Konzepte

Systemnaher

Programmierung

Warum KSP?

- Embedded Systems
- Hochperformante Systeme bei knappsten Ressourcen (Speicherplatz, Ausführungszeit)
- Programme nahe am (Realzeit-) Betriebssystem, manchmal völlig ohne BS direkt auf der Hardware
- Häufigste eingesetzte Sprache: C
- Plattformabhängige Realisierung von plattformunabhängigen Konzepten durch "Virtuelle Maschinen" (Bsp: Java)

Themen

- C-Programmierung
- Speicherverwaltung, Laufzeitorganisation von Programmen
- Bibliotheken
- ~~Betriebssysteme Schnittstelle~~
- Werkzeugkette (Compiler, Assembler, Binder, Lader)
- Garbage-Collectoren
- Interpreter, Virtuelle Maschinen

Aufbau der Veranstaltung

Zwei Alternativen:

a) linear: 4 Vorl. C-Progr., 3 Vorl. Werkzeugkette, ...

b) projektzentriert: anhand eines Projektes lernt man alle notwendigen Konzepte, Tools, Verfahren etc. kennen

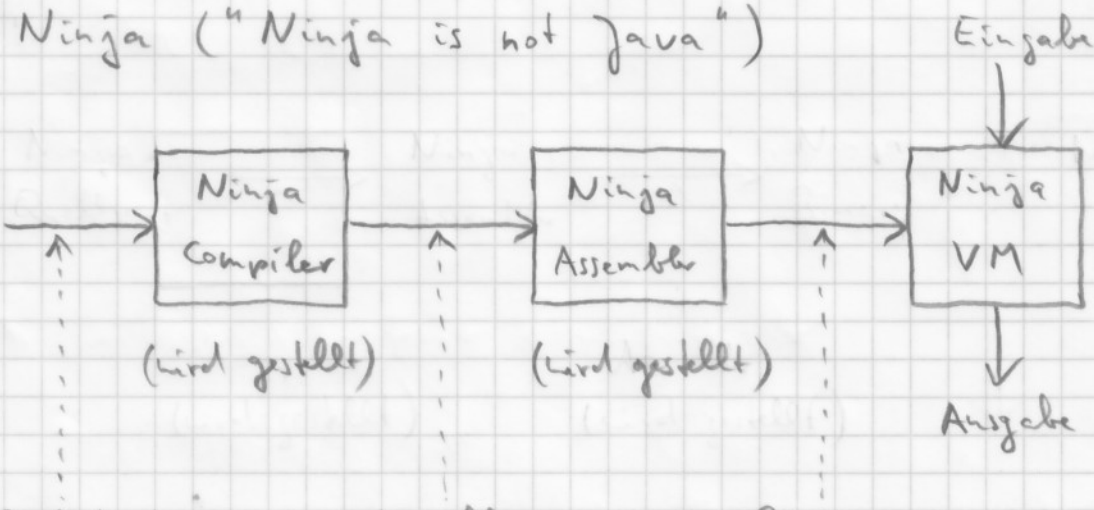
Vorteil a): sehr systematisch

Vorteil b): man lernt, warum die Dinge so sind, wie sie sind und es macht mehr Spaß!

Meine Wahl: b)!

Projekt

Konstruktion einer virtuellen Maschine zur Ausführung von übersetzten Programmen der kleinen Programmiersprache Ninja ("Ninja is not Java")



Bem.: a) Das ist in kleinem Maßstab auch das, was bei Java passiert.

b) Konstruktion "bottom-up": Wir beginnen mit einer minimalen VM und nehmen Features dazu, wenn benötigt

Formalia

Bearbeitung des Projektes in Zweiergruppen

Beginn d. Praktikums: 15.10. (heute!)

Vorbereitung: Linux installieren ~~[illegible]~~ (Tipp: Ubuntu)

Zeitaufwand: ca. 1 Tag/Woche

Zwei akzeptierte Hausübungen sind Voraussetzung für Klausur

Deadlines beachten! Einzelblätter und Skript hier:

http://homepages.thm.de/~hg53/ksp-ws1516

WARNUNG: Kopieren von Code anderer Gruppen ist Betrug und führt zum Ausschluß aller beteiligter Gruppen von der Klausur! Siehe "Plagiatskompaß" der THM!

Literatur

1) Brian W. Kernighan, Dennis M. Ritchie:
The C Programming Language, 2nd ed., Prentice Hall 1988

Drei weitere gute Bücher auf der oben genannten Website

Symbole zur Einordnung des Lehrstoffs

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

C

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 aber Verbunde)
- C kennt keine Referenzen (dafür aber explizite Zeiger)
- C hat keine automatische Speicherbereinigung

  ⇒ Speichermanagement liegt beim Programmierer

Ein "C-Programm" ist 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 (unspezifizierter 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 zurückgibt

Bedeutung der Parameter / Rückgabewert bei main:

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

Bsp.: Wir schreiben unser C-Programm in eine Datei mit dem Namen tst1.c und übersetzen es mit dem Compiler-Aufruf

```
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"

Ausgabe

printf (char *fmt, ...); ("Funktionsprototyp")

fmt: "Format-String" ; wird so ausgegeben, wie er angegeben ist, aber %. wird ersetzt durch den Wert des entspr. Ausdrucks in ...

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

gibt aus: Das ist die 10-te Ausgabe!

%d	Ganzzahl	in Dezimaldarstellung
%x	"	" Hexadezimaldarstellung
%c	"	als character
%s	String (characters bis zum abschließenden '\0')	
%p	Pointer (in maschinenabhängiger Darstellung)	

Verfügbar machen mit: #include <stdio.h>

Stringvergleich

int strcmp (char *s1, char *s2);

Vergleicht die Strings s1 und s2 lexikografisch und liefert Zahl <0, =0, >0 zurück, wenn s1 vor s2 liegt, gleich s2 ist, nach s2 liegt.

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

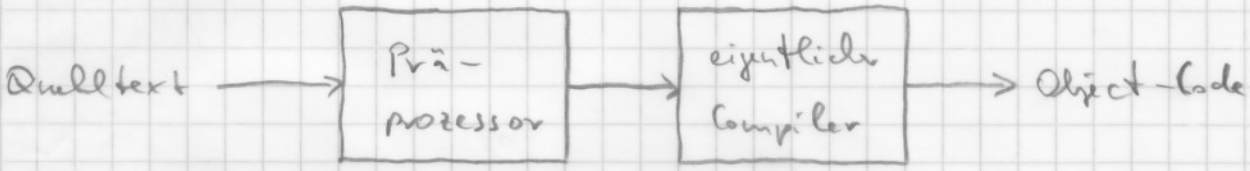
Verfügbar machen mit: #include <string.h>

Beenden des Programms

a) mit return ...; aus main()

b) mit exit(...); aus irgendeiner bel. Funktion
Verfügbar machen mit: #include <stdlib.h>

Der C-Präprozessor

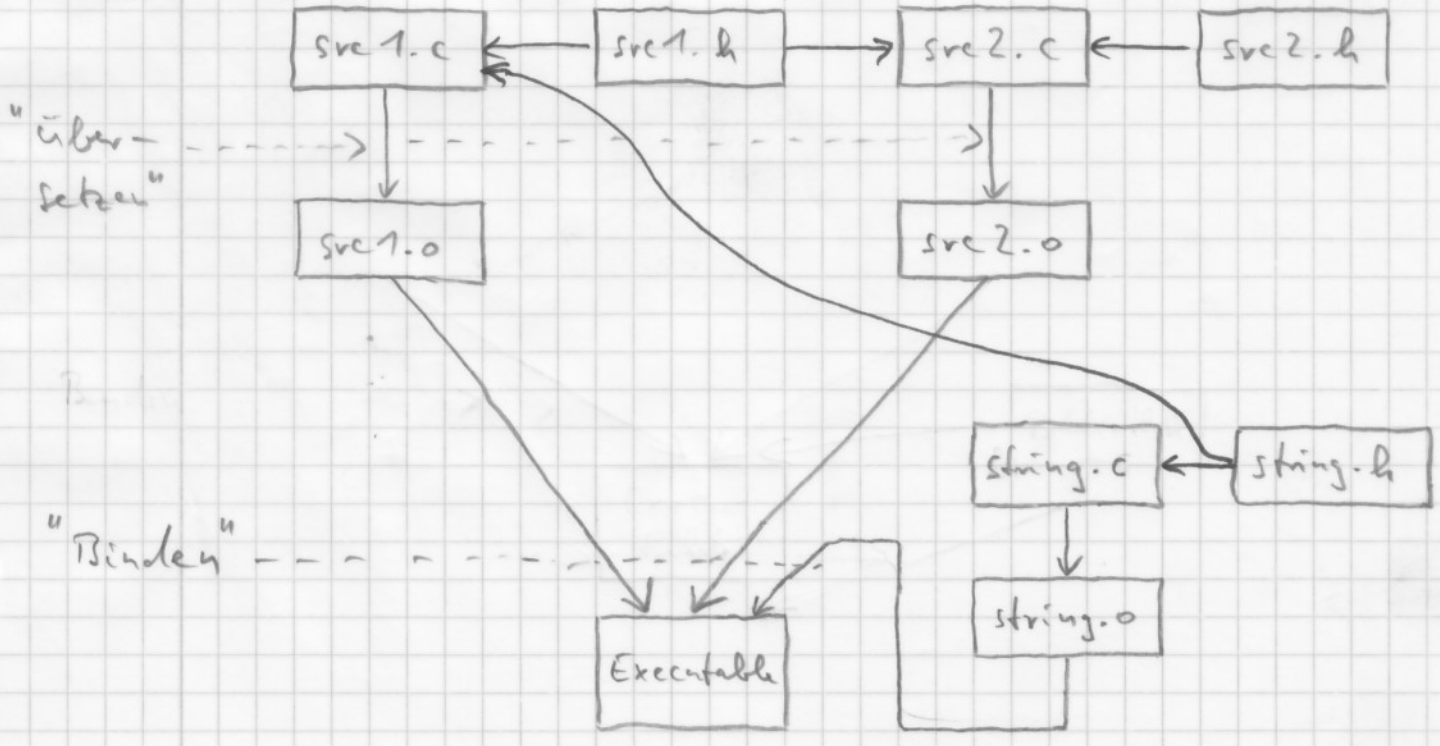


Befehle an den Präprozessor beginnen mit dem Lattenkreuz #

3 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);  ⟶  c = 2*3+1;
```

