Stichworte zu Grundgebiete der Informatik 4

1. endliche Automaten
   5 – Tupel M = (Z, A, δ, z_0, E)
   Z = endliche Menge von Zustände
   A = endliche Menge von erlauben Bandsymbolen = Arbeitsalphabet
   δ = ist die (partielle) Übergangsfunktion mit δ: Z x A → Z
   z_0 Element Z= Anfangszustand
   E Teilmenge Z = Endzustände

   Die Turing Maschine ist mächtiger als ein endlicher Automat, da sie auf ihr Band auch schreiben kann und nicht nur lesend darauf zugreift.

2. Assembler Programmierung
   - Imul und Idiv erlauben keine Immidates
   - der ESP zeigt immer auf das letzte Element des Stacks
   - LEA = Load effecitv Adress (Ziel-),(Quelloperand)
   - BTC = Bit Test and Complement
   - (Bei Zugriff auf den Stack wird SS benutzt)
   - (Bei Zugriff auf das Datensegment wird DS benutzt)
   - JB jump below
   - JS jump sign (negativ)
   - RCL = Rotate through Carry Left

3. Assembler und Assemblierung

Die Übersetzung von asm Dateien in Maschinencode erfolgt in 2 Schritten

1 Phase (First Pass):

In der ersten Phase durchläuft der Assembler einmal vollständig das Programm und wertet dabei zeilenweise die Pseudobefehle (wie z.B. EQU, DD, .486 . . .) aus. Er stellt bei jeder Maschineninstruktion die Länge fest und simuliert mit dem Location Counter (LC) den Instruction Counter, so dass nach der ersten Phase der Assemblierung die Länge des Programms und die genaue Position eines jeden Maschinenbefehles bekannt ist.
In Aufgaben ist hier meist der LC zu jedem Befehl zu berechnen und die Symboltabelle anzugeben. In der Symboltabelle werden alle Symbole, die im Quelltext vorkommen, angegeben, d.h. alle Funktionen, Sprungmarken (Label), Konstanten (Equations) und alle Datenstrukturen (z.B. i DD 4). Neben dem Symbolnamen stehen in der Tabelle der (Sprung-) Typ (near oder far) der Wert des Symbols, d.h. z.B. bei Labels die Adresse oder bei Konstanten deren Wert. Darüber hinaus wird noch vermerkt, ob die Symbole public sind und welches Attribut sie haben, d.h. in welchen Codesegment sie stehen (.text steht für .code) (Pseudobefehle wie EQU etc haben KEIN Atribut)

Symbol (Name)	***Typ***	***Wert***	***Public***	***Atribut***
printf	Far	Undef (00000000)		
main	Near	00000008h	X	.text

Symbol (Name)	*Typ*	*Wert*	*Public*	*Atribut*
Pi	EQU	3.14....		
String	Byte	0000 (Startadresse)		.data

Um den Location Counter zu berechnen, muss man zuerst für jeden Befehl anhand der gegebenen Vereinfachungen seine Länge berechnen. Hierbei ist vor allem auf Displacements und Immediates zu achten (Wichtig: alle Symbole sind Immediates). Da wir nun für jeden Befehl die Länge kennen, können wir nun den Wert des LC für jeden Maschinenbefehl einfach durch Aufaddieren aller Befehlslängen der Befehle die über ihm stehen berechnen.

2 Phase (Second Pass)

Im zweiten Durchlauf wird der Maschinencode erzeugt d.h. die Mnemonics werden mit Hilfe der Machine Operation Table(MOT) in Opcodes umgesetzt (dies ist meisten ebenfalls wieder von Hand durchzuführen, indem mit Hilfe einer gegeben Opcode Tabelle und einer Tabelle für die Adressierungsarten die Mnemonics ersetzt werden).
Ziel ist es eine Objekttdatei zu erzeugen, die dann vom Binder bzw. bindenden Lader verarbeitet werden kann. Sie besteht aus

