Konzepte Systemnaher Programmierung

Prof. Dr. Hellwig Geisse

TeXified by Thomas Bilk

Version 0.3

WS 2015/16

Inhaltsverzeichnis

1 Einführung

1.1 Warum KSP?

1. Embedded Systems
2. Hochperformante Systeme bei knappsten Ressourcen (Speicherplatz, Ausführungszeit)
3. Programme nahe am (Realzeit-) Betriebsystem, manchmal völlig ohne Betriebsystem direkt auf der Hardware
4. Am häufigsten eingesetzte Sprache: C
5. Plattformabhängige Realisierung von plattformunabhängigen Konzepten durch „Virtuelle Maschinen“ (Bsp: Java)

1.2 Themen

1. C-Programmierung
2. Speicherverwaltung, Laufzeitorganisation von Programmen
3. Bibliotheken
4. Werkzeugkette (Compiler, Assembler, Binder, Lader)
5. Garbage-Collectoren
6. Interpreter, Virtuelle Maschinen

1.3 Aufbau der Veranstaltung

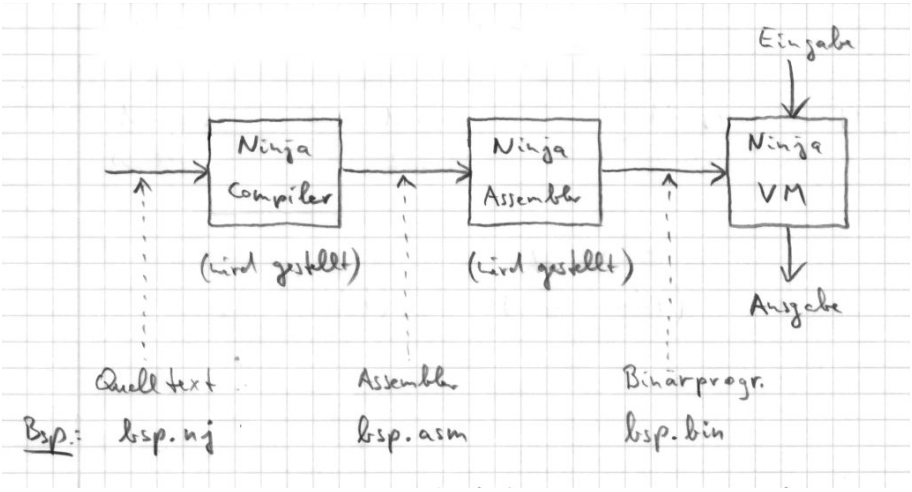
Zwei Alternativen

- linear: 4 Wochen C-Programmieren, 3 Wochen Werkzeugkette, $\cdots$
- projektzentriert: anhand eines Projektes lernt man alle notwendigen Konzepte, Tools, Verfahren etc. kennen

Vorteil von a): sehr systematisch
Vorteil von b): man lernt, warum die Dinge so sind, wie sie sind und es macht mehr Spaß!
Meine Wahl: b) !

1.4 Projekt

Konstruktion einer Virtuellen Maschine zur Ausführung von übersetzten Programmen der kleinen Programmiersprache Ninja („Ninja is not Java“)



Bem.:

- Das ist in kleinem Maßstab auch das, was bei Java passiert.
- Konstruktion „bottom-up“: Wir beginnen mit einer minimalen VM und nehmen Features dazu, wenn benötigt.

1.5 Literatur

1. Brian W. Kernighan, Dennis M. Ritchie:
   The C Programming Language, 2nd Edition, Prentice Hall, 2012
2. Randal E. Bryant, David O'Hallaron:
   Computer Systems - A Programmer's Perspective, Prentice Hall, 2015
3. Richard Jones, Rafael Lins:
   Garbage Collection, John Wiley & Sons, 1996
4. James E. Smith, Ravi Nair:
   Virtual Machines, Morgan Kaufmann, 2005

1.6 Symbole zur Einordnung des Lehrstoffs

C	die Programmiersprache C
VM	die Virtuelle Maschine für Ninja
ASM	der Assembler für Ninja
NJ	die Programmiersprache Ninja

2 C: Grundlagen

Die Syntax und Semantik von Java und C ist für viele Sprachkonstrukte ähnlich (zusammengesetzte Anweisungen, Zuweisungen, if-, while, do-Anweisungen, Rechenausdrücke, Arrays). Wesentliche Unterschiede:

- C hat keine Klassen (dafür Verbunde)
- C kennt keine Referenzen (dafür aber explizite Zeiger)
- C hat keine automatische Speicherbereinigung
  ⇒ Speichermanagement liegt beim Programmierer

Ein „C-Programm“ is eine Sammlung von Funktionen. Genau eine davon muss „main“ heißen; sie wird beim Starten des Programms automatisch aktiviert.

Einfachstes C-Programm:

```
int main(int argc, char *argv[]) {
  return 0;
}
```

Wie liest man Deklarationen? Immer beim Bezeichner starten!

`int argc;`	`argc` ist ein Integer
`char str[100];`	`str` ist ein Array von 100 Zeichen
`int *w;`	`w` ist ein Zeiger auf Integer
`char *argv[];`	`argv` ist ein Array (unspezifischer Größe) von Zeigern auf Zeichen
`int main(int argc, char *argv[]) {}`	`main` ist eine Funktion mit zwei Parametern (ein Integer und ein Array von Zeigern auf Zeichen), die einen Integer zurück gibt

Bedeutung der Parameter/Rückgabewerte bei `main`:

- `argc` gibt an, wie viele Elemente `argv` hat
- `argv` enthält Zeiger auf die Elemente (Strings) der Kommandozeile bei Aufruf des Programms
- `argv[0]` enthält vereinbarungsgemäß immer den Namen des gerade laufenden Programms
- der Rückgabewert signalisiert Erfolg (0) oder Mißerfolg ($\neq 0$) an das Betriebsystem (der Wert kann von anderen Programmen abgefragt werden)

Bsp: Wir schreiben unser C-Programm in eine Datei mit dem Namen `tst1.c` und übersetzen es mit dem Compileraufruf
`gcc -g -Wall -std=c89 -pedantic -o tst1 tst1.c`
dann entsteht das ausführbare Programm `tst1`. Rufen wir dieses auf mit der Kommandozeile
`./tst1 -s 53 hugo`
so ist `argc=4` und `argv` hat folgende Elemente:

```
argv[0] = "./tst1"
argv[1] = "-s"
argv[2] = "53"
argv[3] = "hugo"
```

2.1 Ausgabe

`printf(char *fmt, ...);` („Funktionsprototyp“)
`fmt`: „Format-String“; wird so ausgegeben, wie er angegeben ist, aber `%` wird ersetzt durch den Wert des entsprechenden Ausdrucks in `...`

Bsp: `printf("Das ist die %d-te Ausgabe!\n", 2*5);` gibt aus: Das ist die 10-te Ausgabe!

`%d`	Ganzzahl in Dezimaldarstellng
`%x`	Ganzzahl in Hexadezimaldarstellung
`%c`	Ganzzahl als Character
`%s`	String (Characters bis zum abschließenden `'\0'`)
`%p`	Pointer (in maschinenabhängiger Darstellung)

Verfügbar machen mit `#include <stdio.h>`

2.2 Stringvergleich

`int strcmp(char *s1, char *s2);`
Vergleicht die Strings `s1` und `s2` lexikographisch und liefert Zahl $< 0, = 0, > 0$ zurück, wenn `s1` vor `s2` liegt, gleich `s2` ist, nach `s2` liegt.

Achtung: `s1==s2` ist syntaktisch auch richtig, vergleicht aber die Zeiger und nicht die Strings!

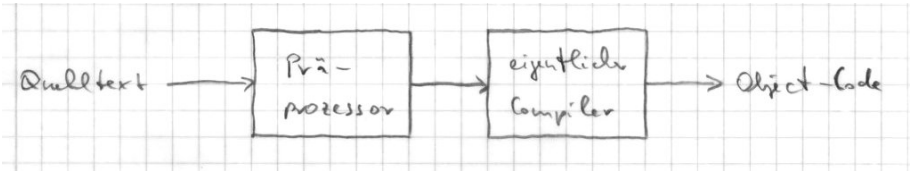
Verfügbar machen mit `#include <string.h>`

2.3 Beenden des Programms

1. mit `return ...;` aus `main()`
2. mit `exit(...);` aus irgendeiner bel. Funktion

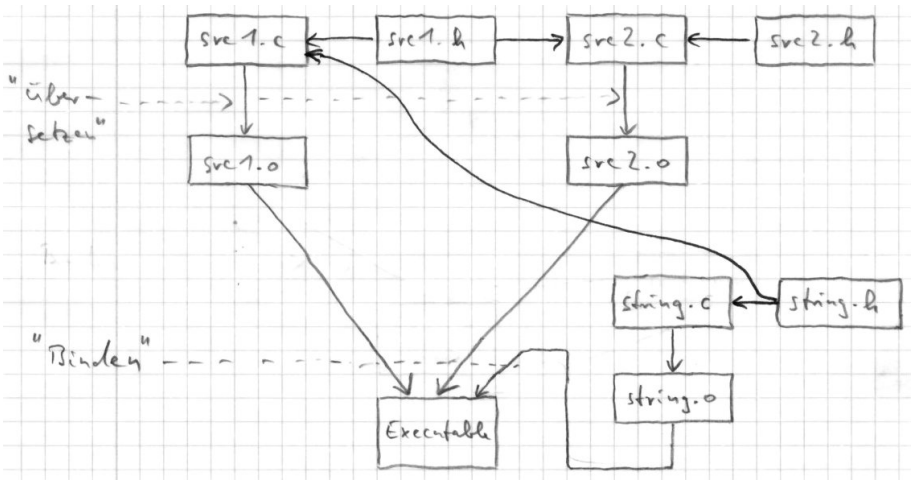
Verfügbar machen mit `#include <stdlib.h>`

2.4 Der C-Präprozessor



Befehle an den Präprozessor beginnen mit dem Lattenkreuz `#`
Drei wichtige Anwendungen:

1. Einschluss („Inklusion“) von Header-Files
   „Header-Files“ (Dateien mit der Endung `.h`) beinhalten Konstanten und Typdefinitionen, sowie Funktionsprototypen. Grund: getrennte Übersetzung von Quelltext-Dateien



Wenn z.B. in `src1.c` ein Aufruf von `strcmp()` vorkommt, muß der Compiler beim Übersetzen über die Parameter/ Rückgabetypen Bescheid wissen. Wodurch? `#include <string.h>`. Ebenso z.B., wenn `src2.c` eine Funktion aus `src1.c` benutzt: `#include "src1.h"`
Achtung Auch `src1.c` muß `#include "src1.h"` enthalten, damit die Deklaration in `src1.h` und die Implementierung in `src1.c` nachprüfbar übereinstimmen!
Bem `#include <...>` und `#include "..."` suchen an verschiedenen Stellen nach der inkludierten Datei; beide Formen schieben den kompletten Inhalt der Datei in den Übersetzungsprozess der inkludierenden Datei ein.