man erwartet: 8 — man erhält: 7 ↯

Konsequenz: Bei Makrodefinitionen Klammern setzen!

3) Bedingte Compilierung

```
#define LINUX
:
#ifdef LINUX
   :                    } wird nur übersetzt, falls LINUX definiert
#endif
```

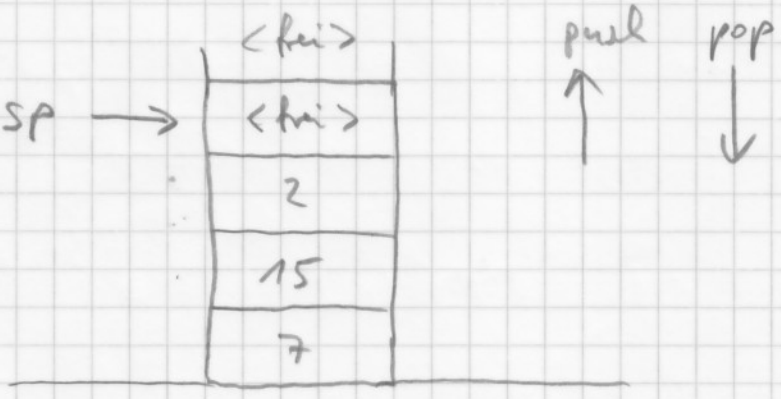
Anwendung: Ausschluß von Mehrfachinklusionen

```
#ifndef  SRC1_H_INCLUDED
#define  SRC1_H_INCLUDED
   :
#endif
```

} Inhalt von src1.h

VM

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!

Instruktionen für die VM

add ... n1 n2 ⟶ ... n1+n2

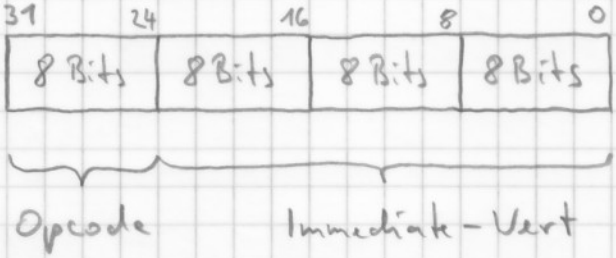
(... unverändert)

entspr. 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)

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
(wird im Augenblick nur für pushc benötigt)

Bsp.: pushc 5

Der Opcode für pushc ist 1, also ist der Code für "pushc 5": `(1 << 24) | 5`

↑ linksschieben ↑ bitweises oder

Achtung: Was passiert bei "pushc -1"?

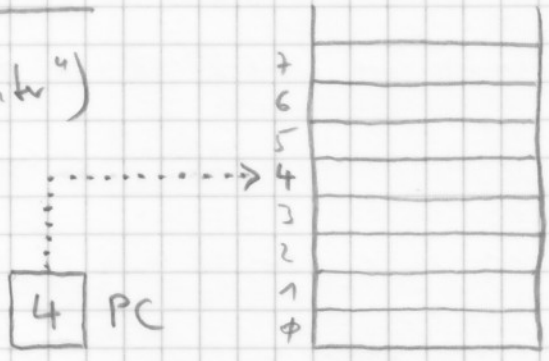
(Hinweis: Zweierkomplement!)

Also besser: `(1 << 24) | (5 & 0x00FFFFFF)`

bitweises und ↑ "Maske"

Programmspeicher und PC

Der PC ("Program Counter") ist der Index der nächsten auszuführenden Instruktion.



Steuerwerk / Instruktionszyklus

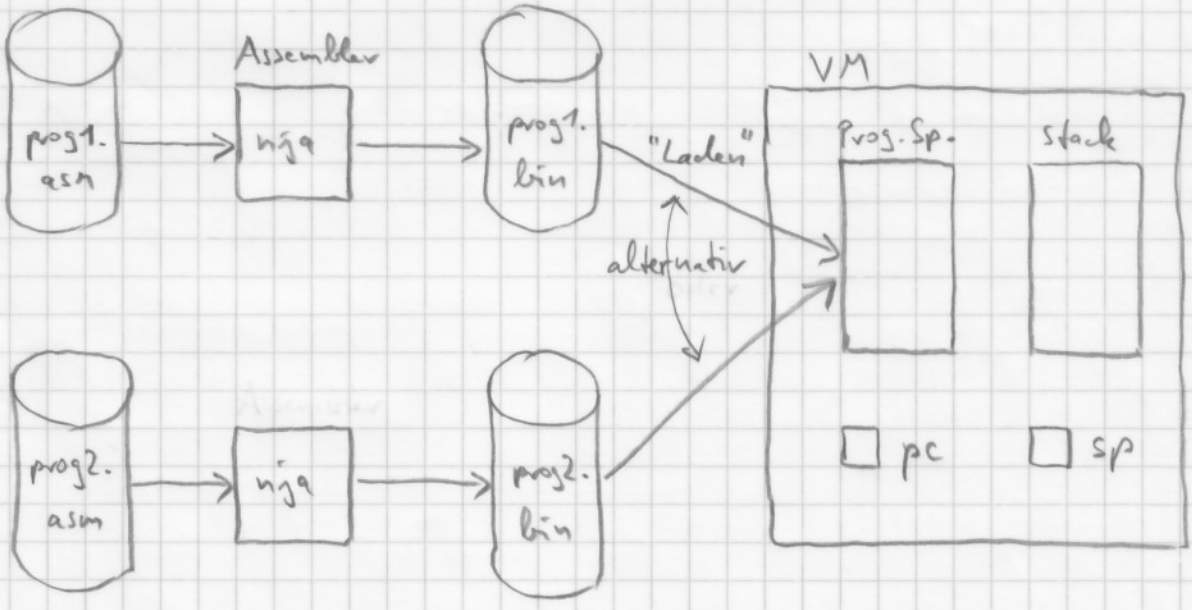
```
while (!halt) {
    IR = program_memory [PC];   // Instruktion holen
    PC = PC + 1;                // PC inkrementieren
    exec (IR);                  // Instruktion ausführen
}
```

(IR heißt "Instruktionsregister")

ASM

Bis jetzt: Fester Programmspeicherinhalt ("ROM")

Ab jetzt: Laden eines bel. Binärprogramms in die VM:



prog1.asm enthält 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!

C

Dateioperationen

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

a) öffnen

```
FILE *fopen(char *path, char *mode);
```

path: welche Datei? z.B. "prog1.bin" oder "/home/hg53/xxx"

mode: Lesen? Schreiben? Anhängen? z.B. "r"

Datenstruktur, die den Zustand der geöffneten Datei hält. Wie das genau aussieht, bleibt dem Benutzer verborgen!

fopen() liefert NULL, falls fehlgeschlagen.

Das muß immer überprüft werden!

b) 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!

c) 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 bel. Datentypen in bel. Anzahl funktionieren!

