Skriptum zur Unterrichtsreihe

MASCHINENNAHE KONZEPTE

– VON DER HOCHSPRACHE ZUR MASCHINENEBENE –

letzte Überarbeitung: 28.02.2002

Inhaltsverzeichnis

Vorbetrachtungen

Eine Programmiersprache wie z. B. DELPHI besitzt einen umfangreichen Befehlsschatz. Du kannst Dir sicher vorstellen, dass die zum Teil sehr komplexen DELPHI-Befehle nicht direkt vom Computer verstanden werden können. Wie sonst sollte auch DELPHI 1.0 aus dem Jahr 1995 auf einem 386-Computer laufen, der 1994 gebaut wurde? Unser Ziel wird es sein, die Hintergründe hinter DELPHI – bzw. das Zwischenstück zwischen DELPHI und Computer – kennen zu lernen. Doch dazu müssen wir zuerst einmal verstehen, wie ein Rechner der heutigen Generation überhaupt funktioniert. Mit einem Rechner der heutigen Generation meine ich nicht nur den Pentium III-Rechner sondern auch alle Vorläufer bis zum Jahr 1946!!! Du hast richtig gehört! Nahezu alle Rechner, die seit dem gebaut wurden, arbeiten alle mit dem gleichen Verarbeitungsprinzip, dem sogenannten von-Neumann-Prinzip. Was hat es nun mit diesem Prinzip auf sich? John v. Neumann war der erste Mensch, der die wesentlichen Vorzüge der „internen Programmspeicherung“ erkannte. Zuvor wurden die Daten, mit denen ein Programm arbeitete, und das Programm an sich in zwei verschiedenen Speichermedien eingespeist. Mit der „internen Programmspeicherung“ – d. h. Daten und Programm wurden in einem Speichermedium festgehalten – war es nun möglich, Ein Programm selbst als Daten aufzufassen, welche manipuliert werden konnten. Das hört sich im ersten Moment eher gefährlich an, doch wie sonst sollte es möglich sein, einen Programmablauf variabel zu gestalten? So ist z. B. eine Verzweigung zur Laufzeit des Programms nur dann möglich, wenn ich zur Laufzeit auch Zugriff auf das Programm selbst habe. Erst durch diese Sichtweise sind Schleifen, Verzweigungen und Sprünge in Programmen überhaupt erst möglich. Mit Hilfe eines variablen Befehlszählers, der festhält, welcher Befehl als nächstes ausgeführt werden soll, ist z. B. ein Sprung über mehrere Befehle hinweg möglich, indem ich diesen Befehlszähler entsprechend erhöhe. Eine Verzweigung wird dadurch möglich, dass ich in Abhängigkeit eines Datenwertes den Befehlszähler auf zwei unterschiedliche Folgebefehle setzen kann.
Das Verarbeitungsprinzip eines von-Neumann-Rechners gliedert sich in die folgenden vier Teilschritte, den sogenannten von-Neumann-Zyklus:

1) Befehl holen (Steuerwerk bringt Befehl in Befehlsregister bzw. Befehlszähler)
2) Befehl dekodieren (Befehlsdekodierer sorgt durch Steuersignale dafür, dass Operanden und Rechenwerk in Verbindung gebracht werden. Dieses erzeugt für Bearbeitung notwendige Steuersignale)
3) Befehl ausführen (Steuersignale werden von anderen Einheiten, die für den entsprechenden Befehl zuständig sind, verarbeitet)
4) Befehlszähler ändern (normalerweise: erhöhe Befehlszähler um die Länge des aktuellen Befehls, jedoch auch Sprung nach beliebige Adresse möglich)

Wie bereits erwähnt. Wir wollen die Schritte, die zwischen der Programmiersprache DELPHI und diesem gerade vorgestellten von-Neumann-Zyklus liegen, nachvollziehen und verstehen. Das heißt insbesondere, eine Möglichkeit zu finden, DELPHI-Programme schrittweise zu „elementarisieren“ also in Maschinensprache zu übersetzen. Genau das tut der sogenannte Compiler von DELPHI.
Nun wird der Compiler die Befehle nicht direkt, also „eins-zu-eins“ in Maschinensprache übersetzen können. Viel mehr sind einige Zwischenschritte nötig, mit der der Übersetzungsvorgang schematisiert wird. Unser erstes Ziel wird es sein, DELPHI-Programme in eine elementarere Programmiersprache RePascal zu übersetzen und das aus gutem Grund:

Die Zwischensprache RePascal

Die geschichtliche Entwicklung der Rechenmaschinen zeigt, dass Ein-, Zwei- bzw. Dreiadressrechner entwickelt wurden. Pascalprogramme verwenden dagegen wesentlich mehr Adressen in einer Anweisung. So benötigt z. B. die Zuweisung

```
x := c * a + 5 * b
```

schon fünf Adressen (die Konstante Zahl 5 ist auch auf einer Adresse abgelegt). Außerdem ist die Anzahl der Operatoren bei Dreiadressrechnern auf einen begrenzt. Im Beispiel sind es bereits 3 Operatoren bei einer Zuweisung. Erstes Ziel ist es also, jedes Pascalprogramm so zu verfeinern, dass alle Anweisungen im Dreiadressformat vorliegen.
Wie in der geschichtlichen Entwicklung der Rechenmaschinen deutlich wird, wird heute das Konzept des von-Neumann-Zyklus verwendet. d. h. insbesondere, dass der Programmablauf über den sogenannten Befehlszähler gesteuert wird. Er gibt an, welcher Befehl als nächstes auszuführen ist. Eine Verzweigung innerhalb des Programms wird somit durch Änderung des Befehlszählers erreicht – der Programmablauf springt sozusagen an eine andere Stelle. Dieser Sprung kann entweder bedingt oder unbedingt geschehen. Zweites Ziel ist es also, jedes DELPHI-Programm [oder besser PASCAL-Programm] so zu verfeinern, dass sämtliche Schleifen und Bedingungsanweisungen durch Sprünge ersetzt werden.
Aus diesen Überlegungen ergibt sich folgender Sprachumfang für den ersten Übersetzungsschritt

Sprachumfang der Minisprache RePascal

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 **`div`** (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	(`Readln, Writeln`)
3.	Unbedingter Sprung zur Marke n	(**`goto`** `n`)
4.	Bedingter Sprung zur Marke n	(**`if`** `Bedingung` **`then goto`** `n`)

Arithmetische Ausdrücke

Mit dem Ziel vor Augen, DELPHI-Zuweisungen auf Maschinenebene im Dreiadressformat zu übersetzen, haben wir im Sprachumfang der Zwischensprache RePascal lediglich Zuweisungen der Form `a := b` *Operator* `c` zugelassen. Da DELPHI bei Zuweisungen allerdings mit wesentlich mehr Operatoren zulässt, müssen wir uns als erstes darum kümmern, jede beliebige Zuweisung in das Dreiadressformat zu überführen.

Pascal → RePascal:

Kümmern wir uns zuerst um das erste Ziel, sämtliche Zuweisungen in das Dreiadressformat umzuwandeln. Betrachten wir dazu das obige Beispiel:

```
x := c * a + 5 * b
```

Eine Zerlegung in das Dreiadressformat ergibt sich durch das bloße Betrachten, wobei wir die Rechenregel Punkt-vor-Strichrechnung berücksichtigen wollen:

```
h1 := c * a;
h2 := 5 * b;
x  := h1 + h2;
```

Dieses bloße Betrachten ist jedoch algorithmisch sehr schwer umzusetzen, genauso wie es bei Sortierverfahren auch schwer umzusetzen ist, durch bloßes Betrachten diejenigen Elemente umzuordnen, die an der falschen Stelle stehen.
Ein mögliches Verfahren für diese Konvertierung ist, mit Hilfe zweier Stacks (Operandenkeller=ODK und Operatorkeller=OTK) die Zuweisung schrittweise in eine Dreiadresszuweisung zu zerlegen. Die obige Zuweisung wird somit zu folgender Zerlegung:

```
h1 := c * a;
h2 := 5 * b;
h3 := h1 + h2;
x  := h3;
```

Schwieriger wird es jedoch, wenn Klammern berücksichtigt werden müssen. Der vollständige Algorithmus für die Zerlegung einer Zuweisung in das Dreiadressformat sieht wie nachfolgend beschrieben aus. Dabei ist zu beachten, dass es sich bei dem Algorithmus lediglich um eine rein verbale Fassung handelt. Die Übersetzung in eine Programmiersprache würde demnach etwas komplexer, allerdings mit unseren zur Verfügung stehenden Methoden durchaus leistbar sein. Wenn Du Lust hast, dann versuche doch mal in einer ruhigen Minute, diesen Algorithmus programmiertechnisch umzusetzen.
Nun aber der Algorithmus:

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

- Richte zwei Keller ein (ODK, OTK)
- Wiederhole folgendes
  - Lies das nächste Zeichen.
  - Falls das Zeichen
    - a) ein Operand ist,
      - kellere ihn im ODK ein.
    - b) ein Operator ist,
      - 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 alle Operatoren aus.

Zur Übung solltest Du die folgenden Zuweisungen mit Hilfe des oben aufgeführten Algorithmus in das Drei-Adress-Format zerlegen.

Aufgabe 1: Zerlege die arithmetische Zuweisung

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

in eine Folge von Dreiadress-Anweisungen in RePascal.

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

```
program Gauss;
var n, s: integer;
begin
  Readln(n);
  s := (n + 1) * n div 2;
  Writeln(s);
end.
```

RePascal →ALI-Assembler:

So, der erste Schritt wäre erst einmal geschafft. Wie in der Vorbetrachtung bereits erwähnt wollen wir uns bei der Übersetzung auf eine abgespeckte Maschinensprache beschränken. Diese abgespeckte Maschinensprache beherrscht allerdings nur das Ein-Adress-Format. Deswegen ist es ratsam, bevor wir direkt auf die Maschinenebene gehen, unsere RePascal-Programme zuerst in eine Assemblersprache zu überführen. Der einzige Unterschied einer Assemblersprache zur Maschinensprache besteht darin, dass statt der Codierung der Befehle

in Nullen und Einsen Pseudonyme für die Befehle verwendet werden. Dadurch ist eine bessere Lesbarkeit der Programme gewährleistet. Feinere Unterschiede zwischen Maschinensprache und Assemblersprache wirst Du im nächsten Kapitel erfahren.
Ein Assembler für unsere abgespeckte Maschinensprache steht uns im Programm ALI zur Verfügung. ALI beherrscht ebenso wie die Maschinensprache nur das Ein-Adress-Format. Dies hört sich allerdings schlimmer an als es ist. Die Übersetzung einer Drei-Adress-Zuweisung in eine Ein-Adress-Zuweisung ist unprobleматisch. Allgemein wird ein Befehl Der Form

```
                                LOAD      ACCU,a
x := a Op b        zu           Operator  ACCU,b
                                STORE     ACCU,x
```

Wie dies mit dem ALI-Assembler geschieht machen wir uns mit einem kleinen Beispiel klar:

```
                                1) L  0,a
h := a + 2                      2) A  0,b
                                3) ST 0,h
```

ALI kennt außer dem +-Operator natürlich auch noch die anderen Operatoren, sowie Eingabe- und Ausgabe-Befehle. Für die erste Betrachtung von kleineren Programmen reicht uns zunächst die folgende Aufstellung der ALI-Befehle:

Befehlsaufbau	Wirkung	Erläuterung
Transportbefehle:		
`L R,ADR`	`R := ADR`	Laden/`LOAD`
`ST R,ADR`	`ADR := R`	Speichern/`STORE`
Ein/Ausgabebefehle:		
`INI ADR`	`Read(ADR)`	Lesen/`IN-Integer`
`OUTI ADR`	`WRITE(ADR)`	Schreiben/`OUT-Integer`
Rechenbefehle:		
`A R,ADR`	`R := R + ADR`	Addieren/`ADD`
`S R,ADR`	`R := R - ADR`	Subtrahieren/`SUBT`
`M R,ADR`	`R := R * ADR`	Multiplizieren/`MULT`
`D R,ADR`	`R := R` **`div`** `ADR`	Dividieren/`DIV`

- `R` ist ein Register im Bereich 0 bis 15 wobei 0 für den `ACCU` 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 ACCU 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 ACCU 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 ACCU 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
`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

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 Programms (incl. Variablen)	Ende
`EOJ`	Ende der Anweisungen	End of Job

Machen wir uns die schrittweise Übersetzung DELPHI → RePascal → ALI an einem Beispiel klar:

Programm in DELPHI	Programm in RePascal	Programm in ALI
`program Beispiel;`	`program Beispiel;`	`Beispiel START 0`
`var a, b, c, x: integer;`	`var a, b, c, x: integer;`	
	`const fuenf = 5;`	
`begin`	`begin`	
`  Readln(a);`	`  Readln(a);`	`INI a`
`  Readln(b);`	`  Readln(b);`	`INI b`
`  Readln(c);`	`  Readln(c);`	`INI c`
		`L 0,c`
`  x := c * a + 5 * b`	`  h1 := c * a;`	`M 0,a`
		`ST 0,h1`
		`L 0,fuenf`
	`  h2 := fuenf * b;`	`M 0,b`
		`ST 0,h2`
	`  h3 := h1 + h2;`	*... usw ...*
	`  x  := h3;`	
`  Writeln(x);`	`  Writeln(x);`	`OUTI x`
`end.`	`end.`	`EOJ`
		`a DS F`
		`b DS F`
		`c DS F`
		`x DS F`
		`fuenf DC ´5´`
		`END Beispiel`

So, jetzt solltest Du in der Lage sein, selber kleinere DELPHI-Programme in ALI zu übersetzen.