2. Textersetzung durch Makros

   (a) Symbolische Namen für Konstante

   ```
   #define ANTWORT_AUF_ALLE_FRAGEN  42
   ...
   if (x == ANTWORT_AUF_ALLE_FRAGEN) { ... }
   ```

   (b) „Inlining“ von einfachen, immer wiederkehrenden Rechnungen

   ```
   #define FUNC(x) (3*(x)+1)
   ...
   c = FUNC(a) + FUNC(b);
   ```

   Achtung der Präprozessor versteht nichts von Priorität und Assoziativität:

   ```
   #define ADD1(x) x+1
   ...
   c = 2*ADD1(3); /* man erwartet 8 */
   /* ist identisch mit */
   c = 2*3+1; /* man erhält 7 */
   ```

   Konsequenz: Bei Makrodefinitionen Klammern setzen!

3. Bedingte Compilierung

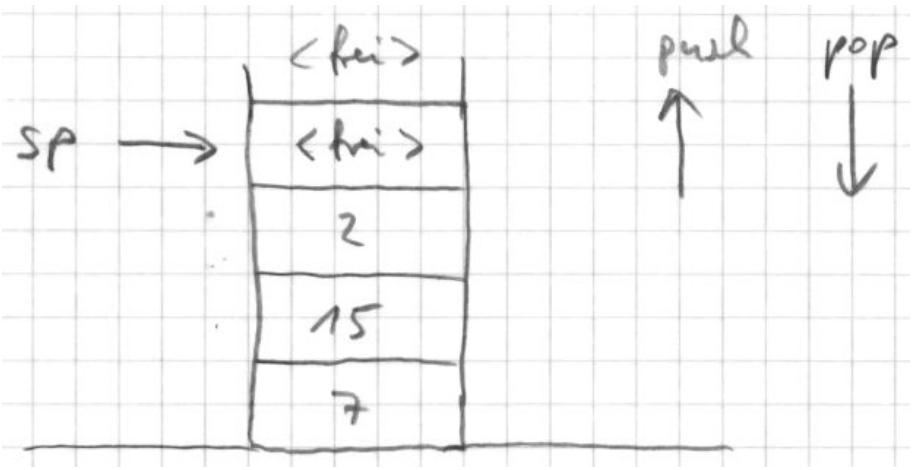
   ```
   #define LINUX
   ...
   #ifdef LINUX
   /* wird nur übersetzt, falls LINUX definiert ist */
   #endif
   ```

   Anwendung: Ausschluss von Mehrfachinklusion

   ```
   #ifndef SRC1_H_INCLUDED
   #define SRC1_H_INCLUDED
   ...
   #endif
   ```

3 Die VM

3.1 Stackmaschine



Rechenoperationen: nehmen die obersten zwei Einträge vom Stack, ermitteln das Ergebnis und legen es auf dem Stack ab.

Auswertung von $2 * 3 + 5$

```
pushc   2
pushc   3
mul
pushc   5
add
```

Auswertung von $2 * (3 + 5)$

```
pushc   2
pushc   3
pushc   5
add
mul
```

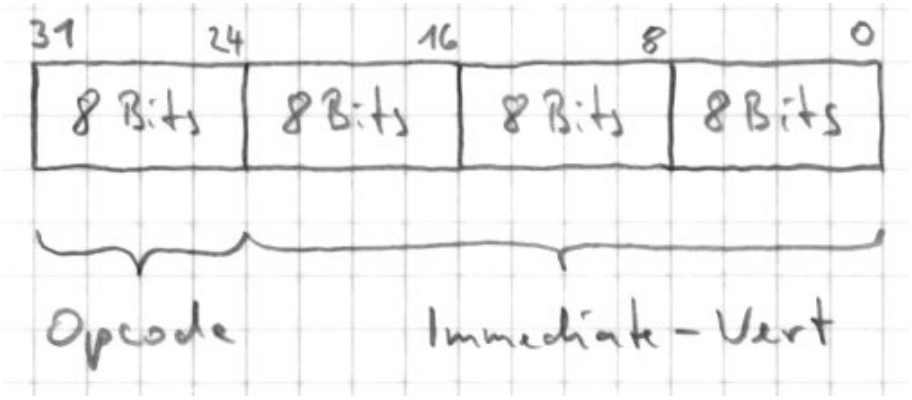
Beachte: Alle Operanden werden genau in der Reihenfolge ihres Auftretens auf den Stack gelegt!

3.2 Instruktionen für die VM

`add ... n1 n2` → `... n1+n2`
entsprechend `sub, mul, div, mod` (Rest beim Dividieren)
`pushc <n> ...` → `... n`
`halt ...` → `...` (hält die VM an)
`rdint ...` → `... n` (liest eine Ganzzahl)
`wrint ... n` → `...` (schreibt eine Ganzzahl)
`rdchr ...` → `... n` (liest ein Zeichen)
`wrchr ... n` → `...` (schreibt ein Zeichen)

3.3 Kodierung von Instruktionen

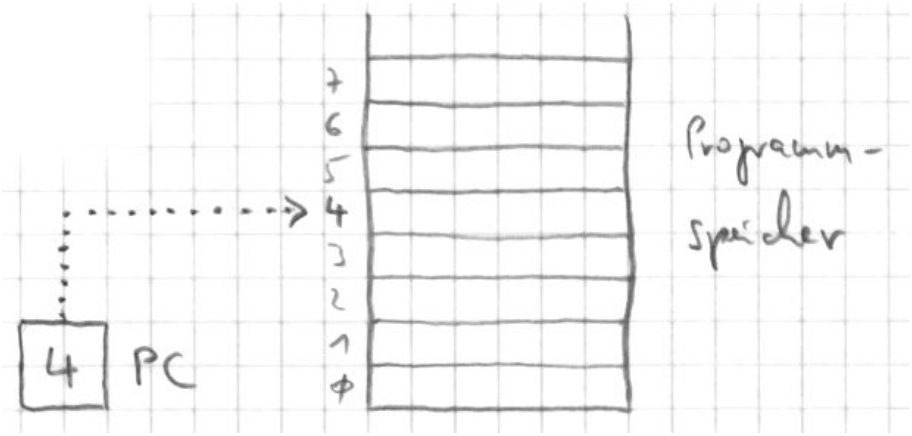
Wir stellen eine Instruktion in einer ganzen Zahl mit 32 Bits ohne Vorzeichen dar („unsigned int“):



Opcode: legt die Operation fest
Immediate-Wert: Parameter für die Operation
Bsp `pushc 5`
Der Opcode für `pushc` ist 1, also ist er Code für `pushc 5`: `(1 << 24) | 5` (`<<` linksschieben, `|` bitweises oder)
Achtung Was passiert bei `pushc -1`? (Hinweis: Zweierkomplement!)
Also besser: `(1 << 24) | (5 & 0x00FFFFFF)` (`&` bitweises und)

3.4 Programmspeicher und PC

Der PC („ProgramCounter“) ist der Index der nächsten auszuführenden Instruktion.



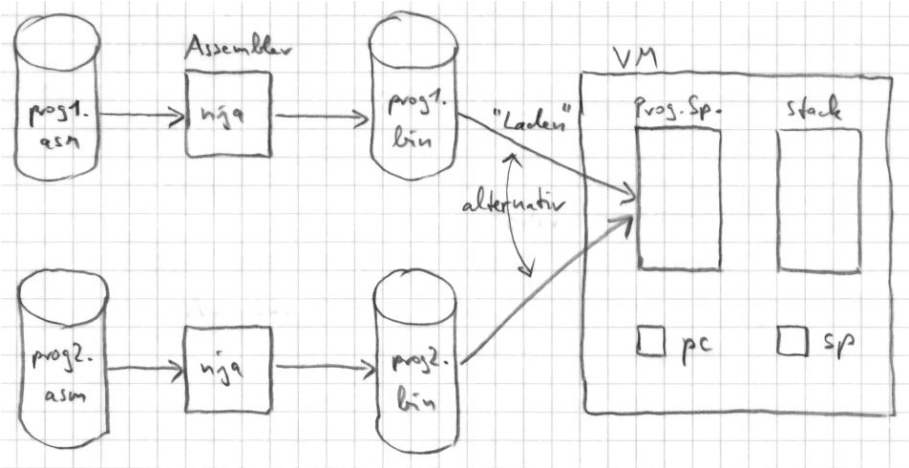
3.5 Steuerwerk/Instruktionszyklus

```
while (!halt) {
  IR = program_memory[PC]; /* Instruktion holen */
  PC = PC + 1;             /* PC inkrementieren */
  exec(IR);                /* Instruktion ausführen */
}
```

(IR heißt Instruktionsregister)

ASM

Bis jetzt: Fester Programmspeicherinhalt (”ROM”)
Ab jetzt: Laden eines beliebigen Binärprogramms in die VM



`prog1.asm` enhält Text, z.B.

```
rdint
pushc 2
mul
wrint
halt
```

`prog1.bin` enthält Binärdaten: können nicht mehr direkt angeschaut werden.

Wie kommt z.B. `prog1.bin` in den Programmspeicher?
Laden mittels Dateioperationen.

3.6 Dateioperationen

C

- Öffnen einer Datei
- Lesen/Schreiben der Datei, evtl. mit Positionieren
- Schließen der Datei

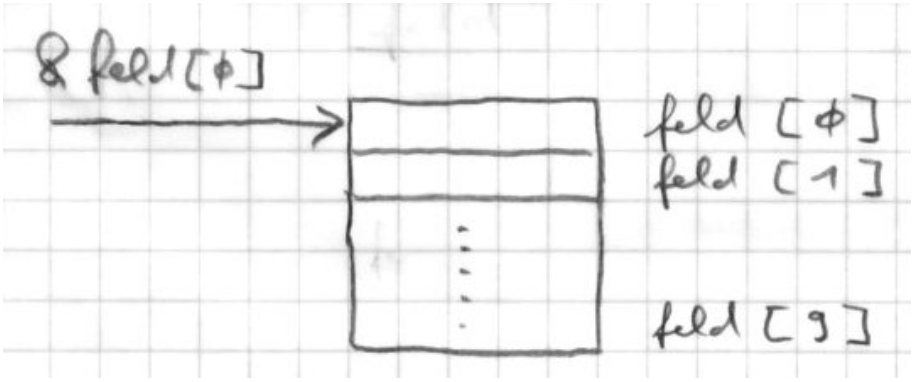
1. Öffnen
   `FILE *fopen(char *path, char *mode);`
   `char *path` sagt welche Datei, z.B. `"prog1.bin"`
   `char *mode` sagt wie die Datei geöffnet werden soll, lesen, schreiben, anhängen, z.B. `"r"` `FILE` ist eine Datenstruktur, die den Zustand der geöffneten Datei hält. Wie das genau aussieht, bleibt dem Benutzer verborgen.
   `fopen()` liefert `NULL` zurück, falls es fehlschlägt. Das muss immer überprüft werden!

