Grundlagen der Informatik III

Wintersemester 2010/2011

Wolfgang Heenes, Patrik Schmittat

TECHNISCHE UNIVERSITÄT DARMSTADT

9. Aufgabenblatt mit Lösungsvorschlag

17.01.2011

Hinweis: Der Schnelltest und die Aufgaben sollen in den Übungsgruppen bearbeitet werden. Die Hausaufgaben sind in der Kalenderwoche 4 (24.01. bis 28.01.) bei den Tutoren in **physikalischer Form** (handschriftlich oder gedruckt) abzugeben. Bei allen Abgaben ist der Name des Tutors und die Übungsgruppe deutlich anzugeben. Bei Teamabgaben wird nur eine Lösung eingereicht, die alle Namen der Teammitglieder enthält.

Schicken Sie Ihre Lösungen von Programmieraufgaben zusätzlich zur schriftlichen Abgabe per E-Mail an Ihren Tutor. Kommentieren Sie Ihren Quellcode.

Aufgabe 1: Schnelltest

Fragen	*Antworten*
1. *Wie geht ein Compiler mit einer* `#include<.>`*-Anweisung um?*	☐ *Die entsprechende Library zur* `.h`*-Datei wird zur Laufzeit vom Lader eingebunden.* ■ *Der Inhalt der* `.h`*-Datei wird durch den Präprozessor eingefügt.* ■ *Die Symbole der* `.h`*-Datei werden zur Compilezeit in die Symboltabelle eingefügt.* ☐ *Die genutzten Funktionen werden automatisch beim Linken eingefügt.* ☐ *Die* `include` *Anweisung ist lediglich eine Konvention und besitzt keine Funktionalität.*
2. *Warum müssen Assembler zwei Durchgänge durchführen?*	☐ *Nicht jeder Opcode kann beim ersten Durchlauf einwandfrei ermittelt werden.* ☐ *Variablen müssen vorher entsprechend ersetzt werden, bevor jeder Befehl assembliert werden kann.* ■ *Adressen von Sprungmarken sind vorher nicht immer bekannt.* ☐ *Der Assembler muss zuerst die Größe der entstehenden Objektdatei ermitteln.* ☐ *Relative Sprünge können erst nach dem zweiten Durchlauf korrekt gesetzt werden.*
3. *Welche Aussagen über absolute und relative Adressen stimmen?*	■ *Das Laden eines Programms mit absoluten Adressen ist am schnellsten durchführbar.* ☐ *Relative Adressen müssen beim Laden nicht verändert werden.* ■ *Ein Programm mit relativen Adressen ist nach dem Laden nicht mehr verschiebbar.* ☐ *Absolute Adressen erlauben es flexibel das Programm im Speicher zu verschieben.* *Fortsetzung nächste Seite. . .*

Fragen	*Antworten*
	☐ *Dynamisches Laden erlaubt es allgemein schnellere Programmausführung zu ermöglichen.*
4. *Welche Aussagen über das Format von ELF-Objektdateien sind korrekt?*	☐ *Das ELF-Format ist unabhängig von Little- und Big-Endian.* ■ *Ausführbarer Code und Datenfragmente werden getrennt voneinander gespeichert.* ■ *Daten die nur gelesen und nicht geschrieben werden, sollten in* `.rodata` *abgelegt werden.* ■ *Im ELF-Header ist die Startadresse von* `_start` *eingetragen.* ☐ *Ein ELF-Header kann erst generiert werden wenn alle externen Symbole aufgelöst wurden.*
5. *Welche Aussagen über das Binden sind korrekt?*	☐ *Prinzipiell werden Libraries immer während der Laufzeit gebunden / geladen.* ■ *Beim Binden mehrerer Objektdateien müssen deren Adressen angepasst werden.* ■ *Ein C Programm muss immer mit mindestens einer Library gebunden werden.* ■ *Falls während der Laufzeit externe Referenzen vorhanden sind, muss der Lader die entsprechende Library laden.* ■ *Die Abschnitte* `.rel.text` *und* `.rel.data` *sind nach dem Binden nicht mehr vorhanden.*

Aufgabe 2: Übersetzungsvorgang

Geben Sie die Reihenfolge der verschiedenen Phasen bei der Übersetzung an. Beschreiben Sie in einem bis zwei Sätze die Aufgaben jeder Phase und geben Sie die genutzten Zwischenprodukte an.

