KSP Aufgaben

Inhaltsverzeichnis

KSP Vorbereitung zum Praktikum

Installation Entwicklungsumgebung

Alternative 1: Installation eigenes Linux

Installieren Sie Linux auf Ihrem Rechner (Tipp: Ubuntu). Machen Sie sich mit dem Installieren von Paketen vertraut (`apt`). Sie benötigen einen Editor Ihrer Wahl, sowie die üblichen Entwicklungswerkzeuge (gcc, bison, flex, make, cmake). Außerdem sind folgende Programme ganz hilfreich: Debugger (gdb) und Speicherprüfer (valgrind).

Alternative 2: Verwendung der vorbereiteten VirtualBox VM

Unter https://hbx.fhhrz.net/getlink/fi73PGkxAW77utjc3F9d1e/debs%2012.ova können Sie ein VM Image für Oracle VirtualBox (https://www.virtualbox.org/) herunterladen, das alle Anforderungen zur Entwicklung der Ninja VM erfüllt. Wenn Sie bereits Linux auf Ihrem System installiert haben, empfiehlt es sich trotzdem die bereitgestellte VM zu verwenden, da dort alle benötigten Pakete installiert sind. Beachten Sie bitte: Wir geben keinen Support für Probleme die auf Eigeninstallationen auftreten. D.h. Benutzen Sie Ihr eigenes System, kümmern Sie sich bitte selbst darum, dass alles läuft. Was in der Regel heißt bestimmte Softwarepakete zu installieren.

Das Passwort für den Benutzer (`student`) der bereitgestellten Virtualbox VM: `student`

Hier noch eine Liste an installierten/verfügbaren IDEs und Editoren:

- Atom (Open Source Editor und IDE)
- Visual Studio Code (Open Source Editor und IDE)
- Sublime Text 3 (installiert, keine Lizenz)
- VIM (Editor, IDE, ...)
- CLION (Jetbrains, muss selbstständig installiert werden. Lizenz kann kostenlos geholt werden: https://www.jetbrains.com/shop/eform/students)

Generell sollte für diesen Kurs auch MacOSX oder ein Windows 10 mit WSL2 ausreichen, solange alle notwendigen Pakete installiert sind. Hier sind sie jedoch auch auf sich alleine gestellt.

Alternative 3: Programmierumgebung für KSP mit CLion und WSL2

1. Windows Subsystem for Linux (WSL) installieren (Tutorial)
2. Ubuntu(WSL) installieren (Tutorial)
   a. Ubuntu starten und die Erstkonfiguration abschließen
   b. Paketquellen und Pakete aktualisieren:
   `sudo apt update && sudo apt upgrade -y`
   c. Benötigte Pakete installieren:
   `sudo apt install build-essential git cmake valgrind python3 python3-pip gdb systemd-coredump -y`
   d. Benötigtes python Paket installieren:
   `python3 -m pip install pipenv`
   Manche dieser Pakete werden nur für die Abgaben der Hausübungen benötigt, sie können jedoch jetzt schon installiert werden.
   e. Optional: Installation von Python Addon für `gdb`: gd-peda

Der Editor CLION dient hier nur als ein Beispiel. Sie können natürlich jeden anderen Editor benutzen. Da das Einrichten und Konfigurieren von CLION etwas aufwändiger ist, wird es hier beispielhaft dargestellt.

3. CLion installieren und mit Hochschullizenz aktivieren *Zum Erhalt der Lizenz muss ein Account*

mit Ihrer Hochschuladresse angelegt werden. (Infos zur Education Lizenz) https://www.jetbrains.com/shop/eform/students

a. WSL Toolchain einrichten (Tutorial)

1. Sollte beim Einrichten der Toolchain der Fehler *"Test CMake run finished with errors"* auftreten, so muss in der WSL folgender Befehl ausgeführt werden:

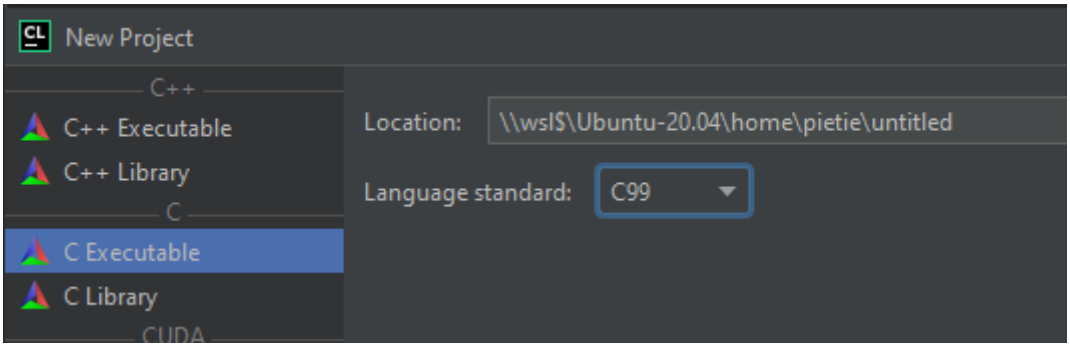
```
sudo bash -c 'cat > /etc/wsl.conf << EOF
[automount]
enabled = true
options = "metadata"
EOF'
```

b. (*Optional*) Valgrind in CLion einrichten (Tutorial)

4. Ich empfehle an dieser Stelle nun ein git Projekt im WSL zu erstellen oder es von dem THM GitLab zu clonen

```
$git clone git@git.thm.de:arin07/KSP_public.git
```

5. Wenn nun in CLion ein neues Projekt erstellen können wir direkt den Ordner aus dem WSL öffnen. (Z.B. `\\wsl$\Ubuntu-20.04...` wobei ⋯ dann der Pfad zu dem git Ordner in WSL ist).
Hierbei sollte natürlich "**C Executable**" mit dem **C99** Standard gewählt werden:



6. In dem Ordner wird dann mit CLion eine `CMakeLists.txt` Datei erstellt, welche dann mindestens folgendes enthalten sollte:

Datei: CMakeLists.txt

```
cmake_minimum_required(VERSION 3.16)
project(njvm C)

set(CMAKE_C_STANDARD 99)

add_compile_options(-g -Wall -pedantic)

add_executable(njvm njvm.c)
```

Nun sollte dem Programmieren mit Clion unter WSL nichts im Wege stehen!