2. Schließen
   `int fclose(FILE *fp);`
   leert die Schreib-/Lesepuffer, gibt die `FILE`-Struktur an die Speicherverwaltung zurück ⇒ Danach darf der Inhalt von `fp` nicht mehr benutzt werden; der Zeiger zeigt auf nicht mehr verfügbaren Speicher!

3. Lesen/Schreiben; z.B. Lesen: `size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);`
   `ptr`: Zeiger auf Puffer, vom Benutzer zur Verfügung zu stellen. Warum `void *`? `fread` soll für beliebigen Datentypen in beliebiger Anzahl funktionieren.
   `void *` ≡ Zeiger auf irgendwas
   Konsequenz: Der Compiler weiß nichts über Typ oder Größe.
   `size`: Größe eines Datenobjekts in Bytes. Diese wird durch den statisch auswertbaren Operator `sizeof` berechnet, z.B. `sizeof(int)` (=4 bei 32-Bit-Rechnern)
   `nmemb`: Anzahl der Datenobjekte, die gelesen werden sollen. Deshalb insgesamt benötigter Platz im Puffer: `size * nmemb`
   `stream`: der von `fopen` gelieferte Filepointer
   Rückgabewert: Anzahl der gelesenen Datenobjekte (Achtung: nicht Zahl der gelesenen Bytes!)
   Bsp Lesen von 10 Integern aus einer Datei `xy.z`:

```
FILE *fp;
int feld[10];
fp = fopen("xy.z", "r");
if (fp == NULL) {
  printf("...");
  exit(99);
}
if (fread(&feld[0], sizeof(int), 10, fp) != 10) {
  printf("...");
  exit(99);
}
...
```

   Bem `&` ≡ Adressoperator, gelesen Adresse von ...
   ⇒ `&feld[0]` ≡ Adresse von `feld[0]`
   Kurzschreibweise: `feld` ≡ `&feld[0]`



   Datentyp von `&feld[0]`: `int *`, Zeiger auf Integer
   Allgemein: `d` hat Typ `T` ⇒ `&d` hat Typ `T*`. Das nennt man Referenzieren (Zeigerkonstruktion). Der Umgekehrte Vorgang heißt Dereferenzieren:
   `p` hat Typ `T*` ⇒ `*p` hat Typ `T`
   Bsp `*(&feld[5])` ≡ `feld[5]` hat Typ `int`

3.7 Speicheranforderung/Freigabe

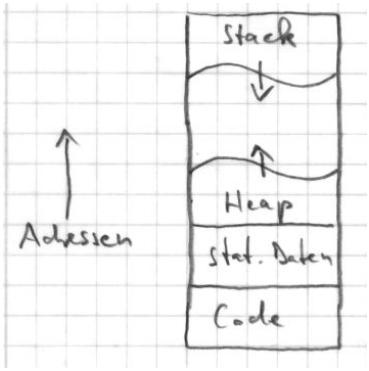
Lokale Variable in C sind automatische Variable, d.h. sie haben ihren Speicherplatz auf dem C-Laufzeitstack.
Bsp `int f(void) {int a,b; ...}`

Ihr Speicherplatz wird automatisch freigegeben, wenn die Funktion verlassen wird, in der sie deklariert sind.
Bem Deshalb ist es ein grober Fehler, einen Zeiger auf eine lokale Variable zurückzugeben:

```
int *f(void) {
  int i = 5;
  return &i; /* Das ist ein Fehler!! */
}
```

Wenn man Datenobjekte anlegen möchte, die länger leben, als die Funktion, die sie anlegt, muss das mit
`void *malloc(size_t size);`
auf dem Heap geschehen.

Adressraum eines laufenden C Programms:



Freigabe des Speichers mit `void free(void *p)`.

4 VM: Variable & Kontrollstrukturen

4.1 Variable

Eine Variable ist ein Ort, an dem ein Wert aufgehoben („gespeichert“) werden kann. In Java gibt es vier Arten von Variablen:

```
class C {
  static int clsVar;       // Klassenvariable
  int instVar;             // Instanzvariable
  void m(int paramVar) {   // Parametervariable
    int localVar;          // lokale Variable
    ...
  }
}
```

In C ebenso:

Klassenvariable	$\longleftrightarrow$	globale/statische Variable
Instanzvariable	$\longleftrightarrow$	Variable auf dem Heap
Parametervariable	$\longleftrightarrow$	Parametervariable
lokale Variable	$\longleftrightarrow$	lokale Variable

4.2 „gobal“ vs. „statisch“

„global“ bezeichnet die Sichtbarkeit der Variablen: sie sind von überall her zugreifbar
„statisch“ bezeichnet die Lebensdauer der Variablen: sie lebt (hat Speicherplatz) die ganze Programmlaufzeit
Also: global $\Rightarrow$ statisch, aber statisch $\not\Rightarrow$ global (siehe Klassenvariablen in Java)

4.3 Globale Variable, Static Data Area

Globale Variable werden in der „Static Data Area“ gespeichert. Jede Variable gibt es genau einmal. Sie wird eindeutig bezeichnet durch ihre Adresse in der Static Data Area. Zum Arbeiten mit globalen Variablen gibt es zwei Instruktionen: `push global variable` und `pop global variable`. Beide haben als Immediate-Wert die Adresse der Variablen kodiert.

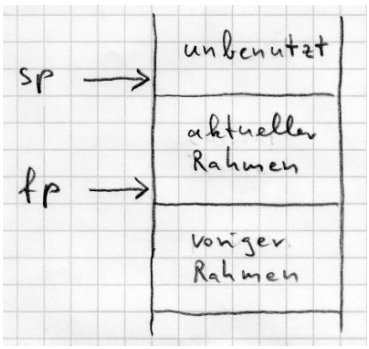
`pushg ...`	$\rightarrow$	`... wert` (legt Wert der globalen Variable auf den Stack)
`popg ... wert`	$\rightarrow$	`...` (speichert top-of-stack in globaler Variable)

Bsp Angenommen `x` soll an der Adresse 2 und `y` an der Adresse 5 in der Static Data Area gespeichert sein. Dann kann die Anweisung `x=3*x+y` berechnet werden durch die Instruktionsfolge:

```
pushc 3
pushg 2
mul
pushg 5
add
popg 2
```

4.4 Lokale Variable, Stack Frames

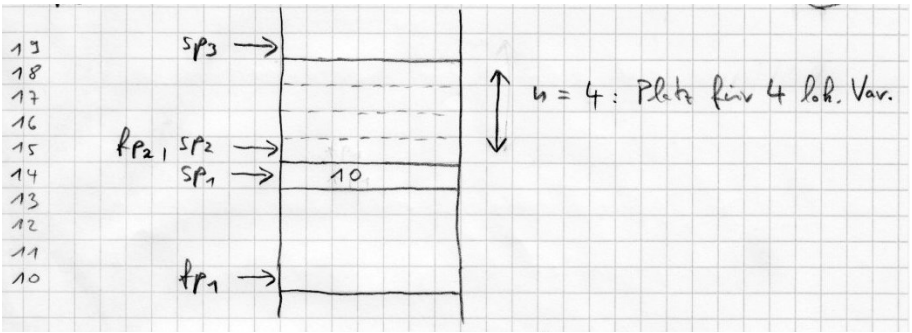
Lokale Variable (und Parametervariable) muss es getrennt für jeden Prozedur- udn Funktionsaufruf („Aktivierung“) geben (wg. Rekursion). Also können diese Variablen nicht statisch gespeichert werden, sondern bekommen ihren Platz auf dem Stack (neu für jede Aktivierung). Der Zugriff über `sp` ist schwierig, da der sich dauernd ändert. Deshalb ein neues Register der VM: `fp` (Framepointer)



Der Bereich zwischen `fp` und `sp` heißt „aktueller Rahmen“ und speichert die lokalen Variablen der momentan ausgeführten Prozedur/Funktion/Methode. Beim Aufruf einer Prozedur muss ein neuer Rahmen angelegt werden, der beim Verlassen wieder vernichtet wird. Dazu zwei neue Instruktionen: `allocate stack frame` und `release stack frame`

```
asf <n>          ≡      push(fp)
                        fp = sp
                        sp = sp + n
rsf              ≡      sp = fp
                        fp = pop()
```

Bem Also wird der „alte“ `fp` im Stack selber aufgehoben!
Bsp `asf 4`



Mit `rsf` wird der Rahmen wieder freigegeben.

4.5 Arbeiten mit lokalen Variablen

Zwei Befehle: `push local variable` und `pop local variable` Beide haben als Immediate-Wert den Abstand zu `fp` kodiert.

`pushl ...` → `... wert` (legt Wert der lokalen Variable auf den Stack)
`popl ... wert` → `...` (speichert top-of-stack in lokaler Variable)

Bsp Angenommen `x` soll an der Stelle 2 und `y` an der Stelle 5 im aktuellen Rahmen gespeichert sein. Dann kann die Anweisung `x=3*x+y` berechnet werden durch die Instruktionsfolge:

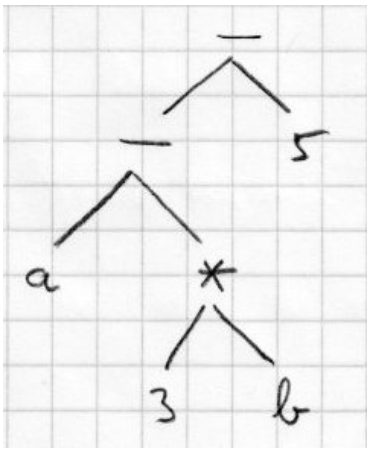
```
pushc 3
pushl 2
mul
pushl 5
add
popl 2
```

NJ und ASM und VM

4.6 Berechnung von arithmetischen Ausdrücken

Bsp $a - 3 * b - 5$
Algorithmus zum Erzeugen der Instruktionen:

1. Schreibe den Ausdruck vollständig geklammert hin (berücksichtige dabei die Priorität und Assoziativität der Operatoren)
   Bsp $((a - (3 * b)) - 4)$
2. Wandle den vollständig geklammerten Ausdruck um in einen Baum:
   (a) Jeder geklammerte Ausdruck („Operatorausdruck“) wird zu einem inneren Knoten des Baumes.
   (b) Jede Zahl und jede Variable wird zu einem Blatt des Baumes



3. Traversiere den Baum einmal (depth first, post-order)
   (a) Bei einem inneren Knoten:
      i. Traversiere den linken Teilbaum
      ii. Traversiere den rechten Teilbaum
      iii. Gib die zum Operator gehörende arithmetische Instruktion aus
   (b) Bei einem Blattknoten:
      i. Steht an dem Blatt eine Zahl gibt `pushc <zahl>` aus
      ii. Steht an dem Blatt eine Variable gib `pushl 3` oder `pushg 3` aus (wobei 3 ein Beispiel für den Abstand des Speicherplatzes zum `fp` bzw. der Adresse bei glob. Variable ist).