"void *" ≙ "Zeiger auf irgendwas"

Konsequenz: Compiler weiß nichts über Typ, Größe!

size: Größe eines Datenobjektes in Bytes. Diese wird durch den statisch auswertbaren Operator "sizeof" berechnet, z.B. sizeof(int) (= 4 bei 32-Bit-Rechner)

nmemb: Anzahl der Datenobjekte, die gelesen werden sollen. Deshalb insgesamt benötigte 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":

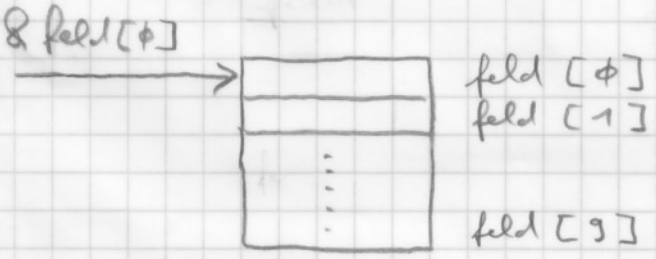
```
FILE *fp;                          <-- int feld[10];
fp = fopen ("xy.z", "r");
if (fp == NULL) {
    printf ("...");
    exit (99);
}
```

```
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"

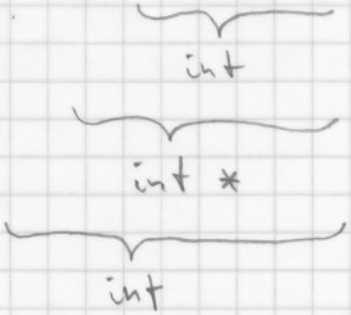
Allgemein: d hat Typ T ⇒ &d hat Typ T*

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



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;   <-- !!!
}
```

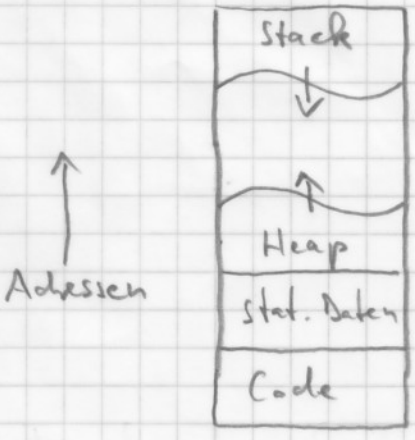
Typ ist korrekt ⇒ der Compiler bemerkt das nicht!

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)`.

VM

Variable

Eine Variable ist ein Ort, an dem ein Wert aufgehoben ("gespeichert") werden kann.

In Java gibt's 4 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	⟷	globale/statische Variable
Instanzvariable	⟷	Variable auf dem Heap
Parametervariable	⟷	Parametervariable
lokale Variable	⟷	lokale Variable

"global" vs. "statisch"

"global" bezeichnet die Sichtbarkeit der Variablen:

- sie ist von überall her zugreifbar

"statisch" bezeichnet die Lebensdauer der Variablen:

- sie lebt (= hat Speicherplatz) die ganze Programmlaufzeit

Also: global ⇒ statisch

statisch ⇏ global (viele Klassenvariablen in Java)

Globale Variable, Static Data Area

Globale Variable werden in der "Static Data Area" gespeichert. Jede Variable gibt's genau einmal. Sie wird eindeutig bezeichnet durch ihr Adresse in der Static Data Area. Zum Arbeiten mit globalen Variablen gibt's zwei Instruktionen: push global variable, pop global Variable. Beide haben als Immediate-Wert die Adresse der Variablen kodiert.

pushg ... → ... wert (legt Wert d. glob. Var. auf Stack)

popg ... wert → ... (speichert top-of-stack in glob. Var.)

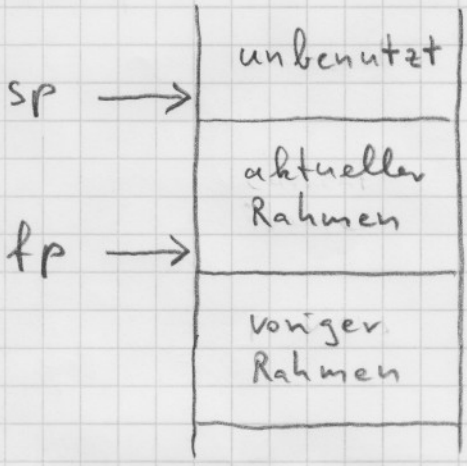
Bsp.: Angenommen, x soll an der Adresse 2 und y an der Adresse 5 in der Static Data Area gespeichert sein. Dann kann die Anweisung x = 3 * x + y berechnet werden durch die Instruktionsfolge:

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

Lokale Variable, Stack Frames

Lokale Variable (und Parametervariable) muß es getrennt für jeden Prozedur- und 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)



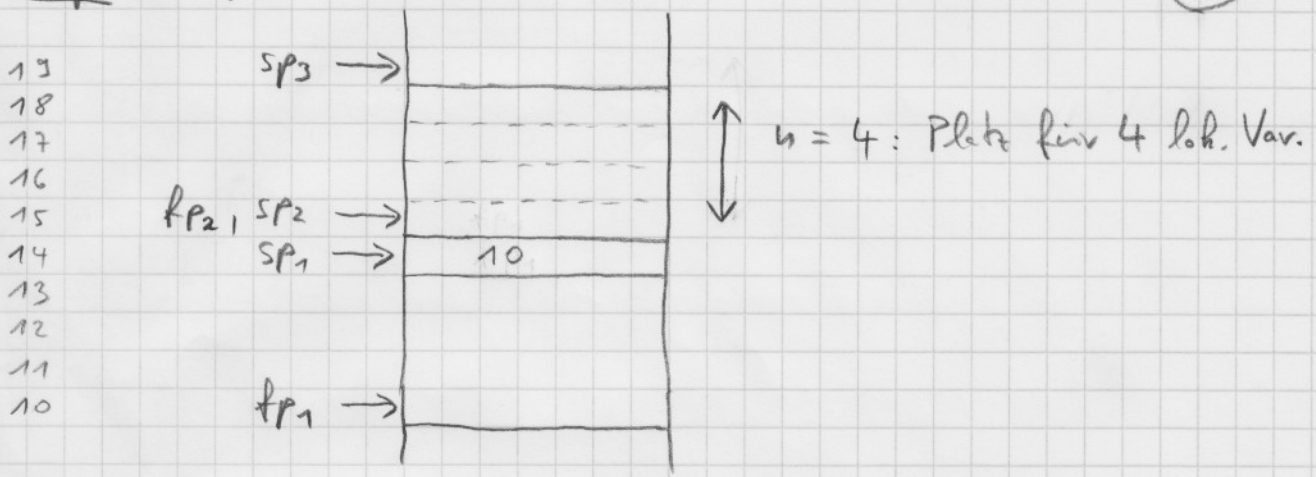
Der Bereich zwischen fp und sp heißt "aktueller Rahmen" und speichert die lokalen Variablen der momentan ausgeführten Prozedur / Funktion / Methode. Beim Aufruf einer Prozedur muß ein neuer Rahmen angelegt werden, der beim Verlassen wieder vernichtet wird. Dazu zwei neue Instruktionen: allocate stack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

Arbeiten mit lokalen Variablen

Zwei Befehle: pushl local variable, popl local variable

Beide haben als Immediate-Wert den Abstand zu fp kodiert.

pushl ... → ... wert (legt Wert der lok. Var. auf Stack)

popl ... wert → ... (speichert top-of-stack in lok. Var.)

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
```

Berechnung von arithmetischen Ausdrücken

Bsp.: $a - 3 * b - 5$

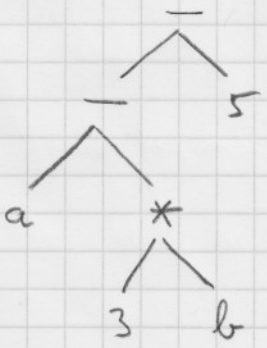
Algorithmus zum Erzeugen der Instruktionen:

1. Schreibe den Ausdruck vollständig geklammert hin (berücksichtige dabei Priorität und Assoziativität der Operatoren)

   Bsp: $((a - (3 * b)) - 5)$

2. Wandle den vollst. gekl. 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

Bsp.:



3. Traversiere den Baum einmal (depth first, post-order)
   a) Bei einem inneren Knoten:
      a1) Traversiere den linken Teilbaum
      a2) Traversiere den rechten Teilbaum

a3) Gib die zum Operator gehörende arithmetische Instruktion aus

b) Bei einem Blattknoten:

b1) Steht an dem Blatt eine Zahl:

Gib "pushc <zahl>" aus

b2) steht an dem Blatt eine Variable:

Gib "pushl <Abstand d. Speicherplatzes zu fp>" (oder "pushg <Adresse>" bei glob. Var.) aus

Bsp.: (ang. Abstand (a) = 10, Abstand (b) = 14)

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

Zuweisung

Bsp.: b = a - 3 * b - 5

Algorithmus zum Erzeugen der Instruktionen:

1. Erzeuge Code für die rechte Seite der Zuweisung
2. Gib "popl <Abstand d. Speicherplatzes der Variablen auf der linken Seite zu fp>" (oder "popg <Adresse der Variablen auf der linken Seite>" bei glob. Var.) aus

Bsp.:
```
  ⋮        } Code für rechte Seite wie oben
popl  14
```

Kontrollstrukturen

a) Einarmiges "if"

if (B) S

b) Zweiarmiges "if"

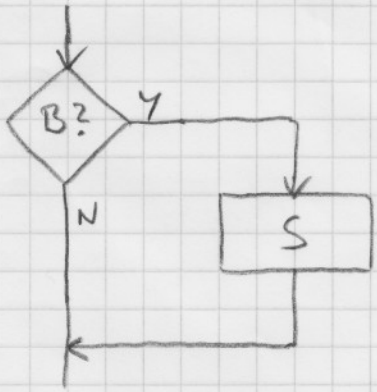
if (B) S_1 else S_2 → A

c) "while" - Schleife

while (B) S → B

d) "do" - Schleife

do S while (B); → C



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

<S> " " " " S " "

```
<B>          // hinterläßt Ergebnis auf dem Stack
brf  L       // springe, wenn Ergebnis == false
<S>

L:           // das nennt man ein "Label" ("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?