Aufgabe 3: Überführe die folgende Anweisungsfolge in das Einadressformat mit Hilfe der Befehle des ALI-Assemblers.

```
h1 := a - b;
h2 := c * h1;
h3 := h1 + h2;
x := h3;
```

Aufgabe 4: Übersetze die folgende Zuweisung zunächst in die Zwischensprache RePascal (Drei-Adress-Format) und dann in ALI-Assembler:

```
x := r + s * (c - d) div (r + s);
```

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

```
program Gauss;
const eins = 1;
      zwei = 2;
var   n, s, h1, h2, h3: integer;
begin
  Readln(n);
  h1 := n + eins;
  h2 := h1 * n;
  h3 := h2 div zwei;
  s  := h3;
  Writeln(s);
end.
```

ALI-Assembler → Maschinensprache:

So, auch dieser Schritt wäre nun geschafft. Der letzte Schritt, die Übersetzung des ALI-Programms in ein Maschinenprogramm ist – wie bereits angedeutet – durch eine nahezu eins-zu-eins-Übersetzung möglich. Du weißt bisher, dass ein Assembler für jeden durch Einsen und Nullen codierten – also als Integer-Zahl codierten – Maschinenbefehl ein Pseudonym zur Verfügung stellt. Wenn wir einmal den Lade-Befehl von ALI (`L 0,ADR`) genauer anschauen, so stellen wir fest, dass zu dem Befehl (`Load`) auch ein Operator (`ADR`) dazugehört. Für einen solchen Befehl benötigt unsere Maschinensprache zwei Bytes zur Speicherung – codiert in der Bytebelegung 88 0 (Die Null steht für das Register 0, dem Accumulator). Beim Operator handelt es sich im Normalfall um eine Adresse. ALI hat einen Speicherbereich mit Adressen von 0 bis 4095, also 2^{12} Speicherplätzen. Um diese Adressen ansteuern zu können benötigt man eineinhalb Bytes, also besser zwei Bytes. Somit werden für den Lade-Befehl in ALI insgesamt vier Bytes in Maschinensprache benötigt. Die gesamte Codierung aller bisher bekannten Befehle ist in der folgenden Tabelle aufgeführt:

Befehl	Codierung (dezimal)	Bezeichnung
`L R,ADR`	88 0 a b	Laden
`ST R,ADR`	80 0 a b	Speichern
`INI ADR`	114 0 a b	Lesen
`OUTI ADR`	115 0 a b	Schreiben
`A R,ADR`	90 0 a b	Addieren
`S R,ADR`	91 0 a b	Subtrahieren
`M R,ADR`	92 0 a b	Multiplizieren
`D 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.

Bei der Übersetzung eines ALI-Programms in Maschinensprache ist die größte Schwierigkeit, allen Variablen deren zugehörige Speicherplätze zuzuordnen. Da die Variablen am Ende des Programms definiert werden, stehen sie auch in den Adressen hinter dem Programm. Das Problem ist, dass wir beim ersten Auftreten einer Variablen noch gar nicht wissen, an welcher Adresse die Variable tatsächlich liegt.
Die Lösung ist einfach. Während der ersten Übersetzung legt man sich ein Adressbuch an, in dem die Zuordnung *Variablenname* → *Adresse* gespeichert werden soll und trägt darin immer dann einen Variablennamen ein, wenn dieser im Programm benötigt wird. Am Ende, wenn die Variablen definiert werden, füllt man das Notizbuch mit den zugehörigen Adressen.
Im zweiten Schritt trägt man dann überall wo noch Variablennamen als Operanden vorkommen die zugehörigen Adressen ein.

Wir machen uns das Prinzip am besten an einem kleinen Beispiel klar. Das ALI-Programm lautet wie folgt:

```
Beispiel START 0
         INI   a
         L     0,a
         M     0,fuenf
         ST    0,b
         OUTI  b
         EOJ
a        DS    F
b        DS    F
fuenf    DC    ´5´
         END   Beispiel
```

Im ersten Schritt übersetzen wir das Programm zu

					Adressbuch	
Adresse	Marke	Befehlscode		Operand(en)	Opd./Marke	Adresse
0	Beispiel	114	0	a	Beispiel	*0*
4		88	0	a	a	*22*
8		92	0	fuenf	fuenf	*26*
12		80	0	b	b	*24*
16		115	0	b		
20		10	4			
22	a	0	0			
24	b	0	0			
26	fuenf	0	5	*{ Konstante Integerzahl ´5´ }*		

Während wir die Variablendeklarationen (ab Adresse 22) bearbeiten füllen wir unser Adressbuch mit den zugehörigen Adressen.

Im zweiten Schritt ersetzen wir dann die Variablennamen durch ihre zugehörigen Adressen. Das fertige Maschinensprache-Programm sieht wie folgt aus:

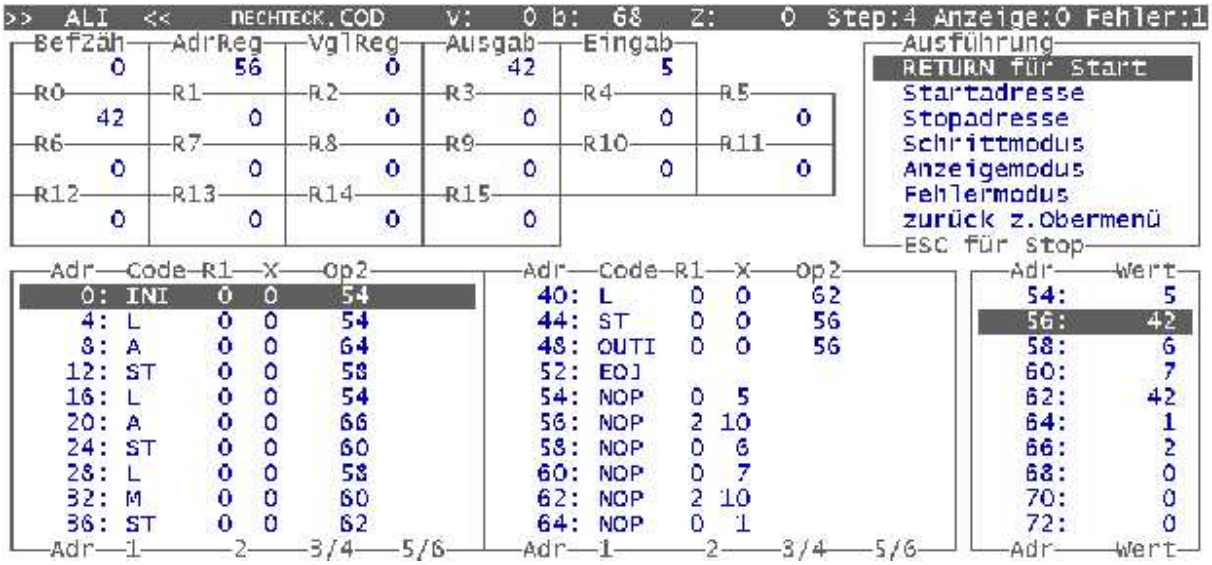
Adresse	Befehl			
0	114	0	0	22
4	88	0	0	22
8	32	0	0	26
12	80	0	0	24
16	115	0	0	24
20	10	4	0	0
24	0	0	0	5

Eigentlich solltest Du jetzt in der Lage sein, selbst ein Programm komplett in Maschinensprache zu übersetzen. Solltest Du damit jedoch noch Probleme haben, so findest Du im Anhang (Seite 44) ein weiteres Beispiel, in dem alle drei Übersetzungsschritte an einem kleine Programm durchgeführt werden.

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 das bekannte Schema.

Der ALI-Modellrechner:

Bevor ich Dir den Modellrechner erläutern möchte, zeige ich Dir ein Bild seines Aufbaus:



Du siehst das Kernstück von ALI – den Modellrechner. „Modellrechner“ deswegen, da er die Funktionsweise des Computers simuliert. Im unteren Teil findest Du das Programm aus dem Anhang in Maschinensprache codiert. im kleineren Fenster unten rechts ist der Speicherbereich abgebildet, in dem die Variablendeklarationen vorgenommen wurden.

In der obersten Zeile sind die wichtigsten Register des Modellrechners:

1. Der Befehlszähler: Hier wird festgehalten, welcher Befehl als nächstes ausgeführt wird (siehe auch von-Neumann-Zyklus)
2. Adressregister: In diesem Register wird festgehalten, welche Operandenadresse aktuell ist.

3. Vergleichsregister: Dazu kommen wir später
4. Ausgabe: Alle Ausgaben (Integer-Zahlen) werden dort angezeigt
5. Eingabe: Alle Eingaben (ebenfalls Integer-Zahlen) werden dort getätigt.

In der zweiten bis vierten Zeile sind Register, die Du für Deine Rechnungen benutzen kannst. Bisher haben wir lediglich das Register 0 (Accumulator) benutzt.

Bevor Es weitergeht solltest Du einige Deiner zuvor erstellten Programme mit ALI testen. Achte bei der Eingabe Deiner Programme im Editor von ALI darauf, dass Du alle Marken in einer Spalte, alle ALI-Befehle in einer Spalte und alle Kommentare in einer Spalte schreibst.

Zusammengesetzte Anweisungen (Kontrollstrukturen)

Nach den arithmetischen Ausdrücken war das zweite Ziel, komplexere Programme mit Schleifen und bedingten Verzweigungen auf eine maschinennahe Gestalt zu bringen. Erinnern wir uns: Die von-Neumann-Struktur erlaubt lediglich bedingte und unbedingte Sprünge. Eine komplexe Struktur wie Schleifen mit Abbruchbedingung oder **`if`**–**`then`**–**`else`**-Verzweigungen existieren nicht. Ziel ist es also, genauso wie bei den arithmetischen Ausdrücken eine Art Übersetzungsschablone herzustellen, die beliebige Kontrollstrukturen in die von-Neumann-Struktur eingliedert.

Bedingte Verzweigungen:

Wenden wir uns zuerst der letzteren **`if`**–**`then`**–**`else`**-Struktur zu:

Pascal → RePascal

Schauen wir uns zuerst ein Beispiel an.

```
program Max;
var a, b, m: integer;
begin
  Readln(a);
  Readln(b);
  if a > b
    then m := a
    else m := b;
  Writeln(m);
end.
```

Die Zwischensprache RePascal erlaubte uns lediglich bedingte Sprünge, nicht jedoch bedingte Anweisungen. Unser Ziel ist es also, die Anweisungen je nach Bedingung „anzuspringen“. Konkret heißt das, ist die Bedingung `a > b` erfüllt, so springe zur Anweisung `m := a;`. Wenn die Bedingung nicht erfüllt ist, so soll das Programm einfach fortgesetzt werden. Allerdings muss man aufpassen, dass man nach der Anweisung `m := b;` die Anweisungen für den **`then`**-Teil überspringt.
Pascal bietet tatsächlich die Methode des bedingten Sprungs. Dafür muss man lediglich vor die Anweisungen, die angesprungen werden sollen, ein Label (Sprungmarke) gefolgt von einem Doppelpunkt setzen. Vor der Variablendeklaration definiert man alle Labels, die im Programmtext vorkommen.
Versuche mit diesen Informationen, das Programm selbständig in RePascal zu übersetzen. Wenn es Dir nicht gelingt, dann kannst Du auf der nächsten Seite die Lösung nachschauen.

```
program Max;
label M1, M;
var a, b, m: integer;
begin
     Readln(a);
     Readln(b);
     if a > b then goto M1;
     m := b;
     goto M;
 M1: m := a;
 M:  Writeln(m);
end.
```

Aufgabe 7: Übersetze das folgende Programm in RePascal:

```
program If_Then_Else;
var x: integer;
begin
  Readln(x);
  if x <= 5
    then Writeln(5)
    else Writeln(x);
end.
```

Du hast bei der letzten Aufgabe wahrscheinlich festgestellt, dass das Prinzip dieser Übersetzung nahezu identisch zur vorherigen Übersetzung ist. Man kann dieses Übersetzungsprinzip – oder besser Schema – allgemeingültig formulieren.

if .. then .. else:

Pascal	RePascal
if Bedingung **then** A1 **else** A2;	**if** Bedingung **then goto** M1; A2; **goto** M; M1: A1; M: ...

Aufgabe 8: Versuche an folgendem Beispiel ein Übersetzungsschema für die einfache **if**-**then**-Anweisung herauszufinden. Versuche dabei, möglichst wenig Sprungmarken zu verwenden. Dazu dürfen auch die negierten Bedingungen abgefragt werden.

```
program If_Then;
var x: integer;
begin
     Readln(x);
     if x <= 5
       then Writeln(-1);
end.
```

Welche Lösung hast Du herausbekommen? Diese hier?