Bsp angenommen Abstand(a)=10, Abstand(b)=14

```
pushl 10
pushc 3
pushl 14
mul
sub
pushc 5
sub
```

4.7 Zuweisung

Bsp `b = a - 3 * b - 5`
Algorithmus zum Erzeugen der Instruktionen:

1. Erzeuge Code für die rechte Seite der Zuweisung
2. Gibt `popl 3` oder `popg 3` aus (Hinweis siehe oben).

Bsp Code für rechte Seite siehe oben

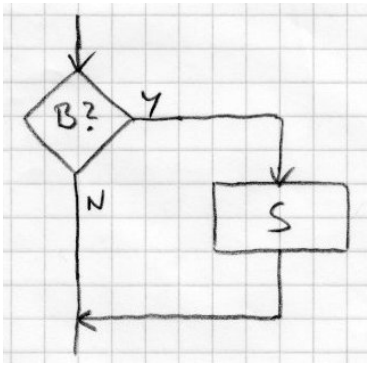
```
...
popl 14
```

NJ und ASM und VM

4.8 Kontrollstrukturen

4.8.1 Einarmiges if

```
if (B) S
```



Notation:
<B> ist die Übersetzung von B in Assembler <S> ist die Übersetzung von S in Assembler

```
  <B>      ; hinterlässt Ergebnis auf dem Stack
  brf L    ; springe zu L, wenn Ergebnis ==false
  <S>
L:         ; das nennt man ein "Label" oder "Marke",
           ; symbolische Form einer Adresse
  ...
```

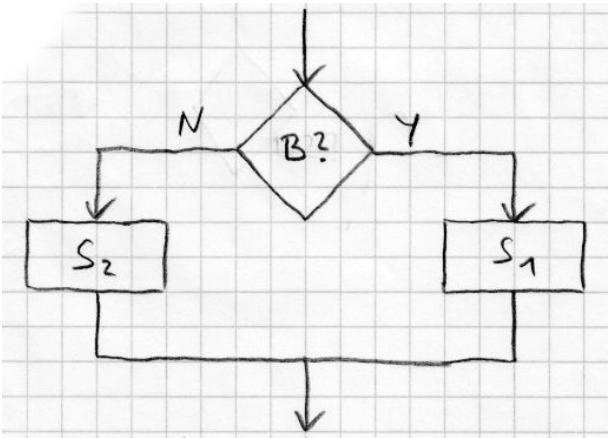
Beachte Der bedingte Sprung nimmt den Boole'schen Wert vom Stack.
Frage Wie kommt der Boole'sche Wert auf den Stack? Was ist `B`? Dazu sechs arithmetische Vergleiche:

```
a==b   →    eq ... a b   →   ... a==b
a!=b   →    ne ... a b   →   ... a!=b
a<b    →    lt ... a b   →   ... a<b
a<=b   →    le ... a b   →   ... a<=b
a>b    →    gt ... a b   →   ... a>b
a>=b   →    ge ... a b   →   ... a>=b
```

Wie wollen wir Boole'sche Werte repräsentieren? Besonders einfach: $0 \equiv$ `false`, $1 \equiv$ `true` (ganze Zahlen)

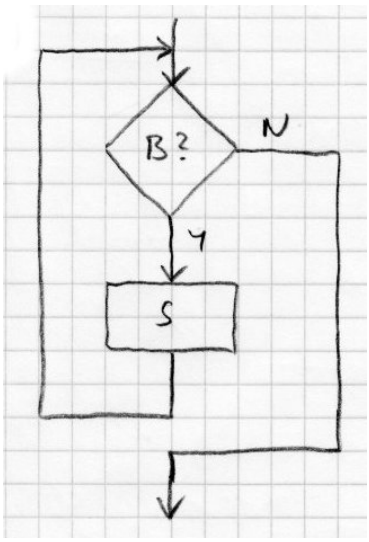
4.8.2 Zweiarmiges if

```
if (B) S1 else S2
```




```
  <B>      ; Ergebnis auf dem Stack
  brf L1   ; springe zu L1, wenn Ergebnis ==false
  <S1>     ; wird ausgeführt, falls B true
  jmp L2   ; weiter bei L2
L1:
  <S2>     ; wird ausgeführt, falls B false
L2:
  ...
```

4.8.3 while-Schleife




```
while (B) S
```

Alternative 1

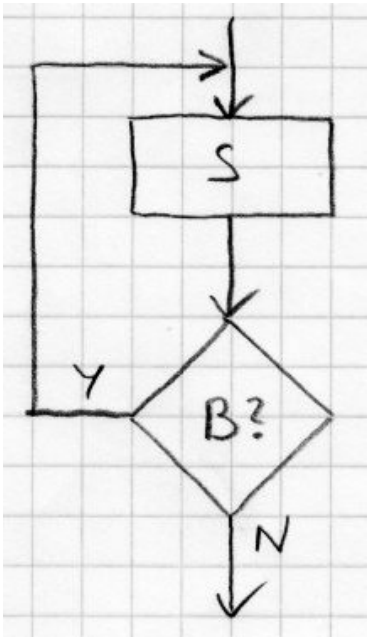
```
L1:
  <B>
  brf L2
  <S>
  jmp L1
L2:
```

Nachteil: zwei Sprünge pro Schleifendurchlauf
Alternative 2

```
  jmp L2
L1:
  <S>
L2:
  <B>
  brt L1
```

4.8.4 do-Schleife

```
do S while (B)
```



Auch hier gibt es zwei Alternativen, aber nur eine sinnvolle:

```
L:
  <S>
  <B>
  brt L
```

Zusammenfassung

1. sechs arithmetische Vergleiche liefern Boole'sche Werte
2. Sämtliche Kontrollstrukturen (mit Ausnahme des Prozeduraufrufs) werden mit `jmp/brf/brt` realisiert. Das Sprungziel wird in Assembler durch eine Marke repräsentiert; in der VM ist das die Adresse der Instruktion, die am Sprungziel steht.

4.9 Auswertung Boole'scher Ausdrücke

Sprache schreibt vor, ob „vollständige Auswertung“ oder „Kurzschlussauswertung“:

1. Vollständige Auswertung
   (a) `B1 && B2`

   ```
   <B1>
   <B2>
   and
   ```

   (b) `B1 || B2`

   ```
   <B1>
   <B2>
   or
   ```

   Nachteil: `if (x != 0 && y / x < 5) {...}` macht nicht das, was es soll!
2. Kurzschlussauswertung

(a) `B1 && B2`

```
    <B1>
    brf L1
    <B2>
    jmp L2
L1:
    pushc 0    ; false
L2:
```

(b) `B1 || B2`

```
    <B1>
    brt L1
    <B2>
    jmp L2
L1:
    pushc 1    ; true
L2:
```

Bem Ninja benutzt Kurzschlussauswertung. Der Compiler erzeugt aber eine etwas modifizierte Instruktionsfolge.

4.10 Unterprogrammaufruf und Rücksprung

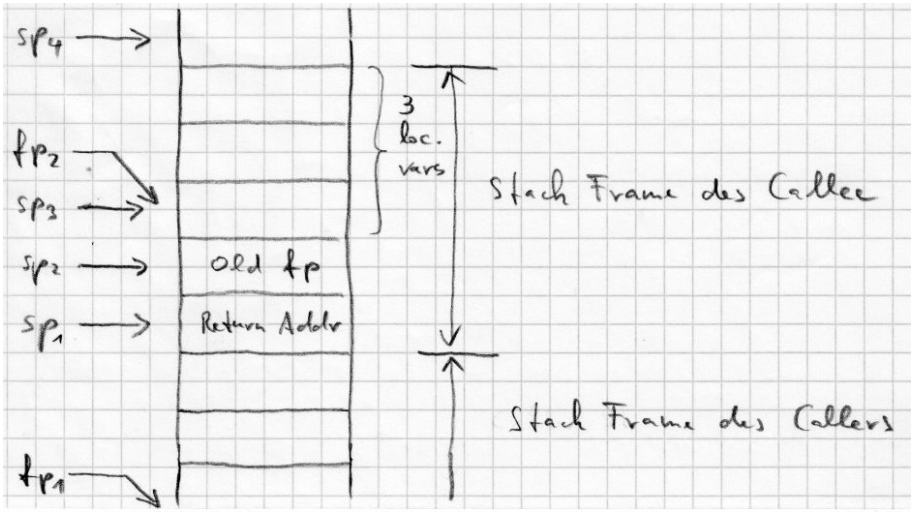
Ein Unterprogramm („Call“) muss eine Adresse des nächsten Befehls („Rückkehradresse“) speichern, damit der Unterprogrammrücksprung („Return“) dorthin zurückfindet. Speicherort ist der Stack.

```
call ...          →     ... ra
ret ... ra        →     ...
```

1. Call ohne Argumente, ohne Rückgabewert
   Caller (= aufrufende Prozedur):
   `call <proc addr>`
   Callee (= aufgerufene Prozedur):

   ```
   asf <num locals>
   <body>
   rsf
   ret
   ```

   Stackaufbau (gezeichnet nach Ausführung von `asf 3`:



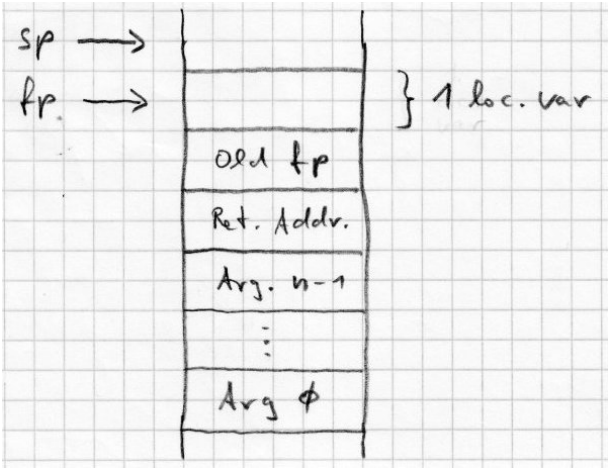
2. Call mit Argumenten, aber ohne Rückgabewert
   Die Werte von Argumentausdrücken müssen vor dem Aufruf einer Prozedur berechnet und gespeichert werden. Wo? Auf dem Stack. Wir nummerieren im Folgenden die n Argumente von 0 („1. Argument“, links) bis n-1 („n-tes Argumet“, rechts).
   Caller:

```
<push arg 0>
...
<push arg n-1>
call <proc addr>
drop <n>     ; löscht n Einträge vom Stack
```

   Callee:

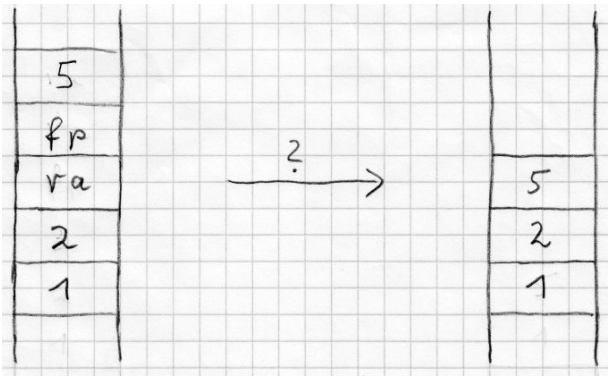
```
asf <num locals>
<body>
rsf
ret
```

   Stackaufbau:



   Zugriff auf Argument `i` (`i=0 ... n=1`) durch negativen Offset zum Framepointer:
   Argument `i` $\longleftrightarrow$ `stack[fp - 2 - n + i]`
   Der Wert von `-2 - n + i` wird als Immediate-Konstante in die Instruktionen `pushl` und `popl` kodiert: so wird auf die Parmeter einer Prozedur zugegriffen.