Ein erstes Programm

1. Editieren Sie folgendes Programm mit einer IDE / einem Editor Ihrer Wahl in einer Datei `hello.c`:

```c
#include <stdio.h>
int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    return 0;
}
```

3. Übersetzen und binden Sie das Programm mit der Kommandozeile

```
$ gcc -g -Wall -std=c99 -pedantic -o hello hello.c
```

4. Lassen Sie das Programm laufen mit der Kommandozeile

```
$ ./hello
```

Es muss den Text "Hello, world!" (ohne die Anführungszeichen, mit einem Zeilenumbruch am Ende) ausgeben.

KSP Aufgabe 0

1. Entwerfen Sie ein C-Programm `njvm.c`, das nur aus der Funktion `int main(int argc, char *argv[])` besteht und beim Aufruf die beiden Strings `Ninja Virtual Machine started\n` und `Ninja Virtual Machine stopped\n` ausgibt.
2. Modifizieren Sie Ihr Programm aus 1. so, dass es alle Kommandozeilenargumente vor den beiden Strings ausgibt.
3. Modifizieren Sie Ihr Programm aus 1. so, dass das Programm sowohl sinnvoll auf die Kommandozeilenargumente `--version` und `--help` reagiert als auch unbekannte Argumente zurückweist. Sie können die folgende Referenzimplementierung als Vorbild für das Verhalten Ihres Programms nehmen (machen Sie das Programm nach dem Download ggf. mit `chmod +x njvm` ausführbar, bevor Sie es mit `./njvm` aufrufen): njvm

Hinweise Aufgabe 0

1. Verwenden Sie den C-Compiler `gcc` mit den Compilerschaltern `-g -Wall -std=c99 -pedantic` und beheben Sie alle Fehler und Warnungen, die der Compiler evtl. ausgibt. Wenn Sie eine Warnung ignorieren wollen, müssen Sie EXAKT erklären können, warum diese spezielle Warnung in diesem speziellen Fall wirklich unschädlich ist.
2. Sie beginnen bereits in dieser Übung, das Programm zu entwickeln, das Sie am Ende des Semesters als Ihre zweite Hausübung abgeben werden und das dann einen erheblichen Umfang

haben wird. Bemühen Sie sich also schon jetzt um Klarheit in der Programmierung, ausreichend viele und aussagekräftige Kommentare, konsistentes Einrücken, usw.!

Beispielausgabe Referenzimplementierung `njvm`

```
$ ./njvm
Ninja Virtual Machine started
Ninja Virtual Machine stopped
$ ./njvm --help
usage: ./njvm [option] [option] ...
  --version        show version and exit
  --help           show this help and exit
$ ./njvm --version
Ninja Virtual Machine version 0 (compiled Sep 23 2015, 10:36:52)
$ ./njvm --foo
unknown command line argument '--foo', try './njvm --help'
$ ./njvm bar
unknown command line argument 'bar', try './njvm --help'
```

KSP Aufgabe 1

Konstruieren Sie die Ninja-VM Version 1, die die in Tabelle 1 gelisteten Instruktionen ausführen kann. Jede Instruktion eines Programms wird in einem `unsigned int` gespeichert, wobei der Opcode (die Zahl, die in der Tabelle hinter jeder Instruktion steht) in den obersten 8 Bits des `unsigned int` abgelegt wird. Die restlichen 24 Bits dienen zur Aufnahme eines Immediate-Wertes. Der wird im Augenblick nur bei der Instruktion `pushc` benötigt, wo er den auf den Stack zu legenden Wert darstellt.

Tabelle 1. VM Instruktionen

Instruktion	Opcode	Stack Layout
`halt`	`0`	`... -> ...`
`pushc <const>`	`1`	`... -> ... value`
`add`	`2`	`... n1 n2 -> ... n1+n2`
`sub`	`3`	`... n1 n2 -> ... n1-n2`
`mul`	`4`	`... n1 n2 -> ... n1*n2`
`div`	`5`	`... n1 n2 -> ... n1/n2`
`mod`	`6`	`... n1 n2 -> ... n1%n2`
`rdint`	`7`	`... -> ... value`
`wrint`	`8`	`... value -> ...`
`rdchr`	`9`	`... -> ... value`
`wrchr`	`10`	`... value -> ...`

Den Effekt einer jeden Instruktion auf den Stack listet die Spalte Stack Layout in der Tabelle 1. Punkte `·····` in der Tabelle stehen jeweils für den durch die Instruktion nicht veränderten Stackinhalt; der `Top-of-Stack` befindet sich jeweils rechts. Die Operanden für arithmetische

Operationen sind immer ganze Zahlen, genauso wie die bei `rdint` und `rdchr` eingelesenen bzw. bei `wrint` und `wrchr` ausgegebenen Werte.

Testprogramme

Die drei hier gelisteten Programme sollen fest in die VM einprogrammiert werden. Der Benutzer kann durch Eingabe eines Kommandozeilenargumentes beim Aufruf der VM eines der Programme auswählen, das dann in den Programmspeicher der VM kopiert wird. Nach dem Anlaufen der VM wird das Programm erst in einer lesbaren Form aufgelistet; eine mögliche Darstellung können Sie der Referenzimplementierung entnehmen. Anschließend wird es ausgeführt. Beachten Sie bitte, dass "Auflisten" und "Ausführen" zwei getrennte Aktionen auf dem Programm sind, die strikt hintereinander passieren sollen.