Pascal	RePascal
if Bedingung **then** A;	**if** Bedingung **then goto** M1; **goto** M; M1: A; M: ...

Diese Lösung ist durchaus korrekt, allerdings ein wenig kompliziert. Es gibt eine Lösung, die mit nur einer Sprungmarke auskommt. Dazu ist es allerdings erforderlich, nicht die Bedingung wie vorliegend zu testen, sondern den bedingten Sprung aufgrund der negierten Bedingung (Bedingung*) durchzuführen. Diese negierte Bedingung ergibt sich durch die folgende Übersetzungsschablone:

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

Somit wird das Übersetzungsschema für die **if**–**then**-Verzweigung vereinfacht zu:

Pascal	RePascal
if Bedingung **then** A;	**if** Bedingung* **then goto** M; A; M: ...

Der erste Schritt in Richtung Maschinensprache wäre geschafft. Jetzt müssen wir uns nur noch um die Übersetzung der RePascal-Befehle in ALI-Assembler-Befehle kümmern

RePascal → ALI-Assembler:

ALI bietet uns die folgenden Vergleichs- und Sprungbefehle (sowohl unbedingt als auch bedingt):

Befehlsaufbau	Wirkung	Bezeichnung
C 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

Eine Übersetzungsschablone für einen bedingten Sprung sieht mit diesen Befehlen wie folgt aus:

RePascal	ALI-Assembler
goto m;	B m
if Bedingung **then goto** m;	L 0,a C 0,b
Bedingung: 1) a = b	1) BE m
2) a <> b	2) BNE m
3) a < b	3) BL m
4) a > b	4) BH m
5) a >= b	5) BNL m
6) a <= b	6) BNH m

Aufgabe 9: Übersetze mit dieser Übersetzungschablone das Programm Max von Seite 12.

```
program Max;
label M1, M;
var a, b, m: integer;
begin
     Readln(a);
     Readln(b);
     if a > b then goto M1;
     m := b;
     goto M;
 M1: m := a;
 M:  Writeln(m);
end.
```

Lösung:

```
Max    START 0
       INI   a
       INI   b
```

Hast Du auch die folgende Lösung herausbekommen?

RePascal (siehe Seite 12)	ALI-Assembler

```
program Max;                    Max     START 0
label M1, M;
var a, b, m: integer;
begin
     Readln(a);                         INI   a
     Readln(b);                         INI   b
                                        L     0,a       1. Vergl.Op.
     if a > b then goto M1;             C     0,b       2. Vergl.Op.
                                        BL    M1        Bed. Sprung
                                        L     0,b
     m := b;                            ST    0,m
     goto M;                            B     M
 M1: m := a;                    M1      L     0,b
                                        ST    0,m
 M:  Writeln(m);                M       OUTI  m
end.                                    EOJ
                                a       DS    F
                                b       DS    F
                                m       DS    F
                                        END
```

Wenn ja, dann freut mich das, wenn nein, dann mache Dir bitte noch einmal die gesamte Übersetzung einer **if**–**then**–**else**-Struktur an der zugehörigen Übersetzungsschablone klar:

if .. then .. else:

Pascal	RePascal	ALI
if Bedingung **then** A1 **else** A2;	**if** Bedingung **then goto** M1; A2; **goto** M; M1: A1; M: ...	L 0,a C 0,b Bxx M1 je nach Bedingung Übersetzung von A2 B M M1 Übersetzung von A1 M ...

Ähnlich der letzten Übersetzungsschablone wird die Schablone für die **if**–**then**-Struktur erstellt:

if .. then:

Pascal	RePascal	ALI
if Bedingung **then** A;	**if** Bedingung* **then goto** M; A; M: ...	L 0,a C 0,b Bxx M je nach Bedingung* Übersetzung von A M ...

Schleifenstrukturen:

Mit den bekannten Möglichkeiten ist prinzipiell jeder beliebige Programmablauf möglich. Dennoch verwendet man in DELPHI selten den bedingten Sprung, um einen Befehlsblock eventuell mehrmals ausführen zu lassen. Man verwendet viel eher eine Schleife. Insgesamt hast Du die **for**-, die **while**- und die **repeat**-Schleife kennengelernt. Kümmern wir uns als erstes um die **while**-Schleife, da diese die am häufigsten verwendete ist und schauen uns dazu das folgende Beispiel an:

```
program Summe;
var a, b, s, i: integer;
begin
  Readln(a);
  Readln(b);
  s := 0;
  i := a;
  while i <= b
    do begin
         s := s + i;
         i := i + 1;
       end;
  Writeln(s);
end.
```

Pascal → RePascal

ALI kennt lediglich den bedingten und unbedingten Sprung, um das lineare Programmverhalten zu durchbrechen. Wollen wir also das Programm in ALI-Assembler übersetzen, so müssen wir zuerst die Schleifenstrukturen in bedingte Sprünge auflösen. Im konkreten Fall der **while**-Schleife bedeutet dies, zwei Sprünge auszuführen. Den ersten bedingten Sprung hinter den Schleifenrumpf führt man dann aus, wenn die Schleifenbedingung nicht mehr erfüllt ist, den zweiten unbedingten Sprung zurück zum Schleifenanfang führt man am Ende des Schleifenrumpfes aus. Schauen wir uns dies am obigen Beispiel an:

```
program Summe;
label M1, M;
var a, b, s, i: integer;
begin
     Readln(a);
     Readln(b);
     s := 0;
     i := a;
 M1: if i > b then goto M
     s := s + i;
     i := i + 1;
     goto M1;
 M:  Writeln(s);
end.
```

Genauso, wie wir für die Verzweigungsbefehle in DELPHI Übersetzungsschemata angelegt haben, wollen wir dies auch für Schleifenstrukturen machen. Die Übersetzungsschablone für die **while**-Schleife sieht damit wie folgt aus:

while .. do:

Pascal	RePascal	ALI
while Bedingung **do** A;	M1 **if** Bedingung* **then goto** M; A; **goto** M1; M: ...	M1 L 0,a C 0,b Bxx M je nach Bedingung* Übersetzung von A B M1 M ...

Aufgabe 10: Übersetze das folgende Programm in RePascal, d. h. eliminiere die **for**-Schleife und ersetze sei durch bedingte Sprünge. Gehe dabei wie folgt vor:
(1) Ersetze die **for**-Schleife durch eine geeignete **while**-.Schleife,
(2) verwende die Übersetzungsschablone für die **while**-Schleife.
Erstelle anschließend eine Übersetzungsschablone für die **for**-Schleife.

```
program Summe2;
var a, b, s, i: integer;
begin
  Readln(a);
  Readln(b);
  s := 0;
  for i := a to b
    do begin
         s := s + i;
       end;
  Writeln(s);
end.
```

Und? Zu welchem Ergebnis bist Du gekommen? Also, die **for**-Schleife lässt sich durch folgende Befehlskombination ersetzen:

```
for i := a to b
  do begin
       < Anweisungen >

     end;
```

```
i := a
while i < b
  do begin
       < Anweisungen >
       i := i + 1;
     end;
```

Die Übersetzungsschablone lautet damit wie folgt:

for .. to:

Pascal	RePascal	ALI
for i := a **to** z **do** A; ⇔ i := a; **while** i <= z **do begin** A; i := i+1; **end;**	i := a; M1 **if** i > z **then goto** M; A; i := i+1; **goto** M1; M: ...	L 0,a ST 0,i M1 L 0,i C 0,z BH M Übersetzung von A L 0,i A 0,´1´ ST 0,i B M1 M ...

Boolsche Verknüpfungen:

Sowohl bei den Schleifen als auch bei den Verzweigungen kann es vorkommen, dass Bedingungen mit **or** oder **and** verknüpft werden müssen. Exemplarisch wollen wir die Übersetzungsschablone für die **and**-Verknüpfung entwickeln.
Eine Verzweigung wie die folgende

```
if (a > b) and (a > c)
  then m := a
  else if (b > c)
         then m := b
         else m := c;
```

liefert das Maximum dreier Zahlen in der Variablen m. Bei der ersten Verzweigung müssen beide Bedingungen `(a > b)` und `(a > c)` erfüllt sein. Ist eine der beiden Bedingungen nicht erfüllt, so kann der **then**-Teil übersprungen werden. Am einfachsten ist es deshalb, nacheinander beide negierten Bedingungen zu testen – ist eine der negierten Bedingungen erfüllt, so überspringe die folgenden Anweisungen.
Diese Überlegung führt zu folgender Übersetzungsschablone:

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

Pascal	RePascal	ALI
`if B1 and B2` `  then A;`	`  if B1* then goto M;` `  if B2* then goto M;` `  A;` `M:  ...`	`  L  0,a` `  C  0,b` `  Bxx  M  je nach B1*` `  L  0,c` `  C  0,d` `  Bxx  M  je nach B2*` `  Übersetzung von A` `M  ...`

Aufgabe 11: Schreibe ein Programm, welches drei Zahlen vom Benutzer einliest und die größte dieser Zahlen ausgibt. Übersetze dein Programm in ALI-Assembler.

So, nun zum Schluss noch eine etwas komplexere Aufgabe. Gegeben ist das folgende Programm, welches eine eingegebene Zahl auf ihre Prim-Eigenschaft testet:

```
 1 program Primtest;
 2 var zahl, test, prim: integer;
 3 begin
 4   Readln(zahl);
 5   prim := 0;          { 0 nicht prim, 1 prim }
 6   if zahl mod 2 <> 0
 7     then begin
 8            test := 3;
 9            prim := 1;
10            while (test * test <= zahl) and (prim = 1)
11            do
12               if zahl mod test = 0
13                 then prim := 0
14                 else test := test + 2;
15          end;
16   Writeln(prim);
17 end.
```

Aufgabe 12: a) Überführe das Programm in die Zwischensprache RePascal. Nehme Dir dabei mehrere Schritte vor, indem Du:

1. zuerst die Rechenoperation **mod** in Zeile 6 mit Hilfe der zulässigen Operationen `+, −, *` und **div** übersetzt.
   Hinweis: `a` **mod** `b = a − (a` **div** `b) * b`
2. danach die **while**-Schleife in Zeile 10 mit Hilfe 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**–**then**–**goto**-Anweisungen ersetzt wurde.
3. dann erst die logische Verknüpfung **and** in der jetzt zur **if**–**then**–**goto**-Struktur gewordenen **while**-Schleife mit Hilfe der Übersetzungsschablonen auflöst und
4. schließlich die **if**–**then**–**else**-Struktur aus den Zeilen 12 – 14 mit Hilfe der Übersetzungsschablone durch **if**–**then**–**goto**-Anweisungen ersetzt.
   Hinweis: Achte auch hier darauf, dass Du die Rechenoperation **mod** wie in Zeile 6 mit Hilfe der zulässigen Operationen übersetzt.

a) Übersetze nun das Programm in den ALI-Assembler und teste es am Modellrechner.

Versuche die Aufgabe erst einmal alleine und schaue dir dann erst die Lösung an.

Zuerst muss geklärt werden, wie die Operation **mod** mittels der zulässigen Operationen `+, −, *` und **div** ersetzt werden kann. Die Zeile

```
6    if zahl mod 2 <> 0
       then Anweisungen
```

wird aufgrund der Beziehung `a` **mod** `b = a − (a` **div** `b) * b` zu:

```
     h1 := zahl div 2;
     h1 := h1 * 2;
     h1 := zahl - h1;
     if h1 = 0 then goto M1;
     Anweisungen
M1:  ...
```

Schließlich muss noch die **while**-Struktur aufgelöst werden. Die Zeilen

```
10   while (test * test <= zahl) and (prim = 1)
11   do Anweisungen
```

kann man mit Hilfe der Übersetzungsschablonen übersetzen zu:

```
M2: if not((test * test <= zahl) and (prim = 1))
    then goto M3;
    Anweisungen
    goto M2;
M3: ...
```

Nun zur Verzweigung mit boolscher Verknüpfung von zwei Bedingungen. Mit Hilfe der De Morganschen Regeln ergibt sich aus

```
M2: if not((test * test <= zahl) and (prim = 1))
```

die Anweisung:

```
M2: if (test * test > zahl) or (prim <> 1)
```

und somit mit Hilfe der Übersetzungsschablonen:

```
M2: h2 := test * test;
    if h2 > zahl then goto M3;
    if prim <> 1 then goto M3;
    Anweisungen
    goto M2;
M3: ...
```

Schließlich noch die **if**–**then**–**else**-Verzweigung in Zeile 12 – 13 mit Hilfe der in Schritt 1 entwickelten Ersetzung des Befehls **mod**. Die Zeilen

```
12  if zahl mod test = 0
13    then prim := 0
14    else test := test + 2;
```

werden übersetzt zu

```
    h3 := zahl div test;
    h3 := h3 * test;
    h3 := zahl - h3;
    if h3 = 0 then goto M4;
    test := test + 2;
    goto M5;
M4: prim := 0;
M5: ...
```

Während der Übersetzung fallen die Marken M1 und M3 übereinander. Somit wird eine der beiden Marken (hier Marke M1) überflüssig und kann eingespart werden.

