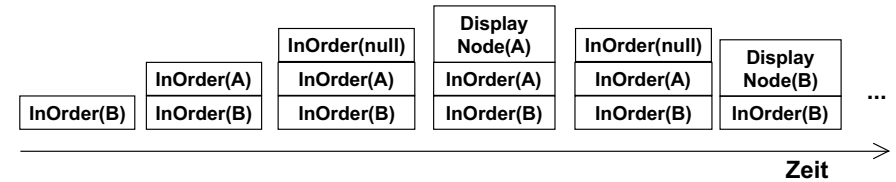


## Darstellungen für Laufzeitstapel und Aktivierungssegment

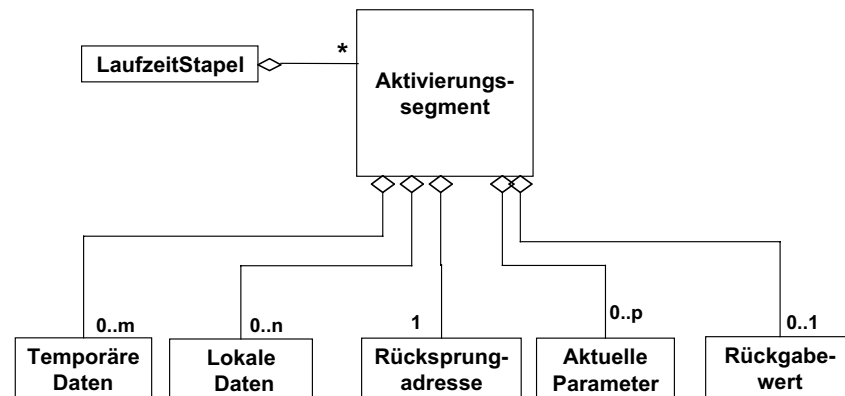
- ❖ Im Übersetzerbau werden Laufzeitstapel und Aktivierungssegmente im allgemeinen nicht in UML gezeichnet, sondern als kontinuierliche Blöcke von Speicherzellen (Insbesondere werden also keine Assoziationen gezeichnet).



## Animation des Laufzeitstapels für InOrder ()

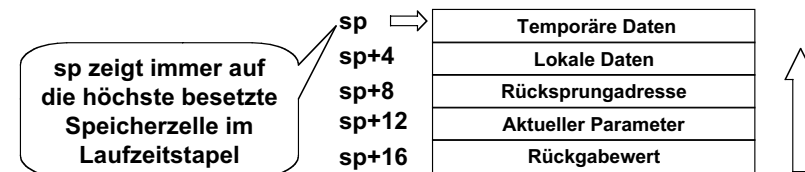


## Modellierung des Stapelaufbaus bei Unterprogrammaufrufen in PMI



## Aufbau eines Aktivierungssegmentes für PMI

Beispiel: Aktivierungssegment für Methode mit einem Parameter



## Übersetzung von Methodenaufrufen

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

### 1. Aufrufender Teil:

- Platz für Rückgabewert
- Kopieren der aktuellen Parameter
- Sprung zum Unterprogramm (jsr)

### 2. Aufgerufener Teil:

- Belegung der lokalen Daten
- Ausführung des Unterprogramms  
Berechnung von temporären Daten
- Rückprung über Rücksprungsadresse



## Beispiel: Fakultätsfunktion

**Deklaration:**

```

...
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-Assemblercode für Fakultätsfunktion

```

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

fakultaet: push @sp+4 // Lokale Daten
test:     comp
        jmpz ende
        jump rekursion

ende:     push 1
        jump ergebnis

rekursion: push 0 // Rückgabewert
        push @sp+4 // Akt. Parameter
        push 1 // i minus 1
        sub
        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

<b>fakultaet (3) ;</b>	Lokale Variable	Offd0	00	00	00	03
	Rücksprungsadresse	Offd4	00	00	00	4f
	Aktueller Parameter	Offd8	00	00	00	03
	Rückgabewert	Offdc	00	00	00	00
<b>fakultaet (4) ;</b>	Lokale Variable	Offe0	00	00	00	04
	Rücksprungsadresse	Offe4	00	00	00	4f
	Aktueller Parameter	Offe8	00	00	00	04
	Rückgabewert	Offec	00	00	00	00
<b>fakultaet (5) ;</b>	Lokale Variable	Offf0	00	00	00	05
	Rücksprungsadresse	Offf4	00	00	00	19
	Aktueller Parameter	Offf8	00	00	00	05
	Rückgabewert	Offfc	00	00	00	00