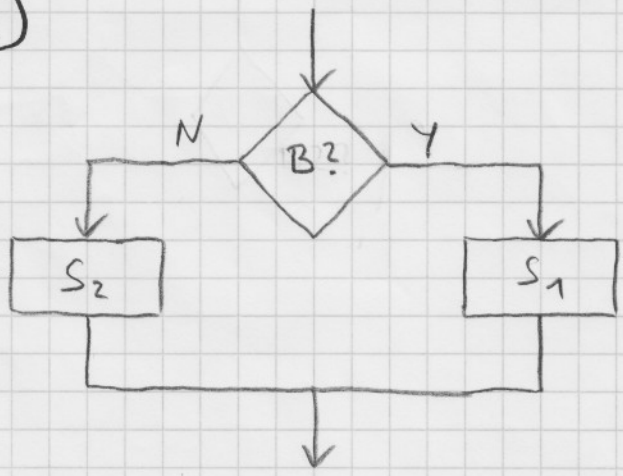
Was ist "B"? Dazu 6 arithmetische Vergleiche:

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	⟶	gt	
a >= b	⟶	ge	

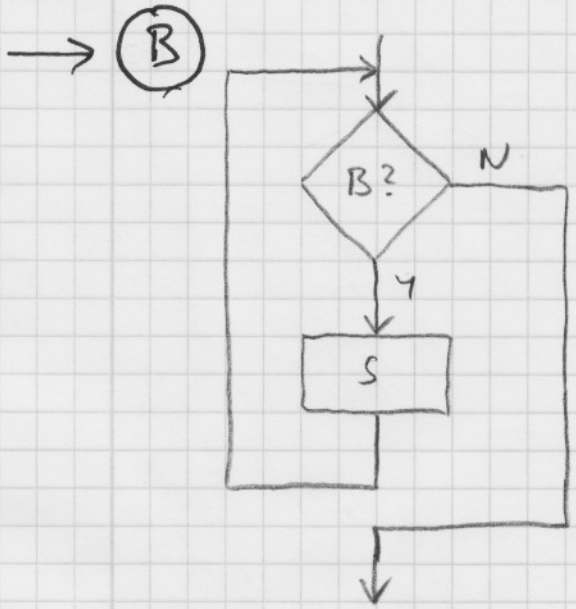
Wie wollen wir Boole'sche Werte repräsentieren?

Besonders einfach: 0 ≙ false, 1 ≙ true (ganze Zahlen)

⟶ (A)




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



Alternative 1:

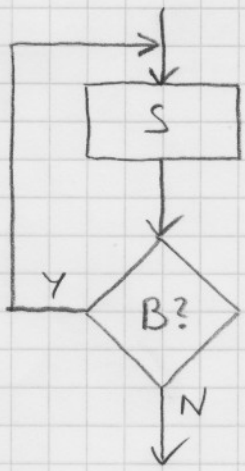
```
L1:
        <B>
        brf   L2
        <S>
        jmp   L1
L2:
```

Nachteil: zwei Sprünge pro Schleifendurchlauf

Alternative 2:

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

→ (C)



Auch hier gibt's zwei Alternativen, aber nur eine sinnvolle:

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

Zusammenfassung:

1) 6 arithmetische Vergleiche liefern Boole'sche Werte
2) Sämtliche Kontrollstrukturen (mit Ausnahme des Prozeduraufrufs) werden mit jmp / brf / brt realisiert. Das Sprungziel wird im Assembler durch eine Marke repräsentiert; in der VM ist das die Adresse der Instruktion, die am Sprungziel steht.

Auswertung Boole'scher Ausdrücke

Sprache schreibt vor, ob "vollständige Auswertung" oder "Kurzschlussauswertung":

a) Vollständige Auswertung

a1) B_1 && B_2 ⟶
```
<B1>
<B2>
and
```

a2) B_1 || B_2 ⟶
```
<B1>
<B2>
or
```

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

b) Kurzschlußauswertung

b1) B_1 && B_2 ⟶
```
<B1>
brf   L1
<B2>
jmp   L2
L1:
pushc 0     // false
L2:
```

b2) B_1 || B_2 ⟶
```
<B1>
brt   L1
<B2>
jmp   L2
L1:
pushc 1     // true
L2:
```

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

Unterprogrammaufruf und -Rücksprung

Ein Unterprogrammsprung ("Call") muß die Adresse des nächsten Befehls ("Rückkehradresse") speichern, damit der Unterprogrammrücksprung ("Return") dorthin zurückfindet. Speicherort ist der Stack:

call ... ⟶ ... ra

ret ... ra ⟶ ...

a) Call ohne Argumente, ohne Rückgabewert

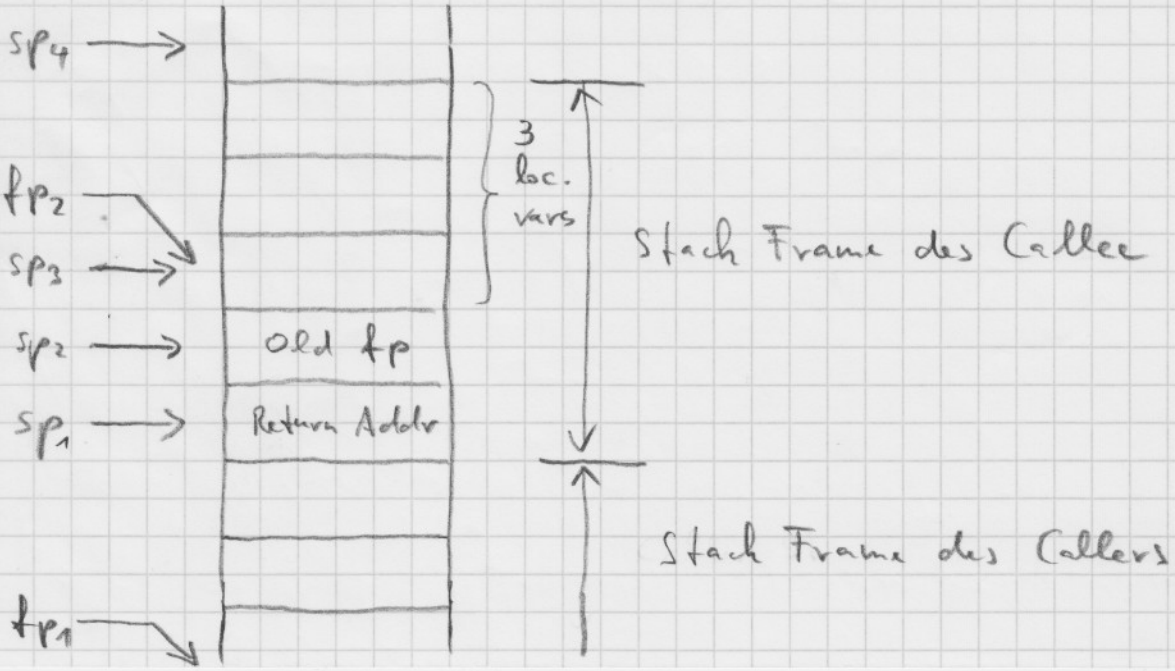
Caller (= aufrufende Prozedur):

```
call  <proc addr>
```

Callee (= aufgerufene Prozedur):

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

Stackaufbau (gezeichnet nach Ausführung von asf 3):



Die Werte von Argumentausdrücken müssen vor dem Aufruf einer Prozedur berechnet und gespeichert werden. Wo? Auf dem Stack! Wir numerieren im Folgenden die n Argumente von ϕ ("1. Argument", links) bis $n-1$ ("n-tes Argument", rechts).