3. Call ohne Argumente, aber mit Rückgabewert
   Der Rückgabewert entsteht (als Wert eines Ausdrucks) auf dem Stack. Er muss auch auf dem Stack erscheinen, allerdings viel tiefer im Stack: vor der Rückkehradresse!
   Bsp `1+2*f()`, mit `f(){return 5;}`



Einfachste Lösung: Rückgabe des Wertes in einem zusätzlichen Spezialregister `r` mit zwei Instruktionen.

```
pushr ...        →     ... rv
popr ... rv      →     ...
```

Caller:

```
call <proc addr>
pushr
```

Callee:

```
asf <num locals>
<body>
<push ret val>
popr
rsf
ret
```

4. Call mit Argumenten und Rückgabewert: Kombination aus 2. und 3.
   Caller:

   ```
   <push arg 0>
   ...
   <push arg n-1>
   call <proc addr>
   drop n
   pushr
   ```

   Callee:
   genau so wie 3., Zugriff auf Argumente: genau so wie 2.

5 VM: Objekte

5.1 Objekte, Objektreferenzen

Rechenobjekte bis jetzt: Integer, Character, Boolean.
Rechenobjekte in Zukunft: Zusammengesetzte Objekte (Records und Arrays, „enthalten“ andere Objekte)
Problem: Die Lebensdauer der Objekte wird sehr oft größer sein müssen als die Ausführungsdauer der sie erzeugenden Prozeduren ⇒ Der Stack ist zum Speichern der Objekte ungeeignet! Wir brauchen einen „Heap“: ein Speicher, in dem Objekte beliebig lange leben können. Später: eigene Heap-Verwaltung mit Garbage-Collector; Jetzt: Benutzung von `malloc` und keine Benutzung von `free()`. Jedes Objekt wird durch einen Zeiger auf seinen privaten Speicherbereich identifiziert, seine Objektreferenz.

C

Wie definiert man einen Typ in C? Genauso wie eine Variable, aber mit vorangestelltem `typedef`.
Bsp

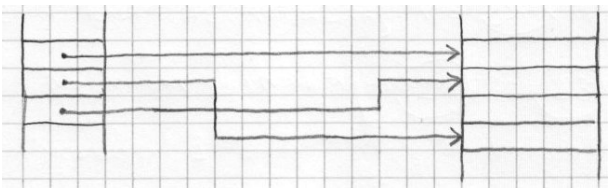
1. `unsigned int U32;` würde eine Variable mit Namen `U32` vom Typ `unsigned int` definieren
   `typedef unsigned int U32;` definiert den Typ mit Namen `U32` als Alias für `unsigned int` (aber keine Variable). Variablendefinition mit dem neuen Typ wie üblich:
   `U32 counter;`
2. `char *Messages[10];` würde eine Variable mit Namen `Messages` definieren (ein Feld mit 10 Zeigern auf `char`)
   `typedef char *Messages[10];` definiert den Typ mit Namen `Messages` als Alias für ein Feld mit 10 Zeigern auf `char`.
   Variablendefinition dann z.B.: `Messages messages;`

Konventionen zur Namensgebung bei Typen:

1. erster Buchstabe ist Großbuchstabe, z.B. `Size`
2. angehängtes `_t` am Namen, z.B. `size_t`

VM

Stack: enthält Objektreferenzen; Heap enthält Objekte



Vorschlag:

```
typedef int Object;        /* Objekt */
typedef Object *ObjRef;    /* Objektreferenz */
```

Zwei Schwierigkeiten

1. Die Objekte werden in Zukunft verschieden groß sein → Größenangabe im Objekt notwendig. Wie macht man das in C?
2. Der Stack muss auch reine Zahlen aufnehmen (`fp, ra`). Wie kann man in C den gleichen Speicher zu verschiedenen Zeiten für Daten verschiedenen Typs nutzen?

5.2 Verbunde („Records“)

C

Feld („Array“): Kollektion von Variablen gleichen Typs, Auswahl durch Zahl („Index“)
Verbung („Record“): Kollektion von Variablen möglicherweise unterschiedlichen Typs, Auswahl durch Namen („Komponente“)
Verbunde gibt es in zwei Ausprägungen:

1. Fixe Verbunde („Structs“) bieten Platz für alle der aufgezählten Komponenten:

   ```
   struct {
     char name[50];
     int tag;
     int monat;
     int jahr;
   } person;
   ```

   definiert eine Verbund-Variable `person` mit vier Komponenten. Auswahl einer Komponente durch den Punkt-Operator, Bsp: `person.jahr = 1954;`

2. Variable Verbunde („Unions“) bieten Platz für irgendeine der aufgezählten Komponenten:

   ```
   union {
     double d;
     unsigned char b[sizeof(double)];
   } inspect;
   ```

   definiert eine Verbund-Variable `inspect`, die Platz für entweder eine `double`-Größe oder ein entsprechend großes Byte-Array hat. Damit kann man z.B. die Darstellung von Fließkomma-Größen sichtbar machen:

   ```
   inspect.d = 3.1415926;
   for (i=0; i<sizeof(double); i++) {
     printf("0x%02x", inspect.b[i]);
   }
   ```

Bem Die Definition von Verbunden geschieht meist in Verbindung mit einer Typdefinition:

```
typedef struct {
  ...
  ...
} Person;        /* Typdefinition */
Person person;   /* Variablendefinition */
```

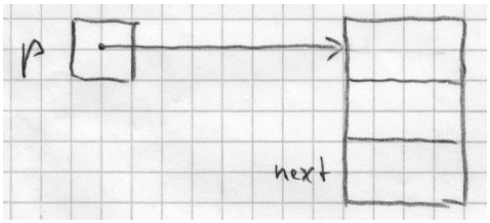
Das reicht nicht aus für rekursive Definitionen!
Bsp Verkette Liste von ganzen Zahlen

```
typedef struct {
  int number;   /* Geht nicht! Der Bezeichner "Liste" wird verwendet */
  Liste *next;  /* bevor er definiert ist. */
} Liste;
```

Deshalb gibt es sogenannte „tag names“: Namen, die nur in Verbindung mit `struct` oder `union` gültig sind.

```
typedef struct liste {
  int number;
  struct liste *next; /* "liste" ist der tag name */
} Liste;
```

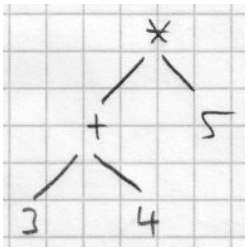
Bem Häufig werden Verbunde auf dem Heap angelegt und mittels Zeigern zugegriffen. Dann beschreibt der Ausdruck `(*p).next` die Dereferenzierung des Zeigers `p` mit anschließender Auswahl der Komponenten `next`.



Abkürzung: `(*p).next` ≡ `p->next`

5.3 Anwendungsbeispiel: Arithmetische Ausdrücke

Bsp: $(3 + 4) * 5$



Was müssen die Knoten speichern?

- innere Knoten: Operation linker Teilbaum, rechter Teilbaum
- Blattknoten: Zahl

Ein Knoten ist entweder innerer Knoten oder Blattknoten ⇒ Das wird dargestellt durch eine `union`. Wenn ein Knoten bearbeitet werden soll, muss das Programm „wissen“, ob es ein innerer oder ein Blattknoten ist ⇒ Darstellung eines Knoten:

```
typedef struct node {
  boolean isLeaf;
  union {
    struct {
      char operation;
      struct node *left;
      struct node *right;
    } innerNode;
    int value;
  } u;
} Node;
```

1. Erzeugen von Knoten auf dem Heap

```
Node *newLeafNode(int value) {
  Node *result;
  result = malloc(sizeof(Node));
  if (result == NULL) error("no memory");
  result->isLeaf = TRUE;
  result->u.value = value;
  return result;
}

Node *newInnerNode(char operation, Node *left, Node *right) {
  Node *result;
  result = malloc(sizeof(Node));
  if (result == NULL) error("no memory");
  result->isLead = FALSE;
  result->u.innerNode.operation = operation;
  result->u.innerNode.left = left;
  result->u.innerNode.right = right;
  return result;
}
```

Bsp

```
expression =
  newInnerNode(
    '*',
    newInnerNode(
      '+',
      newLeafNode(3),
      newLeafNode(4)
    ),
    newLeafNode(5)
  );
```

2. Auswerten von Knoten

```
int eval(Node *tree) {
  int op1, op2, result;
  if (tree->isLeaf) {
    result = tree->u.value;
  } else {
    op1 = eval(tree->u.innerNode.left);
    op2 = eval(tree->u.innerNode.right);
    switch (tree->u.innerNode.operation) {
      case '+':
        result = op1 + op2;
        break;
      case '-':
        ...
    }
  }
  return result;
}
```

Bsp `eval(expression);` liefert 35

3. Übersetzen von Knoten in VM-Befehle

```
void emit(Node *tree) {
  if (tree->isLeaf) {
    printf("pushc %d\n", tree->u.value);
  } else {
```

```
      emit(tree->u.innerNode.left);
      emit(tree->u.innerNode.right);
      switch (tree->u.innerNode.operation) {
        case '+':
          printf("add\n");
          break;
        case '-':
          ...
      }
    }
  }
```

Bsp `emit(expression)` liefert

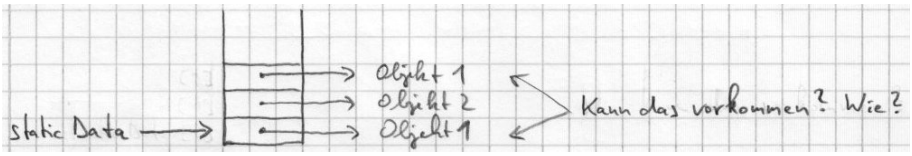
```
pushc 3
pushc 4
add
pushc 5
mul
```

VM

Der Stack ist ein statisch angelegtes Array von „Stack-Slots“:
`StackSlot stack[STACK_SIZE];`
Jeder Stack-Slot muss in der Lage sein, entweder eine Objektreferenz oder eine einfache Zahl aufzunehmen:

```
typedef struct {
  boolean isObjRef; /* slot used for object reference? */
  union {
    ObjRef objRef;  /* used if isObjRef=TRUE */
    int number;     /* used if isObjRef=FALSE */
  } u;
} StackSlot;
```

Die Static Data Area ist ein dynamisch angelegtes Array von Objektreferenzen: `ObjRef *staticData;`



Objektreferenzen zeigen auf Objekte, die selber ihre Größe und eine Variable Anzahl von Bytes speichern:

```
typedef struct {
  unsigned int size;      /* byte count of payload data */
  unsigned char data[1]; /* payload data, size as needed */
} *ObjRef;
```

Wie kann ein Interface zu diesem „Objektspeicher“ aussehen?

