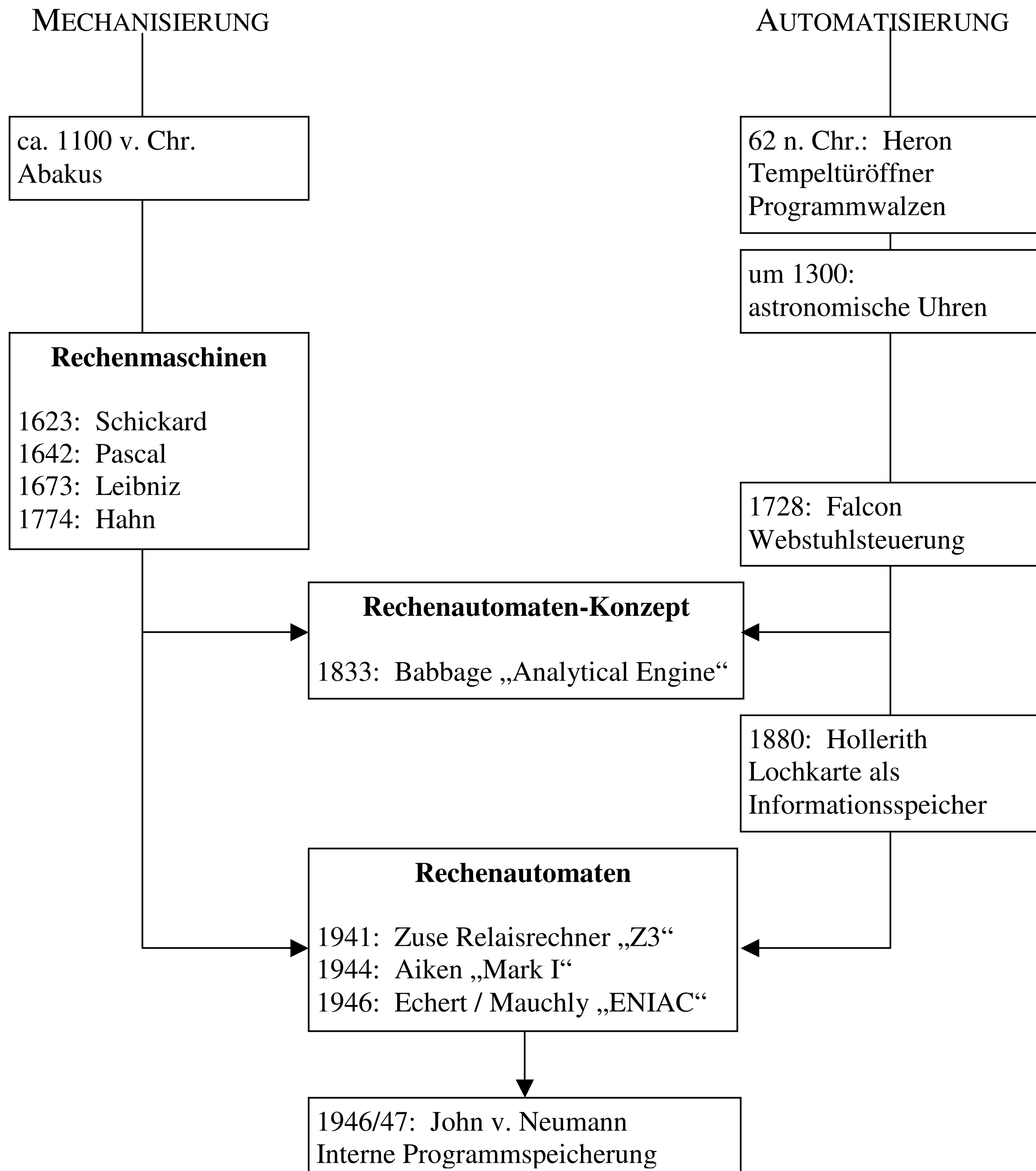
GESCHICHTE DER DATENVERARBEITUNG



SPRACHUMFANG DER MINISPRACHE REJAVA

a) Datentypen und Operationen:

Da die interne Speicherung aller Datentypen genauso wie die Speicherung eines Integer-Datentypen auf die Codierung in Nullen und Einsen hinausläuft, ist es kein Verlust, sich nur auf den Datentypen Integer zu beschränken.

Die einzigen möglichen Operatoren sind:

+ (Addition), **–** (Subtraktion), ***** (Multiplikation) und **/** (Division)

sowie die Vergleichsoperatoren:

<, <=, >, >=, = und **!=**

b) Logische Ausdrücke:

Ein logischer Ausdruck kann wegen des Dreiadressformats nur in der folgenden Form durchgeführt werden:

Ausdruck1 Vergleichsoperator Ausdruck2

Wir nennen einen solchen Ausdruck *Bedingung*

c) Ausdrücke – Zuweisungen:

Bei Zuweisungen sind die folgenden Einschränkungen zu beachten:

1. Es sind nur die oben genannten Operatoren erlaubt.
2. Auf der rechten Seite einer Zuweisung darf höchstens ein Operator stehen.

Eine Zuweisung dieser Art referiert somit auf höchstens Drei Adressen, somit ist das Prinzip des Dreiadressrechners gewahrt.

d) Anweisungen:

Als Anweisungen sind erlaubt:

1. Zuweisung `... = ...`
2. Ein-Ausgabe `... = read(), write(...)`
3. Unbedingter Sprung zur Marke n `goto n`
4. Bedingter Sprung zur Marke n `if (...) goto n`

ARBEITSBLATT ZUM ZERLEGUNGSALGORITHMUS

Der Algorithmus, mit dem man eine arithmetische Zuweisung in eine Folge von Anweisungen im Dreiadressformat zerlegen kann, lautet umgangssprachlich wie folgt:

ALGORITHMUS ***Zerlegung***

I/O-OBJEKTE	Zerlegung:	Zeichenketten (z. B. als Schlange)
Inp-OBJEKT	Ausdruck:	Zeichenkette
HILFSOBJ.	ODK, OTK:	Keller / Stack
	Zeichen:	Zeichen

- Richte zwei Keller ein (ODK, OTK)
- Wiederhole
  - Lies das nächste Zeichen.
  - Falls das Zeichen
    - a) ein Operand ist,
      - kellere ihn im ODK ein.
    - b) ein Operator ist,
      - überprüfe seine Priorität.
      - Solange der Operator keine Priorität gegenüber dem obersten Operator des OTK besitzt,
        tue: • werte den OTK aus.
      - Speichere den neuen Operator im OTK.
    - c) eine öffnende Klammer ist,
      - speichere diese im OTK.
    - d) eine schließende Klammer ist,
      - führe eine Auswertung aller Operatoren zurück bis zur entsprechenden öffnenden Klammer aus
      - entferne die öffnende Klammer aus dem OTK.

  bis Ausdruck ganz abgearbeitet ist.
- Solange der OTK nicht leer ist,
  tue: • werte den OTK aus.

Aufgabe 1: Zerlege die arithmetische Zuweisung

```
x = c * (a - b) - (a + d) / (5 * b)
```

in eine Folge von Dreiadress-Anweisungen in ReJava.