b) Call mit Argumenten, aber ohne Rückgabewert

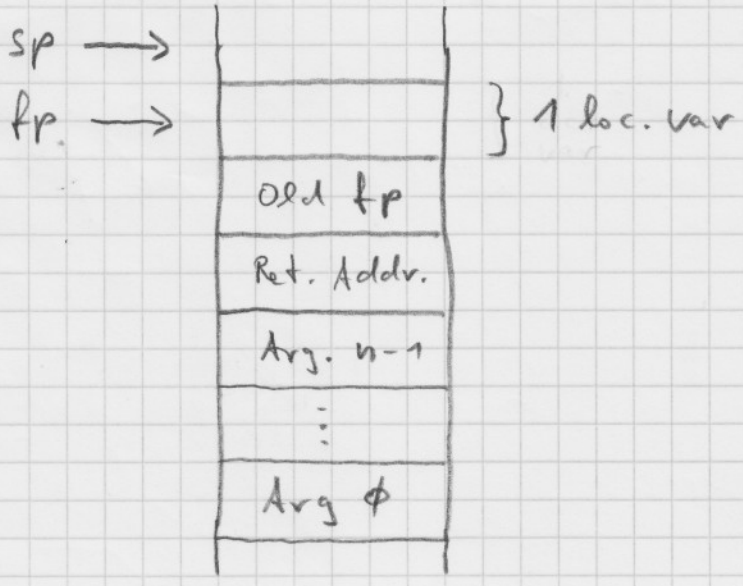
Caller:

```
<push arg φ>
    :
<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 <u>negativen Offset</u> zum Framepointer:

Argument i $\longleftrightarrow$ stack[fp − 2 − n + i]

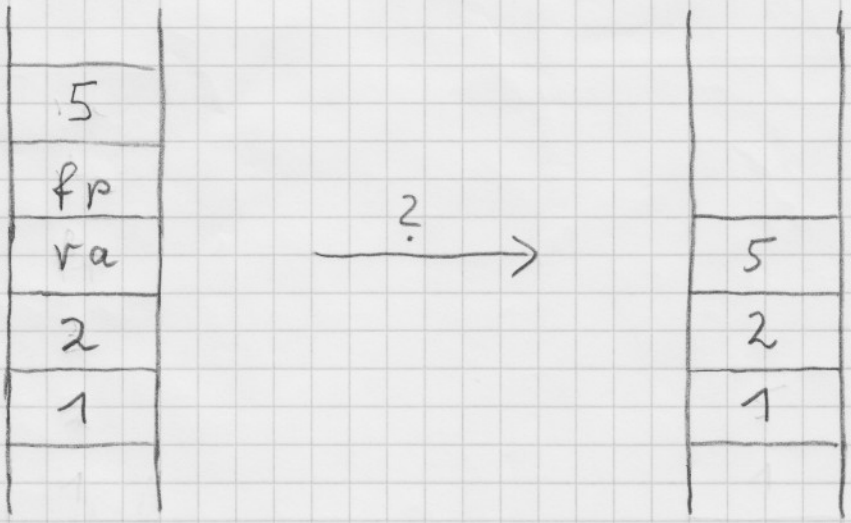
Der Wert von $-2-n+i$ wird als Immediate-Konstante in die Instruktionen "pushl" und "popl" kodiert:

so wird auf die Parameter einer Prozedur zugegriffen

c) 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: <u>vor</u> der Rückkehradresse!

<u>Bsp.</u>: 1 + 2 * f(), mit: int f() { return 5; }



Einfachste Lösung: Rückgabe des Wertes in einem zusätzlichen Spezialregister "r" mit 2 Instruktionen.

pushr ... ⟶ ... rv

popr ... rv ⟶ ...

Caller:

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

Callee:

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

d) Call mit Argumenten und Rückgabewert

Kombination von b) und c).

Caller:

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

Callee:

genauso wie c)

Zugriff auf Argumente: genauso wie b)

VM

Objekte, Objektreferenzen

Rechenobjekte bis jetzt: Integer, Character, Boolean

" 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(); 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.:

a) 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;

b) 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 ...

Variablendefinition dann z.B.: Messages messages;

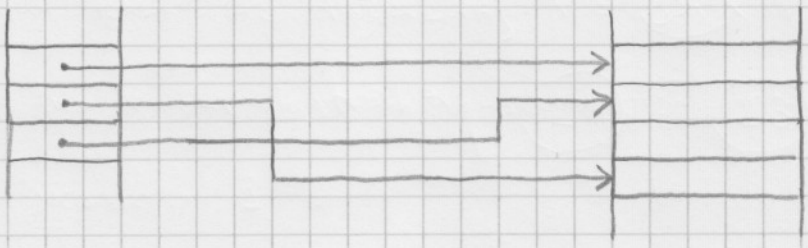
Konventionen zur Namensgebung bei Typen (2 Alternativen):

a) erster Buchstabe ist Großbuchstabe, z.B. Size

b) angehängtes _t am Namen, z.B. size_t

VM

Stack: enthält Objektreferenten Heap: enthält Objekte



Vorschlag: typedef int Object; /* Objekt */

typedef Object *ObjRef; /* Objektreferenz */

Zwei Schwierigkeiten:

a) Die Objekte werden in Zukunft verschieden groß sein → Größenangabe im Objekt notwendig. Wie macht man das in C?

b) 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?

Verbunde ("Records")

Feld ("Array"): Kollektion von Variablen gleichen Typs, Auswahl durch Zahl ("Index")

Verbund ("Record"): Kollektion von Variablen möglicherweise unterschiedlichen Typs, Auswahl durch Namen ("Komponente")

Verbunde gibt's in zwei Ausprägungen:

a) 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 4 Komponenten. Auswahl einer Komponente durch den Punkt-Operator, Bsp.:

```
person.jahr = 1954;
```

b) Variante 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("%x%02X ", inspect.b[i]);
}
```

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

```
typedef struct {
    ...
    ...
} Person;                    } Typ-Definition
Person person;               } Variablen-Definition
```

Das nicht nicht aus für rekursive Datendefinitionen!

Bsp.: Verkettete Liste von ganzen Zahlen

```
typedef struct {
    int member;
    Liste *next;
} Liste;
```

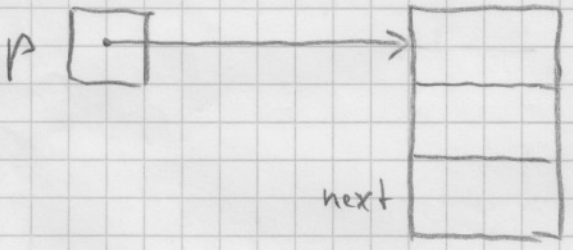
} gilt nicht! Der Bezeichner "Liste" wird verwendet, bevor er definiert ist!

Deshalb gibt's sogenannte "tag names": Namen, die nur in Verbindung mit "struct" oder "union" gültig sind.

```
typedef struct liste {
    int number;
    struct liste *next;
} Liste;
```

"liste" ist der "tag name"

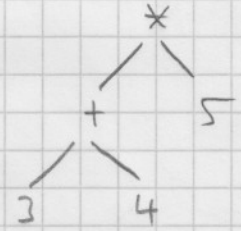
Bem.: Häufig werden Verbunde auf dem Heap angelegt und mittels Zeigern zugegriffen. Dann beschreibt der Ausdruck (*p).next die Dereferenzierung des Zeigers p mit anschließender Auswahl der Komponente "next":



Abkürzung: (*p).next ≡ p->next

Anwendungsbeispiel: Arithmetische Ausdrücke

Bsp: (3+4) * 5



Was müssen die Knoten speichern?

- innerer 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 Knotens:

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

a) 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 -> isLeaf = FALSE;
    result -> u.innerNode.operation = operation;
    result -> u.innerNode.left = left;
    result -> u.innerNode.right = right;
    return result;
}
```

Bsp:

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

b) Auswerten von Knoten

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

Bsp.: eval(expression) liefert 35

c) 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

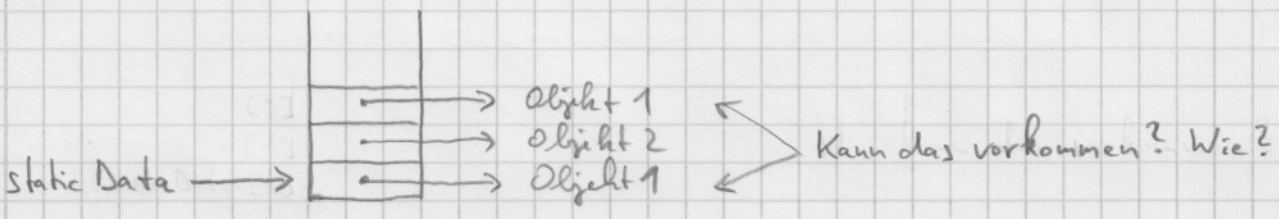
Der Stack ist ein statisch angelegtes Array von "Stack-Slots":

```
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?

a) Anlegen:
```
objRef = alloc(sizeof(unsigned int) + sizeof(int));
objRef->size = sizeof(int);
*(int *)objRef->data = value;
```

b) 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 Arraygrenze

Konsequenz: "size as needed" funktioniert!

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 bel. großen Zahlen

a) Prinzip

Ganze Zahlen werden in einem Stellenwertsystem mit Basis b dargestellt: $z = \sum_{i=0}^{n} d_i \cdot b^i$. Dabei heißen die d_i "Ziffern" der Zahl z, und es ist $0 \leq d_i < b$ f. alle $i = 0 \ldots n$. 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 do stellenweise addiert und dabei einen evtl. auftretenden "Überlauf" in die nächste Stelle beachtet.

Bsp.:
$$\begin{array}{r} 1954 \\ +2073 \\ \hline 4027 \end{array}$$
(Überlauf: 1)

Subtraktion und Multiplikation gehen entsprechend.
Die Division erfordert "Raten": $700020 : 876 = ?$
Dieses Raten muß formalisiert werden (nachzulesen in D. Knuth, The Art of Computer Programming, Vol. 2, Semi-numerical Algorithms).

b) Zahldarstellung

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 unvorhersehbar viele Stellen "vernichtet" werden (Null enthalten), die aber nicht gespeichert werden sollen
→ tatsächlich belegte Anzahl von Bytes mit abspeichern!
Negative Zahlen? → Vorzeichen/Betrags-Darstellung!
⇒ Datentyp für bel. große Zahlen in C:

```
typedef struct {
```

```
    int nd;                    /* number of digits */
    unsigned char sign;        /* sign */
    unsigned char digits[1];   /* the digits */
} Big;
```

(Für Details, z.B. "was ist das Vorzeichen von Φ?" oder "darf die höchste Ziffer Φ sein?" siehe Paket "bigint.tar.gz"!)