1. Anlegen:

```
objRef = malloc(sizeof(unsigned int) + sizeof(int));
objReg->size = sizeof(int);
*(int *)objRef->data = value;
```

2. Benutzen: `*(int *)objRef->data`

ACHTUNG Wie viele Bytes „transportiert“ die Zuweisung `n=*(int *)objRef->data` eigentlich? Antwort: Der dereferenzierte Zeiger ist ein Zeiger auf `int`, also werden `sizeof(int)` Bytes kopiert (bei uns: 4). Die Komponente `data` ist aber als Array von einem Byte erklärt - wieso funktioniert das überhaupt? Zwei Gründe:

- wir haben beim Anlegen genügend viele Bytes angelegt
- C macht keine Prüfung auf Einhalten der Arraygrenzen

Konsequenz: „size as needed“ funktioniert.

6 Richtig große Zahlen

VM

Aufgabe: Berechnung von $\sum_{i=1}^{100} \frac{1}{i}$ als exakter Bruch. Wegen $\frac{1}{a} + \frac{1}{b} = \frac{a+b}{a \cdot b}$ kann der Nenner der gesuchten Zahl in der Größenordnung von $100! \approx 10^{158}$ liegen (Zahl mit 158 Stellen) $\Rightarrow$ Ziel: Rechnen mit beliebig großen Zahlen.

1. Prinzip
   Ganze Zahlen werden in einem Stellenwertsystem mit Basis b dargestellt: $\sum_{i=0}^{n} d_i \cdot b^i$. Dabei heißen die d_i Ziffern der Zahl z, und s ist $0 \leq d_i < v \; \forall i = 0 \ldots n$. Beim Hinschreiben von z reiht man einfach die Ziffern d_i aneinander: $d_n d_{n-1} \ldots d_1 d_0$.
   Bsp $b = 10, z = 1954 = 1 \cdot 10^3 + 9 \cdot 10^2 + 5 \cdot 10^1 + 4 \cdot 10^0$
   Gerechnet wird in einem solchen System, indem man für z.B. eine Addition die zwei zu addierenden Zahlen stellenrichtig untereinander schreibt, beginnend bei d_0 stellenweise addiert und dabei einen evtl. auftretenden „Überlauf“ in die nächste Stelle frachtet.
   Subtraktion und Multiplikation gehen entspechend. Die Division erfordert „Raten“: $700020 : 876 =$? Dieses Raten muss formalisiert werden (nachzulesen in D.Knuth, The Art of Computer Programming, Vol. 2, Seminumerical Algorithms).

2. Zahlendarstellung
   Wir wählen $b = 256$: jede Ziffer der Zahl wird in einem Byte dargestellt. Die gesamte Zahl belegt dann ein genügend großes Array von Bytes. Bei einer Subtraktion können unverhersehbar viele Stellen „vernichtet“ werden (Null enthalten), die aber nicht gespeichert werden sollen $\rightarrow$ tatsächlich belegte Anzahl von Bytes mit abspeichern! Negative Zahlen? $\rightarrow$ Voreichen/Betrags-Darstellung!
   $\Rightarrow$ Datentyp für beliebig große Zahlen in C:

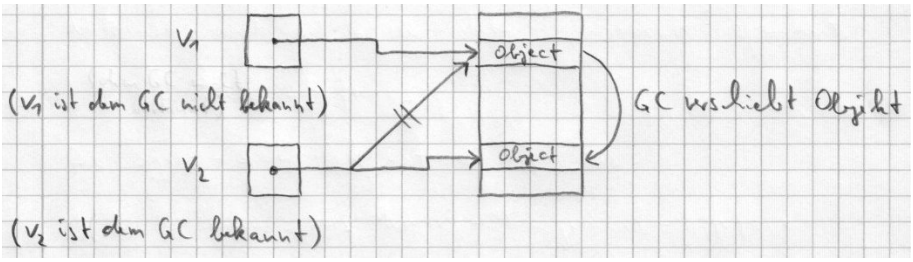
```
typedef struct {
  int ng;                    /* number of digits */
  unsigned char sign;        /* sign */
  unsigned char digits[1];   /* the digits */
} Big;
```

   (Für Details, z.B. „was ist das Vorzeichen von 0?“ oder „darf die höchste Ziffer 0 sein?“ siehe Paket `bigint.tar.gz`!)
   Bem Der Verbund `Big` ist in unseren Objekten die „Nutzlast“ und wird in der `data`-Komponente gespeichert.

3. Bibliotheksinterfaces
   Die Bigint-Bibliothek hat zwei Interfaces:

   (a) Nach „unten“: das Memory-Management-Interface
   Neben einer Funktion zur Fehlerausgabe (und Anhalten des Programms) benötigt die Bibliothek eine Funktion, mit der ein neues Objekt angelegt wird:
   `ObjRef newPrimObject(int dataSize);`
   wobei mindestens `dataSize` Bytes im Objekt zur freien Verfügung stehen müssen und eine Objektreferenz zurückgegeben wird.

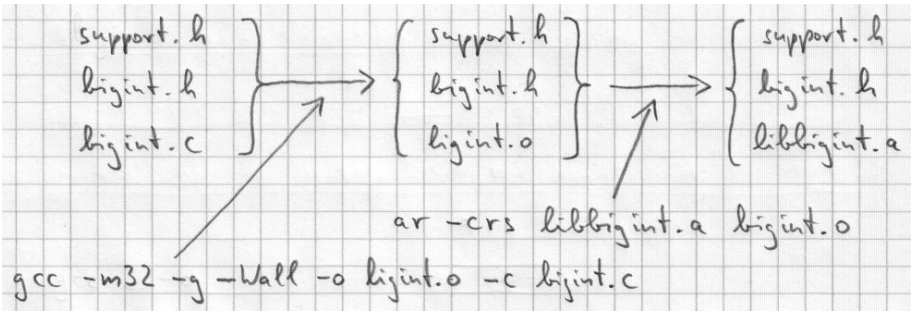
   (b) Nach „oben“: das Benutzer-Programmierinterface
   Hier werden die Arithmetischen Operationen zur Verfügung gestellt. Man würde also etwa für die Addition erwarten:
   `ObjRef bigAdd(ObjRef op1, ObjRef op2);`
   Achtung Das ist wegen der Garbage-Collection nicht möglich! Warum? `bigAdd()` wird für das Resultat ein neues Objekt anlegen und deshalb `newPrimObject()` aufrufen. Wenn der Speicherplatz dafür nicht ausreicht, wird eine Garbage Collection durchgeführt. Dabei können alle Objekte im Speicher verschoben werden!! Also werden alle Objektreferenzen ungültig!! Was tun? Objektrefenzen dürfen nur in Variablen aufgehoben werden, die der Garbage Collector kennt und berichtigt!!



Konsequenz: Es gibt eine globale, strukturierte Variable `bip` („big integer processor“), in die die Argumente von `bigAdd()` gespeichert werden (das sind Objektrefenzen). Dann wird die Funktion aufgerufen. Dabei gibt es möglicherweise eine GC und die Referenzen ändern sich. Das ist OK, da `bip` dem GC bekannt ist. Das Ergebnis der Operation (ebenfalls eine Objektrefenzen) steht danach ebenfalls in `bip`.
Details entnehmen Sie bitte dem Paket `bigint.tar.gz`

4. Einbinden der Bibliothek
   Man könnte den Quelltext der Bibliothek als ein weiteres Modul zur Ninja VM hinzufügen. Da die Bibliothek aber eine genau umgrenzte Funktionalität hat und nicht geändert werden sollte, ist es besser, sie auch formal als Bibliothek („Library“) zu behandeln.

   (a) Erzeugen der Bibliothek



   `gcc -g -Wall -o bigint.o -c bigint.c` und dann
   `ar -crs libbigint.a bigint.o`
   Bem `ar` ist der „Archiver“ (besser wäre „library manager“). Eine „Library“ ist eine Sammlung von Objekt-Dateien (`.o`) und ha einen Namen, der mit `lib` beginnt und mit `.a` endet.

   (b) Benutzung der Bibliothek

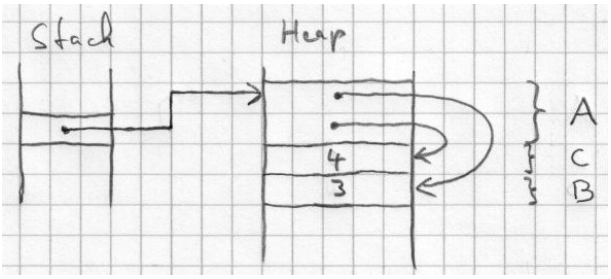
      i. Module, die Funktionen aus der Bibliothek benutzen wollen, müssen die entsprechenden Header inkludieren. Muss man die in sein Entwicklungsdirectory kopieren? Nein! Directory für Header bekannt machen mit z.B. `gcc -I ../build/include ...`
      ii. Beim Binden muss sowohl das Directory angegeben werden, wo die Bibliothek steht, als auch das Einbinden selbst: `gcc -L ../build/lib ... -lbigint`
      Achtung, Namenskonvention: Einbinden von `libbigint.a` mit `-lbigint`!