Programm 1

Ninja Programmfragment	Instruktionssequenz

`writeInteger((3 + 4) * (10 - 6));`
`writeCharacter('\n');`
	`pushc 3`
	`pushc 4`
	`add`
	`pushc 10`
	`pushc 6`
	`sub`
	`mul`
	`wrint`
	`pushc 10`
	`wrchr`
	`halt`

Programm 2

Ninja Programmfragment	Instruktionssequenz

`writeInteger(-2 * readInteger() + 3);`
`writeCharacter('\n');`
	`pushc -2`
	`rdint`
	`mul`
	`pushc 3`
	`add`
	`wrint`
	`pushc '\n'`
	`wrchr`
	`halt`

Programm 3

Ninja Programmfragment	Instruktionssequenz

`writeInteger(readCharacter());`
`writeCharacter('\n');`
	`rdchr`
	`wrint`
	`pushc '\n'`
	`wrchr`
	`halt`

Hier ist wieder die Referenzimplementierung: njvm

Hinweise Aufgabe 1

1. Die Compilerschalter sind wie in Aufgabe 0 auf `-g -Wall -std=c99 -pedantic` zu setzen.
2. Ihre VM benötigt einen Programmspeicher und einen Stack. Beide können Sie global definieren, damit Sie von überall darauf zugreifen können. Wählen Sie die Basis-Datentypen mit Bedacht, insbesondere im Hinblick auf vorzeichenbehaftete (`signed`) vs. vorzeichenlose (`unsigned`) Größen!
3. Ihre VM wird zwei Interpreter für den Programmcode beinhalten: einen für das **Listen des Programms** und einen für die eigentliche **Ausführung**. Beide sind strukturell gleich aufgebaut: eine Schleife (*worüber?*) und darin eine Mehrfachverzweigung (*wozu?*). Beginnen Sie mit dem Programmlister - wenn Sie den haben, ist der Interpretierer für die Ausführung nicht mehr schwer. Beachten Sie aber die zwei unterschiedlichen Abbruchkriterien in den beiden Interpretierern (*welche genau?, und warum sind die eigentlich unterschiedlich?*).
4. Ihr Programm darf unter keinen Umständen abstürzen! Natürlich gibt es Fehlersituationen, in denen Sie das Programm nicht vernünftig weiterlaufen lassen können - dann geben Sie eine verständliche Fehlermeldung aus und beenden das Programm. Eine solche Situation betrifft beispielsweise das Teilen durch 0, eine andere den Stack-Überlauf bzw. -Unterlauf (*wann können diese beiden Fehler auftreten?*).
5. Sie müssen die drei Programme *manuell*, also *per Hand* assemblieren, solange es noch keinen Assembler gibt.
    a. Wie macht man das?
    Nehmen wir als Beispiel `pushc 3` → Der **Opcode** für `pushc` ist `1` (er kommt in die höchsten 8 Bits); die **Immediate-Konstante** ist `3` (sie kommt in die untersten 24 Bits). Also ist das Bitmuster des Befehls (4 Bytes) `00000001 00000000 00000000 00000011`, oder hexadezimal `0x01000003`. Auf gar keinen Fall arbeitet man hier mit Dezimalzahlen! Es geht aber viel eleganter: verwenden Sie den C-Präprozessor, um eine lesbare Codierung der drei kleinen Programme zu ermöglichen! Beispielsweise könnte der Anfang des ersten Programms im C-Quelltext lauten:

```
unsigned int code1[] = {
(PUSHC << 24) | IMMEDIATE(3),
```

```
(PUSHC << 24) | IMMEDIATE(4),
(ADD << 24),
...
}
```

mit den entsprechenden Definitionen für die Opcodes, z.B.

```
#define PUSHC 1
#define ADD 2
```

und für das Kodieren des Immediate-Wertes (*warum braucht man das?*)

```
#define IMMEDIATE(x) ((x) & 0x00FFFFFF)
```

1. Der Immediate-Wert kann auch **negativ** sein. Vielleicht haben Sie Verwendung für dieses Makro:

```
#define SIGN_EXTEND(i) ((i) & 0x00800000 ? (i) | 0xFF000000 : (i))
```

Was macht das Makro eigentlich? Und wie genau funktioniert es?

ACHTUNG: Es ist unbedingt erforderlich, die Makros aus 5. und 6. genau zu verstehen! Machen Sie sich die Wirkungsweise an Beispielen klar und rechnen Sie damit, im Praktikum dazu befragt zu werden!

KSP Aufgabe 2

1. Ab dieser Aufgabe haben Sie einen Assembler zur Verfügung, damit das lästige Kodieren der Befehle nicht mehr per Hand ausgeführt werden muss. Denken Sie daran, dass auch das Programm `nja` ausführbar sein muss! Schreiben Sie die drei kleinen Testprogramme aus der vorigen Aufgabe in jeweils eine Datei, die die Endung `.asm` haben sollte. Lassen Sie die Programme assemblieren; die entstehenden Dateien sollten die Endung `.bin` haben. Inspizieren Sie diese Binärdateien mit dem Kommando `hexdump -C`. Versuchen Sie genau zu erklären, wie die beobachtete Ausgabe zustande kommt! Studieren Sie dazu die Beschreibung des Ninja-Binärformats.

Beachten Sie bitte, dass in Ihrer `.asm` Datei ein Newline `\n` am Dateiende benötigt wird.

2. Jetzt ändern Sie Ihre VM vom vorigen Aufgabenblatt so ab, dass das Binärprogramm aus einer Datei in den Programmspeicher geladen wird.

a) Der Name des zu ladenden Programms soll als Kommandozeilenargument übergeben werden.

b) Bevor Sie etwas mit dem Inhalt der Datei tun können, müssen Sie die Datei öffnen:

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

c) Was Sie mit dem Inhalt einer Ninja-Binärdatei tun müssen, um das darin enthaltene Programm starten zu können, steht ebenfalls in der Beschreibung des Ninja-Binärformats .

d) Sie werden die Funktion `fread()` zum Lesen aus einer Datei brauchen:

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

e) Ebenso brauchen Sie die Funktion `malloc()` zur Speicheranforderung:

```
void *malloc(size_t size);
```

f) Wenn Sie alle Informationen aus der Datei verwertet haben, wird die Datei wieder geschlossen:

```
int fclose(FILE * fp);
```

1. Ergänzen Sie nun Ihre VM auf die Version 2, in dem Sie die neuen Instruktionen aus Tabelle 2 implementieren. Eine Diskussion von globalen Variablen und den zugehörigen Instruktionen finden Sie hier, die Diskussion von lokalen Variablen und Stackframes mit ihren Instruktionen gibt's hier.
2. Prüfen Sie nun das Funktionieren Ihrer VM durch Assemblieren und Ausführen der drei kleinen Programme aus Aufgabenteil 1 sowie der beiden Testprogramme prog1.asm und prog2.asm. Es wird dringend empfohlen, mindestens fünf weitere selbstgewählte Berechnungen im Stackmaschinen-Assembler zu programmieren und ausführen zu lassen. Sie sollten in der Lage sein, zu jedem Zeitpunkt der Ausführung den Stack aufzeichnen zu können!
3. Und hier wie immer die Referenzimplementierung: njvm