Aufgabe 2: Überführe das folgende Programm in ein Programm im Dreiadressformat:

```
public class Gauss {
  public static void main(String args[]) {
    int n = read();
    int s = (n + 1) * n / 2;
    write(s);
  }
}
```

ARBEITSBLATT ZU DEN ALI-BEFEHLEN

Der Modellrechner ALI stellt für die Transport-, Rechen- und Ein/Ausgabebefehle die folgenden Kürzel zur Verfügung:

Befehlsaufbau	Wirkung	Erläuterung
Transportbefehle:		
`LDA R,ADR`	`R = ADR`	Laden/**L**OA**D** **A**kku
`STA R,ADR`	`ADR = R`	Speichern/**ST**ORE **A**kku
Ein/Ausgabebefehle:		
`INI ADR`	`ADR = read()`	Lesen/**IN**-**I**nteger
`OUTI ADR`	`write(ADR)`	Schreiben/**OUT**-**I**nteger
Rechenbefehle:		
`ADD R,ADR`	`R = R + ADR`	**Add**ieren
`SUB R,ADR`	`R = R - ADR`	**Sub**trahieren
`MUL R,ADR`	`R = R * ADR`	**Mul**tiplizieren
`DIV R,ADR`	`R = R / ADR`	**Div**idieren

- `R` ist ein Register im Bereich 0 bis 15 wobei 0 für den `AKKU` steht.
- `ADR` ist eine Adresse im Speicherbereich, die auf folgende Arten angegeben werden kann:
  1. Als **konkrete Zahl** (absolute Adressierung):
     Die Zahl kennzeichnet einen festen Speicherplatz.
     (Beispiel: L 0,1021 bedeutet, dass der Inhalt der Speicherzelle 1021 in den AKKU geladen wird.)
  2. Als **Name** (symbolische Adressierung):
     Dem Namen wird vom Assembler später eine feste Adresse zugeordnet. Dazu muss der Name als Variable oder Konstante deklariert werden.
     (Beispiel: L 0,HUGO bedeutet, dass dem Namen HUGO bei der Übersetzung eine feste Adresse zugeordnet wird, deren Inhalt in den AKKU geladen wird.)
  3. Als **Zahlenwert in Hochkomma** (unmittelbare Adressierung):
     Der Zahlenwert wirkt im Programm wie eine Konstante.
     (Beispiel: L 0,´123´ bewirkt, dass die Zahl 123 in den AKKU geladen wird.

Für die Deklaration von Variablen und Konstanten stellt ALI die folgenden Befehle zur Verfügung:

Befehlsaufbau	Wirkung	Erläuterung
Pseudobefehle:		
`name DS F`	Legt einen Speicherplatz an, der im Programm mit „name“ aufgerufen werden kann.	Deklaration einer ganzzahligen Variablen **D**efine **S**pace
`konst DC wert`	Legt einen Speicherplatz an, der im Programm mit „konst“ aufgerufen werden kann und weist gleichzeitig diesem den angegebenen „wert“ zu.	Deklaration einer ganzzahligen Konstanten **D**efine **C**onstant

Damit der Assembler ein lauffähiges ALI-Programm übersetzen kann, werden noch die folgenden Befehle benötigt:

Befehlsaufbau	Wirkung	Erläuterung
`START 0`	Programmkopf	Start
`END`	Ende des ganzen Programms (inkl. Variablen)	Ende
`EOJ`	Ende des Anweisungsteils	**E**nd **O**f **J**ob

ARBEITSBLATT ZUR ÜBERSETZUNG IN ALI-ASSEMBLER

Aufgabe 3: Eine Anweisung der Form `h = a + 2` im Dreiadressformat kann mit Hilfe des Akkumulators (`ACCU = Register 0`) in eine Anweisungsfolge im Einadressformat umgeformt werden:

```
                  1) LDA 0,a
h = a + 2         2) ADD 0,b
                  3) STA 0,h
```

Überführe die Anweisungsfolge aus *Aufgabe 1* in eine Anweisungsfolge im Einadressformat. Verwende für die arithmetischen Befehle die Kürzel `ADD` (Addition), `SUB` (Subtraktion), `MUL` (Multiplikation) und `DIV` (Division); für die Transportbefehle zwischen Speicher und `AKKU` die Kürzel `LDA` (Load) und `STA` (Store) des ALI-Assemblers.

Aufgabe 4: Übersetze die folgende Zuweisung zunächst in die Zwischensprache (Dreiadressformat) und dann in ALI-Assembler:

```
x = r + s * (c − d) / (r + s);
```

Aufgabe 5: Übersetze das Programm aus Aufgabe 2 in ein ALI-Assembler-Programm. Zur Wiederholung: das Programm in ReJava sah wie folgt aus:

```
public class Gauss {
  public static void main(String args[]) {
    int eins = 1;
    int zwei = 2;
    int n = read();
    int h1 = n + eins;
    int h2 = h1 * n;
    int h3 = h2 / zwei;
    int s = h3;
    write(s);
  }
}
```

ARBEITSBLATT ZUR ASSEMBLIERUNG

Und nun zum letzten Schritt, dem Maschinencode. ALI codiert die bekannten Befehle wie folgt:

Befehl	Codierung (dezimal)	Bezeichnung
`LDA  R,ADR`	88 0 a b	Laden
`STA  R,ADR`	80 0 a b	Speichern
`INI  ADR`	114 0 a b	Lesen
`OUTI ADR`	115 0 a b	Schreiben
`ADD  R,ADR`	90 0 a b	Addieren
`SUB  R,ADR`	91 0 a b	Subtrahieren
`MUL  R,ADR`	92 0 a b	Multiplizieren
`DIV  R,ADR`	93 0 a b	Dividieren
`EOJ`	10 4	End of Job

Information: Die Platzhalter a und b sind die Werte für die Adresse, auf der gearbeitet wird. b für das niederwertige Byte (Low-Byte) und a für das höherwertige Byte (Hi-Byte). Für die Adresse 634 würde demnach **a = 2** (634 **div** 256) und **b = 122** (634 **mod** 256) gelten.

Aufgabe 6: Übersetze das Programm `Gauss` aus Aufgabe 5 in Maschinencode, d. h. führe eine Assemblierung durch. Lege Dir zur Hilfe ein Adressbuch an, in dem Du Dir die Adressen für die Variablen und Konstanten merkst. Verwende also folgendes Schema:

Adresse	Marke	Befehlscode	Operand
0	Gauss	114 0	n

Adressbuch

Operand	Adresse
n	

Aufgabe 7: Notiere das in Aufgabe 6 assemblierte Programm als Maschinenprogramm. Verwende folgendes Schema:

Adresse	Inhalt (dezimal)	Inhalt (binär)
0	114 0 0 46	01110010 00000000 00000000 00101110

VIERSCHRITT VOM PROGRAMM ZUR MASCHINE AM BEISPIEL *RECHTECK* MIT DEM ALI-MODELLRECHNER

Die Programme in Javanotation:

Das Java-Programm

```
public class Rechteck {
  public static void main(String[] args){
    int a = read();
    int b = (a + 1) * (a + 2);
    write(b);
  }
}
```

Das Programm in reduziertem Java:

```
public class Rechteck {
 public static void main(String[] args){
   int a = read();
   int eins = 1;
   int zwei = 2;
   int h1 = a + eins;
   int h2 = a + zwei
   int h3 = h1 * h2;
   int b = h3;
   write(b);
 }
}
```

Die Programme in ALI-Notation:

Das Assemblerprogramm:

```
Rechteck START    0
         INI      a
         LDA      0,a
         ADD      0,eins
         STA      0,h1
         LDA      0,a
         ADD      0,zwei
         STA      0,h2
         LDA      0,h1
         MUL      0,h2
         STA      0,h3
         LDA      0,h3
         STA      0,b
         OUTI     b
         EOJ
a        DS       F
b        DS       F
h1       DS       F
h2       DS       F
h3       DS       F
eins     DC       '1'
zwei     DC       '2'
         END      Rechteck
```

Der Maschinencode:

Adr	Bef.-Code		Operand	
0	114	0	0	54
4	88	0	0	54
8	90	0	0	64
12	80	0	0	58
16	88	0	0	54
20	90	0	0	66
24	80	0	0	60
28	88	0	0	58
32	92	0	0	60
36	80	0	0	62
40	88	0	0	62
44	80	0	0	56
48	115	0	0	56
52	10	4		
54	0	0		
56	0	0		
58	0	0		
60	0	0		
62	0	0		
64	0	1		
66	0	2		
68	0			

TEST ZUM VIERSCHRITT VOM PROGRAMM ZUR MASCHINE

Aufgabe 1: Überführe das folgende Javaprogramm
- **a)** in ein reduziertes Javaprogramm (Dreiadressformat!!!)
- **b)** in ein ALI-Assemblerprogramm und danach
- **c)** in ein Maschinensprachenprogramm. Benutze die Codierungen der untenstehenden Tabelle und denke an die Führung eines Adressbuches, wie unterhalb der Tabelle skizziert.

```
public class Umfang {
  public static void main(String[] args) {
    int a = read();
    int b = read();
    int c = 2 * (a + b);
    write(c);
  }
}
```

Codierungen der ALI-Assemblerbefehle:

Befehl	Codierung (dezimal)	Bezeichnung
LDA R,ADR	88 0 a b	Laden
STA R,ADR	80 0 a b	Speichern
INI ADR	114 0 a b	Lesen
OUTI ADR	115 0 a b	Schreiben
ADD R,ADR	90 0 a b	Addieren
SUB R,ADR	91 0 a b	Subtrahieren
MUL R,ADR	92 0 a b	Multiplizieren
DIV R,ADR	93 0 a b	Dividieren
EOJ	10 4	End of Job

Information: Die Platzhalter a und b sind die Werte für die Adresse, auf der gearbeitet wird. b für das niederwertige Byte (Low-Byte) und a für das höherwertige Byte (Hi-Byte). Für die Adresse 634 würde demnach **a = 2** (634 **div** 256) und **b = 122** (634 **mod** 256) gelten.

Anfang der Übersetzung in Maschinensprache:

Adresse	Marke	Befehlscode	Operand		Adressbuch Operand	Adressbuch Adresse
0		114 0	a		a	

Diese Tabelle bitte auf einem extra Blatt anfertigen

Ersetzung der Operanden durch Adressen aus Adressbuch

Adresse	Befehl			
0	114	0	0	*xxx*

Diese Tabelle bitte auf einem extra Blatt anfertigen

STEUERBEFEHLE IN ALI – SPRÜNGE

Die von Neumann-Struktur eines Rechners sieht keine Schleifenstrukturen vor. Die einzige Möglichkeit der Verzweigung in einem Programm ist durch Veränderung des Befehlszählers gegeben. Dieser kann unbedingt oder bedingt, geändert werden. Ein bedingter Sprung ist z. B. für die Realisierung einer `if-else`-Anweisung nötig.
ALI kennt die folgenden Steuerbefehle:

Befehlsaufbau	Wirkung	Bezeichnung
`CMP R,ADR`	Vergleiche den Inhalt des Registers mit dem Inhalt der Adresse und speichere das Ergebnis im Vergleichsregister. Im Vergleichsregister stehen Verschlüsselungen von ´= =´ (E = equal), ´<´ (L = low), ´>´ (H = high), ´!=´ (NE = not equal) ´>=´ (NL = not low), ´<=´ (NH = not high)	Vergleich (COMPARE)
`B   ADR`	Unbedingter Sprung **goto** ADR	Verzweigung (BRANCH)
`BE  ADR`	Bedingter Sprung: Gehe nach ADR, wenn im Vergleichsregister ´= =´ steht, ansonsten ist das Programm in der nächsten Programmzeile fortzuführen. Der Befehlszähler wird also auf eine bestimmte Speicherzelle gerichtet.	BRANCH ON EQUAL
`BL  ADR`	Falls ´<´, dann ...	BRANCH ON LOWER
`BH  ADR`	Falls ´>´, dann ...	BRANCH ON HIGHER
`BNE ADR`	Falls ´!=´, dann ...	BRANCH ON NOT EQUAL
`BNL ADR`	Falls ´>=´, dann ...	BRANCH ON NOT LOWER
`BNH ADR`	Falls ´<=´, dann ...	BRANCH ON NOT HIGHER

Zur Übersetzung eines Sprungbefehls von der Zwischensprache ReJava in den ALI-Assembler werden folgende Schablonen benötigt:

goto m;	B m

if (<Bedingung>) **goto** m;
 	
Bedingung: 1) a == b
2) a != b
3) a < b
4) a > b
5) a >= b
6) a <= b
	LDA 0,a
	CMP 0,b
	1) BE m
	2) BNE m
	3) BL m
	4) BH m
	5) BNL m
	6) BNH m

Programm Max – vom Programm zur Maschine

Gegeben ist das folgende Programm, welches von zwei eingegebenen Zahlen die größere Zahl ausgibt:

In Java:	In ReJava:

<pre>public class Max {
  public static void main(…){
    int a = read();
    int b = read();
    int m;
    if (a > b)
       m = a;
    else
       m = b;
    write(m);
  }
}</pre>
	<pre>public class Max {
	  public static void main(…){
	     int a = read();
	     int b = read();
	     int m;
	     if a <= b goto M1;
	     m = a;
	     goto M2;
	 M1: m = b;
	 M2: write(m);
	  }
	}</pre>

Aufgabe 1: Überführe das ReJava-Programm mit Hilfe der Übersetzungsschablone für Sprünge in ein ALI-Assemblerprogramm.

Aufgabe 2: Gib eine allgemeine Übersetzungsschablone für die **`if-else`**-Anweisung an, mit welcher sich eine solche Anweisung von ReJava in ALI-Assembler übersetzen lässt.

`if .. else:`

Java	ReJava	ALI-Assembler:
`if` (Bedingung) Anweisung1 ; **`else`** Anweisung2;	**`if`** (! Bedingung) **`goto`** `M1;` Anweisung1; **`goto`** `M;` `M1:` Anweisung2; `M: ...`	

Aufgabe 3: Gib eine allgemeine Übersetzungsschablone für die **`if`**-Anweisung an, mit welcher sich eine solche Anweisung von Java in ReJava und von ReJava in ALI-Assembler übersetzen lässt.

`if:`

Pascal	ReJava	ALI-Assembler:
`if` (Bedingung) Anweisung;		

Programm Summe – vom Programm zur Maschine

Gegeben ist das folgende Programm, welches alle Zahlen zwischen zwei eingegebenen Zahlen aufsummiert und ausgibt:

<table>
<tr><th>In Java:</th><th>In ReJava:</th></tr>
<tr><td>

```
public class Summe {
  public static void main(…){
    int a = read();
    int b = read();
    int s = 0;
    int i = a;
    while (i <= b) {
       s = s + i;
       i = i + 1;
    }
    write(s);
  }
}
```

</td><td>

```
public class Summe {
  public static void main(…){
    int a = read();
    int b = read();
    int s = 0;
    int i = a;
M1: if (i > b) goto M2;
    s = s + i;
    i = i + 1;
    goto M1;
M2: write(s);
  }
}
```

</td></tr>
</table>

Aufgabe 4: Überführe das ReJava-Programm mit Hilfe der Übersetzungsschablone für Sprünge in ein ALI-Assemblerprogramm.

Aufgabe 5: Gib eine allgemeine Übersetzungsschablone für die **`while`**-Anweisung an, mit welcher sich eine solche Anweisung von ReJava in ALI-Assembler übersetzen lässt.