7 Objekte in Objekten

7.1 Objekte

VM

Objekte „beinhalten“ andere Objekte, indem sie Variable zum Speichern von Objektreferenzen bereitstellen:



Bsp `A` ist Instanz eines „Punktes“ mit den Koordinaten `B` und `C` (`B` und `C` sind „primitive Objekte“: nicht zerlegbar).
Objekte gibt es in zwei Varianten:

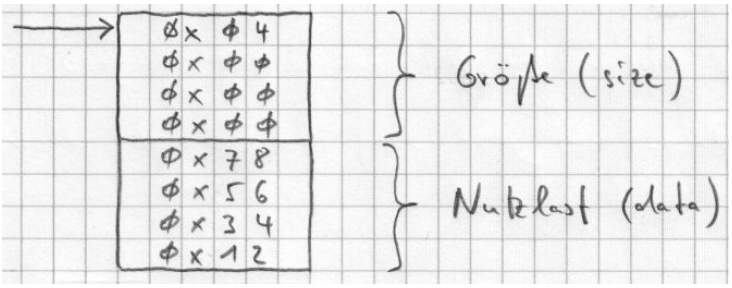
1. „Record“ (in OO-Sprachen „Instanz einer Klasse“ genannt)
   Merkmal: Zugriff auf Variable durch Namen
2. „Array“ (heißt in OO-Sprachen genauso)
   Merkmal: Zugriff auf Variable durch (berechneten) Index

Beiden ist gemeinsam: sie speichern (u.U. viele) Objektreferenzen

7.2 Objekte: Darstellung

Achtung wir haben ein Problem: es gibt zwei ganz verschiedene Sorten von Objekten (primitive und zusammengesetzte)! Wie halten wir die auseinander?
Bsp

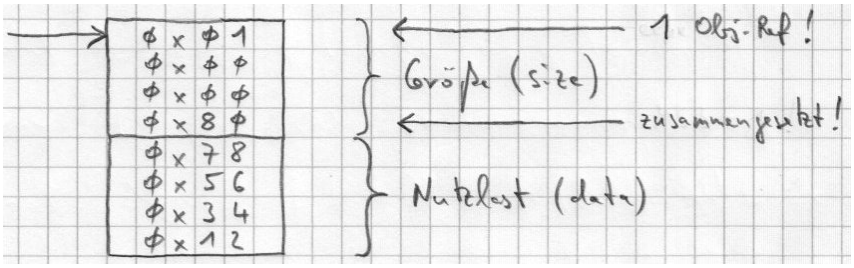
1. ein primitives Objekt mit der Zahl `0x12345678`
2. ein Record mit einer Instanzvariablen, die zufällig mit der Objektreferenz (=Adresse) `0x12345678` belegt ist. Beide sehen im Speicher einer Little-Endian-Maschine so aus:



Wir vereinbaren:

(a) Primitive Objekte bleiben so, wie oben beschrieben.

(b) Bei zusammengesetzten Objekten (Records und Arrays) wird das höchste Bit der Größe auf 1 gesetzt, und die restl. Bits zählen die Objektreferenzen, nicht die Bytes!
Bsp Der Record von oben sieht damit so aus:



Hier ein paar nützliche Makros:

(a) Ist das Objekt ein primitives Objekt?

```
#define MSB              (1 << (8 * sizeof(unsigned int) - 1))
#define IS_PRIM(objRef) (((objRef)->size & MSB) == 0)
```

(b) Wie viele Objektreferenzen beinhaltet das Objekt?

```
#define GET_SIZE(objRef) ((objRef)->size & ~MSB)
```

(c) Berechne einen Zeiger auf die erste Objektreferenzen in Objekt!

```
#define GET_REFS(objRef) ((ObjRef *)(objRef)->data)
```

7.3 Objekte: Referenzen

NJ

Definition:

```
type Point = record {
  Integer x;
  Integer y;
};
```

Erzeugen:

```
local Point p;
p = new(Point);
```

Zugriff:

```
k = p.x;
p.y = 2 * k;
```

ASM VM

Erzeugen: `new <n> ... → ... object` (`n` ist die Anzahl der Objektreferenzen im Objekt; oben: 2)
Zugriff: `(...).x` Objekt und Komponente („Instanzvariable“)

Das Objekt entsteht als Ergebnis einer Berechnung auf dem Stack. Die Komponente gibt an, wo im Objekt der Zugriff erfolgen soll: es ist also die Nummer der Instanzvariable (oben: `x`→0, `y`→1). Der Name und damit die Nummer ist beim Übersetzen bekannt; sie wird als Immediate-Wert in der Instruktion kodiert:

```
getf <i> ... objekt          →     ... value („get field“)
putf <i> ... objekt value    →     ... („put field“)
```

7.4 Objekte: Arrays

NJ

Definition:

```
type Vector = Integer[];
```

Erzeugen:

```
local Vector v;
v = new(Integer[2*n+1]);
```

Komplikation: Anzahl der Elemente zur Laufzeit!

Zugriff:

```
k = v[i];
v[n-i] = 2 * k;
```

Komplikation: Indexberechnung zur Laufzeit!

ASM VM

Erzeugen: `newa ... nelem` → `... array` (`nelem` ist die Anzahl der Objektreferenzen im Array)
Zugriff: `(...)[...]` Objekt und Index

Beides (Objekt und Index) sind Ergebnisse von Berechnungen, liegen also auf dem Stack!

`getfa ... array index` → `... value` („get field of array“)
`putfa ... array index value` → `...` („put field of array“)

Wir haben also dynamische Arrays → man sollte zur Laufzeit die Größe eines Arrays feststellen können!

NJ

Wir vereinbaren also für beliebige Objekte

$$\texttt{sizeof(...)} = \begin{cases} -1 & \text{für ein primitives Objekt} \\ \text{Anzahl der Objektreferenzen} & \text{für ein zusammengesetztes Objekt} \end{cases}$$

Beachte den Unterschied:
In C wird `sizeof(...)` statisch ausgewertet (Compilezeit)
In Ninja wird `sizeof(...)` dynamisch ausgewertet (Laufzeit)

ASM VM

`getsz ... object` → `... size` („get size“)

Implementierung ist einfach, da jedes Objekt seine Größe speichert (aber Achtung: primitive Objekte sollen -1 liefern!)

7.5 Nil, Initialisierungen, Laufzeittests

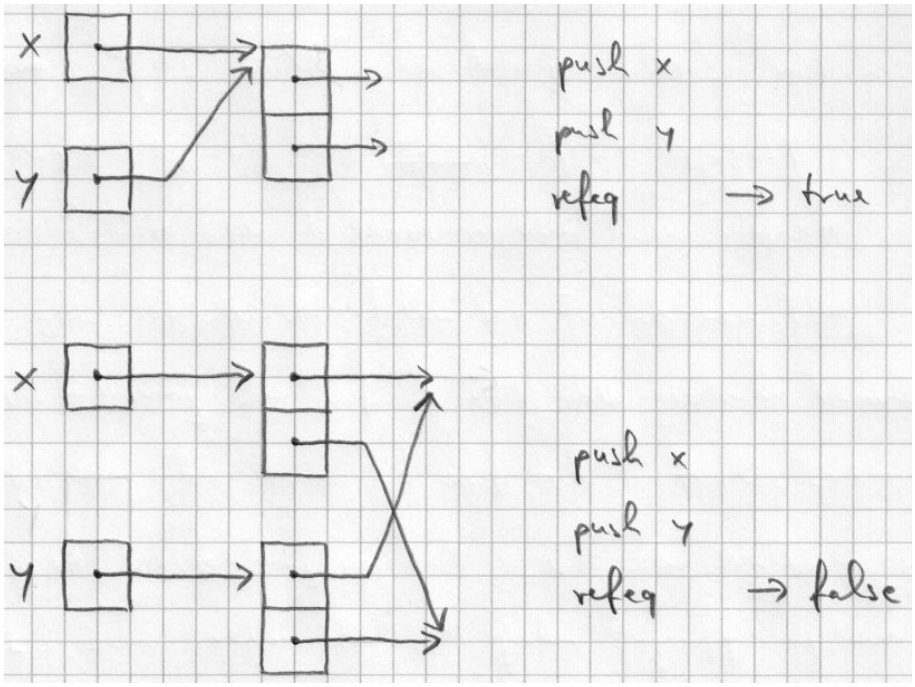
In Java gibt es `null`, in Ninja gibt es `nil`: eine Referenz, die auf kein Objekt verweist. Die VM sollte alle Instanzvariablen neuer Objekte sowie alle lokalen Variablen von Funktionen (bei Ausführung von `asf`) und alle globalen Variablen (direkt nach dem Anlegen) mit `nil` initialisieren. Einfachste Darstellung von `nil` in C: der Nullzeiger `NULL`. Dann sollte die VM jeden Zugriff auf ein Objekt prüfen und mit einem Fehler abbrechen, wenn die Objektrefenz `nil` ist. Zusätzlich muss die VM bei Zugriffen auf Arrays den Index prüfen ($0 \leq i <$ `size`) und ggf. mit Fehler abbrechen. Sie kann dies auch bei Records tun (wenn der Compiler den Code erzeugt hat, ist das aber unnötig).

7.6 Referenzvergleiche

Um Bedingungen wie z.B. `x==nil` berechnen zu können, brauchen wir 3 neue Instruktionen:

`pushn ...`	→	`...` `nil` legt `nil` auf den Stack
`refeq ... x y`	→	`...` `b` prüft Referenzen auf Gleichheit
`refne ... x y`	→	`...` `b` prüft Referenzen auf Ungleichheit

Bem Mit den beiden letzten Instruktionen kann man Objekte auf Identität prüfen:



8 Speicherverwaltung

8.1 Aufgaben

8.1.1 Speicherallokation

Findet statt bei der Erzeugung von Objekten. Methoden:

1. Freiliste: alle freien Bereiche sind verkettet. Vorwiegend verwendet bei konstanter Speichergröße, sonst sehr aufwändig (siehe `malloc` aus C)
2. Freibereich: der freie Bereich des Speichers ist stets zusammenhängend. Das erfordert „Kompaktierung“ des Speichers

Wir haben unterschiedlich große Objekte → Methode 2

8.1.2 Speicherkompaktierung

Wird zur Erzeugung eines zusammenhängenden Freibereichs durchgeführt. Methoden:

1. separater Kompaktierungslauf
2. zusammen mit Speicherfreigabe (bevorzugt)

8.1.3 Speicherfreigabe

Prinzipielle Möglichkeiten:

- explizit (d.h. programmiert: C, C++)
- implizit (d.h. automatisch: Java)

Explizite Speicherfreigabe birgt ein hohes Risiko der falschen Benutzung („dangling pointer problem“)!
Methoden zur impliziten Speicherfreigabe:

1. Referenzzähler
   Idee: Jedes Objekt beinhaltet einen Zähler, dessen Wert die Anzahl der auf das Objekt zeigenden Zeiger ist. Fällt der Wert auf 0: Speicherplatz ist frei
   Vorteil: gleichmäßige Verteilung des zeitlichen Aufwands
   Nachteile:
   - Aufwand ist hoch: aus jeder Zuweisung `p1=p2;` wird `decRef(p1); p1=p2; incRef(p1);`
   - Speicherverwaltung ist über das gesamte Programm verteilt
   - Zyklische Strukturen werden nicht freigegeben, obwohl sie nicht mehr referenziert werden.
2. Müllsammeln (Garbage Collection, GC)
   Idee: Zu bestimmten Zeiten (z.B. wenn der freie Speicher knapp ist) wird ermittelt, welche Objekte noch zugreifbar sind - ausgehend von den Registern und dem Stack der Maschine. Alle anderen Objekte werden freigegeben.
   Vorteile:
   - Speicherverwaltung ist sauber abgegrenzt vom Rest der Maschine
   - Zyklische Strukturen werden gesammelt

8.2 Verfahren zum Müllsammeln

8.2.1 Mark-Sweep-Verfahren

Dieses sind nicht-kompaktierende Verfahren. Sie laufen in zwei Phasen ab:

1. Phase „Mark“: Alle erreichbaren Objekte werden markiert (d.h. ein Flag im Objekt wird gesetzt)
2. Phase „Sweep“: Das ist ein Durchgang durch alle Objekte. Der Speicherplatz von nicht markierten Objekten wird freigegeben (und das Markierungsflag bei allen Objekten zurückgesetzt)

Die Mark-Sweep-Verfahren unterscheiden sich nur in der Implementierung der Mark-Phase.