Tabelle 2. VM Instruktionen

Instruktion	Opcode	Stack Layout
`pushg <n>`	11	`... -> ... value`
`popg <n>`	12	`... value -> ...`
`asf <n>`	13	
`rsf`	14	
`pushl <n>`	15	`... -> ... value`
`popl <n>`	16	`... value -> ...`

Hinweise Aufgabe 2

1. Alle benötigten Bibliotheksfunktionen kann man im Online-Manual nachschlagen. Das ruft man

mit dem Kommando `man` auf. In diesem Manual stehen auch alle Kommandos drin, die das System kennt. Wenn man also z.B. nicht weiß, wie das Manual funktioniert: `man man` hilft. Das Manual ist sehr "dicht" geschrieben; in jedem Halbsatz stehen mehrere wichtige Informationen drin. Studieren Sie deshalb die Manual-Einträge genau; nur überfliegen reicht nicht!

2. Vergessen Sie nicht, alle Rückgabewerte zu prüfen; die aufgerufene Funktion könnte aus diversen Gründen fehlgeschlagen sein (mögliche Ursachen stehen ebenfalls im Manual).

KSP Aufgabe 3

1. Implementieren Sie die neuen Instruktionen zum Testen von Zahlen (`eq`, `ne`, `lt`, `le`, `gt`, `ge`), die die numerischen Vergleiche (`==`, `!=`, `<`, `⇐`, `>`, `>=`) repräsentieren. Die Auswirkungen auf den Stack können Sie wieder in der Spalte Stack Layout der Tabelle 3 sehen. Das Ergebnis eines Vergleichs, ein Boole'scher Wert, wird durch die ganze Zahl `0` für `false` bzw. `1` für `true` dargestellt. Natürlich gibt's einen Assembler , der auch die neuen Instruktionen assemblieren kann.

Tabelle 3. VM Instruktionen

Instruktion	Opcode	Stack Layout
`eq`	17	`... n1 n2 -> ... n1==n2`
`ne`	18	`... n1 n2 -> ... n1!=n2`
`lt`	19	`... n1 n2 -> ... n1<n2`
`le`	20	`... n1 n2 -> ... n1<=n2`
`gt`	21	`... n1 n2 -> ... n1>n2`
`ge`	22	`... n1 n2 -> ... n1>=n2`
`jmp <target>`	23	`... -> ...`
`brf <target>`	24	`... b -> ...`
`brt <target>`	25	`... b -> ...`

1. Nehmen Sie sich dann den unbedingten Sprung `jmp` sowie die beiden bedingten Sprünge (*branch on false* `brf` und *branch on true* `brt`) vor. Der Immediate-Wert in diesen Instruktionen gibt das Sprungziel an. Die bedingten Sprünge prüfen das oberste Stack-Element, um zu entscheiden, ob gesprungen wird. Wenn nicht gesprungen wird, kommt die nächste Instruktion zur Ausführung. Die Auswirkungen auf den Stack können Sie wieder in der Spalte Stack Layout in Tabelle 3 sehen. Der Assembler hat eine neue Kommandozeilenoption `--map`; was kann man eigentlich mit der anfangen?

Teilaufgabe: Interaktiver Debugger

Nun kommt die erste etwas größere Aufgabe: **ein interaktiver Debugger** für Ihre VM. Die Instruktionen werden komplizierter, die auszuführenden Programme umfangreicher - man wünscht sich, mehr über den inneren Zustand der VM zu erfahren, speziell wenn Fehler auftreten.

a) Im Gegensatz zu den früheren Aufgaben soll ab jetzt das Programm nach dem Laden nicht mehr automatisch gelistet werden, sondern einfach nur ausgeführt werden. Wenn man aber die VM mit dem Kommandozeilenschalter `--debug` startet, soll sich nach dem Laden des Programms der

interaktive Debugger melden und auf Kommandos warten, die er dann ausführt.

b) Schreiben Sie eine kurze, aber exakte Spezifikation, welche Kommandos Ihr Debugger verstehen soll. Als kleiner Hinweis für diejenigen, die sich wenig unter einem solchen Teil vorstellen können: Was wünscht man sich, um den Programmverlauf verfolgen oder vielleicht sogar beeinflussen zu können? Lassen Sie doch einfach mal dieses Beispielprogramm 1 laden und stellen Sie sich vor, es würde nicht das gewünschte Ergebnis (den größten gemeinsamen Teiler zweier positiver Zahlen) liefern. Natürlich können Sie das auch ganz praktisch ausprobieren, indem Sie irgendeine Instruktion (z.B. das zweite `brf`) in eine andere Instruktion (z.B. `brt`) umändern, oder noch besser: von Ihrem anderen Grupenmitglied umändern lassen, ohne dass Sie wissen, was da passiert ist. Was braucht man dann für Hilfsmittel, um den Fehler zu lokalisieren?

Als Minimalmenge müssen die Kommandos: *Anzeigen des Stacks*, *Anzeigen der statischen Daten*, *Listen des Programms*, *Ausführen des nächsten Befehls*, *Weiterlaufen ohne Anhalten* und *Verlassen der VM* aufgenommen werden.

Sehr zweckmäßig ist auch das *Setzen eines Breakpoints* (wenn das Programm bei der Ausführung später dort vorbeikommt, hält es an und der Debugger übernimmt die Kontrolle). Anregungen können Sie sich natürlich wie immer auch von der Referenzimplementierung holen: njvm

c) Implementieren Sie nun die von Ihnen vorgesehen Kommandos des Debuggers. Prüfen Sie mit dem Beispielprogramm 1 und Beispielprogramm 2 sowie anderen, von Ihnen geschriebenen Testprogrammen, ob der Debugger das leistet, was Sie sich von ihm erwarten. Wahrscheinlich fallen Ihnen noch Verbesserungen ein - dann gehen Sie zurück zu b).