`while`:

Java	ReJava	ALI-Assembler:

`while` (Bedingung)
Anweisung;
	`M1` **`if`** (! Bedingung) **`goto`** `M;`
	Anweisung;
	`goto` `M1;`
	`M: ...`	

Aufgabe 6: Gib eine Übersetzungsschablone für die relativ allgemeine **`for`**-Anweisung `for (i = a; i <= b; i++) Anweisung;` an, mit welcher sich eine solche Anweisung von Java in ReJava und von ReJava in ALI-Assembler übersetzen lässt. **Hinweis:** Überführe die **`for`**-Schleife zuerst in eine **`while`**-Schleife!

`for`:

Java	ReJava	ALI-Assembler:
`for (i = a; i <= b; i++)` Anweisung;		

ÜBERSETZUNGSSCHABLONEN FÜR KONTROLLSTRUKTUREN:

`A, A1, A2` seien Anweisungen
`M, M1, M2` seien Sprungmarken
mögliche Bedingungen: `a == b, a < b, a > b, a >= b, a <= b, a != b`
negierte (!) Bedingungen: `a != b, a >= b, a <= b, a < b, a > b, a == b`

if:

if (Bedingung)
Anweisung;
	if (! Bedingung) **goto** M;
	Anweisung;
	M: ...
		LDA 0,a
		CMP 0,b
		Bxx M je nach ! Bedingung
		Übersetzung von Anweisung
		M ...

if .. else:

if (Bedingung)
Anweisung1 ;
else
Anweisung2;
	if (! Bedingung) **goto** M1;
	Anweisung1;
	goto M;
	M1: Anweisung2;
	M: ...
		LDA 0,a
		CMP 0,b
		Bxx M1 je nach ! Bedingung
		Übersetzung von Anweisung1
		B M
		M1 Übersetzung von A2
		M ...

while:

while (Bedingung) Anweisung;	M1 **if** (! Bedingung) **goto** M; Anweisung; **goto** M1; M: ...	M1 LDA 0,a CMP 0,b Bxx M je nach ! Bedingung Übersetzung von Anweisung B M1 M ...

for:

for (i = a; i <= b; i++)
Anweisung;
⇔
i = a;
while (i <= b) {
Anweisung;
i = i + 1;
}
	i = a;
	M1 **if** (i > z) **goto** M;
	Anweisung;
	i = i + 1;
	goto M1;
	M: ...
		LDA 0,a
		STA 0,i
		M1 LDA 0,i
		CMP 0,z
		BH M
		Übersetzung von Anweisung
		LDA 0,i
		ADD 0,´1´
		STA 0,i
		B M1
		M ...

&&-Operator

(Bedingung B1 mit Operanden a und b, Bedingung B2 mit Operanden c und d seien zwei Bedingungen):

if (B1 && B2)
Anweisung;
	if (! B1) **goto** M;
	if (! B2) **goto** M;
	Anweisung;
	M: ...
		LDA 0,a
		CMP 0,b
		Bxx M je nach ! Bedingung1
		LDA 0,c
		CMP 0,d
		Bxx M je nach ! Bedingung2
		Übersetzung von Anweisung
		M ...

DER PRIMZAHLTEST – VOM PROGRAMM ZUR MASCHINE

Gegeben ist das folgende Programm, welches eine eingegebene Zahl auf ihre Prim-Eigenschaft testet:

```
 1 public class Primtest {
 2   public static void main(String[] args) {
 3     int zahl = read();
 4     int prim = 0;         { 0 nicht prim, 1 prim }
 5     if (zahl % 2 != 0) {
 6        int test = 3;
 7        prim = 1;
 8        while ((test * test <= zahl) && (prim == 1)) {
 9          if (zahl % test == 0)
10              prim = 0
11          else
12              test = test + 2;
13       }
14     }
15     write(prim);
16   }
17 }
```

Aufgabe 1: Überführe das Programm in die Zwischensprache ReJava. Nimm dir dabei mehrere Schritte vor, indem du:

1. zuerst die Rechenoperation **%** in Zeile 5 mithilfe der zulässigen Operationen +, –, * und / übersetzt. **Hinweis:** a % b = a – (a / b) * b
2. danach die **while**-Schleife in Zeile 8 mithilfe der Übersetzungsschablone auflöst. **Hinweis:** Störe dich zuerst nicht daran, dass die Bedingung in der **while**-Schleife eine logische Verknüpfung aufweist. Diese wird erst aufgelöst, wenn die **while**-Schleife durch **if-goto**-Anweisungen ersetzt wurde.
3. dann erst die logische Verknüpfung **&&** in der jetzt zur **if-goto**-Struktur gewordenen **while**-Schleife mithilfe der Übersetzungsschablonen auflöst und
4. schließlich die **if-else**-Struktur aus den Zeilen 9 – 12 mithilfe der Übersetzungsschablone durch **if-goto**-Anweisungen ersetzt. **Hinweis:** Achte auch hier darauf, dass du die Rechenoperation **%** wie in Zeile 6 mithilfe der zulässigen Operationen übersetzt.

Aufgabe 2: Übersetze nun das Programm in den ALI-Assembler und teste es im Modellrechner.

DIE DATENSTRUKTUR ARRAY

Wir wollen uns im folgenden wie schon bei den Kontrollstrukturen auf den Datentypen integer beschränken. Eine Erweiterung auf andere Datentypen erfolgt analog.

Erweiterung der Zwischensprachen ReJava:

1) Die Deklaration eines Arrays erfolgt wie in Java:
   `int[] feld = new int[anz];`
2) Die Zugriffe auf Arrayelemente (lesend/schreibend) sind in der Form `A[i]` möglich. Dabei ist *A* ein Array und *i* eine Konstante bzw. eine Variable. Somit sind Ausdrücke als Index nicht erlaubt bzw. entsprechend zu vereinfachen (z. B. muss `A[x + 5]` vereinfacht werden zu `h1 = x + 5; ...A[h1]...`).
3) Ein Arrayzugriff gilt als ein Operand. Damit ist dann z. B. eine Zuweisung der Form
   `A[i]= B[j] + C[k]`
   ein erlaubtes Dreiadressformat.

Wir benötigen folgende **ALI-Befehle:**

Befehlsaufbau	Wirkung	Erläuterung
`name DS anzahlF`	Legt `anzahl` Speicherplätze an.	Deklaration „**D**efine **S**pace“
`... ... name(Reg)`	Zum Wert der symbolischen Adresse „`name`“ (Fußpunkt des Arrays) wird der Inhalt des Registers „`Reg`“ addiert (der Offset).	symbolische Adressierung

und die zugehörigen **Übersetzungsschablonen** an dem konkreten Beispiel der Variablendeklaration `int[] feld = new int[6];` :

`int[] feld = new int[6];`	`feld DS 6F`	Die Symbolische Adresse „`Feld`“ ist der Fußpunkt des Arrays.

`x := i + 2;`
 	
`feld[x]:= 90;`
	`LDA 0,i`
	`ADD 0,´2´`
	`STA 0,x`
	`LDA 5,x`
	`MUL 5,´2´`
	`LDA 0,´90´`
	`STA 0,feld(5)`
		Variable i initialisieren.
		
		Berechnung der Distanz vom Fußpunkt (Offset).
		
		indizierte Adresse über Register 5.

DIE DATENSTRUKTUR ARRAY – ÜBUNGSAUFGABEN

Aufgabe 1: Die folgenden Arrayzugriffe sind in der Zwischensprache ReJava nicht zulässig. Forme diese in ein zulässiges Dreiadressformat um.

a) `A[B[i]] := A[B[i+1]]`

b) `A[i*j] := B[2*j]`

Aufgabe 2: Überführe das folgende Javaprogramm zuerst in die Zwischensprache ReJava und schließlich in ALI-Assemblersprache.

```
public class Fuelle {
  public static void main(String[] args) {
     int[] feld = new int[6];
     int i;
     for (i = 0; i < 6; i++)
       feld[i] = 90;
  }
}
```

Aufgabe 3: Erstelle ein Java-Programm, welches von einem Feld mit n Elementen das arithmetische Mittel berechnet und ausgibt. Zuvor sollte der Benutzer die Möglichkeit haben, das Feld mit Werten zu belegen.
Übersetze dieses anschließend in ReJava und schließlich in ALI-Assembler.