Insgesamt ergibt sich folgendes Programm:

```
 1    program Primtest;
 2    label M1, M2, M4, M5;
 3    var zahl, test, prim: integer;
 4        h1, h2, h3: integer;
 5    begin
 6       Readln(zahl);
 7       prim := 0;          { 0 nicht prim, 1 prim }
 8       h1 := zahl div 2;
 9       h1 := h1 * 2;
10       h1 := zahl - h1;
11       if h1 = 0 then goto M1;
12       test := 3;
13       prim := 1;
14   M2: h2 := test * test;
15       if h2 > zahl then goto M1;
16       if prim <> 1 then goto M1;
17       h3 := zahl div test;
18       h3 := h3 * test;
19       h3 := zahl - h3;
20       if h3 = 0 then goto M4;
21       test := test + 2;
22       goto M5;
23   M4: prim := 0;
24   M5: goto M2;
25   M1: Writeln(prim);
26  end.
```

Datenstrukturen

In diesem Kapitel wollen wir uns mit verschiedenen Datenstrukturen beschäftigen. Da wir grundsätzlich nur den Datentypen `integer` zulassen wollen, läuft die gesamte Betrachtung von Datenstrukturen im wesentlichen auf die Verwaltung eines Speicherbereichs fester Größe hinaus. Dieser Speicherbereich kann als eine Art Array aufgefasst werden, weshalb wir uns zuerst mit dieser Datenstruktur beschäftigen wollen.

Die Datenstruktur ***Array***

Pascal bietet die Möglichkeit einer Reihung (**`array`**) von Elementen. Eine solche Reihung kann in Pascal mit beliebigen Datentypen durchgeführt werden – wir hingegen wollen uns auf die Reihung von Integertypen beschränken. Das Prinzip ist in jeder Hinsicht übertragbar.

Die Zwischensprache RePascal muss, wenn eine solche Datenstruktur erlaubt sein soll, wie folgt erweitert werden:

1) Die Deklaration eines Arrays erfolgt wie in Pascal:
   `var` `Feld:` **`array`**`[u..o]` **`of`** `integer`
2) Der Indextyp `u..o` ist auf den Typ integer beschränkt. Außerdem muss gelten: u ≤ o.
3) 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 (keine Zahl!!!). Somit sind Ausdrücke (wie z. B. i+5) als Index nicht erlaubt bzw. entsprechend zu vereinfachen.
4) Ein Arrayzugriff gilt als 1 Operand. Damit ist dann z. B. eine Zuweisung der Form
   `A[i]:= B[j] + C[k]`
   ein erlaubtes Dreiadressformat.

Die Frage nach der internen Speicherabbildung liegt durch den Begriff Reihung auf der Hand. Ein Array belegt einen zusammenhängenden Block im Speicher, so dass alle Elemente darin Platz finden. Die Anzahl der Elemente in einem solchen Array lässt sich dabei aus der Formel `A = o – u + 1` berechnen.

Wie adressiert man nun ein Element der Datenstruktur

```
Feld: array[-2..3] of integer?
```

Greife ich auf Feld[1] zu, so meine ich damit nicht das erste Element im reservierten Block, sondern das dritte Element (mit 0 beginnend gezählt). Um eine genaue Adressierung möglich zu machen, ist es nötig, den Array genauer zu spezifizieren:

Der **Arraydeskriptor** legt folgende Rahmenbedingungen eines Arrays fest:

F	Fußpunkt, d. h. erstes Byte der Struktur	Eine Adresse, z. B. 1382
u	kleinster erlaubter Index	im obigen Beispiel –2
o	größter erlaubter Index	im obigen Beispiel 3
A	Anzahl der Feldelemente A = o-u+1	im obigen Beispiel 6
L	Komponentenlänge in Bytes	i. d. R.. 2 für Datentyp Integer

Mit diesem Deskriptor ist nun eine Adressberechnung möglich. Soll z. B. auf `Feld[1]` zugegriffen werden, so berechnet man zuerst die logische Stelle des Elements:

```
Stelle := Index – u,
```

d. h. im konkreten Beispiel: `Stelle := 1 – (–2) = 3`.

Mit der Stelle kennen wir die Position, wobei `Stelle = 0` das erste Element der Reihung meint. Da wir es im Normalfall mit unterschiedlichen Datentypen zu tun haben berechnet sich der Offset, d. h. die relative physikalische Adresse, aus der Stelle in der Reihung und der Komponentenlänge:

```
Offset := Stelle*Länge,
```

d. h. im konkreten Beispiel eines Integer-Arrays: `Offset := 3*2 = 6`.

Schließlich muss nun noch der Fußpunkt hinzuaddiert werden, um die tatsächliche physikalische Adresse zu erhalten. Im konkreten Beispiel heißt das:

```
Adresse := 1382+6 = 1388.
```

Die benötigten Syntaxelemente von ALI lauten wie folgt:

Befehlsaufbau	Wirkung	Erläuterung
`name DS anzahlF`	Legt anzahl Speicherplätze an.	Deklaration „Define Space“
`name(Reg)`	Zum Wert der symblosichen Adresse „name“ wird der Inhalt des Registers „Reg“ addiert. Bei einem Feld ist dies der Fußpunkt.	symbolische Adressierung

Mit Hilfe von Übersetzungsschablonen lassen sich nun komplexe Programme mit Arrays in ALI-Code übersetzen. Grundsätzlich sollte zur Indizierung eines Arrays ein festes Register (z. B. das Register 5) verwendet werden. Die Übersetzungsschablonen sehen dabei wie folgt aus:

Pascal	ALI	Erläuterung
`const u = -2;` `o = 3;` `var feld: array[u..o]` `of integer`	`u DC ´-2´` `o DC ´3´` `feld DS 6F`	Deklaration der Feldgrenzen weil o-u+1=6. Festlegung von 6 Speicherstellen. Damit ist die Symbolische Adresse „feld“ der Fußpunkt.
`i := 2;` `Feld[i]:= 90;`	`L 0,´2´` `ST 0,i` `L 5,i` `S 5,u` `M 5,´2´` `L 0,´90´` `ST 0,Feld(5)`	Variable i initialisieren. Berechnung der Distanz vom Fußpunkt (Offset). indizierte Adresse über Register 5.

Aufgabe 13: Gebe zur folgenden Variablendeklaration den zugehörigen Arraydeskriptor an:

```
const u = 12;
      v = 17;
var   Feld: array[u..v] of integer;
```

Aufgabe 14: Die folgenden Arrayzugriffe sind in der Zwischensprache RePascal 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 15: Überführe das folgende Pascalprogramm zuerst in die Zwischensprache RePascal und schließlich in ALI-Assemblersprache.

```
program Fuelle;
var Feld: array[-2..3] of integer;
    i: integer;
begin
for i := -2 to 3 do
  Feld[i] := 90;
end.
```

Die letzte Aufgabe war schon etwas komplizierter. Deshalb soll dazu auch die Lösung angegeben werden. Allerdings solltest Du die Aufgabe zuerst alleine versuchen.

RePascal	ALI-Assembler
program Fuelle; **const** u = -2 o = 3 **var** Feld: **array**[u..o] **of** integer; i: integer; **begin** i := u; M1: **if** i > o **then goto** M2; Feld[i]:= 90; i := i + 1; **goto** M1; M2: **end.**	Fuelle START 0 L 0,u ST 0,i M1 L 0,i C 0,o BH M2 L 5,i S 5,u M 5,Zwei L 0,Neunzig ST 0,Feld(5) L 0,i A 0,Eins ST 0,i B M1 M2 EOJ Eins DC '1' Zwei DC '2' Neunzig DC '90' u DC '-2' o DC '3' Feld DS 6F i DS F END Fuelle

Nach dieser ausführlichen Lösung solltest Du in der Lage sein, folgende Aufgabe zu lösen:

Aufgabe 16: Gegeben ist der folgende Sortieralgorithmus:

```
program Sort;
const n=10;
var swaped: integer;
    i,j: integer;
    hilf: integer;
    Feld: array[1..n] of integer;
begin
  for i:= 1 to n do Readln(Feld[i]);

  swaped:= 1;
  i:= 1;
  while swaped=1 do
  begin
    swaped:= 0;
    for j:= 1 to n-i do
    begin
      if Feld[j]>Feld[j+1]
      then begin
             hilf:= Feld[j];
             Feld[j]:= Feld[j+1];
             Feld[j+1]:= hilf;
             swaped:= 1;
           end;
    end;
    i:= i+1;
  end;
end.
```

a) Um welchen Algorithmus handelt es sich hierbei?

b) Übersetze den Algorithmus in ALI-Assemblersprache.

Ich hoffe, Du hast die Lösung herausbekommen. Es handelte sich bei dem Algorithmus um den Bubble-Sort-Algorithmus. Die vollständige Übersetzung findest Du im Anhang (Seite 45).

Mehrdimensionale `Arrays` und `Records`

Sicherlich ist Dir bekannt, dass neben der behandelten Datenstruktur eines eindimensionalen Arrays auch mehrdimensionale Array in Pascal zugelassen sind. Die Frage die sich stellt ist, wie können solche Arrays in Maschinensprache übersetzt werden.

Aufgabe 17: Gegeben ist die Datenstruktur eines mehrdimensionalen Feldes (Matrix):

```
const u1 = 2;
      o1 = 5;
      u2 = 3;
      o2 = 8;
var   Matrix: array[u1..o1,u2..o2] of integer;
```

a) Wie muss eine solche Datenstruktur in der Zwischensprache deklariert werden, wenn in RePascal lediglich eindimensionale Felder erlaubt sind?

```
const u1 = 2;
      o1 = 5;
      u2 = 3;
      o2 = 8;
var   Feld: ____________________________ of integer
```

b) Gebe den Arraydeskriptor für diese Datenstruktur an, nachdem sie in RePascal mit Hilfe eines eindimensionalen Feldes übersetzt wurde.

c) Übersetze die folgende Pascal-Anweisung (Zugriff auf ein Matrixelement) in RePascal und anschließend in ALI:

```
Matrix[x,y] := 1;
```

Ein weiteres Problem sind Records. Diese sind von der Struktur her ähnlich wie eindimensionale Arrays mit dem einzigen Unterschied, dass die Komponenten eines Records unterschiedliche Größe haben können.

Aufgabe 18: Gegeben ist die folgende Datenstruktur:

```
type TName    = array[1..20] of char;
     TWohnort = array[1..25] of char;
     TStrasse = array[1..30] of char;
     TPerson  = record
                  Name, Vorname: TName;
                  Wohnort: TWohnort;
                  Strasse: TStrasse;
                end;
var  Person: TPerson;
```

a) Ähnlich wie für den Datentyp **array** wollen wir auch zum **record** ein Speicherabbild konstruieren. Nehmen wir an, die Variable Person wird im Speicher ab der Zelle Nr. 17 generiert. Fülle dazu die folgende Tabelle aus:

Variable:	Länge in Bytes	Offset	Adresse
Person			17
Person.Name			
Person.Vorname			

Die Datenstruktur Stack

Bevor wir damit anfangen, die Datenstruktur des Stapels in ALI zu übersetzen, sollten wir uns noch einmal überlegen, wie diese Datenstruktur definiert ist und welche Operationen auf dieser Struktur arbeiten. D. h. wie sieht der ADT Stack in Delphi aus? Wir wollen die Implementierung über einen Array (dies wird auch die Methode bei ALI sein) verfolgen. Dabei soll der Stack von unten her aufgebaut werden. Wird ein neues Element eingefügt, so muss also zuerst der „Kopf-Zeiger“ um eins erniedrigt werden, bevor dann das zu speichernde Element eingefügt wird.

```
type TInhalt = integer;
       TStack = class
                   Kopf: integer;
                   Keller: array[1..256] of TInhalt;
                   constructor create;
                   procedure push(wert: TInhalt);
                   procedure pop;
                   function  top: TInhalt;
                   function  empty: boolean;
                 { evtl. auch noch }
                   function  full: boolean;
                  end;
```

mit den Implementierungen:

```
constructor TStack.create;
begin
  Kopf:= 257;
end;

procedure TStack.push(i: TInhalt);
begin
  if not full
  then begin
         Kopf:= Kopf-1;
         Keller[Kopf]:= i;
       end;
end;

procedure TStack.pop;
begin
  if not empty
  then Kopf:= Kopf+1;
end;

function TStack.top: TInhalt;
begin
  if not empty
  then top:= Keller[Kopf];
end;

function TStack.empty: boolean;
begin
  empty:= (Kopf=257);
end;

function  TStack.full: boolean;
begin
  full:= (Kopf=1);
end;
```

Die Frage nach der Übersetzung in ALI ergibt sich anhand der obigen Implementierung des Stacks automatisch: Der Keller wird als Array im Speicher abgelegt. Welche Informationen benötigen wir allerdings dafür?
a) Maximalgröße (wir nehmen 256 Integer-Werte ⇒ 512 Bytes)
b) Zeiger auf das oberste Element (realisiert durch Register mit Nr. 5)
c) Erkennung, ob Keller leer ist benötigt einen Anfangs-Zeiger (`k_Anf`)
d) Erkennung, ob Keller voll ist benötigt einen Ende-Zeiger (`k_End`)

Die Deklaration des Kellers sieht also wie folgt aus:

Zwischensprache	ALI-Assembler
var Stack: TStack; *d. h.* Keller: array[1..256] of integer; *und* Kopf: integer;	k_anf DS F k_end DS F k_laenge DC ´512´ keller DS 256F

Übersetzung der einzelnen Routinen:

constructor create

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

Antwort: 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	

Übersetzungschablone:

Zwischensprache	ALI-Assembler
Kopf:= 257;	LA 5,keller ST 5,k_end A 5,k_laenge ST 5,k_anf

`LA` heißt `Load Adress` 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. Wie du siehst ist anschließend das Register 5 (der „Kopf-Zeiger“) mit der richtigen Adresse initialisiert. Bei dieser Realisierung des Stacks ist die Routine `empty` kein Problem mehr:

`function empty: boolean` (vor pop- und top-Aufruf nötig)
Wie oben erwähnt ist durch einen Vergleich zwischen Kopf und `k_anf` der Wert `empty` definiert. Ein Test, ob der Keller leer ist wird in der Zwischensprache zu dem Test, ob `Kopf = 257` ist.