Lösungsvorschlag:

1. Präprozessor: sorgt für das korrekte Auflösen von Pragmas wie z. B. `include, define`.
2. Compiler: optimiert und übersetzt das gegebene Quellprogramm in die gewählte Zielsprache (meistens eine Assemblervariante). Zusätzlich führt er syntaktische wie semantische Prüfungen durch.
3. Assembler: übersetzt das gegebene Assemblerprogramm in die Maschinensprache und führt letzte Optimierungen für die spezielle Architektur durch.
4. Linker: bindet multiple Objektdateien zu einer ausführbaren Objektdatei und löst genutzte Libraryfunktionen auf. Weiterhin werden die Pfade von dynamischen Libraries eingebettet.

Aufgabe 3: Übersetzung verschiedener Sprachen

Geben Sie zu den folgenden Übersetzungsrichtungen an ob diese problemlos durchführbar sind. Falls dies nicht der Fall sein sollte, geben Sie die auftretenden Probleme an.

a) *IA32 Assembler → Maschinencode*

Lösungsvorschlag:

Es treten keine Probleme auf.

b) *Maschinencode → IA32 Assembler*

Lösungsvorschlag:

Die Unterscheidung von Daten und Code kann Probleme verursachen. Liegt z. B. nur ein kleiner Ausschnitt eines Speicherauszugs vor, kann ggf. nicht zwischen Befehlen und Daten unterschieden werden. Ebenso sind die Information über Labels und Namen verloren.

c) *C → IA32 Assembler*

Lösungsvorschlag:

Es treten keine Probleme auf. Anzumerken gilt, dass beliebig viele verschiedene Übersetzungen existieren.

d) *IA32 Assembler → C*

Lösungsvorschlag:

Diese Richtung ist prinzipiell nicht möglich. Allein die resultierende Leserlichkeit des C-Quellcodes ist nicht gegeben und auch ausgerollte Schleifen können nicht richtig detektiert werden. Weiterhin existiert wie auch schon von Maschinencode zu IA32 Assembler das Problem der Labels und Namen. Es existieren Programme, die diese Übersetzungsrichtung durch die Anwendung von Heuristiken ausführen.

Aufgabe 4: Kontextfreie Grammatiken I

Welche der gegebenen Folgen von Terminale können mit der entsprechenden Grammatik hergeleitet werden?

a)

Startsymbol: X	~~*423.55.324*~~
$X \rightarrow YXY$	~~*82299228*~~
$X \rightarrow .$	~~*4521.254*~~
$Y \rightarrow 1\|2\|3\|4\|5\|6\|7\|8\|9\|0$	*5833.1294*

b)

Startsymbol: S	~~*acbbba*~~
$S \rightarrow aXY \mid Z$	~~*caabcba*~~
$X \rightarrow aX \mid c$	*acbcba*
$Y \rightarrow bYb \mid X$	*aaacaaaaac*
$Z \rightarrow Za \mid XY$	~~*aaabcb*~~

Aufgabe 5: Kontextfreie Grammatiken II

Für die folgenden Aufgaben sei Σ die Menge der verfügbaren Terminale, geben Sie für jede Aufgabenstellung eine entsprechende kontextfreie Grammatik an.

a) *Sei $\Sigma := \{+, -, /, *, (,), 0, ..., 9\}$. Geben Sie eine Grammatik für die Menge aller arithmetischen Terme mit Ganzahlen an.*

Lösungsvorschlag:

Startsymbol: A
A → AOA | (A) | N
O → + | - | / | *
N → 0 | 1 | ... | 9 | NN

b) *Sei $\Sigma := \{+, -, /, *, 0, ..., 9, \neg\}$. Geben Sie eine Grammatik für die Menge aller arithmetischen Terme in Postfixnotation mit Ganzahlen an.*

Hinweis: ¬ soll hierbei als Trennzeichen zwischen den Zahlen dienen und wäre mit der Return-Taste auf einem Taschenrechner gleichbedeutend. 1¬43+

Lösungsvorschlag:

Startsymbol: A
A → A¬AO | N
O → + | - | / | *
N → 0 | 1 | ... | 9 | NN

c) *Sei $\Sigma := \{a, b, c, \$\}$. Geben Sie eine Grammatik für die Menge aller möglichen Palindrome an, die in der Mitte ein $ enthalten.*

Lösungsvorschlag:

Startsymbol: S
S → aSa | bSb | cSc | S | $

d) *Sei $\Sigma := \{I, V, X, L, C\}$. Geben Sie eine Grammatik für die Menge aller römischen Zahlen bis 100 'C' an.*

Hinweis: Sie können dabei zwischen folgenden Darstellungen frei wählen. 99 → IC oder XCIX