Headerdatensätzen :
H|Modulname(6)|Startadresse des Maschinenprogramms (8)|Modullänge in bye (8)
Textdatensätzen:
T|Startadresse des Feldes(8)|Länge des Felde(2)|Maschinencode(max 68)
Enddatensätzen:
E|Adresse der ersten ausführbaren Instruktion des Moduls(8)

Defintionsdatensätze:
D|Name des exportierten Symbols(6)|Adresse relativ zum Modul(OFFSET)(8)| weiter Symbole mit ihrem Offset(max.54)

Referenzdatensätze:
R|Name des referenzierten externen Symbols (6)|Weitere Symbolnamen (max. 62)

Modifikationsdatensätze:
M|Adresse des zu modifizierenden Adressfeldes relativ zum Modul(8)|Breite des zu modifizierenden Adressfelde (4)|+ oder -|Name des Symbols dessen Adresse zur Modifikation verwendet werden soll (6)

alle Angaben in hex; (Breite)
die Speicher Adressierung erfolgt byteweise

Beispiel: siehe Skript S. 81

5.Binder und Lader

1.Phase (Allocation)

- Einlesen der Namen externer Symbole unter Verwendung von R- und D-Datensätzen
- Festlegen der Startadressen der einzelnen Module (Allocation)
- Eintragen der endgültigen Adressen in die Symboltabelle. Diese ergeben sich durch Startadresse des Moduls + Offset (steht im D-Datensatz)

2.Phase (Loading, Relocation, Linking)

- Laden der Module an die in Phase 1 festgelegten Adressen (Loading)
- Anpassen von Referenzen mittels der Symboltabelle aus Phase 1 und der M Datensätze der Objektdateien (Relocation und Linking)

PROGADDR: Startadresse des gesamten Programms, wie das BS sie festlegt
(MADDR: Startadresse des aktuelle Moduls)
(MLTH: Länge des aktuelle Moduls)
EXECADDR: Adresse der ersten auszuführenden Anweisung
ESTAB: Adressen von externen Symbolen. Entspricht der ST des Assemblers

Die ESTAB sie so aus:

Symbol	***Adresse***
Main	400000

Bindender Lader

Dynamisches Binden
Durch positionsunabhängigen Code (alle Sprünge relativ, externe Symbole werden durch die GOT aufgelöst) wird es möglich erst beim Laden in den Speicher das Programm zu binden. Die Global Offset Table (GOT) ist quasi die Symboltabelle des dynamischen Binders. WICHTIG: jedes Modul hat seine eigene GOT, die aber alle auf der selben logischen Adresse liegen. Deshalb ist es nur mit Hilfe der virtuellen Speicherverwaltung möglich, die richtige GOT zuzuordnen.

GOT vor dem Laden (steht im Header)

Position	***Name***	***Typ***
0	Create Window	Funktion

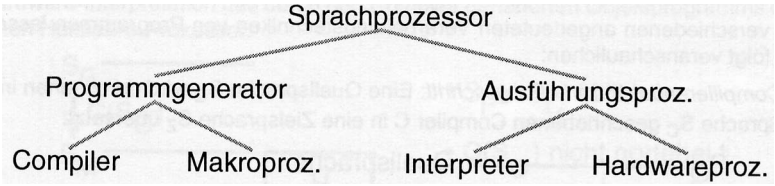
Die GOT sieht nach dem Laden so aus.

Position oder Name	***Adresse***
	modulrelative Adresse + Startadresse des Moduls (user32_base + 0000BC2F)

Procedure Lookup Table (PTL)
Für jede externe Funktion (f) wird eine lokale Dummy-Funktion (f_dummy) die folgendes macht:
f_PLT:
JMP [EBX+ f_GOT*4]
(hier sieht man, wie ein Funktionsaufruf mit GOT funktioniert. Wichtig: In EBX muss die Adresse der GOT liegen.)

f_dummy:
PUSH f_PLT_ID (Dies ist eine Nummer, welche die Funktion f eineindeutig identifiziert (für den lazy-Binder))
JMP lazy_binder
(Hier wird nun der lazy-Binder aufgerufen, der feststellt, wo die Funktion f liegt diese Adresse in die GOT einträgt und dann die Funktion aufruft, deshalb wird auch mit jmp aufgerufen, da er nicht zu f_dummy zurückspringen soll, sonder f ausführen soll. Es wird also die alte Rücksprungadresse des CALL-Befehls weiterhin benutzt.)