Aufgabe 4: Gegeben ist der folgende Sortieralgorithmus:

```
public class Sort {

  private static final int N = 10;
  public static void main(String[] args) {
    int[] feld = new int[N];

    for (int i = 0; i < N; i++)
      feld[i] = read();

    int swaped = 1;
    int i= 0;
    while (swaped == 1) {
      swaped = 0;
      for (int j = 0; j < n-i; j++) {
        if (feld[j] > feld[j+1]) {
           int hilf = feld[j];
           feld[j] = feld[j+1];
           feld[j+1] = hilf;
           swaped = 1;
        }
      }
      i++;
    }
  }
}
```

a) Um welchen Algorithmus handelt es sich hierbei?
b) Übersetze den Algorithmus in ALI-Assemblersprache.

DIE DATENSTRUKTUR STACK (KELLER) IN ALI

VORÜBERLEGUNGEN:

Eine Möglichkeit, die Datenstruktur des Keller in ALI zu übersetzen liegt in der Verwendung eines Arrays zur Speicherung der Kellerelemente (von unten her gefüllt). Zur Erkennung eines leeren bzw. vollen Kellers werden jeweils ein Anzeiger für den Anfang (`k_Anf`) und für das Ende (`k_End`) des Kellers benötigt. Um diese angeben zu können muss festgelegt werden, wie viele Kellerelemente maximal (`k_Laenge`) gespeichert werden können.

EINE MÖGLICHE DELPHI-DEFINITION DES STACKS

```
public class Stack {
   private int kopf;
   private int[] keller = new int[256];
   public Stack() {…}
   public void push(int wert) {…}
   public void pop() {…}
   public int top() {…}
   public boolean empty() {…}
//eventuell noch:
   public boolean full() {…}
}
```

Die Deklaration des Kellers sieht also wie folgt aus (Vereinfachung zum Beispiel):

Zwischensprache	**ALI-Assembler**

`Stack stack = new Stack();`
d. h. `keller: int[256]`
und `kopf: int;`
	`k_anf    DS F`
	`k_end    DS F`
	`k_laenge DC ´512´`
	`keller   DS 256F`

Die Speicherung des Kopf-Zeigers geschieht im folgenden mit dem Register 5, welches wie beim Array für die indizierte Adressierung benötigt wird.

ÜBERSETZUNG DER EINZELNEN ROUTINEN:

constructor Stack:

<u>Frage:</u> Wie muss `k_anf` initialisiert sein, damit die Erkennung eines leeren Kellers möglich wird?

<u>Antwort:</u> Der Kopf steht immer auf dem obersten Kellerelement. Wenn ein Vergleich auf Gleichheit mit `k_anf` Erfolg haben soll, so muss `k_anf` die Adresse annehmen, welche direkt auf den durch `keller` belegten Speicherbereich folgt (hier 1844). Das Ende (markiert durch `k_end`) muss dementsprechend die oberste Speicheradresse des Kellers enthalten (hier 1332). Dies hat zur Folge, dass ein push-Befehl zuerst den „Kopf-Zeiger" auf den nächsten Speicherplatz setzt (Register 5 um eins erniedrigen) und erst dann das Element an der entsprechenden Stelle abgelegt wird.

1326	k_anf = 1844
1328	k_end =1332
1330	k_laenge = 512
1332	keller
1334	
1336	
1338	
1340	
1342	
…	…
1842	
1844	

Zwischensprache	ALI-Assembler

`kopf = 256;`	`LA  5,keller`
	`STA 5,k_end`
	`ADD 5,k_laenge`
	`STA 5,k_anf`

LA heißt **L**oad **A**dress und bedeutet, dass nicht der Wert der Variablen Keller, sondern die Adresse, an der Keller abgelegt wurde, in das Register 5 („Kopf-Zeiger") geladen wird. Da der Kopf-Zeiger immer auf das erste Element im Keller zeigt, muss dieser bei einem leeren Keller auf die Adresse zeigen, welche im direkten Anschluss an den von `keller` belegten Speicherplatz folgt (das Element an Adresse „`keller`" + „`k_laenge`").

boolean empty() (vor pop- und top-Aufruf nötig)

Zwischensprache	ALI-Assembler

`if (kopf == 256) goto leer;`	`      CMP 5,k_anf`
	`      BE  leer`
	`      ...`
	`leer  ...`

boolean full() (vor push-Aufruf nötig)

Zwischensprache	ALI-Assembler

`if (kopf == 0) goto voll;`	`      CMP 5,k_end`
	`      BE voll`
	`      ...`
	`voll  ...`

void push(int wert)

Zwischensprache	ALI-Assembler

`kopf = kopf - 1;`
`keller[kopf] = wert;`
	`      SUB 5,'2'`
	`      CMP 5,k_end`
	`      BE  voll`
	`      LDA 0,wert`
	`      STA 0,0(5)`
	`      ...`
	`voll  ...`

Die indizierte Adressierung geschieht hier ab Adresse 0, da im Register 5 schon die absolute Adresse für das aktuelle Kopfelement steht. Somit gelangt man mit 0+Reg.5 an die gewünschte Adresse.

void pop()

Zwischensprache	ALI-Assembler

`kopf:= kopf + 1;`	`      CMP 5,k_anf`
	`      BE  leer`
	`      ADD 5,'2'`
	`      ...`
	`leer  ...`

int top()

Zwischensprache	ALI-Assembler

`wert = keller[kopf];`	`      CMP 5,k_anf`
	`      BE  leer`
	`      LDA 0,0(5)`
	`      STA 0,wert`
	`      ...`
	`leer  ...`

ÜBUNGSAUFGABEN ZUR DATENSTRUKTUR STACK

Löse mit Hilfe der Überlegungen zur Datenstruktur Stack die folgenden Aufgaben:

Aufgabe 1: Was leistet folgendes Programm?

```
public class WasTueIch {

  private static Stack k;

  public static void main(String[] args) {
    k = new Stack();
    do {
      int wert = read();
      if (wert != 0) k.push(wert);
    } while (wert != 0);

    while ( ! k.empty() ){
      wert = k.top();
      write(wert);
      k.pop();
    }
  }
}
```

Aufgabe 2: Übersetze das obige Programm
- **a)** in die Zwischensprache (ReJava)
- **b)** in ALI
  Teste das Programm mit dem Modellrechner.

Aufgabe 3: Gesucht ist ein Programm, welches eine Zahl n einliest und daraufhin die Zahlen n, n–1, n–2, ..., 2, 1 in einen Stack schreibt. Im Anschluss daran soll das Programm nach und nach den Stack leeren und die in ihm abgelegten Zahlen aufsummieren. Abschließend soll die Summe ausgegeben werden.
- **a)** Gib ein entsprechendes Java-Programm an.
- **b)** Übersetze das Programm in ReJava und anschließend in ALI. Test das Programm.