Lösungsvorschlag:

Die folgende Grammatik steht für die Darstellung: 99 → IC. Der Übersichtlichkeit halber werden hier unüblicherweise Terminale groß und Nicht-Terminale klein geschrieben. Startsymbol: s

s → e | z | ze | C
z → p | XL | L | Lp | XC
e → o | IV | V | Vo | IX
o → I | II | III
p → X | XX | XXX

e) *Sei $\Sigma := \{0, ..., 9, .\}$. Geben Sie eine Grammatik für die Menge aller gültigen IP4 Adressen an.*

Lösungsvorschlag:

Startsymbol: S
S → N.N.N.N
N → C | ZC | 1CC | 2AC | 25B
A → 0 | 1 | 2 | 3 | 4
B → 0 | 1 | 2 | 3 | 4 | 5
C → 0 | Z
Z → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Hausaufgabe 1: Compilerausfall (5 Punkte)

Gegeben ist folgendes ANSI-C Codefragment. Leider ist ihr Compiler nicht in der Lage diese Zeilen zu übersetzen. Bilden Sie die folgenden Schritte eines Compilers nach und ermitteln Sie die Zwischenergebnisse des Übersetzungsvorgangs sowie die Symboltabelle selbstständig.

- *Lexikalische Analyse*
- *Syntaktische Analyse*
- *Semantische Analyse*
- *Zwischencodeerzeugung*
- *Codeoptimierer*
- *IA32 Codegenerator*

Informationen über die Variablen werden in die Symboltabelle übernommen:

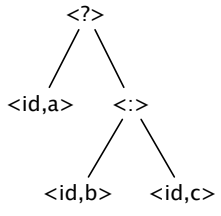
```
char m,n,b;
int i;
float f;
```

Codefragment:

```
f = 2 * f + i / b ? m : n;
```

Hinweis: Nutzen Sie für die Anweisung `a ? b : c` *folgenden verschiedenen Darstellungsformen.*

Syntaxbaum



Zwischencodeerzeugung

```
  if a goto L1
  t1 = c
  goto L2
L1:
  t1 = b
L2:
```

Lösungsvorschlag:

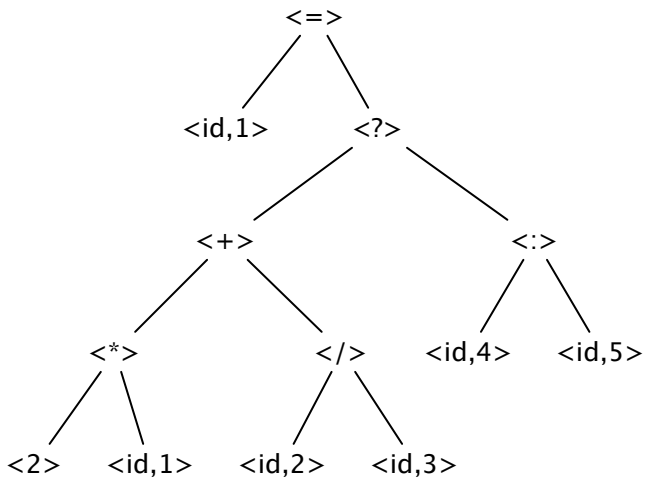
- Lexikalische Analyse

  <id,1> <=> <2> <*> <id,1> <+> <id,2> </> <id,3> <?> <id,4> <:> <id,5> <;>

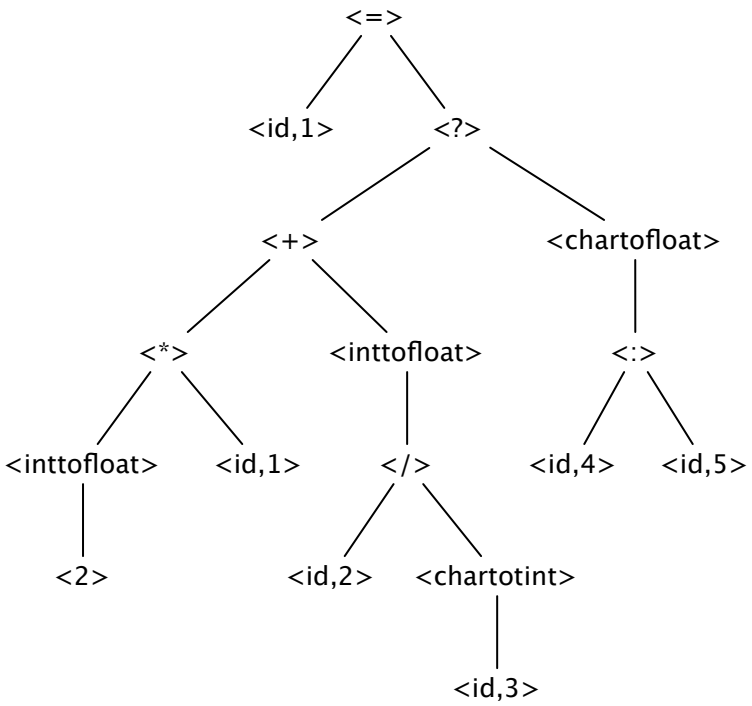
  Symboltabelle:

ID	Name	Typ
1	f	float
2	i	int
3	b	char
4	m	char
5	n	char

- Syntaktische Analyse



- Semantische Analyse



- Zwischencodeerzeugung

```
t1 = inttofloat(2)
t2 = t1 * id1
t3 = chartoint(id3)
t4 = id2 / t3
```

```
    t5 = inttofloat(t4)
    t6 = t2 + t5
    if t6 goto L1
    t7 = id5
    goto L2
  L1:
    t7 = id4
  L2:
    id1 = t7
```

- Codeoptimierer

```
    t1 = 2.0 * id1
    t2 = chartoint(id3)
    t3 = id2 / t2
    t4 = inttofloat(t3)
    t5 = t1 + t4
    if t5 goto L1
    id1 = id5
    goto L2
  L1:
    id1 = id4
  L2:
```

- IA32 Codegenerator

```
    # t1 = 2.0 * f
    fld id1
    fld1
    fld1
    faddp
    fmulp
    # t2 = chartoint(id3)
    movzbl id3, %ebx
    # t3 = id2 / t1
    movl id2, %eax
    cdq
    idiv %ebx
    # t5 = t2 + t4
    pushl %eax
    fiaddl (%esp)
    # if t5
    fldz
    fcomip %st(1)
    jz L1
    # id1 = id5
    movzbl id5, %eax
    jmp L2
L1:
    # id1 = id4
    movzbl id4, %eax
L2:
    movl %eax, (%esp)
    fildl (%esp)
    popl %eax
    fstp id1
```

Hausaufgabe 2: Analyse von Sprungtabellen (5 Punkte)

Die Implementierung von Sprungtabellen ist auf unterschiedliche Weise möglich. Eine Implementierung ist im Folgenden dargestellt.

```
.data
intout:     .string "Ergebnis␣%d␣\n"
 i:         .long 3 # Auswahl
stab:       .long .L1,.L2,.L3 # Sprungmarken
 c:         .long 0
.text
.globl main
main:
     movl i,%ebx # %ebx mit i laden
     subl $1,%ebx # Anfang ohne Offset
     jmp *stab(,%ebx,4)

.L1: movl $7,c
     jmp .ende
.L2: movl $3,c
     jmp .ende
.L3: movl $8,c
     jmp .ende

.ende:   movl c,%eax
         pushl %eax # Wert im %eax ausgeben
         pushl $intout
         call printf

# Exit
movl $1, %eax
int $0x80
```

a) *Welchen Nachteil hat diese Implementierung bzgl. einer korrekten und sicheren Ausführung? Geben Sie an, welche Ausgaben bei welchen Eingaben zu erwarten sind.*

Lösungsvorschlag:

Durch das Ablegen der Sprungtabelleneinträge im Datenteil können die Sprungtabellen aus Versehen und/oder absichtlich manipuliert werden. Für die Eingabe von 1, 2 oder 3 ergibt sich:
1 ⇒ 7
2 ⇒ 3
3 ⇒ 8

b) *Die Variable c für das Ergebnis ist in dieser Implementierung mit 0 initialisiert. Dies ist allerdings nicht notwendig, da die Variable ja geschrieben wird, bevor sie gelesen wird. Welche Möglichkeiten gibt es uninitialisierte Variablen in Assembler anzugeben? Wird in dem verschiebbaren Objekt-File für eine uninitialisierte Variable Speicher reserviert?*

Lösungsvorschlag:

Für uninitialisierte Variablen gibt es die Vereinbarung **.bss**. In dem verschiebbaren Objekt-File wird kein Speicher reserviert. Die folgende Implementierung zeigt die Vereinbarung mit **.bss**

```
.bss
 c:         .long
.data
intout:     .string "Ergebnis␣%d␣\n"
```

```
 i:         .long 3 # Auswahl
stab:       .long .L1,.L2,.L3 # Sprungmarken
.text
.globl main
main:
     movl i,%ebx # %ebx mit i laden
     subl $1,%ebx # Anfang ohne Offset
     jmp *stab(,%ebx,4)

.L1: movl $7,c
     jmp .ende
.L2: movl $3,c
     jmp .ende
.L3: movl $8,c
     jmp .ende

.ende:   movl c,%eax
         pushl %eax # Wert im %eax ausgeben
         pushl $intout
         call printf

# Exit
movl $1, %eax
int $0x80
```