6. Sprachen Kompiler und Interpreter



Programmgenerator
Ein Programmgenerator leistet die mehrstufige Transformation eines Quellprogramms in ein direkt ausführbares Objektprogramm.

- Analyse des Quellprogramms
- Interpretation Anhand von semantischen Aktionen
- Generierung des Objektprogramms aus der in Schritt 2 erzeugten Repräsentation

compilierender Verarbeitungsschritt
Eine Quellsprache S_q wird durch einen in der Sprache S_c geschriebene Compiler in eine Zielsprache S_z übersetzt. (z.B. Assembler in Maschinencode)

interpretativer Verarbeitungsschritt
Eine Quellsprache S_l wird durch einen in der Sprache S_l geschriebenen Interpreter interpretiert

Hardware Prozessor
Eine Maschinensprache S_m wird durch einen Hardware-Prozessor H auf der Maschine M interpretiert (Ausführung von x86 Binaries auf einem X86 ?)

Zweistufige Zweiprogrammverarbeitungen

- Übersetzung
- Hardware-Interpretation

Dreistufige Programmverarbeitung

- Übersetzung (eines in S_p geschriebenen Programms mit dem Compiler C in die Sprache S_a)
- Übersetzung (eines in S_a geschriebenen Programms mit dem Compiler A in die Maschinensprache M)
- Hardware-Interpretation (des Maschinenprogramms S_m durch einen Hardware Prozessor)

Beispiel: Die Übersetzung eines C/C++ Programms (vereinfacht)

- c/c++ → Assembler
- Assembler → Maschinencode (binär)
- Ausführung auf der CPU

Übersetzung und Interpretation

- Übersetzung (Sp → Zwischensprache Ss)
- Interpretation (möglichst Hardwarenahe)

Beispiel:
Java oder C#

Übersetzung und Laufzeit
Die Übersetzung und anschließende Interpretation auf einer Hardware mit softwaremäßigen Laufzeitsystem ist eine Mischung aus Maschinencode und einem Interpreter (Laufzeitsystem). Software-Interpreter wie die Java VM sind im Vergleich zu binär vorliegen Programm (compiliert) um ein vielfaches langsamer, da sie viel weniger optimieren können und der Prozessor für die Ausführung jedes Befehles des Programms einige Befehle des Interpreters ausführen muss, so dass viel Rechenzeit verloren geht. Um nun die Vorteile beider Methoden, die Portabilität des Interpreter und die Schnelligkeit von compilierten Programmen zu nutzen, werden nun die Teile des Programms, die nicht auf Funktionen und Eigenheiten des OS angewiesen sind, direkt compiliert und können direkt ausgeführt werden, der Rest wird nur in eine Zwischensprache übersetzt und vom Interpreter ausgeführt.

Emulator (von S_m auf H_2)
ist die softwaremäßige Simulation eines Hardware-Prozessor (für die Sprache S_m) auf einer nicht notwendiger weise anderen Hardware H_2)
(= Interpreter für S_m auf H_2)

Translator
ist ein Compiler, der den Quelltext einer Hochsprache in den einer anderen übersetzt