Übersetzungschablone:

Zwischensprache	ALI-Assembler
`if Kopf = 257 then goto leer;`	`C 5,k_anf` `BE leer` `...` `leer ...`

`function full: boolean` (vor push-Aufruf nötig)
Die Vorsichtsmaßnahmen, dass der Speicher evtl. voll ist, übernimmt der Compiler für den Pascalprogrammierer. Deshalb gibt es keine Funktion `full(...)`. Gerade deshalb ist es nötig, sich bei der Übersetzung eines `push`-Befehls auch Gedanken darüber zu machen, ob genügend Speicherplatz vorhanden ist – eine `full`-Funktion wird notwendig:

Übersetzungschablone:

Zwischensprache	ALI-Assembler
`if Kopf = 1 then goto voll;`	`C 5,k_end` `BE voll` `...` `voll ...`

Mit Hilfe dieser Funktion ist ein `push`-Befehl wie folgt in die Zwischensprache zu übersetzen:

`procedure push(wert: TInhalt)`

Übersetzungschablone:

Zwischensprache	ALI-Assembler
`Kopf:= Kopf – 1;` `Keller[Kopf]:= wert;`	`S 5,'2'` `C 5,k_end` `BE voll` `L 0,wert` `ST 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.

Genauso wie der `push`-Befehl auf Bereichsgrenzen achten sollte müssen auch der `pop`- und der `top`-Befehl auf Bereichsgrenzen achten:

procedure pop

Übersetzungschablone:

Zwischensprache	ALI-Assembler
`Kopf:= Kopf + 1;`	`C 5,k_anf` `BE leer` `A 5,'2'` `...` `leer ...`

function top: TInhalt

Übersetzungschablone:

Zwischensprache	ALI-Assembler
`Wert:= Keller[Kopf];`	`C 5,k_anf` `BE leer` `L 0,0(5)` `ST 0,wert` `...` `leer ...`

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

Aufgabe 19: Was leistet folgendes Programm?

```
program Was_tue_ich;

uses Keller;

var k: TStack;
    wert: TInhalt;

begin
  k:= TStack.create;

  repeat
    Readln(wert);
    if wert <> 0 then k.push(wert);
  until wert = 0;

  while not k.empty do
  begin
    wert:= k.top;
    Writeln(wert);
    k.pop;
  end;

end.
```

Aufgabe 20: Übersetze das obige Programm

a) in die Zwischensprache (RePascal)

b) in ALI

Teste das Programm mit dem Modellrechner.

Hast Du es herausbekommen? Das Programm liest der Reihe nach Zahlen ein, bis der Benutzer die Zahl Null eingibt. Danach werden alle eingelesenen Zahlen in der umgekehrten Reihenfolge wieder ausgegeben. Die Implementierung sieht dabei wie folgt aus:

RePascal	ALI-Assembler
program in_keller_und_wieder_raus;	inout START 0
label eingabe, ausgabe, ende, voll;	
var k: **array**[1..256] **of** integer; kopf, wert: integer;	
begin	
kopf:= 257;	LA 5,keller ST 5,k_end A 5,k_laen ST 5,k_anf
eingabe: **if** kopf=1 **then goto** voll;	eingabe C 5,k_end BE voll
Readln(wert);	INI wert
if wert = 0 **then goto** ausgabe;	L 0,wert C 0,'0' BE ausgabe
kopf:= kopf-1; k[kopf]:= wert;	S 5,'2' L 0,wert ST 0,0(5)
goto eingabe;	B eingabe
ausgabe: if kopf=257 **then goto** ende;	ausgabe C 5,k_anf BE ende
wert:= k[Kopf];	L 0,0(5) ST 0,wert
Writeln(wert); kopf:= kopf+1; **goto** ausgabe;	OUTI wert A 5,'2' B ausgabe
voll: Writeln('Keller voll!');	voll OUTI '-1'
ende:	ende EOJ
	k_anf DS F k_end DS F wert DS F k_laen DC '512' keller DS 256F
end.	END inout

Prozedurtechnik

Statische Verwaltung der Rücksprungadresse

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 durchgeführt werden kann, muss auch der Rücksprung aus der Prozedur variabel sein. Konkret heißt das, dass vor dem Aufruf der Prozedur die Rücksprungadresse variabel festgelegt werden muss. Wir wollen diese Rücksprungadresse statisch festhalten. Statisch heißt hierbei, wir wollen ein Register

reservieren, in dem die Rücksprungadresse abgelegt wird. 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

Wir wollen diese Befehle an einem konkreten Beispiel erläutern. Schauen wir uns dazu zuerst das folgende Programm an:

```
program Prozedur_Beispiel;

  procedure lies_zwei_Zahlen_und_schreibe_Summe;
  var summe, zahl1, zahl2: integer;
  begin
    Readln(zahl1)
    Readln(zahl2)
    summe := zahl1 + zahl2;
    Writeln(summe);
  end;

begin
  ...
  lies_zwei_Zahlen_und_schreibe_Summe;
  ...
  lies_zwei_Zahlen_und_schreibe_Summe;
  ...
end.
```

Die erste Frage, die uns beschäftigen sollte ist die, wo eine Unterprozedur im ALI-Programm implementiert wird. Wir wollen dazu die folgenden Festsetzungen machen:

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`

Wie Du an der Skizze schon erkennen kannst, wird vor einem Aufruf eines Unterprogramms die Rücksprungadresse im Register 4 abgelegt. Am Ende eines Unterprogramms wird mit Hilfe des Registers 4 an die Rücksprungadresse zurückverzweigt. In unserem Beispiel heißt das, dass es bei jedem Aufruf der Prozedur `lies_zwei_Zahlen_und_schreibe_Summe`

erforderlich wird, die Rücksprungadresse festzuhalten. Dieses festhalten der Rücksprungadresse wollen wir im RePascal-Programm verdeutlichen. Allerdings ist diese Änderung so klein, dass man häufig auf eine explizite Angabe des RePascal-Programms verzichtet.

```
program Prozedur_Beispiel;
label rueck1, rueck2;
  procedure lies_zwei_Zahlen_und_schreibe_Summe;
  var summe, zahl1, zahl2: integer;
  begin
    Readln(zahl1)
    Readln(zahl2)
    summe := zahl1 + zahl2;
    Writeln(summe);
  end;

begin
          ...
          lies_zwei_Zahlen_und_schreibe_Summe;
  rueck1: ...
          lies_zwei_Zahlen_und_schreibe_Summe;
  rueck2: ...
end.
```

Die Übersetzung in ALI kann nun mit Hilfe der oben angegebenen Befehle und dem darunter aufgeführten Schema durchgeführt werden. Zuerst das Hauptprogramm

1. Hauptprogramm	`ProzBsp START 0` `...` `LA 4,rueck1` `B lzuds` `rueck1 ...` `LA 4,rueck2` `rueck2 ...` `EOJ`

Jetzt den Deklarationsteil. Dieser ist hier allerdings leer (es gibt keine globalen Variablen oder Konstanten.

2. Konstanten und Variablen des Hauptprogramms wie gewohnt	`EOJ` **noch kein END**

Aber es gibt eine Prozedur. Diese (abgekürzt durch `lzuds`) muss jetzt folgen:

3. erstes Unterprogramm bzw. erste Prozedur	`lzuds INI zahl1` `INI zahl2` `L 0,zahl1` `A 0,zahl2` `ST 0,summe` `OUTI summe` `BR 4` *{ Rücksprung }*

Und die dazugehörigen Variablen und Konstanten:

4. zugehörige lokale Konstanten und Variablen **nach** dem Rücksprung!	`BR 4` *{ Rücksprung }* `zahl1 DS F` `zahl2 DS F` `summe DS F`

Da es keine weiteren Prozeduren mehr gibt, folgt jetzt das

6. Programmende	`END ProzBsp`

Du solltest nun für Dich selbst die folgende Aufgabe lösen:

Aufgabe 20: Übersetze das folgende Programm in ALI-Assembler. Achte dabei darauf, dass die oben angegebene Strukturierung des Programms eingehalten wird.

```
program Sprung_Uebung;
var wert, zahl1, zahl2: integer;

  procedure Warte;
  var hilf: integer;
  begin
    Readln(hilf);
  end;

  procedure lies_zwei_Zahlen;
  begin
    Readln(zahl1)
    Readln(zahl2)
  end;

  procedure schreibe_Summe;
  begin
    wert := zahl1 + zahl2;
    Writeln(wert);
  end;

  procedure schreibe_Produkt;
  begin
    wert := zahl1 * zahl2;
    Writeln(wert);
  end;

  procedure schreibe_Quotient;
  begin
    wert := zahl1 div zahl2;
    Writeln(wert);
  end;

begin
  lies_zwei_Zahlen;
  schreibe_Summe;
  Warte;
  schreibe_Produkt;
  Warte;
  schreibe_Quotient;
end.
```

Dynamische Verwaltung der Rücksprungadresse

Wir haben uns bis jetzt mit der statischen Verwaltung der Rücksprungadresse begnügt. Folgendes Beispiel zeigt allerdings, dass die statische Verwaltung nicht ausreichend ist.

```
programM Rekursionsdemo;
label Haupt1;
var n, m, Summe: integer;

  procedure Berechne_Summe;
  label Unter1, Unter2;
  begin
              if n<= 0 then goto Unter2;
              n := n - 1;
              Berechne_Summe;
     Unter1: m:= m + 1;
              Summe:= Summe + m;
     Unter2:
  end;

begin
            m := 0;
            Summe:= 0;
            Readln(n);
            Berechne_Summe;
   Haupt1: Writeln(Summe);
end.
```

Verwenden wir die altbekannte Methode zur Rettung der Rücksprungadresse, d. h. wird bei der Übersetzung dieses Programms die Rücksprungadresse statisch in einem Register gespeichert, so wird spätestens beim ersten Rekursionsaufruf die alte Rücksprungadresse `Haupt1` mit der neuen Rücksprungadresse `Unter1` überschrieben. Dies zeigt dir hoffentlich, dass eine dynamische Verwaltung der Rücksprungadresse erforderlich wird. Die

Frage ist nur, wie kann eine solche Rücksprungadresse dynamisch festgehalten werden. Am besten kann man sich die Verwaltung der Rücksprungadressen so vorstellen: Bevor ich in ein Unterprogramm einspringe schreibe ich mir die Adresse des Rücksprungs auf einen Zettel und lege diesen auf einen Stapel, wo alle Rücksprungadressen liegen. Erreiche ich das Ende einer Prozedur, so nehme ich den obersten Zettel und springe zur angegebenen Adresse. Dieses Verfahren bezeichnet man als Laufzeitkeller oder Runtime-Stack. Auf dem Stapel werden der Reihe nach alle Rücksprungadressen draufgelegt und bei Bedarf wieder heruntergeholt. Die Implementierung eines Stacks hatten wir ja bereits, also müssten wir in der Lage sein, das Programm durch Verwendung der dynamischen Rücksprungtechnik zu übersetzen. Wir wollen dies schrittweise am obigen Beispiel tun. Dazu nehmen wir uns als erstes das Hauptprogramm vor.

Pascal	RePascal	Assembler
begin m := 0; Summe:= 0; Readln(n); Berechne_Summe; Haupt1: Writeln(Summe); **end.**	**begin** stack:= TStack.create; m := 0; Summe:= 0; Readln(n); stack.push(*Adr. von Haupt1*); **goto** Berechne_Summe; Haupt1: ... **end.**	Rekurs START 0 LA 5,stack ST 5,k_end A 5,k_laenge ST 5,k_anf L 0,´0´ ST 0,m ST 0,Summe INI n S 5,´2´ LA 0,Haupt1 ST 0,0(5) B Berechne Haupt1 OUTI Summe EOJ

Wie Du siehst ist also zuallererst eine Initialisierung des Laufzeitkellers nötig. DELPHI übernimmt dieses bei jedem Programm für Dich, egal ob Du Prozeduren benötigst oder nicht. Deshalb steht dieser Befehl auch ganz am Anfang. Die einzige Neuigkeit in diesem Programm ist der Pseudobefehl `stack.push(`*`Adresse von Haupt1`*`)`. Damit ist nur angedeutet, dass die Rücksprungadresse auf dem Stack abgelegt wird. Deutlich wird dies am ALI-Programm, in dem die alten Methoden des Stacks sichtbar werden.

