(D1) Übersetzen, Binden, Laden und Ausführen von Programmen

Der Programmierer formuliert seine Algorithmen zumeist in einer problemorientierten Programmiersprache, wie z.B. C, C++, JAVA, Pascal, Prolog, Basic, Perl. Diese Form muss jedoch umgesetzt werden in eine semantisch äquivalente Form, die der Rechner abarbeiten kann. Der Rechner kann jedoch nur Programme in seiner Maschinensprache, eine Bytefolge, abarbeiten.

Für die Unterstützung des Programmierers und diese Umsetzung gibt es verschiedene Alternativen:

- Interpreter: Ein Programm interpretiert die Anweisungen der Programmiersprache, bei jedem Durchlauf durch die Anweisungen. (Beispiel: BASIC-Interpreter)

  Nachteil: Da die Anweisung bei jedem Durchlauf neu analisiert und interpretiert werden muss, ist die Laufzeit sehr schlecht.

  Vorteil: Es kann z.B. nach einer Änderung sofort wieder mit der Interpretation begonnen werden evtl. sogar eine laufende fortgeführt werden. Die Interpreter enthalten daher i.A. einen integrierten Editor.

- Batch (Kommandozeilen)-Compiler: Der Programmierer erstellt das Quellprogramm mit einem beliebigen Texteditor. Dies ist die Eingabe für den Compiler. Der Compiler, z.B. gcc, gibt ein Bindemodul aus und evtl. auch ein Protokoll mit Fehlermeldungen. Das Bindemodul wird mit anderen Modulen und den Standardbibliotheken zu einem ausführbaren Programm gebunden. Dieses Programm wird dann gestartet.

  Nachteil: Bis zu einem Programmlauf muss der Programmierer viele Programme auf der Kommandozeile aufrufen (Compiler, Binder, ausführbares Programm). Um dies zu automatisieren gibt es spezielle Programme (make) , die diese Schritte in der richtigen Reihenfolge ausführen.

  Vorteil: Bei der Übersetzung werden die Anweisungen analisiert und in eine Folge von semantisch äquivalenten Machinenbefehlen umgesetzt. Durch entsprechende Optimierungen während der Übersetzung kann man so sehr effiziente Programme erzeugen.

- Integrierte Entwicklungsumgebung (IDE= Integrated Development Envirenment): Hier sind Editor, Compiler, Binder und Makefunktion in einem Programm vereinigt. Die erste solche IDE, die Maschinencode erzeugte, war Turbo-Pascal (1983).

  Nachteil: Lange Zeit war der große Speicherbedarf solcher IDE ein Problem. Die besondere Leistung beim ersten Turbo-Pascal war, dass Editor und Compiler mit ca 32k Byte realisiert wurden. Bei einer Programmänderung muss natürlich wieder neu übersetzt werden. Bei einer IDE wird meist ein so genannter Entwicklungscompiler benutzt, der nicht so stark optimiert, dafür jedoch schnell übersetzt. Ist das Programm fertig entwickelt, kann es noch mal mit einem so genannten Produktionscompiler, mit allen Optimierungen übersetzt werden.

  Vorteil: Die IDE vereinigt die Vorteile von Interpreter und Compiler.

Der von einem Compiler erzeugt Code, ist jeweils nur auf der Zielmaschine des Compilers lauffähig. Bei Anwendungen, die über das Internet geladen werden, ist dies jedoch nachteilig. Um ein Programm auf mehren Architekturen laufen zulassen, wird wie z.B. in JAVA, ein Zwischencode benutzt.

```
Quelle (.java) -> Compiler (javac) -> Byte-Code (.class)
Byte-Code (.class) -> geladen (Class-Loader)
                   -> interpretiert  (Java-Virtual-Machine)
```

Beim Laden eines Programms aus dem Internet, wird der Byte-Code geladen. Auf der Maschine muss dann nur eine Java-Virtual-Machine zur Verfügung stehen.

Um den Laufzeitnachteil der Interpretation des Byte-Codes durch die Java-Virtual-Machine zu vermeiden, wird heute ein Just-In-Time (JIT)-Compiler benutzt. Bei der ersten Interpretation eines Codestückes, wird dies in Maschinensprache übersetzt. So hat man gegenüber einem normalen Compiler nur den zusätzlichen Aufwand, dass der Bytecode bei jedem Ablauf des Programms neu übersetzt wird. Diese Übersetzungszeiten können jedoch heute vernachlässigt werden.