Bem.: Der Verbund "Big" ist in unseren Objekten die "Nutzlast" und wird in der "data"-Komponente gespeichert.

c) Bibliotheksinterfaces

Die Bigint-Bibliothek hat zwei Interfaces:

1. 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.

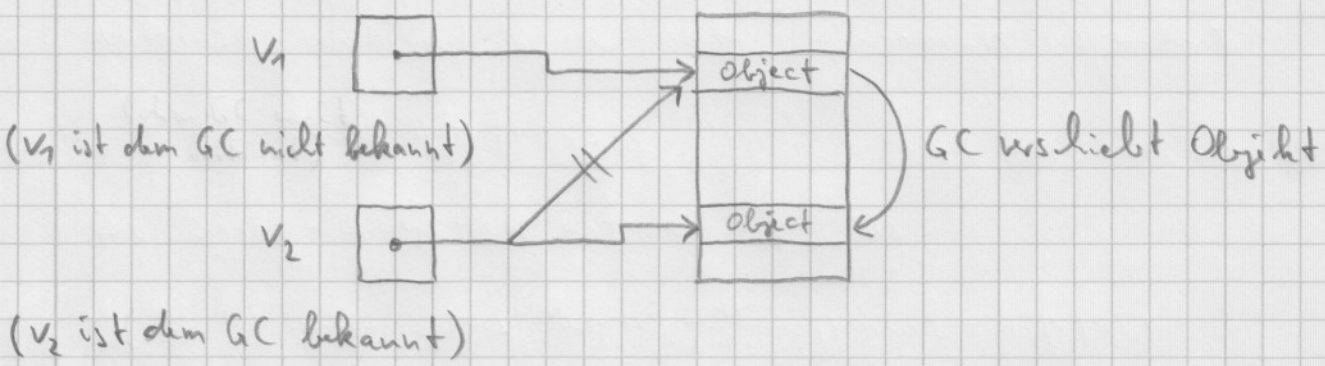
2. Nach "oben": das Benutzer-Programmierinterface

   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? Objektreferenzen dürfen nur in Variablen aufgehoben werden, die der Garbage Collector kennt und berichtigt !!



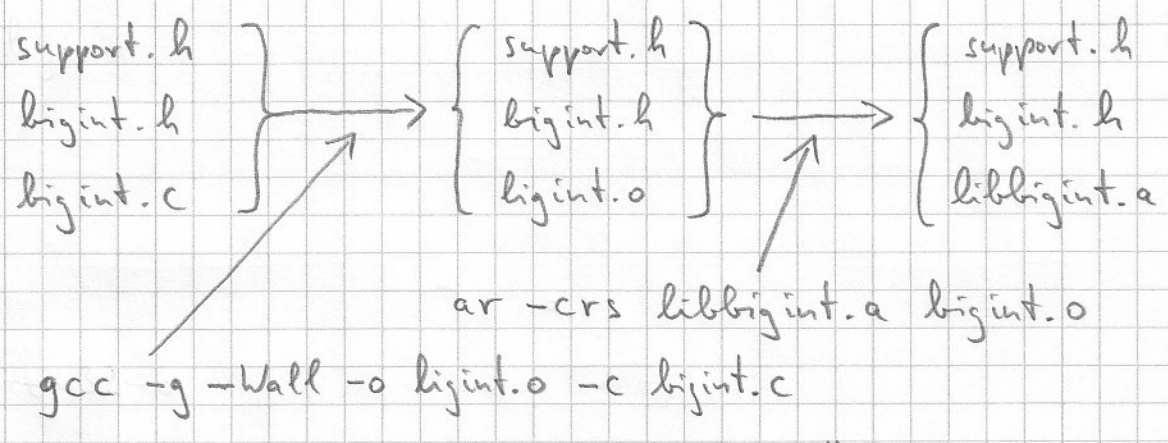
Konsequenz: Es gibt eine globale, strukturierte Variable "bip" ("big integer processor"), in die die Argumente von bigAdd() gespeichert werden (das sind Objektreferenzen!). Dann wird die Funktion aufgerufen. Dabei gibt's möglicherweise eine GC und die Referenzen ändern sich. Das ist ok, da "bip" dem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

d) 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.

1. Erzeugen der Bibliothek



Bem: ar ist der "archiver" (besser wäre: "library manager"). Eine "Library" ist eine Sammlung von Object-Dateien (".o") und hat einen Namen, der mit "lib" beginnt und mit ".a" endet.

2. Benutzen der Bibliothek

a) Module, die Funktionen aus der Bibliothek benutzen wollen, müssen die entspr. Header inkludieren. Muß man die in sein eigenes Entwicklungsdirectory kopieren? Nein! Directory für Header bekannt machen mit z.B.

gcc -I../build/include ...

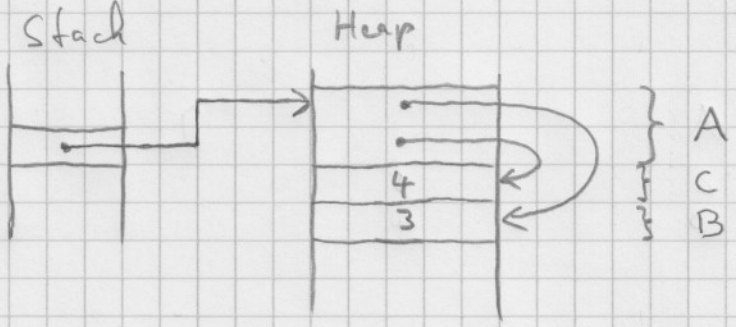
b) Beim Binden muß sowohl das Directory angegeben werden, wo die Bibliothek steht, als auch das Einbinden selbst: gcc -L../build/lib ... -lbigint

Achtung, Namenskonvention: Einbinden von libbigint.a mit -lbigint!

VM

Objekte

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



Bsp.: A ist Instanz eines "Punktes" mit den Koord. B und C (B und C sind "primitive Objekte": nicht weiter zerlegbar).

Objekte gibt's in zwei Varianten:

a) "Record" (in OO-Sprachen "Instanz einer Klasse" genannt)

Merkmal: Zugriff auf Variable durch Namen

b) "Array" (heißt in OO-Sprachen genauso)

Merkmal: Zugriff auf Variable durch (berechneten) Index

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

Objekte: Darstellung

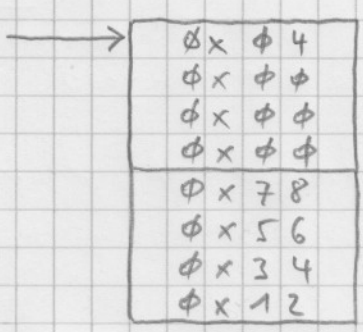
ACHTUNG, wir haben ein Problem: es gibt zwei ganz verschiedene Sorten von Objekten (primitive und zusammengesetzte)! Wie halten wir die auseinander?

Bsp.:

1. ein primitives Objekt mit der Zahl Φx 12345678
2. einen Record mit einer Instanzvariablen, die zufällig mit

der Objektreferenz (= Adresse) 0x12345678 belegt ist

Beide sehen im Speicher einer Little-Endian-Maschine so aus:

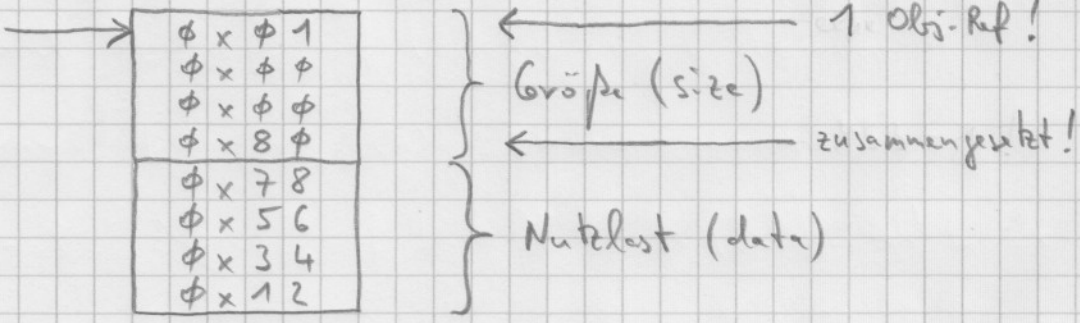


Wir vereinbaren:

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 Objektreferenz im Objekt!