ARBEITSBLATT ZUR PROZEDURTECHNIK

Beim Aufruf einer Prozedur muss das aufrufende Programm verlassen werden und nach Abarbeitung der Prozedur an der Stelle hinter dem Prozeduraufruf fortgeführt werden. Da der Aufruf der Prozedur an verschiedenen Stellen erfolgen kann, ist auch der Rücksprung aus der Prozedur unterschiedlich. Konkret heißt das, dass vor dem Aufruf der Prozedur die Rücksprungadresse festgehalten werden muss, damit später dorthin zurückgesprungen werden kann. ALI bietet zum Retten der Rücksprungadresse und für den Rücksprung in das aufrufende Programm folgende Befehle:

ALI-Befehl	Wirkung	Erläuterung
`LA    i,weiter`	Die Adresse der Marke „weiter“ wird in Register i abgelegt.	Retten der Rücksprungadresse Load Adress
`B     u_prog`	Sprung zur Marke „u_prog“.	Verzweigung in die Prozedur
`BR    i`	Sprung zur Adresse, die in Register i steht.	Rücksprung aus der Prozedur in das aufrufende Programm

Vereinbarungen zur Anordnung lokaler und globaler Variablen sowie des Programmcodes:

1. Hauptprogramm	`Prog    START 0`
	`        ...`
	`        LA    4,weiter`
	`        B     uProg1`
	`weiter  ...`
	`        EOJ`

2. Konstanten und Variablen des Hauptprogramms wie gewohnt	`        EOJ`
	`var     DS    F`
	`konst   DC    ´17´`
	noch kein END

3. erstes Unterprogramm bzw. erste Prozedur	`uProg1  INI ...`
	`        ...`
	`        BR    4`

4. zugehörige lokale Konstanten und Variablen **nach** dem Rücksprung!	`        BR    4`
	`var1    DS    F`
	noch kein END

5. nächstes Unterprogramm ...	`uProg2  ...`
6. Programmende	`        END   Prog`

Aufgabe 1: Übersetze folgendes Pascal-Programm in ALI-Assembler. Eine Überführung in ReJava ist nicht unbedingt nötig, da in diesem Beispiel kaum Unterschiede zu Java bestehen.

```
public class Sprung_Uebung {
  private int wert, zahl1, zahl2;

  public void warte() {
    int hilf = read();
  }

  public void lies_zwei_Zahlen() {
    zahl1 = read();
    zahl2 = read();
  }

  public void schreibe_Summe() {
    wert = zahl1 + zahl2;
    write(wert);
  }

  public void schreibe_Produkt() {

    wert = zahl1 * zahl2;
    write(wert);
  }

  public void schreibe_Quotient(){
    wert = zahl1 / zahl2;
    write(wert);
  }

  public static void main(...) {
    lies_zwei_Zahlen();
    schreibe_Summe();
    warte();
    schreibe_Produkt();
    warte();
    schreibe_Quotient();
  }
}
```

ARBEITSBLATT ZUR PROZEDURTECHNIK

Aufgabe 1: Was leistet das folgende Java-Programm? Übersetze es anschließend in ALI.

```
public class Rekursionsdemo {
  private int n, m, summe;

  public void berechne_Summe() {
             if (n <= 0) goto Unter2;
             n = n - 1;
             berechne_Summe();
     Unter1: m = m + 1;
             summe = summe + m;
     Unter2:
  }

  public static void main(...) {
             m = 0;
             summe = 0;
             n = read();
             berechne_Summe();
     Haupt1: write(summe);
  }
}
```

Aufgabe 2: Die Ausführung des Programms führt nicht zu dem gewünschten Ergebnis. Erläutere, wie man mit Hilfe eines Stacks zum richtigen Ergebnis gelangt. Implementiere anschließend deine Idee in ALI.

Aufgabe 3: Formuliere eine Art Übersetzungsschablone für Prozeduraufrufe mit Hilfe derer man beliebige Aufrufe von Java über ReJava in ALI übersetzen kann.

ARBEITSBLATT ZUR PROZEDURTECHNIK MIT DYNAMISCHER VERWALTUNG DER RÜCKSPRUNGADRESSE

Übersetzungsschablone für Prozeduraufrufe ohne Parameter mit Hilfe eines dynamisch verwalteten Stacks:

<table>
<tr><th>Pascal</th><th>RePascal</th><th>Assembler</th></tr>
<tr><td><pre>
class Schema {

  void unter() {
     ...
  }

  void main(...){
        ...
        unter();
  Weiter: ...
  }
}
</pre></td><td><pre>
class Schema {

  void main(...){
        Stack s = new Stack();
        ...
        s.push(Adresse von Weiter);
        goto unter;
  Weiter: ...
}

Unter: ...
       Rücksprungadresse := s.top();
       s.pop();
       goto Rücksprungadresse;
</pre></td><td><pre>
        START 0
        LA   5,stack
        STA   5,k_end
        ADD   5,k_laenge
        STA   5,k_anf
        ...
        SUB   5,´2´
        LA   0,Weiter
        STA   0,0(5)
        B   Unter
Weiter ...
        EOJ

Unter   ...
        LDA   4,0(5)
        ADD   5,´2´
        BR   4
</pre></td></tr>
</table>

Aufgabe 1: Übersetze das Programm Aufgabe1 in ein äquivalentes Assemblerprogramm.

Aufgabe 2: Übersetze das Programm vom letzten Arbeitsblatt (Programm Sprunguebung) in ein Assemblerprogramm, jetzt allerdings unter Berücksichtigung der dynamischen Rücksprungtechnik.

Aufgabe 3: Formuliere zum Programm Aufgabe3 mit Rekursionsaufruf ein äquivalentes Assemblerprogramm.

```
public class Aufgabe1 {
  private int a, b;

  private void aaa(){
    do
      a = read();
      if (a != 0)
        b = b + a;
    while (a != 0);
  }

  public static void main(…){
    b = 0;
    aaa();
    write(b);
  }
}
```

```
public class Aufgabe3 {
  private int a, b;

  private void aaa(){
    a = read();
    if (a != 0) {
       b = b + a;
       aaa();
    }
  }

  public static void main(…){
    b = 0;
    aaa();
    write(b);
  }
}
```

ARBEITSBLATT ZU LOKALEN DATENRÄUMEN

Aufgabe 1: Was leistet das folgende Programm? Übersetze das Programm anschließend in ALI und teste dessen Funktionsweise.

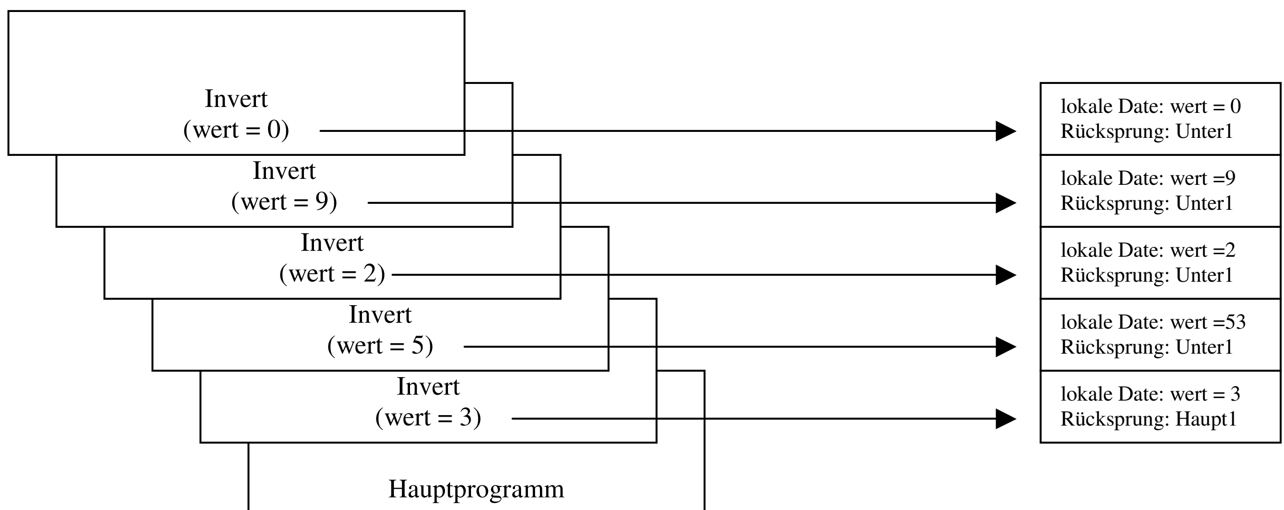
```
public class Zahleninversion {

  public void invert() {
            int wert;
            wert = read();
            if (wert == 0) goto Unter2;
            invert();
    Unter1: write(wert);
    Unter2:
  }

  public static void main(String[] args) {
            invert();
    Haupt1:
  }
}
```