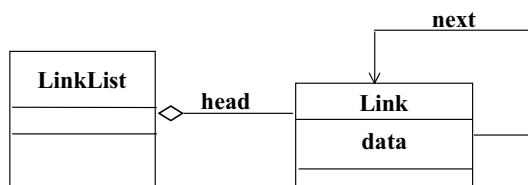
## Übersetzung rekursiver Datenstrukturen

- ❖ 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-Elements 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

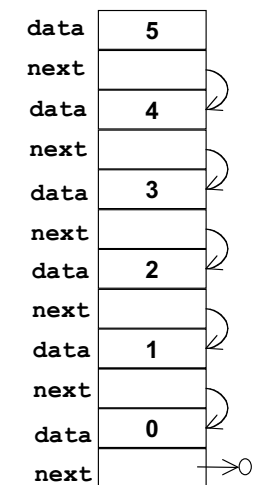
## Beispiel: Verkettete Liste



- ❖ Aus Info I: Grundbaustein der verketteten Liste ist das Listenelement (**link**). Ein Listenelement enthält zwei Attribute:
  - Applikationsspezifische Daten-Elemente (**data**)
  - Eine Referenz auf das nächste Listenelement (**next**)
- ❖ Im folgenden Beispiel gehen wir davon aus, dass das Attribut **data** einer ganzen Zahl (**int**) entspricht.

## Repräsentation von Listen auf der Halde

- 1) Repräsentation jedes Listenelements
    - **data** (4 bytes): eine ganze Zahl
    - **next** (4 bytes): Nachfolgeradresse des nächsten Listen-Eintrags
  - 2) Das letzte Element der Liste kennzeichnen wir dadurch, dass wir **next** auf den Wert **0** setzen.
  - 3) Berechnung der Nachfolgeradresse eines Listen-Elements:
    - Ausgehend von der Adresse des Listen-Elements greifen wir auf die Adresse **next** zu.
    - Mit **indirekter Adressierung** auf **next** holen wir uns von dort die Nachfolgeradresse.
- ❖ **Allgemein:** Indirekte Adressierung wird benötigt, um auf Daten, die im Arbeitsspeicher abgelegt sind, anhand ihrer Adresse gezielt zugreifen zu können.



## Java-Beispiel: Erzeugung einer Liste mit 5 ganzen Zahlen

```

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

```

Listenende erreicht?

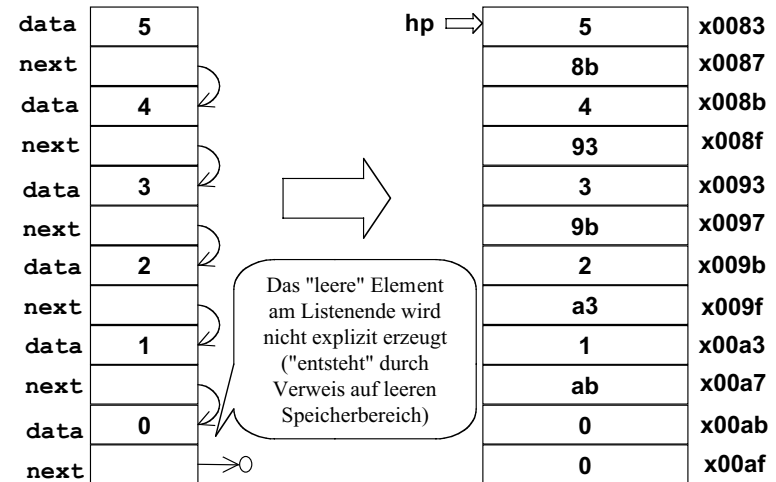
Irgendwo in einer Methode

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

Beispiel mit a=0x0083 und v = 5:

5	x0083
8b	x0087
	x008b

## Speicherung der Liste auf der Halde



## PMI-Implementation der verketteten Liste

```

// Java-Code: int i = 5;
// while (i > 0) {
//     int address = sucheEnde(); neuesElement(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 // "neuesElement(address, i);"
        del          // Aufrufargument "i" löschen
        del          // Aufrufargument "address" löschen
        push 1       // "i--;"
        sub
        jump test
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 `neuesElement()`