KSP Aufgabe 4

Jetzt wird's ernst... ;-) Mit der Lösung dieser Aufgabe steht Ihnen das komplette Ninja (mit Ausnahme der Array- und Record-Objekte) zur Verfügung. Sie können damit z.B. die folgenden Programme ausführen: a) ggT b) n! rekursiv c) n! iterativ

Aber schön der Reihe nach...

Neue Instruktionen der VM

Tabelle 4. VM Instruktionen

Instruktion	Opcode	Stack Layout
`call <target>`	`26`	`... -> ... ra`
`ret`	`27`	`... ra -> ...`
`drop <n>`	`28`	`... a0 a1...an-1 -> ...`
`pushr`	`29`	`... -> ... rv`
`popr`	`30`	`... rv -> ...`
`dup`	`31`	`... n -> ... n n`

1. Hier ist eine Diskussion der Instruktionen zum Unterprogrammaufruf und in Tabelle 4 eine Liste aller Instruktionen der VM dieser Version.

2. Realisieren Sie Unterprogrammsprünge und -rücksprünge mit den Instruktionen `call` und `ret`. Hier ist ein Testprogramm ohne Argumente und ohne Rückgabewert. Sie sollten die Ausführung mit Hilfe Ihres Debuggers im Einzelschrittverfahren genau verfolgen können.
3. Implementieren Sie die Instruktion `drop` und testen Sie den Zugriff auf die Argumente einer Prozedur mit diesem Testprogramm.
4. Fügen Sie das Rückgaberegister zur Maschinenarchitektur hinzu und implementieren Sie die Instruktionen `pushr` und `popr`. Sie können die Rückgabe eines Wertes mit diesem Testprogramm überprüfen.
5. Dann sollte schließlich dieses Testprogramm ohne weitere Arbeit funktionieren.
6. Implementieren Sie die Instruktion `dup`, die den obersten Stackeintrag dupliziert: `⋯ n -> ⋯ n n`
7. Der zu dieser Aufgabenstellung gehörende Assembler ist natürlich auch verfügbar: njа
8. Zu guter Letzt hier wieder die Referenzimplementierung: https://git.thm.de/arin07/KSP_public/-/blob/master/aufgaben/a4/njvm

Der Ninja-Compiler

Nun gibt's eine erste Version des Ninja-Compilers - noch ohne Arrays, Records und anderen Schnickschnack, aber mit Prozeduren und Funktionen, Kontrollstrukturen, Zuweisungen und Ausdrücken. Die Grammatik zusammen mit den Tokens und den vordefinierten Bezeichnern beschreibt, wie ein korrektes Ninja-Programm (in unserem noch etwas reduzierten Sprachumfang) aussieht.

1. Übersetzen Sie die drei ganz oben genannten Testprogramme mit Hilfe des Ninja-Compilers. Der Output ist Ninja-Assembler und kann z.B. mit einem Texteditor angeschaut werden. Versuchen Sie, jedes vom Compiler generierte Assembler-Statement einer Ninja-Quelltext-Anweisung zuzuordnen. Wenn Sie Fragen dazu haben, diskutieren Sie diese bitte mit den Betreuern im Praktikum!
2. Assemblieren Sie die drei Testprogramme und lassen Sie sie durch Ihre VM ausführen. Verfolgen Sie die Ausführung mit dem Debugger Ihrer VM.
3. Schreiben Sie mindestens fünf weitere Testprogramme, entweder in Ninja oder in Assembler, um sicherzugehen, dass die VM korrekt funktioniert. Loten Sie dabei auch Konstruktionen aus, die bisher nicht Bestandteil der Tests waren. Ein Beispiel: Kann eine Funktion in ihrem Return-Statement eine andere Funktion aufrufen? Geht das auch geschachtelt?
4. Welche Sequenz von Instruktionen erzeugt der Compiler für das logische `und` und das logische `oder`? Erklären Sie genau, wie damit die Kurzschlussauswertung implementiert wird!

KSP Aufgabe 5

1. Die Aufgabenstellung hier ist ganz einfach, bedingt aber einige Änderungen in der VM: Verlagern Sie die Rechenobjekte (bis jetzt sind das ja nur ganze Zahlen) auf den Heap und halten Sie in der Static Data Area, auf dem Stack und im Rückgabewertregister nur noch Zeiger in den Heap. Der Speicherplatz für die Objekte wird mit `malloc()` beschafft und innerhalb Ihres

Programms nicht wieder freigegeben (das ist nämlich ohne Garbage Collector praktisch nicht möglich, und wie ein solches Teil funktioniert, werden wir in der Vorlesung erst später behandeln). Was genau sind unsere "Rechenobjekte"? Nun, das ist einfach: alle Objekte in der Static Data Area, alle Objekte auf dem Stack (mit Ausnahme von Rücksprungadressen und Framepointerwerten) sowie der Inhalt des Rückgabewertregisters. Alle anderen in der VM benutzten Dinge, wie z.B. Codespeicher, Programmzähler, Stackpointer und Framepointer sind keine Rechenobjekte: man kann sie von einem Ninja-Programm aus nicht unmittelbar beeinflussen.

2. Der Instruktionssatz ist gegenüber Aufgabe 4 unverändert; auch der Compiler und der Assembler bleiben exakt gleich (bis auf die Versionsnummern).
3. Ihr Debugger muss beim Anzeigen des Stacks kenntlich machen, welche Einträge Objektreferenzen und welche Einträge einfache Zahlen (Rücksprungadressen und gespeicherte Framepointer) sind.
4. Bauen Sie in Ihren Debugger eine Möglichkeit ein, Objekte zu inspizieren. Warum? Wenn Sie z.B. wissen wollen, welche Zahl ganz oben auf dem Stack liegt, dann lassen Sie sich den Stack anzeigen. Dort finden Sie aber nur die Adresse des Objektes auf dem Heap. Also wünschen Sie sich eine Möglichkeit, den Inhalt eines Objektes bei gegebener Objektadresse anzuschauen. Das wird im Übrigen auch in der übernächsten Aufgabe gebraucht, wenn in Objekten Referenzen auf andere Objekte gespeichert werden.
5. Hier wie immer die Referenzimplementierung: njvm