INFORMATIONEN ZUM ARBEITSBLATT ZU LOKALEN DATENRÄUMEN

Durch den rekursiven Aufruf der Prozedur `Invert` wird die lokale Date `wert` jedes Mal mit der neuen Eingabezahl überschrieben. Somit steht nach dem Rekursionsabbruch in der Adresse mit dem Namen `wert` die Zahl 0, welche nun beim rekursiven Rücklauf vier mal ausgegeben wird. Damit das Programm jedoch richtig arbeitet, müsste bei jedem Rekusionsaufruf die **neue** Eingabezahl in einer **neuen** Adresse abgelegt werden.

Das gleiche Problem hatten wir bereits bei der Rettung der Rücksprungadresse. Hier war das gleiche Problem gegeben, da bei jedem Rekursionsaufruf die alte Rücksprungadresse überschrieben wurde. Erst mit Hilfe eines Kellers, in dem nach und nach die Rücksprungadressen „eingekellert" wurden, konnten wir das Problem umgehen.

Die gleichen Überlegungen führen nun auch bei diesem Problem zur Lösung. Bei jedem Aufruf der Prozedur `Invert` muss für die lokale Variable `wert` eine neue Adresse reserviert und belegt werden. Nun könnten wir für diese – und jede zusätzliche – lokale Variable einen neuen Keller anlegen, der uns diese Arbeit abnimmt. Sinnvoller ist es jedoch, wenn die lokale Date wert **und** die Rücksprungadresse in einem gemeinsamen Keller verwaltet werden. Schauen wir uns dazu diese Situation in einer Grafik an:



Man erkennt deutlich, dass der Keller um die lokale Date `wert` erweitert wurde. Da dieser Keller nun nicht mehr nur die Rücksprungadresse, sondern auch lokale Daten während der Programmausführung – also während der Laufzeit – aufnimmt, nennen wir diesen Keller nun **LAUFZEITKELLER**. Diese Erweiterung des Rücksprungkellers zu einem Laufzeitkeller hat nun natürlich Folgen für seine Verwaltung:

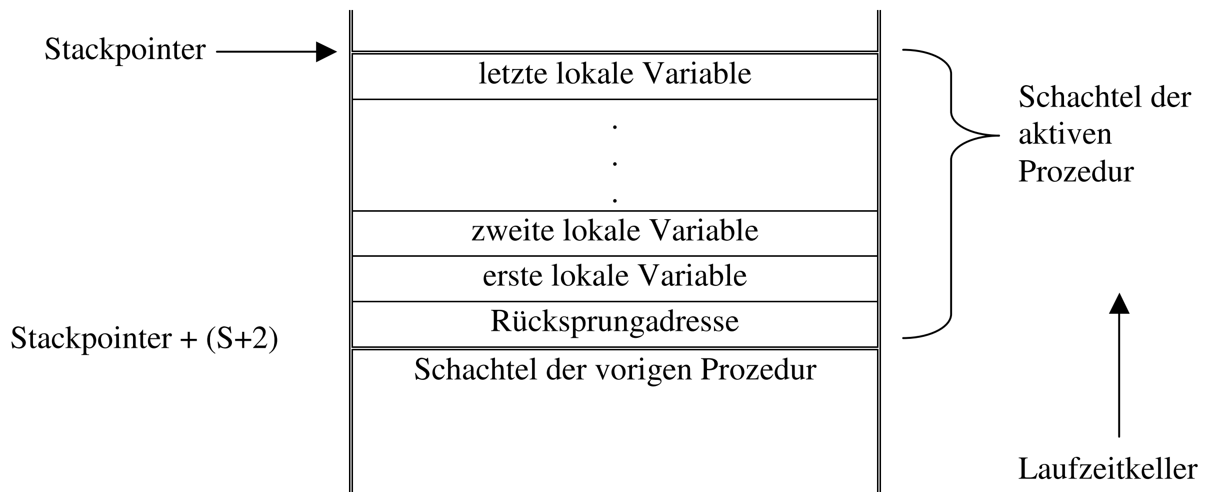
1. Beim Aufruf bzw. Abschluss einer Prozedur muss der Stackpointer nicht mehr um konstant 2 Byxtes verändert werden, sondern zusätzlich noch um den Platz, der für die jeweiligen lokalen Variablen benötigt wird. Diesen Bereich wollen wir im folgenden als „**LOKALEN DATENRAUM**" bezeichnen.

   Somit müssen vor dem **Aufruf** einer Prozedur folgende Schritte durchgeführt werden:
   a) Ermittle den Speicherplatz „S", den die lokalen Variablen der aufzurufenden Prozedur benötigen.
   b) Schaffe im Laufzeitkeller Platz für den neuen lokalen Datenraum und die Rücksprungadresse, d. h. vermindere den Stackpointer um S+2.
   c) Lege die Rücksprungadresse im neugeschaffenen lokalen Datenraum ab.

Beim **Abschluss** einer Prozedur ergibt sich analog:

a) Ermittle die zuletzt abgespeicherte Rücksprungadresse. Lade sie in ein beliebiges Register i.
b) Gib die zuletzt angelegte Schachtel des Laufzeitkellers frei, d. h. erhöhe den Stackpointer um S+2.
c) Springe zur Adresse, die in Register i steht.

2. Bei Prozeduren mit umfangreicheren lokalen Variablen muss festgehalten werden, an welcher Stelle des lokalen Datenraums sich welche Variable befindet. Hierbei ist es ganz nützlich, sich Bilder von der konkreten Speicherbelegung zu machen. Wir werden den lokalen Datenraum immer so aufbauen, dass die lokalen Variablen in der Reihenfolge ihrer Deklaration im Laufzeitkeller abgespeichert werden, d. h. nachdem die Rücksprungadresse in der höchsten Adresse der zugehörigen Schachtel des Laufzeitkellers abgespeichert worden ist, wird darüber die erste deklarierte lokale Variable dieser Prozedur abgespeichert usw.:



3. Da verschiedene Prozeduren i. A. einen verschieden großen lokalen Datenraum benötigen, haben die einzelnen Schachteln des Laufzeitkellers natürlich auch unterschiedliche Größe. Aus diesem Grund kann der Laufzeitkeller auch nicht mehr adäquat mit den bekannten Sprachelementen der Zwischensprache RePascal beschrieben werden. Statt dessen benutzen wir die umgangssprachliche Formulierung (siehe oben auf dieser Seite und unten auf der letzten Seite), um seine Verwaltung zu beschreiben.