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

Objekte: Records

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> ... ⟶ ... objekt

(n ist die Anzahl der Objektreferenzen im Objekt; oben: 2)

Zugriff: (....).x
(....): Objekt; x: 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")

Objekte: Arrays

NJ

Definition: `type Vector = Integer[];`

Erzeugen:
```
local Vector v;
v = new(Integer[2*n+1]);
```
(Komplikation: Anzahl d. 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, [....] = 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 für bel. Objekte:

$$\text{sizeof}(\underbrace{\ldots}_{\text{Objekt}}) = \begin{cases} -1 & \text{für ein primitives Objekt} \\ \text{Anzahl d. Obj.-Ref. für ein zusammengs. Obj.} \end{cases}$$

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!)

Nil, Initialisierungen, Laufzeittests

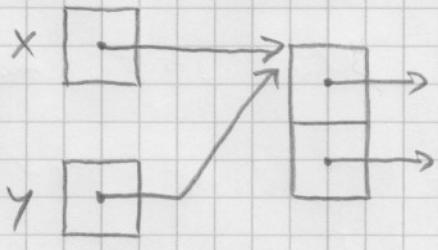
In Java gibt's "null", in Ninja gibt's "nil":
eine Referenz, die auf kein Objekt verweist.
Die VM sollte alle Instanzvariablen neuer Objekte sowie alle lokale Variablen von Funktionen (bei der Ausführung von "asf") und alle globale Variablen (direkt nach dem Anlegen) mit nil initialisieren.
Einfachste Darstellung von nil in C: der Nullzeiger NULL. Dann sollte die VM jeden Zugriff auf ein Objekt prüfen und mit Fehler abbrechen, wenn die Objektreferenz 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).

Referenzvergleiche

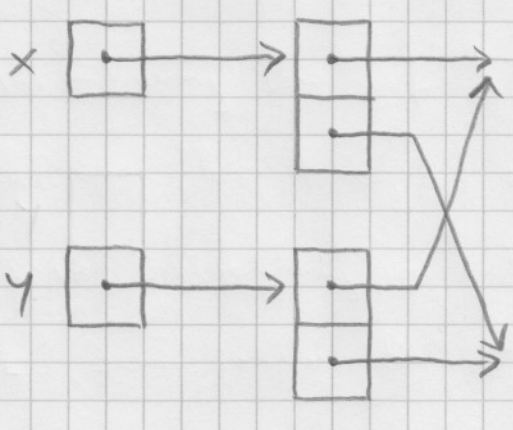
Um Bedingungen wie z.B. x == nil berechnen zu können, brauchen wir 3 neue Instruktionen:

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

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



push x
push y
refeq → true



push x
push y
refeq → false

VM (Speicherverwaltung)

1. Aufgaben

1.1. Speicherallokation

Findet statt bei der Erzeugung von Objekten. Methoden:

a) Freiliste: alle freien Bereiche sind verkettet.
Vorwiegend verwendet bei konstanter Objektgröße, sonst sehr aufwendig (siehe malloc() aus C)

b) Freibereich: der freie Bereich des Speichers ist stets zusammenhängend. Das erfordert "Kompaktierung" des Speichers.

Wir haben unterschiedlich große Objekte → Methode b)

1.2. Speicherkompaktierung

Wird zur Erzeugung eines zusammenhängenden Freibereichs durchgeführt. Methoden:

a) separater Kompaktierungslauf

b) zusammen mit Speicherfreigabe (bevorzugt)

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:

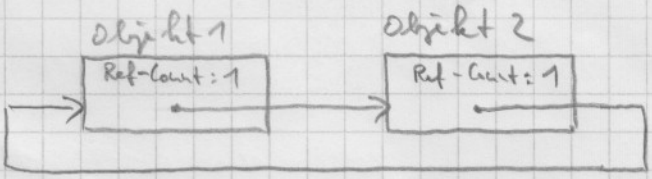
a) Referenzzähler

Idee: Jedes Objekt beinhaltet einen Zähler, dessen Wert die Anzahl der auf das Objekt zeigenden Zeiger ist. Fällt der Wert auf ϕ: Speicherplatz ist frei

Vorteil: gleichmäßige Verteilung des zeitl. Aufwandes

Nachteile:

- Aufwand ist hoch: aus jeder Zuweisung `p1 = p2;` wird `decRef(p1); p1 = p2; incRef(p1);`
- zyklische Strukturen werden nicht freigegeben, obwohl sie nicht mehr referenziert werden:



b) Müllsammeln (Garbage Collection, GC)

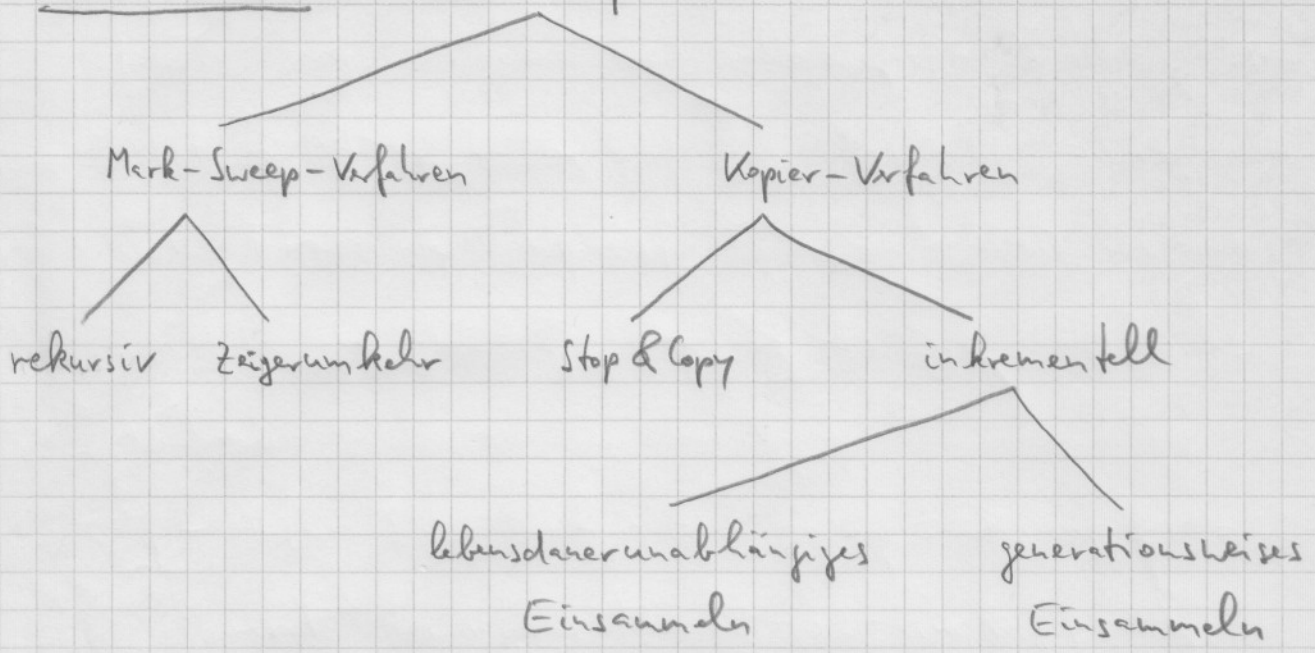
Idee: Zu best. Zeiten (z.B. wenn der freie Speicher knapp ist) wird ermittelt, welche Objekte noch zugreifbar sind – ausgehend von den Registern und dem Stack der Maschine. Alle anderen Objekte werden freigegeben.

Vorteile:

- Speicherverwaltung ist sauber abgegrenzt vom Rest der Maschine
- zyklische Strukturen werden gesammelt
- Speicherverwaltung ist über das gesamte Programm verteilt

2. Verfahren zum Müllsammeln

Übersicht:



2.1. Mark-Sweep-Verfahren

Dieses sind nicht-kompaktierende Verfahren. Sie laufen in 2 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 <u>alle</u> Objekte. Der Speicherplatz von nicht-markierten Objekten wird freigegeben (und das Markierungsflag bei <u>allen</u> Objekten zurückgesetzt).