Grundzüge eines Compilers

Der Compiler muss eine Anweisung:

```
A:=B+C;
```

umsetzen in eine Folge von Maschinenbefehlen:

```
MOV AX,[B]
ADD AX,[C]
MOV [A],AX
```

A, B, C sind im Zielcode Angaben, die der Binder dann in Adressen umsetzen kann.
In Turbo-Pascal sind dies z.B. Paare Bereichsnummer und Offset. Der Binder ordnet den Bereichen der einzelnen Module dann Speicherplatz zu, sodass diese Angaben in Adressen umgesetzt werden können.

Realisierung von Prozeduren

Prozeduren können in den meisten Programmiersprachen rekursiv aufgerufen werden. Dadurch kann man mehr als eine Instanz einer Prozedur in einer Aufrufverschachtelung haben.

Prog
P
Q
P
Q
R

Die Parameter und die lokalen Variablen einer Prozedur können daher nicht statisch gehalten werden, sondern müssen auf einem Stapel (stack) abgelegt werden.

Für diese Ablage gibt es verschiedene Varianten:

PASCAL-Schema:

```
procedure P(a:integer; b:integer);
var c:integer;
begin c:=a+b end;

P(X,4);
```

Der Aufruf liefert folgenden Code:

```
push [X]
push 4
call P
```

Für die Prozedur wird folgender Code erzeugt:

```
P:
push BP
mov BP,SP
sub SP,2
mov AX,[BP+8]
add AX,[BP+6]
mov [BP-2],AX
mov SP,BP
pop BP
ret 4
```

Die Belegung des Stacks:

A	+8
B	+6
RUA	+4
	+2
BPold	<- BP
C	-2

Kennzeichen des Pascal-Schemas:
Die Parameter werden durch das aufrufende Programm von links nach rechts auf den Stack gebracht. Die Entfernung der Parameter erfolgt durch das gerufene Programm. Aktuelle und formale Parameter müssen bezüglich Typen und Anzahl übereinstimmen. Implizite Parameter werden hinten angefügt. Dadurch ist ihre Adresse immer die gleiche, unabhängig von der Anzahl der Parameter. Der letzte Parameter hat immer die Adresse BP+6.

Übersetzung von Methodenaufrufen

```
type t=object
 a:integer;
 procedure p(b:integer);
end;

var x:t;
x.p(3);

push 3
push x.segment
```

```
push x.offset
call t.p

procedure t.p(b:integer);
begin a:=b; end;

push BP
mov BP,SP
mov AX,BP+10
les DI,BP+6
mov ES:DI, AX
mov SP,BP
pop BP
ret 6
```

3	+10
X.segment	
X.offset	+6
RUA	+2
BPold	<-BP

I.A. werden die Methoden jedoch virtuell deklariert. Der Zugriff erfolgt dann über eine virtuelle Methodentabelle (VMT).

VMT

---V

Länge
1. Methode
2. Methode
3. Methode

Da jede Klasse die Methoden ihrer Superklasse erbt, die sie evtl. überschreiben kann, bleibt die Position der Methode in der VMT in allen Subklassen gleich.

C-Methode:

Die Spreche C erlaubt Prozeduren mit variabler Parameterzahl.
Beispiele:

```
printf(" <Formatstring> ", Wert1,Wert2,Wert3);
sum(3,a,b,c);
```

Die Anzahl und Typen der folgenden Parameter ergeben sich aus den bereits bekannten Parametern. Daraus folgt:

- Die Position des ersten Parameters muss bekannt sein.
  -> Die Parameter kommen von rechts nach links auf den Stack
- Es muss je nach Aufruf eine unterschiedliche Anzahl von Parametern vom Stack entfernt werden.
  -> Die Parameter werden durch das rufende Programm entfernt.

```
Aufruf: SUM(3,X,Y,Z);

push [Z]
push [Y]
push [X]
push 3
call SUM
sub SP,8
```

```
SUM:
push BP
mov BP,SP
sub SP,4
s:=0;
for i:=1 to [BP+6] do s:=s+[BP+6+2*i]
mov AX,[s]
mov SP,BP
pop BP
ret
```

Z	
Y	
X	
3	
RET	
BPold	<-BP
i	
sum	

Bei C werden implizite Parameter z.B. Objekte bei Methoden ebenfalls zuletzt werggespeichert, d.h. sie werden links von den anderen Parametern eingefügt.