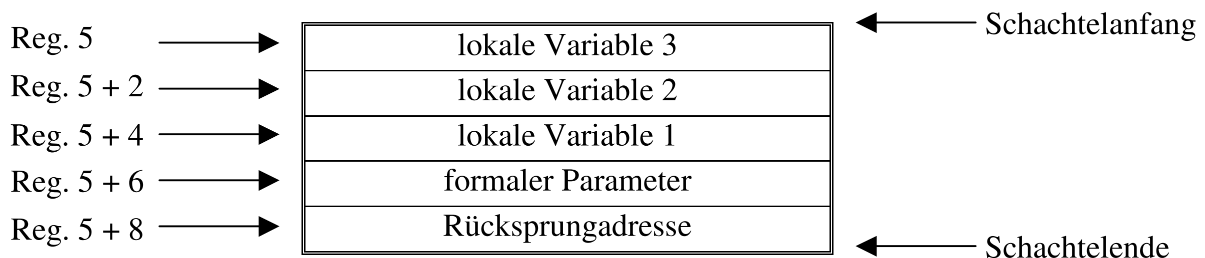
Aufgabe 4: Übersetze nun das Programm Zahleninversion in ein ALI-Programm. Wie gewohnt soll Register 5 den Stackpointer beinhalten.
Der Speicherplatz S für den lokalen Datenraum der Prozedur `Invert` beträgt 2 Bytes, d. h. vor jedem Prozeduraufruf muss der Stackpointer um 4 vermindert werden. Diesen Speicherbereich teilen wir wie oben beschreiben auf.
Diese Aufteilung hat zur Folge, dass der Wert der lokalen Variablen `wert` nun stets an der absolut indizierten Adresse `0(5)` (Register 5 ist Stackpointer), die Rücksprungadresse an der Adresse `2(5)` steht. Verwende also statt des symbolischen Namens `wert` den Ausdruck `0(5)` und zum Laden der Rücksprungadresse den Befehl `LDA 3,2(5) ... BR 3`.

INFORMATIONEN ZUR REALISIERUNG EINER PARAMETERÜBERGABE

Der Datenaustausch zwischen Prozeduren verlangt – wenn man nicht nur auf globale Variablen zurückgreifen möchte – die Möglichkeit einer Parameterübergabe. Zwei verschiedene Arten kennen wir:

1) Paramterübergabe mit formalen Parametern, d. h. ein sogenannter formaler (Wert)parameter wird in die aufgerufene Prozedur übergeben,
2) Parameterübergabe mit Funktionsrückgaben d. h. ein Ergebnis wird in die aufgerufene Prozedur zurückübergeben.

Kümmern wir uns zuerst um erstere Übergabeart. Diese bedeutet, dass die aufgerufene Prozedur einen neuen Speicherplatz reserviert, in den der Wertparameter abgelegt wird. Nach Beendigung der Prozedur wird der entsprechende Speicherplatz wieder freigegeben. Prinzipiell verhält sich also die Variable, in der der Wertparameter abgelegt wird, wie eine lokale Variable. Dementsprechend ist es sinnvoll, die Prozedurschachtel des Laufzeitkellers um einen Speicherplatz pro formalen Parameter zu erweitern. Eine Schachteleinteilung für eine Prozedur mit drei lokalen Variablen und einem formalen Parameter sähe damit wie folgt aus:



Der Aufruf einer Prozedur mit dieser Prozedurschachtel geschieht also mit der Befehlsfolge

```
SUB 5, '10'             Prozedurschachtel hat Größe 10
LA  0, Sprungmarke      Rücksprungadresse laden
STA 0, 8(5)             und im Laufzeitkeller speichern
LDA 0, Wertparameter    Wertparameter laden
STA 0,6(5)              und im Laufzeitkeller speichern.
```

Innerhalb der Prozedur kann man nun – genauso wie man auf lokale Daten zugreift – auf die formalen Parameter zugreifen.

Die Befehlssequenz

```
LDA 0,4(5)
ADD 0,6(5)
STA 0,2(5)
```

bedeutet hier: „lokale Variable 2“ := „lokale Variable 1“ + „formaler Parameter“

ARBEITSBLATT ZU WERTPARAMETERN

Aufgabe 1: Übersetze folgendes ReJava-Programm in ALI-Assembler. Mache Dir zuvor ein Bild der Prozedurschachtel von `Invert`.

```
public class Zahleninversion {
  private int anzahl;

  public void invert(int n) {
              int wert;
              if (n == 0= goto Unter2;
              wert = read();
              n = n - 1;
              invert(n);
    Unter1: write(wert);
    Unter2:
  }

  public static void main(String[] args) {
              anzahl = read();
              invert(anzahl);
    Haupt1:
  }
}
```

Aufgabe 2: Übersetze folgendes Java-Programm in ReJava und schließlich in ALI-Assembler. Mache Dir auch hier zuvor ein Bild der Prozedurschachtel von `Lies_zahlen_und_drucke_summe`.

```
public class Summe {
  private int anzahl;

  private void lies_zahlen_und_drucke_summe(int n) {
    int summe, zahl, index;
    summe = 0;
    index = 0;
    do {
      index = index + 1;
      zahl = read();
      summe = summe + zahl;
    while (index < n);
    write(summe);
  }

  public static void main(String[] args) {
    anzahl = read();
    lies_zahlen_und_drucke_summe(anzahl);
  }
}
```

LÖSUNG ZUM ARBEITSBLATT ZU WERTPARAMETERN – AUFGABE 1

```
ZInv     START    0
         INI      anzahl
         LDA      5,anfang          Laufzeitkeller einrichten (Reg. 5)
         SUB      5,s_laeng         Pruefen, ob geung
         CMP      5,ende            Platz fuer Schachtel
         BL       fehler            vorhanden ist
         LA       1,Haupt1          Ruecksprungadresse in Reg 1 laden
         STA      1,4(5)            und in Schachtel ablegen
         LDA      0,anzahl          formaler Parameter anzahl laden
         STA      0,2(5)            und in Schachtel kopieren
         B        invert            Sprung ins Unterprogramm
Haupt1   EOJ                        Ende des Hauptprogramms

fehler   OUTI     '-1'
         EOJ

anzahl   DS       F                 globale Variable (wird zu form. Param.)
anfang   DC       '4000'            Anfangsadresse des Laufzeitkellers
ende     DC       '1000'            Endadresse des Laufzeitkellers
s_laeng  DC       '6'               (Laenge der Schachtel: 1 lok. Var. +
*                                   1 form. Par. + 1 Ruecksprådr. = 6 Byte)
*
* Prozedur invert
* ------------------------------
* R5 zeigt auf Anfang der Schachtel auf dem Laufzeitkeller
* R4 wird als Hilfsregister benutzt (Rettung des Schachtelanfangs)
*
*               ##########################
* R5      --->  # lokale Variable "wert" #
*               # ---------------------- #
* R5 + 2  --->  # formaler Parameter "n" #
*               # ---------------------- #
* R5 + 4  --->  # Ruecksprungadresse     #
*               ##########################
*
*
invert   LDA      0,2(5)            if n
         CMP      0,'0'              = 0
         BE       Unter2            then goto Unter2
         INI      0(5)              Readln(wert)
         LDR      4,5               aktueller Schachtelanfang in
*                                   Hilfsregister 4 retten (Load Register)
         SUB      5,s_laeng         Pruefung, ob genug Platz
         CMP      5,ende                      fuer Schachtel
         BL       fehler                      vorhanden ist
         LA       1,Unter1          Ruecksprungadr. in Register 1 laden
         STA      1,4(5)            und in Prozedurschachtel ablegen
         LDA      0,2(4)            form. Param. aus akt.(Reg. 4) Schachtel
         SUB      0,'1'             laden und um eins erniedrigen
         STA      0,2(5)            und in die neue Schachtel kopieren
         B        invert            und Rekursionsaufruf
Unter1   OUTI     0(5)              lokale Variable "Wert" ausgeben
Unter2   LDA      1,4(5)            Rueckkehradresse in Register 1 laden
         ADD      5,s_laeng         Stackpointer auf vorherige
*                                   Prozedurschachtel setzen
         BR       1                 Rueckkehr zum Aufrufenden

         END      ZInv
```