Bootstrap
Technik zu Portierung von Compilern auf andere Hardware mit minimalen Aufwand
Compiler = 3 Tupel (Q, Z, I) (= übersetzt Sprache Q nach Z und ist in I implementiert/ausgeführt)

	von	mit	nach
1.	C1= (Q, Z1, Q)	→ (manuell)	C' = (Q, Z2, Q)
2.	C' = (Q, Z2, Q)	→ (mit C* = (Q1, Z1, Z1)	C" (Q, Z2, 1)Z
3.	C' = (Q, Z2, Q)	→ (mit C" = (Q1, Z2, Z1)	C2 = (Q, Z2, Z1)

Makroprozessor

Makrodefinition: &DEFINE &name(&Par1, &Par2) &AS
Makrorumpf: #Makrocode (wichtig vor Parameter immer &)#

Ein Token ist die lexikalische Grundeinheit, die ein Parser bearbeitet. (=Eingabesymbole des Parsers)

Die **Expansion** von Makros erfolgt wieder in 3 Schritten

1. **Lexikalische Analyse**
   In diesen ersten Schritt wird das Programm von einem Scanner in Token zerlegt und dabei schon zwischen Namen für Makros, Schlüsselworten (z.B. &DEFINE), Parametern und Terminalsymbolen (z.B: #) unterschieden. Damit der Makroprozessor weiß, was er bei der Programmgenerierung machen muss, wird die Programmtabelle (PT) erstellt. Sie enthält folgende Eintragstypen:
   K: für das Kopieren der Zeichenkette
   M: Expandieren eines Makros
   P: Definition eines Makroparameters
   PT

Tätigkeit	***Position***	***Länge***	***MD***	***Parameter***
K/P/M			Verweis auf einen Eintrag in der MD	Anzahl der Parameter
K	0	J		
M			0	2
P	J+30	4		

   Der Makrorumpf wird NICHT kopiert oder sonst etwas mit ihm gemacht, wichtig sind nur die Makroaufrufe.

2. **Syntaktische Analyse**
   Die syntaktische Analyse fasst die Token zu lexikalischen Einheiten zusammen und wertet diese aus. So werden z.B. Makrodefinition und Aufrufe erkannt. Diese Informationen werden in der interen Repräsentation in der Makrodefinitionstabelle (MD) und der Makrorumpftabelle (MR) gespeichert
   MD

Position	***Name***	***Verweis auf MR***	***Anzahl Token***
0	&testmakro	0	3

   Anzahl der Token = wie viele Zeilen in der MR zu dem Makro gehören

   MR

Position	***Tätigkeit***	***Anfangsposition***	***Länge***	***Parameter***
0	E			1
1	K	I+42	21	
2	E			2

   Die Positionsspalten können bei der MD und MR auch weggelassen werden. Wichtig

3. **Programmgenerierung**
   Es werden vordefinierte Makros aus den Makrobibliotheken kopiert und vor alle die PT abgearbeitet und danach das Programm mit den fertig expandierten Makros ausgegeben.

Compiler

Ein Compiler übersetzt die komplette Eingabe in eine Repräsentation, die die direkte Ausführung des Codes auf einen Prozessor erlaubt.

1. **Lexikalische Analyse**

Auch hier findet wieder zuerst eine lexikalische Analyse statt, die folgende Strukturen erzeugt

Uniform Symbol Tabelle (UST)

Token	***Typ***	***Index***
Main	Key	1
{	TRM	1
Var	IDN	1
Pi	LIT	1
Undeklarierte var	Any	1

Identifier and Names (IDN)

Index	***Name***
1	Var

Literals (LIT)
Konstanten und ihre Werte

Index	***Wert***
1	3.14...

Unknown Tokens (ANY)

Index	***Name***
1	Undeklarierte var

Terminals (TRM)

Index	***Terminalsymbol***
1	{

Key (KEY)
alle Schlüsselworte

Index	***Schlüsselwort***
1	main

2. **Syntaxanalyse**

Der Parser kann nun die syntaktische Struktur der Eingabe überprüfen um übergeordnete syntaktische Einheiten oder Fehler zu finden, um so die Übersetzung vorzubereiten.
Er benötigt dazu neben den obigen Datenstrukturen auch noch die grammatikalischen Regeln der Sprache, die z.B. in BNF oder Syntax-Diagrammen visualisiert werden können.