8.2.1.1 Rekursiver Depth-First-Durchlauf

```
void mark(ObjRef p) {
  if (p->markFlag) return;
  p->markFlag = TRUE;
  for (each ObjRef q in the object pointed to by p) {
    mark(q);
  }
}
```

Vorteil: Sehr einfache Implementierung
Nachteil: Platzbedarf des Stacks für Rekursion groß (schlechtester Fall: genau so groß wie der gesamte Heap!) → unbrauchbar.

8.2.1.2 Iterativer Depth-First-Durchlauf mit Zeigerumkehr (Dentsch, Schorr, Waite 1965)

Idee: Die Information, wohin man im Graphen „zurück“ muss, wird nicht im Stack, sondern im Graphen selbst gespeichert.
Bsp Binärbaum
Das ist das bei konstanter Objektgröße vorwiegend verwendete Verfahren.

8.2.2 Kopier-Verfahren

Dieses sind kompaktierende Verfahren, die einen zusammenhängenden Freibereich erzeugen.

8.2.2.1 Stop & Copy (Minsky, Fenichel, Yochelson, 1969)

Der Arbeitsspeicher wird in zwei Hälften geteilt:
„Ziel-Halbspeicher“: dort werden die Obejkte allokiert
„Quell-Halbspeicher“: ist unbenutzt
System läuft, bis ein Objekt allokiert werden soll, das zu groß für den verbleibenden Platz im Ziel-Halbspeicher ist. Situation dann:
Jetzt wird der GC gestartet. Aktionen:

1. Flip (Vertauschen der Halbspeicher)
2. Kopieren der aus den Registern (und dem Stack) der VM verwiesenen Objekte (sog. „Root Objects“)
3. Kopieren aller anderen zugänglichen Objekte (der sog. „lebenden“ Objekte); diese Phase des GC heißt „Scan“.

Problem: Wenn ein Objekt kopiert wurde, wird es Zeiger sowohl im Quell- als auch im Ziel-Halbspeicher geben, die noch auf das alte Objekt zeigen. Wie werden diese Zeiger auf das kopierte Objekt geändert?
Idee: Ein kopiertes Objekt wird gekennzeichnet (durch das „Broken-Heart-Flag“) und in ihm wird die Adresse des kopierten Objektes gespeichert (der sog. „Forward-Pointer“). Während der Scan-Phase werden alle Zeiger korrigiert.

Algorithmus zum Umwandeln eines Zeigers auf ein Objekt im Quellspeicher in einen Zeiger auf das entsprechende Objekt im Zielspeicher:

```
ObjRef relocate(ObjRef orig) {
  ObjRef copy;
  if (orig == NULL) {
    /* relocate(nil) = nil */
    copy = NULL;
  } else if (orig->brokenHeart) {
    /* Objekt ist bereits kopiert, Forward-Pointer gesetzt */
    copy = orig->forwardPointer;
  } else {
    /* Objekt muss noch kopiert werden */
    copy = copyObjectToFreeMem(orig);
    /* im Original: setze Broken-Heart-Flag und Forward-Pointer */
    orig->brokenHeart = TRUE;
    orig->forwardPointer = copy;
  }
  /* Adresse des kopierten Objektes zurück */
  return copy;
}
```

Situation nach dem Kopieren der Root-Objekte:

Algorithmus für die Scan-Phase:

```
scan = zielhalbspeicher;
while (scan != freizeiger) {
  /* es gibt noch Objekte, die gescannt werden müssen */
  if (Objekt enthält Zeiger) {
    for (jeder Zeiger q im Objekt, auf das scan zeigt) {
      scan->q = relocate(scan->q);
    }
  }
  scan += Größe des Objektes, auf das scan zeigt;
}
```

Am Ende der Scan-Phase befinden sich alle lebenden Objekte im Ziel-Halbspeicher und alle Verweise innerhalb von Objekten zeigen in diesen Teil des Speichers.

Bem

1. Die Scan-Phase macht einen Breadth-First-Durchgang durch alle Objekte
2. Nach dem Kopieren eines Objektes ist dessen Inhalt irrelevant; das Broken-Heart-Flag und der Forward-Pointer können ihn bedenkenlos überschreiben.
3. In unserer VM eignet sich das zweithöchste Bit von `size` als Broken-Heart-Flag und die restlichen 30 Bits als Forward-Pointer, gespeichert als Byte-Offset relativ zum Beginn des Halbspeichers.

Nachteil des Stop & Copy-Verfahrens: Während eines GC-Laufs halten alle anderen Berechnungen an.

8.2.2.2 GC von Baker (1978)

Idee: Verteilen der Rechenzeit für die GC (bei jeder Objekt-Allokation wird ein kleiner Teil der Scan-Phase abgearbeitet). Aufteilung des Ziel-Halbspeichers:
Komplikation: Broken-Heart-Objekte zu jeder Zeit vorhanden!

8.2.2.3 GC von Liebermann und Hewitt (1980)

Beobachtung: Bei Baker's GC bleibt die Lebensdauer der Objekte unberücksichtigt.
Idee: Der mittlere Aufwand zur Erzeugung eines Objektes wird geringer, wenn es gelingt die kurzlebigen Objekte „billiger" zu machen.
Realisierung: Aufteilen des Speichers in mehrere kleine „Baker-Regionen". Objekte ähnlichen Alters sin in der gleichen Region. Jüngere Regionen werden öffter entmüllt als ältere.
Vorteil: Sehr viel weniger Kopierarbeit als bei Baker.
Komplikation: Objektreferenzen über die Regionengrenzen hinweg müssen behandelt werden.

9 Lose Enden

9.1 Zwei Bedeutungen des Schlüsselwortes `static`

1. Bei Variablendeklarationen innerhalb von Funktionen („lokalen Variablen“) bewirkt `static` das Anlegen des Speicherplatzes für die Variable im (statischen) Datensegment, nicht im Stack (wie sonst bei lokalen Variablen). Damit bleibt der Speicherplatz und der Wert darin über den Aufruf der Funktion hinaus erhalten.
   Bsp Eine Funktion soll mitzählen, wie oft sie aufgerufen wurde, ohne eine globale Variable zu benutzen.

   ```
   void f(void) {
     static int n = 0;
     n++;
     printf("ich wurde %d mal aufgerufen \n", n);
   }
   ```

   Achtung: Initialisierung der statischen Variablen passiert nur einmal.

2. Bei Variablendefinitionen außerhalb von Funktionen („globale Variable“) und bei Funktionsdefinitionen beschränkt `static` die Sichtbarkeit des Namens auf die Quelltextdatei, in der die Definition steht. Verwendung: Verbergen von nur lokal gebrauchten Namen.

   ```
   #include <stdio.h>
   static int counter;
   static int f(int x) { ... }
   void doIt(int what) { ... }
   ```

   Nach außen ist nur die Funktion `doIt()` sichtbar.

9.2 Zeigerarithmetik

Wenn `p` ein Zeiger auf ein Objekt vom Typ `T` ist und `n` eine ganze Zahl, dann ist `p±n` ein Zeiger auf ein Objekt vom Typ `T`, das sich `n` Objekte (nicht Bytes!) weiter hinten (also vorne) im Speicher befindet.

Natürlich können die so errechneten Zeiger auch dereferenziert werden: `*(p+2)` liefert das Objekt `a[4]` (wenn `p` auf den Index 2 zeigt).
Erinnerung: `a` ≡ `&a[0]`, also ein Zeiger aufs erste Element.
⇒ `a[n]` ≡ `*(a+n)` und `&a[n]` ≡ `a+n`. Das gilt für beliebige Zeiger auf `a`!

Typische Anwendung: Mitführen von Zeigern statt Indizierung.
Bsp Kopierschleife: `while(n--) *q++ = *p++;`

9.3 Funktionszeiger

Wenn `T f(..) {..}` eine Funktionsdefinition ist, dann ist `f` ein Zeiger auf diese Funktion. Dieser Zeiger kann als Argument in eine Funktion hineingereicht werden, oder auch in einer Variable abgespeichert werden: `p=f`; Dabei muss die Variablendeklaration so aussehen: `T (*p)(..);`. Soll die an `p` hängende Funktion aufgerufen werden, schreibt man `(*p)(...)`.

Typische Anwendung: Auswahl einer Funktion aus einer Menge von Funktionen zur Laufzeit.

```
int inspect(int arg) {...}
int simulate(int arg) {...}
...
typedef struct {
```

```
  char *name;
  int (*fkt)(int arg);
} Command;
Command cmdlist[] = {
  {"anzeigen", inspect},
  {"simulieren", simulate},
  ...
};
int execute(char *cmdname, int arg) {
  int i;
  for (i  =0; i < sizeof(cmdlist)/sizeof(cmdlist[0]); i++) {
    if (strcmp(cmdname, cmdlist[i].name) == 0) {
      /* command found */
      return (*cmd[i].fkt)(arg);
    }
  }
  /* command not found */
  error("command not found");
  return -1;
}
```

9.4 Funktionen mit variabel vielen Argumenten

Deklaration mit drei Punkten in der Parameterliste, z.B. `int printf(char *fmt, ...);`
Zugriff auf die Argumente innerhalb der Funktion durch einen Satz von Makros, definiert in `<stdarg.h>`.

Typische Verwendung: eigene Fehlerausgabe, soll so verwendet werden können wie `printf`.

```
void error(char *fmt, ...) {
  va_list ap;
  va_start(ap, fmt);
  printf("Error: ");
  vprintf(fmt, ap);
  printf("\n");
  va_end(ap);
  exit(1);
}
```

Aufruf z.B. mit `error("unbekannte Instruktion %x an Adresse %x", code[pc], pc);`