Interessanter wird es nun beim Unterprogramm, d. h. bei der Prozedur. Wie muss ein Rücksprung implementiert werden? Schauen wir uns das auch an dem obigen Beispiel an:

Pascal	RePascal	Assembler
procedure Berechne_Summe; **label** Unter1, Unter2; **begin** **if** n<= 0 **then** **goto** Unter2; n := n - 1; Berechne_Summe; Unter1: m:= m + 1; Summe:=Summe+m; Unter2: **end;** **end.**	Berechne: **if** n<= 0 **then** **goto** Unter2; n := n - 1; *{ Jetzt Rücksprungadr. retten }* stack.push(*Adr. von Unter1*); **goto** Berechne; *{ und Unterprogramm fortsetzen }* Unter1: m:= m + 1; Summe:= Summe + m; *{ und Rücksprung letzter Aufruf }* Unter2: *Rücksprungadr.*:= stack.top; stack.pop; **goto** *Rücksprungadresse;* **end.**	Berechne L 0,n C 0,´0´ BNH Unter2 L 0,n S 0,´1´ ST 0,n S 5,´2´ LA 0,Unter1 ST 0,0(5) B Berechne Unter1 L 0,m A 0,´1´ ST 0,m L 0,Summe A 0,m ST 0,Summe Unter2 L 4,0(5) A 5,´2´ BR 4 END Rekurs

Wenn Du mit den einzelnen Teilstücken nicht so gut klarkommst, so kannst Du Dir die gesamte Übersetzung im Anhang (Seite 46) anschauen.

Diese Methode bietet sich allerdings nicht nur für die Rekursion an. Alle Prozeduraufrufe können so dynamisch verwaltet werden. Deshalb wollen wir hierfür auch eine Art Übersetzungsschablone festhalten, mit welcher sich jeder Prozeduraufruf (noch ohne Parameter) realisieren lässt.

Übersetzungsschablone für Prozeduraufrufe ohne Parameter:

Pascal	RePascal	Assembler
`program Schema;` `  procedure Unter;` `  begin` `    ...` `  end;` `begin` `          ...` `          Unter;` `  Weiter: ...` `end.`	`program Schema;` `begin` `          stack:= TStack.crate;` `          ...` `          stack.push(Adresse von Weiter);` `          goto Unter;` `  Weiter: ...` `end.` `Unter: ...` `       Rücksprungadresse := stack.top;` `       stack.pop;` `       goto Rücksprungadresse;`	`        START  0` `        LA     5,stack` `        ST     5,k_end` `        A      5,k_laenge` `        ST     5,k_anf` `        ...` `        S      5,´2´` `        LA     0,Weiter` `        ST     0,0(5)` `        B      Unter` `Weiter  ...` `        EOJ` `Unter   ...` `        L      4,0(5)` `        A      5,´2´` `        BR     4`

Aufgabe 21: Übersetze das Programm Aufgabe21 in ein äquivalentes Assemblerprogramm.

Aufgabe 22: Übersetze das Programm von Aufgabe 20 (Programm Sprung_Uebung) in ein Assemblerprogramm, jetzt allerdings unter Berücksichtigung der dynamischen Rücksprungtechnik.

Aufgabe 23: Formuliere zum Programm Aufgabe23 mit Rekursionsaufruf ein äquivalentes Assemblerprogramm.

```
program Aufgabe21;
var a, b: integer;

  procedure aaa;
  begin
    repeat
      Readln(a);
      if a<>0 then b := b + a;
    until a = 0;
  end;

begin
  b := 0;
  aaa;
  Writeln(b);
end.
```

```
program Aufgabe23;
var a, b: integer;

  procedure aaa;
  begin
    Readln(a);
    if a<>0
    then begin
           b := b + a;
           aaa;
         end;
  end;

begin
  b := 0;
  aaa;
  Writeln(b);
end.
```

Lokale Datenräume

Die Sache mit der dynamischen Rücksprungtechnik funktioniert soweit ganz gut. Problematisch wird es allerdings, wenn lokale Daten hinzukommen. Diese hatten uns bisher nur wenig interessiert, da unsere Variablen global angelegt waren. Dass dort allerdings Schwierigkeiten auftauchen können zeigt das folgende Programm:

```
program Zahleninversion;
label Haupt1

  procedure Invert;
  label Unter1, Unter2;
  var wert: integer;
  begin
            Readln(wert);
            if wert = 0 then goto Unter2;
            Invert;
    Unter1: Writeln(wert);
    Unter2:
  end;

begin
          Invert;
  Haupt1:
end.
```

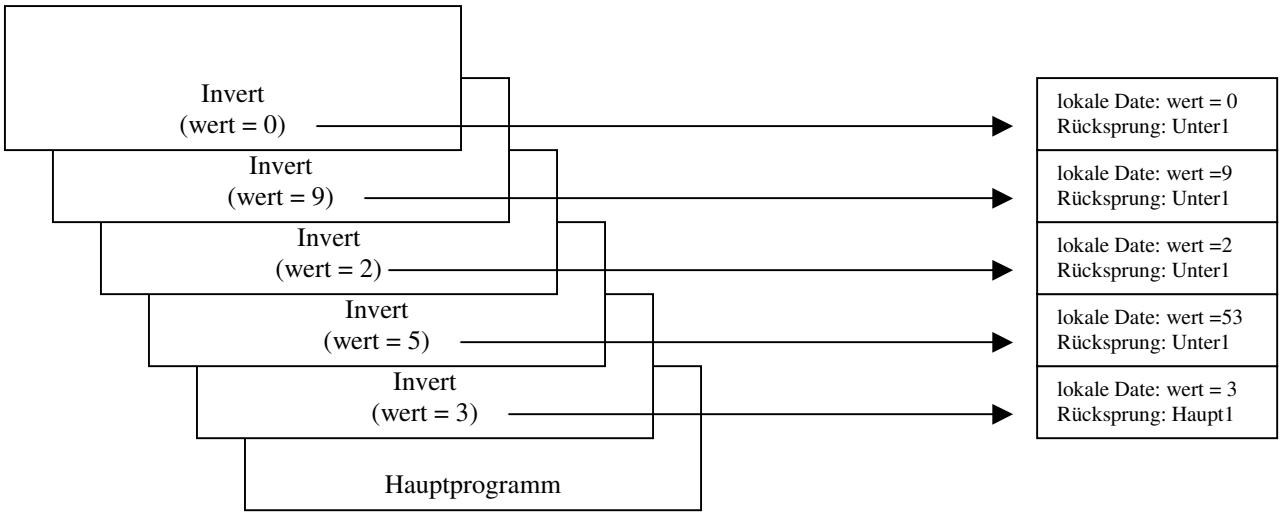
Bevor wir uns mit dem Problem genauer beschäftigen, sollten wir erst mal unsere alte Methode der Übersetzung anwenden. Vielleicht gibt es ja gar kein Problem.

Aufgabe 24: Beschreibe die Wirkung des Programms. Welche Ausgabe wird bei Eingabe der Zahlenfolge 3, 5, 2, 9 und 0 erfolgen?

Aufgabe 25: Übersetze das Programm nach dem bekannten Verfahren in ALI und teste es im Modellrechner mit der bekannten Eingabefolge 3, 5, 2, 9, 0.

Du hast wahrscheinlich auch festgestellt, dass das Programm vier mal die Zahl 0 ausgibt. Woran liegt das? Das Problem liegt in der Variablen `wert`. 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:



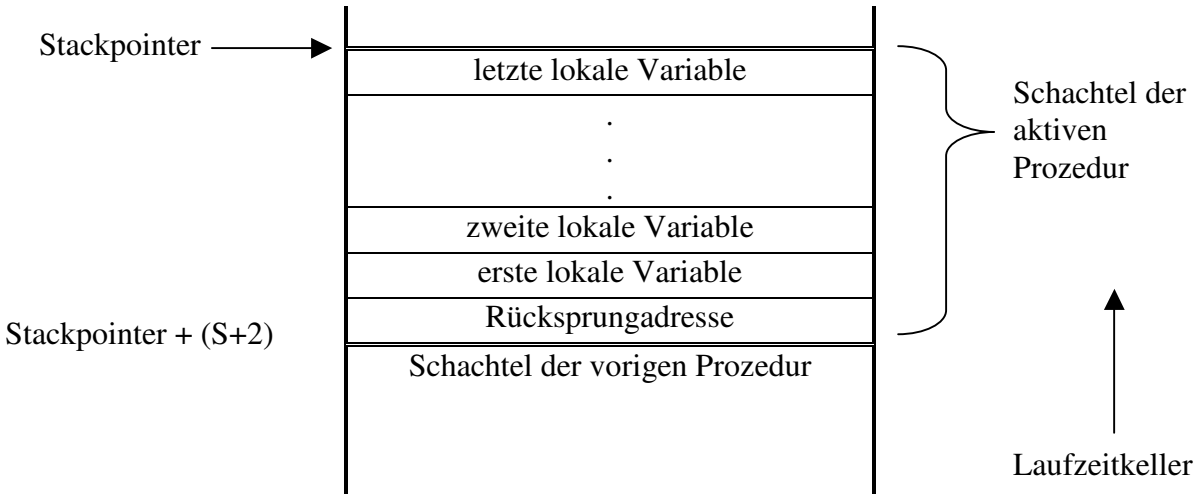
Man erkennt deutlich, dass der Keller um die lokale Date `wert` erweitert wurde. Da dieser Keller nun nicht mehr nur die Rücksprungadresse, sondern auch lokale Daten während der Programmausführung – also während der Laufzeit – aufnimmt, nennen wir diesen Keller nun **LAUFZEITKELLER**. Diese Erweiterung 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 26: 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 `L 3,2(5) ... BR 3`.

Bevor Du Dir die ALI-Lösung im Anhang (Seite 47) anschaust, solltest Du es zuerst alleine versuchen. Diese Übung ist nämlich grundlegend für das Verständnis den folgenden Prozeduraufrufen mit Paramterübergabe.

Ich hoffe, Du hast es hinbekommen. Schaue Dir jetzt ruhig die Lösung im Anhang an. Dort erkennst Du, wie mit der lokalen Variable `wert` umgegangen wird. Überall dort, wo diese Variable eigentlich auftauchen müsste, wird mit Hilfe der Adresse `0(5)` gearbeitet. Mache Dir dies noch mal am Beispiel deutlich!

Also, bevor Du weiterliest solltest Du nochmals die Lösung genau studieren, damit Du die folgenden Überlegungen nachvollziehen kannst. Schließlich bist jetzt schon ein erfahrener ALI-Experte und solltest auch mit einer Lösung zurechtkommen, die nicht von Dir stammt.

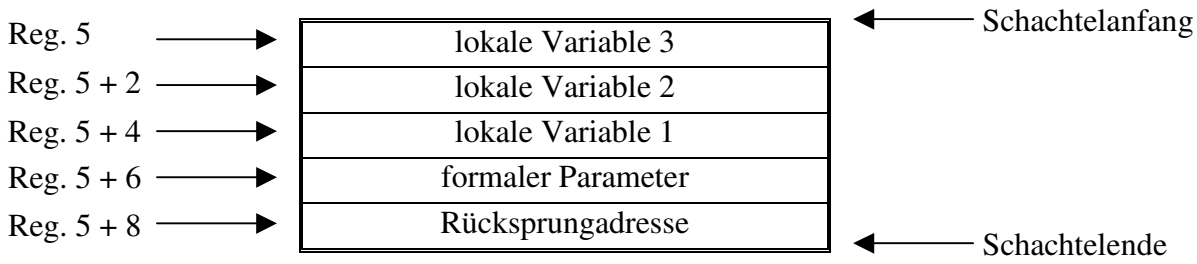
Prozeduren mit Parametern

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 Call by Value (CbV), d. h. ein sogenannter formaler (Wert)parameter wird in die aufgerufene Prozedur übergeben,
2) Parameterübergabe mit Call by Reference (CbR), d. h. ein sogenannter Referenzparameter wird in die aufgerufene Prozedur übergeben.

Prozeduren mit Call-by-Value-Übergabe

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: (Beachte die Reihenfolge lokale Variablen, formale Parameter, Rücksprungadresse)



Die Übergabe eines formalen Parameters geschieht also mit der Befehlsfolge

```
L   0, Wertparameter
ST  0,6(5)            { Register 5 ist Stackpointer }
```

Innerhalb der Prozedur kann man nun – genauso wie man auf lokale Daten mit Hilfe der absoluten Indizierung zugreift – auf die formalen Parameter zugreifen.
Die Befehlssequenz

```
L   0,4(5)
A   0,6(5)
ST  0,2(5)
```

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

Bevor Du Dich jetzt um die Lösung der folgenden Aufgabe kümmerst, sei der bereits bekannte Stack noch einmal angesprochen. Es ist ratsam, den Stack fest zu definieren. Das heißt, nach dem Hauptprogramm sollte eine Variablen-/Konstantendeklaration der folgenden Art auftauchen

```
anzahl   DS       F
anfang   DC       '4000'    Anfangsadresse des Laufzeitkellers
ende     DC       '1000'    Endadresse des Laufzeitkellers
s_laeng  DC       'xxx'     Länge der Prozedurschachtel
```

Damit wird der Stack von unten nach oben im freien Speicherbereich 1000-4000 aufgebaut. Dieser Stack sollte groß genug sein, um auch Rekursionen aufnehmen zu können.