Die folgende Implementierung setzt auch eine Sprungtabelle um.

```
.data
intout:     .string "Ergebnis␣%d␣\n"
 i:         .long 3 # Auswahl
stab:       .long .L1,.L2,.L3 # Sprungmarken
 c:         .long 0
.text
.globl main
main:
     leal i,%edi
     movl (%edi),%ebx
     subl $1,%ebx
     movl %ebx, 8(%edi)
     jmp *stab(,%ebx,4)

.L1: movl $7,c
     jmp .ende
.L2: movl $3,c
     jmp .ende
.L3: movl $8,c
     jmp .ende

.ende:   movl c,%eax
         pushl %eax # Wert im %eax ausgeben
         pushl $intout
         call printf

# Exit
movl $1, %eax
int $0x80
```

c) *Geben Sie an, welche Ausgaben bei welchen Eingaben zu erwarten sind. Woher können ggf. auftretende Probleme kommen?*

Lösungsvorschlag:

Für die Eingabe von 1, 2 oder 3 ergibt sich:
1 ⇒ 7
2 ⇒ Segmentation Fault
3 ⇒ 8

Das Überschreiben der Sprungtabelle sorgt für einen Segmentation Fault.

Betrachtet wird nochmal das erste Programm.

```
.data
intout:     .string "Ergebnis␣%d␣\n"
 i:         .long 3 # Auswahl
stab:       .long .L1,.L2,.L3 # Sprungmarken
 c:         .long 0
.text
.globl main
main:
     movl i,%ebx # %ebx mit i laden
     subl $1,%ebx # Anfang ohne Offset
     jmp *stab(,%ebx,4)

.L1: movl $7,c
     jmp .ende
.L2: movl $3,c
     jmp .ende
.L3: movl $8,c
     jmp .ende

.ende:   movl c,%eax
         pushl %eax # Wert im %eax ausgeben
         pushl $intout
         call printf

# Exit
movl $1, %eax
int $0x80
```

d) *Ändern Sie dieses Programm so, dass die Sprungtabelle nicht mehr im* **.data** *Segement vereinbart ist. Variablen deren Wert zu Übersetzungszeit 0 sind, sollen als uninitialisierte Variablen vereinbart werden.*

Lösungsvorschlag:

```
.bss
 c:         .long
.data
intout:     .string "Ergebnis␣%d␣\n"
 i:         .long 3 # Auswahl
.section .rodata
 stab:      .long .L1,.L2,.L3 # Sprungmarken
.text
.globl main
main:
     movl i,%ebx
     subl $1,%ebx
     jmp *stab(,%ebx,4)
```

```
.L1: movl $7,c
     jmp .ende
.L2: movl $3,c
     jmp .ende
.L3: movl $8,c
     jmp .ende

.ende:   movl c,%eax
         pushl %eax # Wert im %eax ausgeben
         pushl $intout
         call printf

# Exit
movl $1, %eax
int $0x80
```

e) *Übersetzen Sie das Programm und analysieren Sie das verschiebbare Objekt-File. Wie sehen die Segmente* **.txt**, **.data** *und* **.rodata** *aus. Erläutern Sie die Einträge des* **.rodata** *Segments mit den Bezügen zum* **.txt** *Segment.*

Lösungsvorschlag:

```
case_good.o:     file format elf32-i386

Contents of section .text:
 0000 8b1d0e00 000083eb 01ff249d 00000000  ..........$.....
 0010 c7050000 00000700 0000eb18 c7050000  ................
 0020 00000300 0000eb0c c7050000 00000800  ................
 0030 0000eb00 a1000000 00506800 000000e8  .........Ph.....
 0040 fcffffff b8010000 00cd80             ...........
Contents of section .data:
 0000 45726765 626e6973 20256420 0a000300  Ergebnis %d ....
 0010 0000                                 ..
Contents of section .rodata:
 0000 10000000 1c000000 28000000           ........(...
```

Man sieht im **.rodata** Segment die Adressen der auszuführenden Befehle. An der Adresse 10, 1c und 28 stehen die Opcodes des movl (vgl. Assemblerprogramm).