3. **Interpretationsphase**

Die Eingabe wird nun vollständig in eine Zwischenrepräsentation konvertiert (z.B. Binärbaum oder Matrixdarstellung)

4. **Maschinenunabhängige Optimierung**
   - Eliminierung gemeinsamer Unterausdrücke
   - Berechnung von Konstanten
   - Verkürzung von Ausdrücken durch bekannte Rechenregeln
   - Auslagerung (Vorziehen) von Schleifen-Invarianten, also Ausdrücken, die sich in der Schleife nicht ändern würden.

5. **Speicherplatzzuteilung**
   Bei der Speicherplatzzuteilung wird für die IDN, die LIT und die Zwischenergebnisse der benötigte Speicherplatz reserviert. Man kann also am Besten die Speicherplatzzuteilung berechnen, wenn man die IDN und die LIT kennt und die Matrixdarstellung vorliegen hat, da man aus ihr einfach die Anzahl und die Namen für die Zwischenergebnisse ablesen kann.
6. **Codegenerierung**
   = Umsetzung der Zwischenrepräsentation in Maschinencode
   Dies geschieht nach der Berechnung der Belegung der Register des Prozessors durch zeilenweise Umsetzung der Matrixdarstellung in Maschinencode bzw. Assembler.
   Dieser Code ist leider sehr ineffizient, da er sämtlich Besonderheiten des Prozessors noch nicht ausnutzt.
7. **Maschinenabhängige Optimierung**
   Durch
   - die Benutzung effizienter Befehle (z.B. Inc und Dec)
   - Vermeidung unnötiger Speicherbefehle
   - Ausnutzung der Registerstruktur für die Optimierung

   wird die Ausführungsgeschwindigkeit erhöht und der Speicherplatz (meist auch die Codelänge) verringert.

Interpreter

Ein Interpreter übersetzt jedes Sprachkonstrukt (z.B. Zuweisungen, Funktionsaufruf) einzeln und auch wiederholt bei erneuten Aufruf.

Befehlsinterpreter

Befehlsinterpreter sind Interpreter für ausführbaren Maschinencode. Sie bilden prinzipiell Code für einen hypothetischen (oder auch virtuellen) Prozessor auf einen realen ab. Näheres siehe Seite 136-138 im Skript

Kommandointerpreter sind Softwareinterpreter für Steuersprachen (UNIX Shell)

Sprachinterpreter
Sprachinterpreter sind Interpreter für höhere Programmiersprachen. Wir betrachten die Interpretation auf der Ebene der Uniform Symbol Table (UST)

UST

Token	***Tabelle***	***Nr.***
	IDN/ TRM/ LIT	Zeilennummer

TRM

Symbol	***Nr.***
z.B. +,/, ...	Zeilennummer

IDN

Name	***Nr.***	***Wert***
x	1	42

LIT

Pseudoname	***Nr.***	***Wert***
&L1	1	42

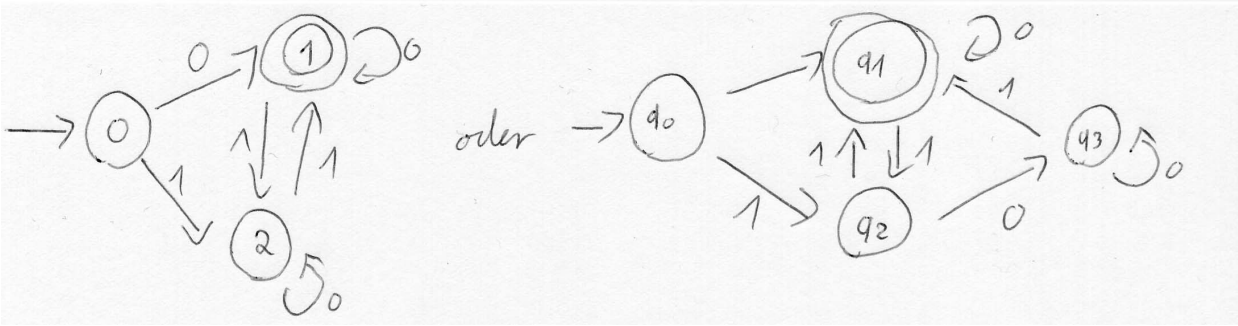
Die Spalten Wert bei der IDN bzw. LIT sind quasi die Veranschaulichung der dynamischen Speicherplatzzuweisung. Sie ermöglichen es Werte zuzuweisen und Zwischenergebnise zu speichern.