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

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);
    }
}
```

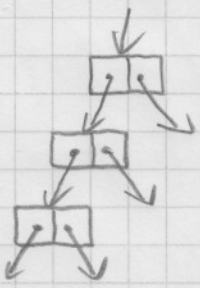
Vorteil: sehr einfache Implementierung

Nachteil: Platzbedarf des Stacks für Rekursion groß (schlechtester Fall: genauso groß wie der gesamte Heap!) → unbrauchbar

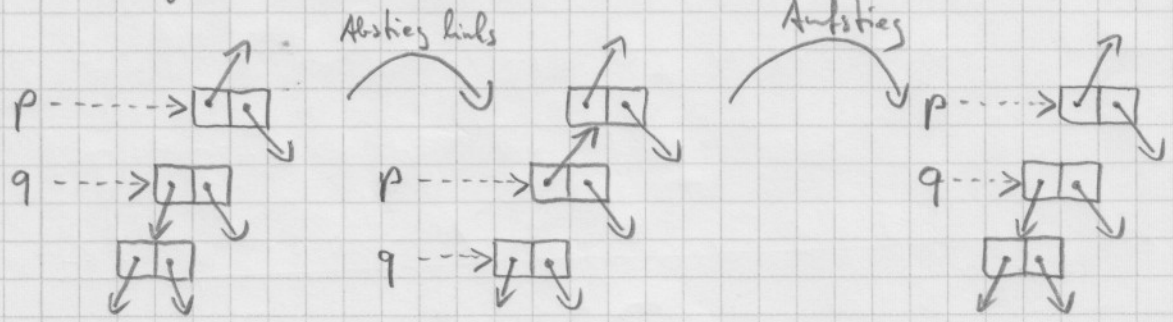
2.1.2. Iterativer Depth-First-Durchlauf mit Zeigerumkehr

(Deutsch, Schorr und Waite 1965)

Idee: Die Information, wohin man im Graphen "zurück" muß, wird nicht im Stack, sondern im Graphen selbst gespeichert. Bsp.: Binärbaum



Zeigerumkehr:



entspr. Abstieg rechts. Das ist das bei konstanter Objektgröße

vorwiegend verwendete Verfahren.

2.2. Kopier-Verfahren

Dieses sind kompaktierende Verfahren, die einen zusammenhängenden Freibereich erzeugen.

2.2.1. Stop & Copy

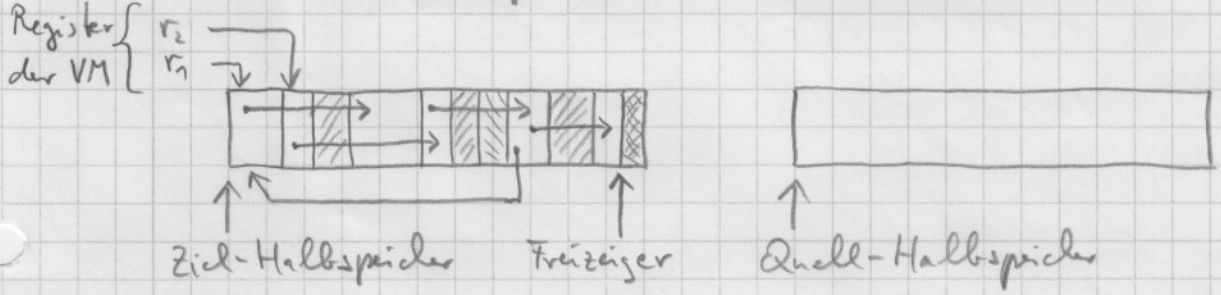
(Minsky, Fenichel, Yochelson, 1969)

Der Arbeitsspeicher wird in zwei Hälften geteilt:

"Ziel-Halbspeicher": dort werden die Objekte alloziert

"Quell-Halbspeicher": ist unbenutzt

System läuft, bis ein Objekt alloziert werden soll, das zu groß für den verbliebenen 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 erreichbaren Objekte (sog. "Root Objects")
3) Kopieren aller anderen zugänglichen Objekte (der sog. "lebenden" Objekte); diese Phase des GC heißt "Scan"

Problem: Wenn ein Objekt gerade kopiert wurde, wird es Zeiger sowohl im Quell- als auch im Ziel-Halbspeicher geben, die noch auf das "alte" Objekt zeigen. Wie werden diese in Zeiger auf das kopierte Objekt geändert?

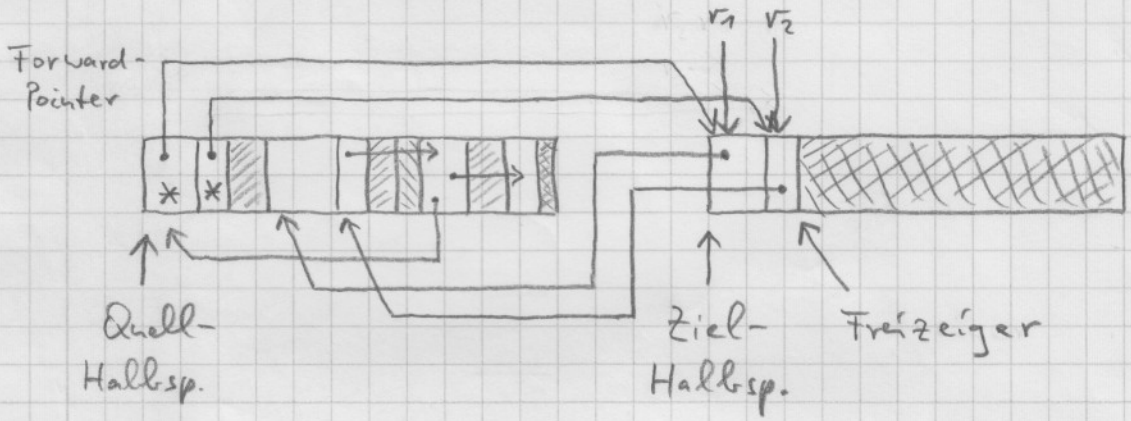
Idee: Ein kopiertes Objekt wird gekennzeichnet (durch das "Broken-Heart-Flag") und in ihm wird die Adresse des kopierten Objektes gespeichert (der sog. "Forward-Pointer"). Während der Scan-Phase werden alle Zeiger korrigiert.

Algorithmus zum Umwandeln eines Zeigers auf ein Objekt im Quellspeicher in einen Zeiger auf das entspr. 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:



* ≙ Broken-Heart-Flag gesetzt, Forward-Pointer gültig

Algorithmus für die Scan-Phase:

```
scan = zielhalbspeicher;
while (scan != freizeiger) {
    /* es gibt noch Objekte, die gescannt werden müssen */
    if (Objekt enthält Zeiger) {
        for (jeden 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.:

a) Die Scan-Phase macht einen Breadth-First-Durchgang durch alle Objekte.

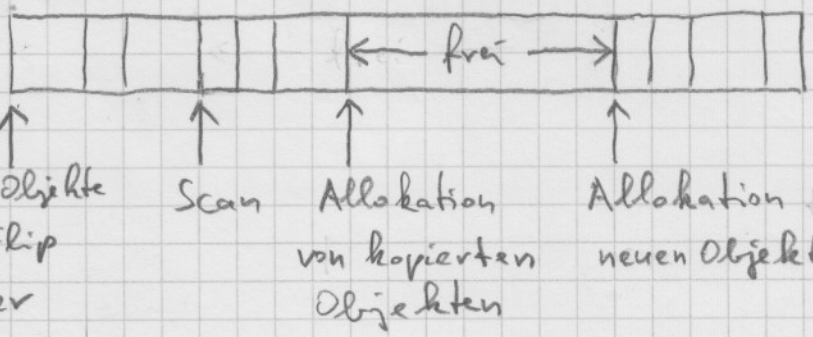
b) Nach dem Kopieren eines Objektes ist dessen Inhalt irrelevant; das Broken-Heart-Flag und der Forward-Pointer können ihn bedenkenlos überschreiben.

c) 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.

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!

2.2.3. GC von Lieberman 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 sind in der gleichen Region. Jüngere Regionen werden öfter entmüllt als ältere. Vorteil: Sehr viel weniger Kopierarbeit als bei Baker. Komplikation: Objektreferenzen über die Regionengrenzen hinweg müssen behandelt werden.

C "Lose Enden"

1. Zwei Bedeutungen des Schlüsselwortes "static"

a) 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: Die Initialisierung der statischen Variablen passiert nur einmal!

b) 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. Bsp.:

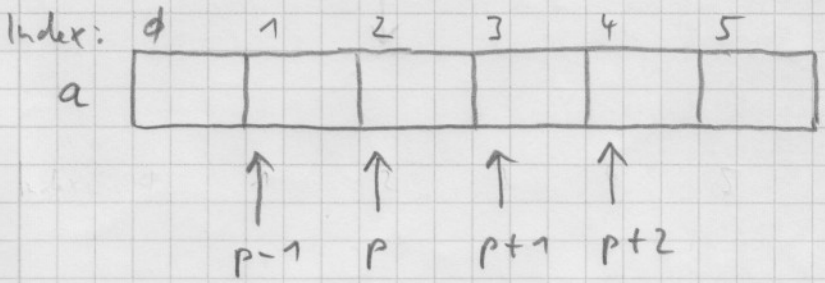
```
#include <stdio.h>
static int counter;
static int f(int x) { ... }
void dolt(int what) { ... }
```

Nach außen ist nur die Funktion dolt() sichtbar.

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 weiter hinten (weiter vorne) im Speicher befindet:

(nicht Bytes!)



Natürlich können die so errechneten Zeiger auch dereferenziert werden: $*(p+2)$ liefert das Objekt $a[4]$.

Erinnerung: $a \equiv \&a[0]$, also ein Zeiger aufs erste Element

$$\Rightarrow \boxed{\begin{aligned} a[n] &\equiv *(a+n) \\ \&a[n] &\equiv a+n \end{aligned}}$$ } Das gilt für bel. Zeiger a !!!

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, von einer Funktion zurückgegeben 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: Mitführen von Zeigern statt Indizierung

Bsp.: Kopierschleife

```
while (n--)  *q++ = *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 (*cmdlist[i].fkt)(arg);
        }
    }
    /* command not found */
    error("command not found");
    return -1;
}
```

Deklaration mit drei Punkten in der Parameterliste,
z.B. int printf(char *fmt, ...);
Zugriff auf die Argumente innerhalb der Funktion durch
eines 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);
```