```
// Java-Code: mem[ a] = v; int next = a+8; mem[ a+4] = next;

// PMI-Unterprogramm neuesElement
// Erster Parameter: Einfüge-Adresse "a"
// Zweiter Parameter: Elementwert "v"
neuesElement: 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

- ❖ Alle Objekte werden wie Daten-Elemente dynamisch auf der Halde erzeugt
- ❖ Unterschied zu reinen 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 Methodenadress-Tabelle mit den Adressen aller Methodenrümpfe (Unterprogramme), die auf Instanzen dieser Klasse ausführbar sind.
  - Jede Instanz instanziiert ihre eigene Methodenadress-Tabelle
  - Bindung: für jeden Methodenaufruf wird aus der Instanz-spezifischen Methodentabelle des Objektes mit indirekter Adressierung die Adresse des entsprechenden Methodenrumpfs ausgewählt.
- ❖ Für Interessierte: PMI-Beispielprogramm `Klasse.pmi`
- ❖ **Hauptstudiumsvorlesung: Übersetzung objektorientierter Sprachen**

## Zusammenfassung

- ❖ Die Übersetzung eines Programms mit einem Übersetzer gliedert sich in
  - **Analyse** (lexikalische, syntaktische und semantische Analyse)
    - Bei der syntaktischen Analyse wird ein Syntaxbaum erzeugt.
  - **Synthese** (Zwischencode-Erzeugung, Optimierung, Code-Erzeugung)
    - Bei der Code-Erzeugung wird maschinennaher Code für die Zielmaschine erzeugt.
- ❖ Manuelle Übersetzung von Java-Ausdrücken, Zuweisungen, Schleifen, Operationsaufrufen und rekursiven Datenstrukturen.
- ❖ Übersetzung von Operationsaufrufen mit Hilfe von **Aufrufbaum, Kontrollkeller und Aktivierungssegment**.
- ❖ Der **Laufzeitstapel** und die **Halde** werden zur Speicherung von Daten benutzt, die erst zur Laufzeit erzeugt werden: Der Laufzeitstapel enthält die Aktivierungssegmente der aufgerufenen Operationen. Die Halde enthält nicht-lokale Daten von rekursiven Datenstrukturen und Objekten (als Instanzen von Klassen).

### ***Exkurs: Was ist, wenn PMI sich selber liest?***

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

### ***Exkurs: Das Halteproblem (1)***

- ❖ Wenn es einen Algorithmus geben würde, der das Halteproblem löst, dann könnte man ihn benutzen, um unendliche Schleifen bereits während der Übersetzung zu entdecken. Das wäre *sehr* nützlich!
- ❖ Wir skizzieren jetzt einen indirekten Beweis, dass das Halteproblem unlösbar ist:
- ❖ Nehmen wir an, wir haben ein Programm **P**, welches das Halteproblem löst. Ausserdem nehmen wir an, dass **P** eine boolesche Variable **Terminiert** hat, die es folgendermaßen setzt:
  - wenn **P** ein haltendes Programm **Q** als Eingabe bekommt, setzt es **Terminiert** auf **true**.
  - wenn **P** ein nicht-haltendes Programm **Q'** als Eingabe bekommt, setzt es **Terminiert** auf **false**.
- ❖ Wir erstellen jetzt eine neue Version **P'**, die mit **P** identisch ist, abgesehen von folgender Änderung: Da, wo **P** **Terminiert** auf **true** oder **false** setzt, enthält **P'** zusätzlich die Schleife **while (Terminiert == true) do {};**

### ***Exkurs: Das Halteproblem (2)***

- ❖ **P'** hat folgendes Verhalten:
  - Wenn **P'** ein terminierendes Programm als Eingabe bekommt, dann führt **P'** eine Endlos-Schleife aus, terminiert also nicht.
  - Wenn **P'** ein nicht-terminierendes Programm als Eingabe bekommt, dann führt **P'** die Endlos-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 hält es nicht, wenn es **P'** als Eingabe hat.
  - Wenn **P'** ein nicht-terminierendes Programm ist, dann hält es, wenn es **P'** als Eingabe hat.
- ❖ Unsere Annahme, dass **P** das Halte-Problem löst, führt also zu einem Widerspruch.
- ❖ Wir müssen deshalb annehmen, dass es kein Programm gibt, welches das Halteproblem lösen kann. Das Halteproblem ist also unlösbar.