KSP Aufgabe 6

1. In dieser Version der VM wollen wir als Rechenobjekte beliebig große ganze Zahlen zulassen. Diese neuen Rechenobjekte ersetzen die alten 32-Bit-Zahlen und haben ihren Speicherplatz ebenfalls auf dem Heap. Auf dem Stack, in der *Static Data Area* und im Rückgabewertregister stehen wie gehabt Zeiger auf die Rechenobjekte.
2. Die Algorithmen zum Rechnen mit beliebig großen ganzen Zahlen (*Multiple Precision Arithmetic*) sind entnommen aus [D. Knuth: *The Art of Computer Programming, Vol. 2, Seminumerical Algorithms*]. Eine Implementierung in C finden Sie hier. Entpacken Sie das Paket und studieren Sie die darin enthaltene Datei README. Bauen Sie das Paket und lassen Sie die Tests laufen! Machen Sie sich mit der Struktur der Zahlendarstellung vertraut! Hinweise: Jede Ziffer der Zahl ist ein Byte, d.h. die Zahldarstellung benutzt die Basis 256. Die Ziffern werden in einem genügend großen Array gespeichert, das zusammen mit der Größenangabe in einer Struktur ("Objekt") auf dem Heap steht.
3. Die *Multiple Precision Arithmetic* braucht eine winzige Support-Bibliothek, deren Funktionalität aber in Ihrer VM schon enthalten ist. Stellen Sie die benötigten Funktionen aus Ihrer VM zur Verfügung!
4. Studieren Sie das Benutzer-Programmierinterface und passen Sie Ihre VM an. Achten Sie darauf, ALLE Operationen mit Rechenobjekten der Bibliothek zu überlassen; dazu gehören auch die Vergleiche!
5. Der Instruktionssatz ist gegenüber Aufgabe 5 unverändert; auch der Compiler und der Assembler bleiben exakt gleich (bis auf die Versionsnummern).

6. Hier wie immer die Referenzimplementierung: njvm

Aufgaben für Tests a) Schreiben Sie ein kleines Ninja-Programm zur Berechnung von 100! = 1*2*3*...*100. Fällt Ihnen am Ergebnis etwas auf? Kann das denn überhaupt richtig sein?

b) Die Fibonacci-Folge ist definiert als `F(0) = 0, F(1) = 1, und F(n) = F(n-1) + F(n-2)` für alle `n > 1`. Schreiben Sie ein kleines Ninja-Programm zur Berechnung von `F(100)`. Hinweis: Nur die iterative Version wird in annehmbarer Zeit zum Ergebnis führen!

c) Schreiben Sie ein kleines Ninja-Programm zur Berechnung der Summe aller Brüche `\frac{1}{i}` für `i = 1, \dots, 100` (die einhundertste "harmonische Zahl"). Das Ergebnis soll als exakter Bruch in gekürzter Darstellung angegeben werden. Hinweise: Halten Sie den Zähler und den Nenner des Ergebnisses in zwei verschiedenen Variablen. Wenn Sie einen neuen Term addieren wollen, müssen Sie die Brüche auf einen gemeinsamen Nenner bringen. Beim Kürzen (das Sie entweder bei jeder Rechnung oder aber einmal ganz am Ende aller Rechnungen durchführen können) leistet der größte gemeinsame Teiler gute Dienste! Können Sie die Zahl in Dezimalschreibweise mit z.B. 10 Stellen nach dem Komma ausgeben? Dazu sind nur Ganzzahloperationen nötig!

KSP Aufgabe 7

In dieser Aufgabenstellung soll das Rechnen mit Komponenten von Records und Arrays implementiert werden. Bei uns sind diese beiden Datenstrukturen (im Gegensatz zu C) "richtige Objekte", d.h. alle Instanzen von Records und Arrays leben auf dem Heap und werden nur über Zeiger zugegriffen. Das ist in Ninja nicht anders als in Java, und deshalb heissen die Komponenten auch "Instanzvariable". Um Ihnen einen Eindruck zu vermitteln, welche Berechnungen nun möglich sind, gibt es hier drei Beispielprogramme:

- listrev.nj (Umdrehen einer Liste von Zahlen)
- twodim.nj (Belegen und Ausgeben eines zweidimensionalen Arrays)
- matinv.nj (Invertieren einer 2x2-Matrix mit Brüchen als Komponenten)

Vorgehen

1. Studieren Sie die neu hinzugekommenen Instruktionen des dann kompletten Befehlssatzes unserer VM, wobei Sie die Diskussion der Struktur von Objekten und die Erläuterungen zu Referenzvergleichen hinzuziehen sollten.
2. Überprüfen Sie Ihr Verständnis der neuen Befehle! Vorschlag: Sie lassen das o.g. Programm `matinv.nj` übersetzen und entnehmen dem entstandenen Assemblerprogramm die Funktionen newFraction , subFraction und newMatrix . Dann ergänzen Sie die Funktionen Zeile für Zeile um einen aussagekräftigen Kommentar, was da jeweils genau passiert. Sie sollten zu jedem Zeitpunkt den Stack der VM zeichnen können!
3. Implementieren Sie die neuen Instruktionen. Sie können mit der Umsetzung der Objektstruktur beginnen. Es gibt ja zwei Sorten von Objekten: primitive Objekte (die eine Anzahl Bytes speichern) und zusammengesetzte Objekte (die eine Anzahl Objektreferenzen speichern). Das höchste Bit des Zählers, der in jedem Objekt enthalten ist, bestimmt, ob der Zähler Bytes oder Instanzvariable zählt. Deswegen kann man als Interface zum Objektspeicher die beiden

folgenden Funktionen vorsehen:

   a. `ObjRef newPrimitiveObject(int numBytes);`

   b. `ObjRef newCompoundObject(int numObjRefs);`

4. Dann realisieren Sie `new`, `getf` und `putf`. Denken Sie daran, alle Instanzvariablen mit `nil` zu initialisieren! Als nächstes kommen die Instruktionen für Arrays an die Reihe; bitte auch hier die Initialisierung nicht vergessen. Und wo wir schon dabei sind: die lokalen Variablen einer Methode müssen ebenfalls mit `nil` initialisiert werden, genauso wie die globalen Variablen. Noch ein Hinweis: Ihre VM muss erkennen, wenn auf ein Objekt zugegriffen werden soll, aber nur eine `nil`-Referenz vorliegt. Ebenso müssen Sie einen Zugriff auf ein Array mit unzulässigem Index abfangen. Testen Sie alle Befehle und Fehlerbedingungen ausführlich! Vergessen Sie nicht die Instruktion `getsz` zum Bestimmen der Größe eines Objektes, sowie die Instruktionen zu den Referenzvergleichen!
5. Passen Sie Ihren Debugger der veränderten Situation an. Insbesondere die Inspektion von Objekten wird wahrscheinlich auf das MSB des Zählers Rücksicht nehmen müssen.
6. Natürlich gibt's einen erweiterten Compiler , der mit Records und Arrays umgehen kann (dazu die Grammatik , die Tokens , und die vordefinierten Bezeichner), sowie einen passenden Assembler , der die neuen Instruktionen assemblieren kann.
7. Hier wie immer die Referenzimplementierung: njvm