Die Interpretation erfolgt mittels zweier Stacks: dem Evaluationsstack für die Operanden und dem Operationsstack für die Operationen. Um Rechenregeln etc. berücksichtigen zu können, muss der Interpreter wissen, ob er z.B. zuerst addieren und dann multiplizieren soll oder umgekehrt. Dies wird ihm mittels der Präzedenzmatrix beigebracht.

- Op1 > Op2 = Op1 hat Vorrang vor Op2 (Op = Operand)
- Op1 < Op2 = Op2 hat Vorrang vor Op2
- Op1 = Op2 = beide Operanden sind gleichrangig; hier wird meist der erste Ausdruck ausgewertet
- Op1 0 Op2 = eine nicht erlaubte Folge von Operanden

Präzedenzmatrix:
Zeile = Op1; Spalte = Op2

Antworten auf die Fragen der letzten Kleingruppenübung

1. Bei Sprachprozessoren lassen sich in 2 Unterklassen unterscheiden:
   Programmgenerator, Ausführungsprozessoren; (siehe Bild oben oder S.111 Skript)
2. Hardwareprozessor und Interpreter
3. Call Marke
   Marke:
   POP EAX
4. Software-Programm, dass einen Quellcode zur Laufzeit einliest, analysiert und ausführt.
   Bsp: Befehls-, Kommando- und Sprachinterpreter
5. Little Endian: Die niederwertigste Byte liegt an der niederwertigsten Stelle (wird vom PIV benutzt)
   Big Endian: Die niederwertigste Byte liegt an der höchstwertigen Stellen
6. portabler (positionsunabhängiger Code) und ein Compiler, der die Erstellung solcher Bibliotheken unterstützt.
7. 

8. Das keine externen Symbole gelinkt werden müssen (?)
9. lexikalische Analyse, syntaktische Analyse, Programmgenerierung
10. Programmgeneratoren
11. MOV EAX, i
    while:
    CMP EAX,0
    JE while_ende
    DEC EAX
    JMP while
    while_ende:
    MOV i, EAX
12. Complex Instruction Set Computer (viel auf teilweise sehr komplexe Befehle)
13. Reduced Instruction Set Computer (weniger Befehle, dafür schneller)
14. MOV [EAX + EBX +42] ohne Skalierung
    (MOV [EBX + ECX *4 + 42) mit Skalierung (wichtig scale = 2,4,8 sein)
15. Laden, Verschieben, Binden (s.o. Oder S. 90)
16. des Ausführungsprozessors
17.

Speicherindex	*0*	*1*	*2*	*3*
Wert	FB	FF	FF	FF

18.
    - Argumente (Parameter der Fkt) werden von rechts nach links auf den Stack gelegt

- aufrufendes Programm muss den Stack wieder aufräumen
- Ergebnis in EAX

19. Turing Maschine TM = (z, A, δ, z_0, E) mit z =Zustandsraum, A = Alphabet, δ = Übergangsfunktion, z_0 = Anfangszustand, E = Menge der Endzustände
20. ja, mitdie Übertragung in zweites Register
21. M Datensätze dienen dazu, dem Linke oder Lader mitzuteilen, an welchen Stellen des Programms Adressen abgeändert werden müssen und wie die geschehen soll.