Aufgabe 27: Übersetze folgendes RePascal-Programm in ALI-Assembler. Mache Dir zuvor ein Bild der Prozedurschachtel von `Invert`.

```
program Zahleninversion;
label Haupt1;
var anzahl: integer;

  procedure Invert(n: integer);
  label Unter1, Unter2;
  var wert: integer;
  begin
            if n = 0 then goto Unter2;
            Readln(wert);
            n:= n - 1;
            Invert(n);
    Unter1: Writeln(wert);
    Unter2:
  end;

begin
          Readln(anzahl)
          Invert(anzahl);
  Haupt1:
end.
```

Aufgabe 28: Übersetze folgendes Pascal-Programm in RePascal und schließlich in ALI-Assembler. Mache Dir auch hier zuvor ein Bild der Prozedurschachtel von `Lies_zahlen_und_drucke_summe`.

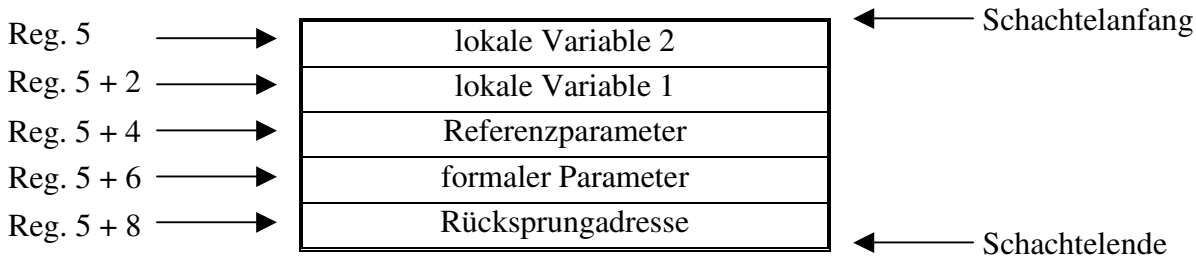
```
program Summe;
var anzahl: integer;

  procedure lies_zahlen_und_drucke_summe(n: integer);
  var summe, zahl, index: integer;
  begin
    summe:= 0;
    index:= 0;
    repeat
      index:= index + 1;
      Readln(zahl);
      summe:= summe + zahl;
    until index = n;
    Writeln(summe);
  end;

begin
  Readln(anzahl);
  lies_zahlen_und_drucke_summe(anzahl);
end.
```

Zur Aufgabe 27 ist eine Lösung im Anhang (Seite 48) abgedruckt. Schaue Dir dort noch mal genau an, wie das mit dem festen Stack gemeint ist und übertrage diese Lösung auf Deine Lösung zur Aufgabe 28. Dies sollte nun eigentlich machbar sein.

Prozeduren mit Call-by-Reference-Übergabe

Im Gegensatz zur Parameterübergabe Call by Value, bei der eine echte Kopie des formalen Parameters in der neuen Prozedurschachtel angelegt werden musste, reicht es bei der Parameterübergabe Call by Reference schon aus, lediglich die Adresse des übergebenen Objekts mitzuteilen. Somit kann in der aufgerufenen Prozedur direkt über die Adresse auf den Referenzparameter zugegriffen werden.

Denken wir uns ein Beispiel, in der sowohl lokale Variablen als auch Wertparameter als auch Referenzparameter in einer Prozedurschachtel festgehalten werden müssen. Eine Prozedurschachtel könnte dann wie folgt aussehen (Beachte auch jetzt wieder die Reihenfolge lokale Variablen, Referenzparameter, formale Parameter, Rücksprungadresse):



Die Referenzparameterübergabe – d. h. die Übergabe der Adresse der zu übergebenen Variablen – kann dann durch folgende ALI-Befehlssequenz realisiert werden:

```
LA   0,Referenzparameter    { Wichtig ist hier das Load Adress }
ST   0,4(5)                 { Register 5 ist Stackpointer }
```

Beim Zugriff auf einen Referenzparameter innerhalb der Prozedur spielt nun die Tatsache, dass der Inhalt des entsprechenden Parameters eine Adresse darstellt, eine entscheidende Rolle. Um auf die Variable zugreifen zu können muss erst diese Adresse in ein Hilfsregister (z. B. Register 4) geladen werden und danach erst das Objekt selbst mit diesem Register über die absolut-indizierte Adressierung (mit Absolutadresse 0) angesprochen werden.

```
L    4,distanz(5)    { Adresse aus der Prozedurschachtel in Hilfsregister laden }
L    0,0(4)          { und dann mit dieser Adresse absolut-indiziert
                       auf das Objekt zugreifen }
```

Im Obigen Beispiel der Prozedurschachtel erfordert die Erhöhung des Referenzparameters um die lokale Variable 1 den Umweg über ein Hilfsregister. So bedeutet die Befehlssequenz

```
L    4,4(5)     { Lade Adresse des Referenzparameters in Register 4. }
L    0,0(4)     { Lade den Wert, der an dieser Adresse steht. }
A    0,2(5)     { Addiere den Wert der „lokalen Variablen 1“ hinzu. }
ST   0,0(4)     { Speichere den neuen Wert im Referenzparameter }
```

hier: „Referenzparameter“ := „ Referenzparameter“ + „lokale Variable 1“

Versuch nun mit diesen Informationen die folgenden Programme zu übersetzen.

Aufgabe 29: Übersetze folgendes RePascal-Programm in ALI-Assembler. Mache Dir zuvor ein Bild der Prozedurschachtel von `lies_zahlen_uns_summiere`. Die Lösung zu diesem Programm findest Du im Anhang (Seite 49).

```
program Summe;
label ausgabe;
var summe, anzahl: integer;

  procedure lies_zahlen_und_summiere(n: integer;
                                     var s: integer);
  label m;
  var zahl, index: integer;
  begin
        index:= 0;
    m:  index:= index + 1;
        Readln(zahl);
        s:= s + zahl;
        if index <> n then goto m;
  end;

begin
          summe := 0;
          Readln(anzahl);
          lies_zahlen_und_summiere(anzahl, summe);
  ausgabe: Writeln(summe);
end.
```

Aufgabe 30: Im Zusammenhang mit Referenzparametern kann es vorkommen, dass der aktuell zu übergebene Parameter bereits ein Referenzparameter ist, d. h. die Adresse des eigentlichen Objektes enthält. Betrachte dazu folgendes Programm:

```
program Beispiel;
var anzahl: integer;

  procedure zaehle(var n: integer);
  label m;
  begin
        if n = 0 then goto m
        n := n - 1;
        zaehle(n);
    m:
  end;

begin
  Readln(anzahl);
  zaehle(anzahl);
end.
```

Welche Problematik entsteht hierdurch und wie könnte man diese Problematik umgehen? Überlege Dir dazu genau, wann es nötig ist, die Adresse des Übergabeobjektes in die Prozedurschachtel zu kopieren und wann es vielleicht ausreicht lediglich den Wert des aktuellen Parameters weiterzureichen. Versuche anschließend Dein Glück und übersetze das obige RePascal-Programm mit Hilfe Deiner Übergabestrategie in ein hoffentlich funktionierendes ALI-Programm.

Schlussbetrachtung

So – es ist Zeit, uns noch einmal vor Augen zu führen, was Du auf diesen 50 Seiten gelernt hast. Du weißt jetzt,

- wie man grundlegende arithmetische Befehle schrittweise in Maschinensprache übersetzt,
- wie man Sprünge in einer Maschinensprache übersetzt und dass Sprünge zur Programmsteuerung unerlässlich sind,
- wie man Schleifen und Kontrollstrukturen aus höheren Programmiersprachen schrittweise übersetzen kann,
- wie Datenstrukturen – sowohl statische als auch dynamische – in Maschinensprache angelegt werden können,
- wie Prozeduren übersetzt werden können,
- wie Parameter von Prozeduren behandelt werden,

Dabei ist zu beachten, dass für alle Probleme eine Art Übersetzungsschablone existiert. Die Aufgabe eines Compilers ist es nur, diese Übersetzungsschablonen richtig anzuwenden. Wie ein solcher Compiler genau arbeitet, das werden wir uns in einer gesonderten Unterrichtsreihe anschauen. Die Von-Hand-Übersetzung ist jedoch für Dich kein Problem mehr.
Als letzte Übung – so quasi als Abiturvorbereitung – möchte ich Dir die folgende Aufgabe ans Herz legen. Dort wird noch mal so einiges von dem gebraucht, was Du bis jetzt kennen gelernt hast. Also, viel Erfolg.

Aufgabe 31: Übersetze das folgende Programm schrittweise in ALI-Assembler-Sprache. Erinnere Dich dabei an die Methode, wie eine x := a **mod** b – Anweisung in die Zwischensprache RePascal übersetzt wurde.

```
program Quersum;
var n, qw: integer;

  procedure quer (var w: integer; q: integer);
  var h1, h2, h3: integer;
  begin
      h1 := q div 10;
      if h1 = 0
        then w := q
        else begin
               quer( h2, h1 )
               h3 := q mod 10;
               w := h3 + h2;
             end;
  end;

begin
  Readln(n);
  quer( qw, n );
  Writeln(qw);
end.
```

Die Lösung zu diesem Programm findest Du ebenfalls im Anhang (Seite 50).

Anhang

DREISCHRITT VOM PROGRAMM ZUR MASCHINE AM BEISPIEL `Rechteck` MIT DEM ALI-MODELLRECHNER

Die Programme in Pascalnotation:

Das Pascalprogramm

```
program Rechteck;
var a, b: integer;



begin
  Readln(a);
  b := (a + 1) * (a + 2)



  Writeln(b);
end;
```

Das Programm in reduziertem Pascal:

```
program Rechteck;
const eins = 1;
      zwei = 2;
var   a: integer;
      b: integer;
begin
  Readln(a);
  h1 := a + 1;
  h2 := a + 2;
  h3 := h1 * h2;
  b  := h3;
  Writeln(b);
end;
```

Die Programme in ALI-Notation:

Das Assemblerprogramm:

```
Rechteck START    0
         INI      a
         L        0,a
         A        0,eins
         ST       0,h1
         L        0,a
         A        0,zwei
         ST       0,h2
         L        0,h1
         M        0,h2
         ST       0,h3
         L        0,h3
         ST       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	

LÖSUNG ZUM PROGRAMM `Bubblesort` (AUFGABE 16)

```
program Bubblesort;
const n=10;
var swaped: integer;
    i,j: integer;
    hilf: integer;
    Feld: array[1..n] of integer;
begin
  for i:= 1 to n do
    Readln(Feld[i]);

  swaped:= 1;
  i:= 1;

  while swaped=1 do
  begin
    swaped:= 0;
    for j:= 1 to n-i do
    begin
      if Feld[j]>Feld[j+1]
      then begin
             hilf:= Feld[j];
             Feld[j]:= Feld[j+1];
             Feld[j+1]:= hilf;
             swaped:= 1;
           end;
    end;
    i:= i+1;
  end;
end.
```

```
program Bubblesort;
label L1, L2, L3, L4, L5, L6, L7;
const n=10;
      null=0;
      eins=1;
      zwei=2;
var swaped: integer;
    i,j: integer;
    hilf: integer;
    Feld: array[1..n] of integer;
    h1, h2: integer;
begin
      i:= eins;
  L1: if i>n then goto L2;
      Readln(Feld[i]);
      i:= i+eins;
      goto L1;

  L2: swaped:= eins;
      i:= eins;

  L3: if swaped<>eins then goto L4;
      swaped:= null;
      j:= eins;
      h1:= n-i;
  L5: if j>h1 then goto L6;
      h2:= j+1;
      if Feld[j]<=Feld[h2] then goto L7;
      hilf:= Feld[j];
      Feld[j]:= Feld[h2];
      Feld[h2]:= hilf;
      swaped:= eins;
  L7: j:= j+eins;
      goto L5;
  L6: i:= i+eins;
      goto L3;
  L4: ;
end.
```