Vollständige Liste der Instruktionen der VM

Tabelle 5. VM Instruktionen

Instruktion	Opcode	Stack Layout
`halt`	`0`	`... -> ...`
`pushc <const>`	`1`	`... -> ... value`
`add`	`2`	`... n1 n2 -> ... n1+n2`
`sub`	`3`	`... n1 n2 -> ... n1-n2`
`mul`	`4`	`... n1 n2 -> ... n1*n2`
`div`	`5`	`... n1 n2 -> ... n1/n2`
`mod`	`6`	`... n1 n2 -> ... n1%n2`
`rdint`	`7`	`... -> ... value`
`wrint`	`8`	`... value -> ...`
`rdchr`	`9`	`... -> ... value`
`wrchr`	`10`	`... value -> ...`
`pushg <n>`	`11`	`... -> ... value`
`popg <n>`	`12`	`... value -> ...`
`asf <n>`	`13`	
`rsf`	`14`	
`pushl <n>`	`15`	`... -> ... value`
`popl <n>`	`16`	`... value -> ...`
`eq`	`17`	`... n1 n2 -> ... n1==n2`

Instruktion	Opcode	Stack Layout
ne	18	... n1 n2 -> ... n1!=n2
lt	19	... n1 n2 -> ... n1<n2
le	20	... n1 n2 -> ... n1<=n2
gt	21	... n1 n2 -> ... n1>n2
ge	22	... n1 n2 -> ... n1>=n2
jmp <target>	23	... -> ...
brf <target>	24	... b -> ...
brt <target>	25	... b -> ...
call <target>	26	... -> ... ra
ret	27	... ra -> ...
drop <n>	28	... a0 a1...an-1 -> ...
pushr	29	... -> ... rv
popr	30	... rv -> ...
dup	31	... n -> ... n n
new <n>	32	... --> ... object
getf <n>	33	... object --> ... value
putf <n>	34	... object value --> ...
newa	35	... number_elements --> ... array
getfa	36	... array index --> ... value
putfa	37	... array index value --> ...
getsz	38	... object --> ... number_fields
pushn	39	... --> ... nil
refeq	40	... ref1 ref2 --> ... ref1==ref2
refne	41	... ref1 ref2 --> ... ref1!=ref2

Aufgaben für Tests

a. Entwerfen Sie eine Datenstruktur zum Speichern von Bäumen für arithmetische Ausdrücke in Ninja (ganz so, wie wir das in der Vorlesung für C gemacht haben). Stellen Sie die entsprechenden Konstruktoren bereit, sowie eine Methode, die einen Baum mit korrekter Einrückung ausgibt. Überprüfen Sie Ihr Programm mit dem Ausdruck `5 - (1 + 3) * (4 - 7)`: diese Konstruktoraufrufe sollten einen Baum erzeugen, der, wenn er ausgegeben wird, diese Ausgabe produzieren sollte. Nein, das ist kein Fehler in der Aufgabenstellung... ;-)

b. Ersetzen Sie die Ausgabe in a) durch eine Auswertefunktion ("Evaluator"). Kommt bei dem o.g. Beispiel wirklich 17 heraus?

c. Ersetzen Sie jetzt den Evaluator aus b) durch eine Prozedur zum Erzeugen von Ninja-Assembler. Wenn Sie den erzeugten Code für das o.g. Beispiel assemblieren und ausführen lassen, kommt dann wieder 17 heraus?

KSP Aufgabe 8

Statten Sie Ihre VM mit einem kompaktierenden Garbage-Collector nach dem "Stop & Copy"-Verfahren aus!

Vorbereitungen

1. Legen Sie beim Programmstart den Stack dynamisch mit `malloc()` an. Sehen Sie eine Kommandozeilenoption `--stack n` vor, die einen Stack mit `n` KiB bereitstellt (**ACHTUNG**: `n * 1024` Bytes, NICHT Stack Slots, NICHT `n * 1000` Bytes). Der Defaultwert soll n = 64 sein, falls diese Kommandozeilenoption nicht angegeben wird. Hinweis: Sie werden möglicherweise Ihren Test auf Stackoverflow anpassen müssen!
2. Legen Sie beim Programmstart mit `malloc()` dynamisch einen eigenen Heap an, dessen Größe man durch die Kommandozeilenoption `--heap n` festlegen kann. Dabei soll n die Gesamtgröße des Heaps in KiB sein (ACHTUNG: `n * 1024` Bytes, NICHT `n * 1000` Bytes). Der Defaultwert soll `n` = `8192` sein, falls diese Kommandozeilenoption nicht angegeben wird.
3. Teilen Sie den Heap in zwei gleich große Hälften. Ändern Sie das Anlegen von Objekten so ab, dass Sie den benötigten Speicherplatz fortlaufend von einer Hälfte des Heaps nehmen. Brechen Sie Ihr Programm ab, wenn diese Hälfte erschöpft ist.

Garbage Collector

1. Programmieren Sie einen kompaktierenden Garbage-Collector nach dem Verfahren *Stop & Copy*, der anstelle des Programmabbruchs einen Sammeldurchlauf ausführt. Steht nach dem Durchlauf immer noch nicht genügend Platz für die verlangte Allokation zur Verfügung, sollte Ihre VM mit einer passenden Fehlermeldung terminieren.
   - Die Zeiger zu den "Root-Objekten" findet man an allen Stellen in unserer VM, die Objektreferenzen halten: der statische Datenbereich, das Return-Value-Register, diejenigen Stack-Slots, die auf Objekte verweisen, und - nicht vergessen! - die vier Komponenten des Big-Integer-Prozessors `bip`.