```
Bubble  START   0
        L       0,eins
        ST      0,i
L1      L       0,i
        C       0,n
        BH      L2
        L       5,i
        S       5,eins
        M       5,zwei
        INI     Feld(5);
        L       0,i
        A       0,eins
        ST      0,i
        B       L1
L2      L       0,eins
        ST      0,swaped
        L       0,eins
        ST      0,i
L3      L       0,swaped
        C       0,1
        BNE     L4
        L       0,null
        ST      0,swaped
        L       0,eins
        ST      0,j
        L       0,n
        S       0,i
        ST      0,h1
L5      L       0,j
        C       0,h1
        BH      L6
        L       0,j
        A       0,eins
        ST      0,h2
        L       5,j
        S       5,eins
        M       5,zwei
        L       0,Feld(5)
        L       5,h2
        S       5,eins
        M       5,zwei
        C       0,Feld(5)
        BNH     L7
        L       5,j
        S       5,eins
        M       5,zwei
        L       0,Feld(5)
        ST      0,hilf
        L       5,h2
        S       5,eins
        M       5,zwei
        L       0,Feld(5)
        L       5,j
        S       5,eins
        M       5,zwei
        ST      0,Feld(5)
        L       0,hilf
        L       5,h2
        S       5,eins
        M       5,zwei
        ST      0,Feld(5)
        L       0,eins
        ST      0,swaped
L7      L       0,j
        A       0,eins
        ST      0,j
        B       L5
L6      L       0,i
        A       0,eins
        ST      0,i
        B       L3
L4      EOJ
n       DC      '10'
null    DC      '0'
eins    DC      '1'
zwei    DC      '2'
swaped  DS      F
i       DS      F
j       DS      F
hilf    DS      F
Feld    DS      10F
h1      DS      F
h2      DS      F
        END     Bubble
```

LÖSUNG ZUM PROGRAMM Rekursionsdemo

RePascal	Assembler
program Rekursionsdemo;	
label Haupt1, Unter1, Unter2,	
Berechne, Ende;	
var n, m, Summe: integer;	
{ Hauptprogramm }	
begin	Rekurs START 0
stack:= TStack.create;	LA 5,stack
	ST 5,k_end
	A 5,k_laenge
	ST 5,k_anf
m := 0;	L 0,´0´
	ST 0,m
Summe:= 0;	ST 0,Summe
Readln(n);	INI n
{ Jetzt Rücksprungadr. retten }	
stack.push(*Adresse von Haupt1*);	S 5,´2´
	LA 0,Haupt1
	ST 0,0(5)
goto Berechne;	B Berechne
{ und Hauptprogramm fortsetzen }	
Haupt1: Writeln(Summe);	Haupt1 OUTI Summe
end.	EOJ
	m DS F
	n DS F
	Summe DS F
	k_anf DS F
	k_end DS F
	k_laenge DC ´512´
	stack DS 256F
{ Prozedur Berechne_Summe }	
begin	
Berechne: **if** n<= 0 **then goto** Unter2;	Berechne L 0,n
	C 0,´0´
	BNH Unter2
n := n – 1;	L 0,n
	S 0,´1´
	ST 0,n
{ Jetzt Rücksprungadr. retten }	
stack.push(*Adresse von Unter1*);	S 5,´2´
	LA 0,Unter1
	ST 0,0(5)
goto Berechne;	B Berechne
{ und Unterprogramm fortsetzen }	
Unter1: m:= m + 1;	Unter1 L 0,m
	A 0,´1´
	ST 0,m
Summe:= Summe + m;	L 0,Summe
	A 0,m
	ST 0,Summe
{ und Rücksprung letzter Aufruf }	
Unter2: *Rücksprungadresse* := stack.top;	Unter2 L 4,0(5)
stack.pop;	A 5,´2´
goto *Rücksprungadresse*;	BR 4
end;	END Rekurs

LÖSUNG ZUM PROGRAMM `Zahleninversion` (AUFGABE 26)

```
ZInv     START    0
         LA       5,keller
         ST       5,k_end
         A        5,k_laeng
         ST       5,k_anf          Laufzeitkeller einrichten (Reg. 5)
         S        5,s_laeng        Pruefen, ob geung
         C        5,k_end          Platz fuer Schachtel
         BL       fehler           vorhanden ist
         LA       4,Haupt1         Ruecksprungadresse in Reg 4 laden
         ST       4,2(5)           und in Schachtel ablegen
         B        invert           Sprung ins Unterprogramm
Haupt1   EOJ                       Ende des Hauptprogramms

fehler   OUTI     '-1'
         EOJ

anzahl   DS       F
k_anf    DS       F                Anfangsadresse des Laufzeitkellers
k_end    DS       F                Endadresse des Laufzeitkellers
k_laeng  DC       '1000'           Kellergröße
s_laeng  DC       '4'              (Laenge der Schachtel: 1 lok. Variable
                                                      + 1 Ruecksprадr.)

* Prozedur invert
* ---------------------------
* R5 zeigt auf Anfang der Schachtel auf dem laufzeitkeller
*
*               ############################
* R5      --->  # lokale Variable "wert" #
*               # ---------------------- #
* R5 + 2  --->  # Ruecksprungadresse     #
*               ############################
*

invert   INI      0(5)             Readln(wert)
         L        0,0(5)           if wert
         C        0,'0'             = 0
         BE       Unter2           then goto Unter2
         S        5,s_laeng        Pruefung, ob genug Platz
         C        5,k_end                    fuer Schachtel
         BL       fehler                     vorhanden ist
         LA       4,Unter1         Ruecksprungadr. in Register 4 laden
         ST       4,2(5)           und in Prozedurschachtel ablegen
         B        invert           und Rekursionsaufruf
Unter1   OUTI     0(5)             lokale Variable "Wert" ausgeben
Unter2   L        4,2(5)           Rueckkehradresse in Register 4 laden
         A        5,s_laeng        Stackpointer auf vorherige
                                              Prozedurschachtel setzen
         BR       1                Rueckkehr zum Aufrufenden

         END      ZInv
```

LÖSUNG ZUM PROGRAMM `Zahleninversion` (AUFGABE 27)

```
ZInv     START    0
         INI      anzahl
         L        5,anfang          Laufzeitkeller einrichten (Reg. 5)
         S        5,s_laeng         Pruefen, ob geung
         C        5,ende            Platz fuer Schachtel
         BL       fehler            vorhanden ist
         LA       1,Haupt1          Ruecksprungadresse in Reg 1 laden
         ST       1,4(5)            und in Schachtel ablegen
         L        0,anzahl          formaler Parameter anzahl laden
         ST       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 Ruecksprадr. = 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   L        0,2(5)            if n
         C        0,'0'              = 0
         BE       Unter2            then goto Unter2
         INI      0(5)              Readln(wert)
         LR       4,2(5)            aktueller Schachtelanfang in
                                    Hilfsregister 4 retten (Load Register)
         S        5,s_laeng         Pruefung, ob genug Platz
         C        5,ende                      fuer Schachtel
         BL       fehler                      vorhanden ist
         LA       1,Unter1          Ruecksprungadr. in Register 1 laden
         ST       1,4(5)            und in Prozedurschachtel ablegen
         L        0,2(4)            form. Param. aus akt.(Reg. 4) Schachtel
         S        0,'1'             laden und um eins erniedrigen
         ST       0,2(5)            und in die neue Schachtel kopieren
         B        invert            und Rekursionsaufruf
Unter1   OUTI     0(5)              lokale Variable "Wert" ausgeben
Unter2   L        1,4(5)            Rueckkehradresse in Register 1 laden
         A        5,s_laeng         Stackpointer auf vorherige
                                    Prozedurschachtel setzen
         BR       1                 Rueckkehr zum Aufrufenden

         END      ZInv
```

LÖSUNG ZUM PROGRAMM Summe (AUFGABE 29)

```
summe    START    0
         L        0,'0'
         ST       0,summe          summe:= 0
         INI      anzahl           Readln(anzahl)
         L        5,anfang         Laufzeitkeller einrichten (Reg. 5)
         S        5,laeng          Pruefung, ob genug Platz
         C        5,ende                    fuer Schachtel
         BL       fehler                    vorhanden ist
         LA       1,ausgabe        Rueckkehradresse in Schachtel
         ST       1,8(5)             ablegen (Push Rueckkehradr.)
         L        1,anzahl         Wert des Wertparamters anzahl in den
         ST       1,6(5)             formalen Parameter n der Schachtel kopieren
         LA       1,summe          Adresse des Referenzparameters summe in den
         ST       1,4(5)             formalen Parameter s der Schachtel kopieren
         B        lzus             Sprung in die Prozedur
ausgabe  OUTI     summe            Writeln(summe)
         EOJ

fehler   OUTI     '-1'             Fehlerabbruch
         EOJ

* Deklarationen des Hauptprogramms und technische Konstanten

anzahl   DS       F
summe    DS       F
anfang   DC       '4000'
ende     DC       '1000'
laeng    DC       '10'             Schachtellaenge = 2 lokale Variablen
*                                  + 1 Wertparameter + 1 Referenzparameter
*                                  + 1 Ruecksprungadr. = 10 Bytes
*
* Prozedur "lies_zahlen_und_summiere"
* -----------------------------------
* R5 zeigt auf den Angang der aktuellen Schachtel im Keller
* R4 wird fuer die Ruecksprungadresse gebraucht
* R3 ist Hilfsregister f□r die absolut-indizierte Adressierung des Referenzparameters
*
*             ##############################
*  R5     --> # lokale Variable "index"    #
*             # -------------------------- #
*  R5 + 2 --> # lokale Variable "zahl"     #
*             # -------------------------- #
*  R5 + 4 --> # Referenzparameter "s"      #
*             # -------------------------- #
*  R5 + 6 --> # formaler Wertparameter "n" #
*             # -------------------------- #
*  R5 + 8 --> # Ruecksprungadresse         #
*             ##############################

lzus     L        0,'0'            index := 0
         ST       0,0(5)
m        L        0,0(5)           index :=
         A        0,'1'            index + 1;
         ST       0,0(5)
         INI      2(5)             Readln(zahl)
         L        3,4(5)           Adresse von summe in R3 laden
         L        0,0(3)           und Wert von summe laden
         A        0,2(5)           s := s + zahl
         ST       0,0(3)           und Wert von s an Adr. von summe ablegen

         L        0,0(5)
         C        0,6(5)           index = n?
         BNE      m                falls ungleich zurueck zur Schleife
         L        4,8(5)           Rueckkehradresse in R4 laden
         A        5,laeng          Stackpointer um Schachtellaenge zuruecksetzen
         BR       4                Rueckkehr zur Aufrufenden Stelle (Adresse in R4)

         END      summe
```

LÖSUNG ZUM PROGRAMM `Quersumme` (AUFGABE 31)

In RePascal:

```
program Quersum;
var n, qw: integer;

  procedure quer(var w: integer;
                     q: integer);
  label m1, m2;
  var h1, h2, h3: integer;
  begin
      h1 := q div 10;
      if h1 = 0 then goto m1;
      quer( h2, h1 )
      h3 := q div 10;
      h3 := h3 * 10;
      h3 := q - h3;
      w := h3 + h2;
      goto m2;
  m1: w := q;
  m2: end;

begin
  Readln(n);
  quer( qw, n );
  Writeln(qw);
end.
```

So, mit dieser letzten Übung soll unsere Reihe zu ALI dem Ende entgegen gehen. Ich hoffe, Du hast diese letzte Übung auch noch zu Deiner Zufriedenheit gemeistert und hattest vielleicht auch ein bisschen Spaß daran, mit ALI zu experimentieren.

In ALI-Assembler:

```
quers    START   0
         INI     n           Readln(n)
         L       5,anfang    Stack vorbereiten
         S       5,s_laeng   Aufruf vorbereiten
         LA      0,ausgabe   Ruecksprungadresse
         ST      0,10(5)       sichern
         L       0,n         Formalen Parameter
         ST      0,6(5)        sichern
         LA      0,qw        Referenzparameter
         ST      0,8(5)        sichern
         B       quer        erster Aufruf
ausgabe  OUTI    qw          Writeln(qw)
         EOJ

n        DS      F
qw       DS      F
anfang   DC      '4000'
ende     DC      '1000'
s_laeng  DC      '12'

* Prozedurschachtel ist wie folgt aufgebaut:
*
*              ##########################
* R5      -->  # lokale Variable "h3"  #
*              # -------------------- #
* R5 + 2  -->  # lokale Variable "h2"  #
* R5 + 4  -->  # lokale Variable "h1"  #
* R5 + 6  -->  # Wertparameter "q"     #
* R5 + 8  -->  # Referenzparameter "w" #
* R5 + 10 -->  # Ruecksprungadresse    #
*              ##########################

quer     L       0,6(5)      q
         D       0,'10'        div 10
         ST      0,4(5)              --> h1
         C       0,'0'       if h1 = 0 then
         BE      m1            goto m1
         LR      4,5         Alten SPointer merken
         S       5,s_laeng   Stackpointer - s_laeng
         LA      0,weiter    Ruecksprungadresse
         ST      0,10(5)       sichern
         LA      0,2(4)      Referenzparameter (h2)
         ST      0,8(5)        sichern
         L       0,4(4)      formaler Parameter (h1)
         ST      0,6(5)        sichern
         B       quer        Rekursionsaufruf

weiter   L       0,6(5)      q
         D       0,'10'        div 10
         ST      0,0(5)              --> h3
         M       0,'10'      h3 * 10
         ST      0,0(5)              --> h3
         L       0,6(5)      q
         S       0,0(5)        - h3
         ST      0,0(5)              --> h3
         A       0,2(5)      h3 + h2
         L       4,8(5)              -->
         ST      0,0(4)                  w
         B       m2

m1       L       0,6(5)      q
         L       4,8(5)              -->
         ST      0,0(4)                  w
m2       L       1,10(5)     Ruecksprадr. holen
         A       5,s_laeng     und Schachtel freig.
         BR      1             und zurueckspringen

         END     quers
```