Als "Broken Heart Flag" lässt sich sehr gut das zweithöchste Bit der `size` -Komponente benutzen. Der eigentliche "Forward Pointer" (= Byte-Offset in den Zielhalbspeicher, wo sich das kopierte Objekt befindet) belegt die restlichen 30 Bits. Das geht problemlos, da die `size`-Komponente bei bereits kopierten Objekten nicht mehr benötigt wird.

Alternativen zur Speicherung in der `size`-Komponente — also die Verwendung von zusätzlichen Komponenten im struct `ObjRef` — wurden in der Veranstaltung besprochen.

2. Da man beim Integrieren eines Garbage-Collectors leicht einmal einen eigentlich zu verfolgenden Zeiger vergessen kann, stellen Sie eine Kommandozeilenoption `--gcpurge` zur Verfügung, die bewirkt, dass direkt nach einem Sammeldurchlauf der alte Halbspeicher mit Nullen überschrieben wird. So sollten sich Fehler der genannten Art schneller bemerkbar

machen.

3. Instrumentieren Sie Ihren Garbage-Collector so, dass man mit der Kommandozeilenoption `--gcstats` beim Sammeldurchlauf die folgenden Werte angezeigt bekommt:
   - Anzahl der seit dem letzten Durchlauf angelegten Objekte (sowie die davon belegte Zahl von Bytes),
   - Anzahl lebender Objekte (sowie die davon belegte Zahl von Bytes), freier Platz nach dem Sammeldurchlauf (Bytes im momentan benutzten Halbspeicher).
   - **Außerdem**: Sie sollten nach dem Ausführen der `halt`-Instruktion den Garbage-Collector einmal explizit aufrufen, damit man auch für kleine Programme mit wenig Speicherbedarf, die den GC sonst nicht triggern würden, diese statistischen Angaben erhält.
4. Der Instruktionssatz ist gegenüber Aufgabe 7 unverändert; auch der Compiler und der Assembler bleiben exakt gleich (bis auf die Versionsnummern).
5. Hier nun die endgültige Version der Referenzimplementierung: njvm

Testprogramme

Die folgenden zwei etwas größeren Testprogramme muss Ihre VM (bei verschiedenen Heapgrößen) fehlerlos ausführen; dann wird sie mit guter Wahrscheinlichkeit auch als Hausübung akzeptiert.

Die Faktorisierung großer Zahlen

Dieses Ninja-Programm hilft, die folgende Frage zu beantworten: Gegeben sind die Zahlen `10^n+1` bei `n = 1, \ldots, 30`. Was sind die Primfaktoren jeder Zahl?

Bemerkung: Man muss hier zwischen Tests auf Zusammengesetztheit, Primalitätsbeweisen und Methoden zur Faktorisierung unterscheiden. Es gibt verschiedene Algorithmen in jedem der drei Gebiete. Ein gutes Buch, das auch diese Themen behandelt, ist [Henry Cohen: *A Course in Computational Algebraic Number Theory*, Springer 1993].

Ein kleines Computeralgebra-System

Computeralgebra-Systeme werden in ihrem Kern oft als Interpreter in einer LISP-ähnlichen Sprache implementiert. Deshalb wird hier njlisp zur Verfügung gestellt, eine Re-Implementierung des alten muLISP-80-Systems in Ninja. Der Makefile dient zum Zusammenbauen und Laufenlassen. Da das System auf mehreren Ebenen interpretiert abläuft, und diese Interpreter alle geladen werden müssen, folgt eine kurze Erklärung der Makefile-Targets, jeweils zusammen mit einem kleinen Beispiel:

1) `make` → erzeugt die Binärdatei `njlisp.bin`.

2) `make run` → lässt `njlisp` laufen. Probieren Sie als Eingaben:

a. `(PLUS 3 4)`

b. `(TIMES (PLUS 3 4) (DIFFERENCE 10 7))`

c. `(PUTD (QUOTE SQUARE) (QUOTE (LAMBDA (X) (TIMES X X))))`

d. `(SQUARE 12345)`

e. `(SQUARE (SQUARE (SQUARE 12345)))`

f. `(OBLIST)`

3) `make musimp` → lässt `njlisp` laufen und lädt das LISP-Programm `musimp.lsp`, das die mathematische Notation von Ausdrücken erlaubt und eine eigene Programmiersprache (mit gefälligerer Syntax als LISP) zur Verfügung stellt. Es werden dann interaktiv Eingaben erwartet, die berechnet und ausgegeben werden. Probieren Sie als Eingaben:

a. `123456789*987654321;`

b. `1-2*3+4*5;`

c. `FUNCTION F(N), WHEN N=0 OR N=1, N EXIT, F(N-1)+F(N-2) ENDFUN;`

d. `F(10);`

e. `FUNCTION G(N), A:0, B:1, LOOP WHEN N=0, A EXIT, H:A+B, A:B, B:H, N:N-1 ENDLOOP ENDFUN;`

f. `G(100);`

4) `make mumath`: lädt nach `musimp.lsp` auch noch die Mathematik-Module `arith.mus` (rationale Arithmetik) und `algebra.ari` (elementare Algebra). Probieren Sie als Eingaben:

a. `5/9 + 7/12;`

b. `((236 - 3*127) * -13) ^ 16;`

c. `GCD(861, 1827);`

d. `(-24) ^ (1/3);`

e. `(-4) ^ (1/2);`

f. `#E ^ (3 * #I * #PI / 2);`

g. `5*X^2/X - 3*X^1;`

h. `(5*X)^3 / X;`

i. `EXPD((3*Y^2 - 2*Y + 5)^3);`

j. `FCTR(6*X^2*Y - 4*X*Y^2/Z);`

Das ist nur ein kleiner Ausschnitt aus dem Funktionsumfang des damaligen Systems, das auf Mikrocomputern mit maximal 64 KiB (!) Hauptspeicher lief. Es gab ca. 15 Pakete (wie die oben benutzten `arith.mus` und `algebra.ari`), die Aufgaben aus den Bereichen Matrizen, Gleichungen, Trigonometrie, Logarithmen, Differential- und Integralrechnung sowie Summen- und Grenzwertbildung lösen konnten.