Vorgänge beim Binden:

Begonnen wird mit dem Code des Hauptprogramms.

```
a:=b  =>
(1) MOV AX,[b] => Referenzbereich von b =b'
(2) MOV [a],AX => Referenzbereich von a =a'
```

d.h. die Bereiche a' und b' werden als zu ladend notiert.. Ferner wird in (1) ein Bezug auf b'.b bzw. in (2) ein Bezug auf a'.a notiert.
Wenn alle als zu ladend notierten Bereiche geladen sind, sind die Anfabgsadressen aller Bereiche bekannt
=> alle notierten Bezüge können durch absolute bzw. programmrelative Adressen ersetzt werden.
=> .EXE Datei

Vorgänge beim Laden:

Inhalt der .EXE Datei:

```
Header:    Kennung
           Speicherbedarf (Minimum, Maximum )
           Startwert für SS,SP,IP,CS
Relocation-Table: Besteht aus Eintragungen mit folgender
Struktur:
                  offset,segment
Codetext
```

Zunächst wird geprüft, ob genügend Speicher für den minimalen Speicherbedarf vorhanden ist.
Wenn ja wird maximal der maximale Speicherbedarf reserviert.
Programmsegmentpräfix bilden.
Codetext in den Speicher übertragen
Für jeden Eintrag in der Relokationstabelle:

segment:=segment im Codetext+startsegment

D.h. ein Programmrelativer Segmentbezug wird durch einen absoluten Segmentbezug ersetzt.

SS:=$_{SSHeader+Startsegment}$
SP:=SP$_{Header}$
DS,ES:=PSP
Startadresse:=CS$_{Header+}$Startsegment,IP$_{Header}$

Nachteile dieses Verfahrens:

Ein Programm kann nach dem Laden nicht mehr im Speicher verschoben werden.

Vorgehen bei Protected Mode Programmen:

Segmentbezüge im Programm sind Programmrelative Selectoren 0,1,2,3,...,n
Nach dem Laden wird für jedes Segment ein Selector über den DPMI-Server besorgt. Jeder Segmentbezug wird dann durch diesen Selector ersetzt. Für jedes Segment wird Speicher reserviert.

Bei WIN32 oder UNIX wird ein linearer Adressraum benutzt. D.h. es gibt nur ein oder zwei (Code, Daten) Segmente.

Modulaufbau bei UNIX:

Header MagicNumber
Programmcode "text"
initialisierte Daten
Relokationstabelle
Symboltabelle

In der Relokationstabelle enthält ein Eintrag folgende Informationen:

```
r_address: long;      Adresse im Programm
r_symbolnum:24bit;    Nr. des Symbols in der Symboltabelle
r_pcrel:1bit;         schon PC-relativ
r-length:2bit;        Angabe ob byte,word,long
r_extern:1bit;        Kennzeichen ob Externbezug

r_extern=0: r_symbolnr ist die Nummer eines Segments. Adresse
ist der offset im Segment
r_extern=1: r_symbolnr ist die Nummer eines Eintrags in der
Symboltabelle. Adresse ist des Offset zu der Adresse des
Symbols.
```

Die Symboltabelle enthält Einträge, die wie folgt aufgebaut sind:

```
n_name: pstring oder n_strx:long; // Index in Stringtabelle
n_type:byte;    // Typ
n_value:long    // Wert bzw. Adresse des Symbols
```

Mögliche Typen:

```
n_abs:     absolute Adresse bzw. absoluter Wert
n_text:    Adresse ist offset im Codeteil
n_data:    Adresse ist offset im Datenbereich
n_bss:     Adresse ist offset im nicht initialiesierten
Datenbereich
n_fn:      Wert ist Index in einer Tabelle von Dateinamen
```

Beispiel:

Programmcode:

```
Var a: record b,c:integer end;
a.b:=a.c;
P; {Aufruf von Prozedur P}
```

Erzeugter Maschinencode:

```
MOV AX,[2]
       ^1
MOV [0],AX
    ^2
CALL [0]
     ^3
```

Relocationstabelle:

```
1: [1] 4Byte extern
2: [1] 4Byte extern
3: [2] 4Byte extern
```

Symboltabelle:

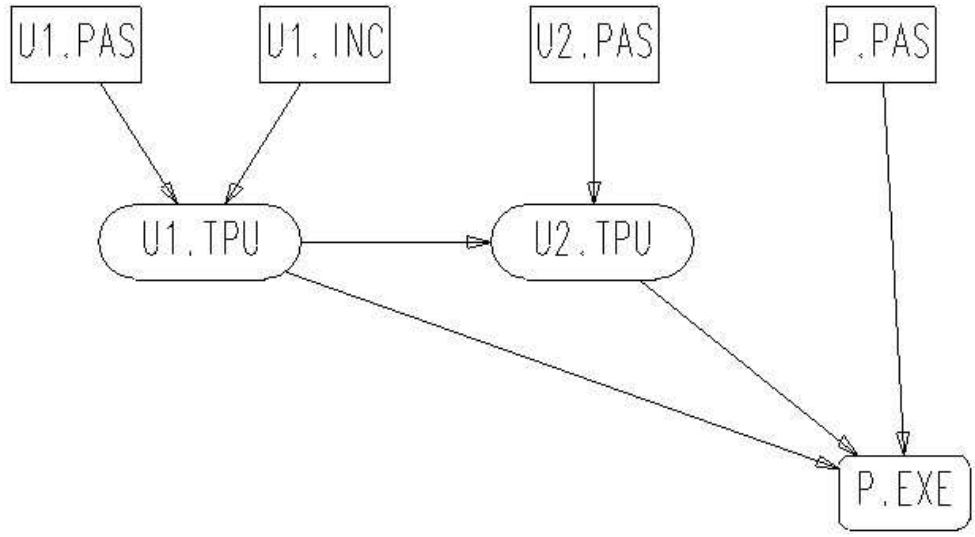
```
[1] ´a´ n.data <offset von a im Datenbereich>
[2] ´P´ n_code <offset non P im Codebereich>
```

Funktionsweise eines Debuggers

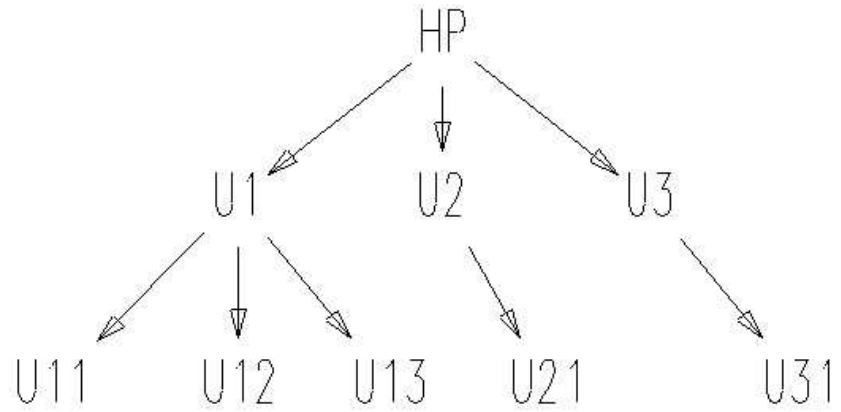
Der Binder erzeugt eine Tabelle, die für jede Programmzeile, die Adresse der ersten zugehörigen Anweisung enthält.
Wird auf eine Programmzeile ein Breakpunkt gesetzt, wird der entsprechende Befehl durch einen Interruptbefehl ersetzt (bei der INTEL- Architektur der 1-Byte Befehl INT3 =CCh).
Wird dieser befehl erreicht wird durch den Interrupt der Debugger angesprungen.
Jetzt kann der Speicher inspiziert werden. Die Adresse der Variablen kann über die Symboltabelle ermittelt werden.
Soll das Programm fortgesetzt werden, wird der INT3 wieder durch das ursprüngliche Befehlsbyte ersetzt. Dieser Befehl wird dann im Einzelschrittmodus ausgeführt. Danach kann wieder der Befehl INT3 eingesetzt werden und das Programm wird normal fortgesetzt.

Make-Funktion

Ein Programm P werde aus folgenden Dateien erzeugt:



Die Pascal-Quelldatei U1 benutzt eine Includedatei U1.INC. Die Unit U2 benutzt die Unit U1.
Bei Make p wird getestet:
U1.TPU ist abhängig von U1.PAS und U!.INC. Ist das Datum einer dieser Dateien jünger als das von U1.TPU muss U1.PAS neu übersetzt werden. In U1.TPU steht eine Prüfsumme über den Interfaceteil.
Ist U2.TPU älter als U2.PAS wird U2.Pas neu übersetzt. In U2.TPU befindet sich die Prüfsumme von U1.TPU zum Zeitpunkt der letzten Übersetzung von U2. Ist diese Prüfsumme ungleich der aktuellen Prüfsumme in U1.TPU muss U2.PAS auf jeden Fall neu übersetzt werden, da sich der Interfaceteil der benutzten Unit U1 geändert hat.




```
Make HP
 Testen U1
```

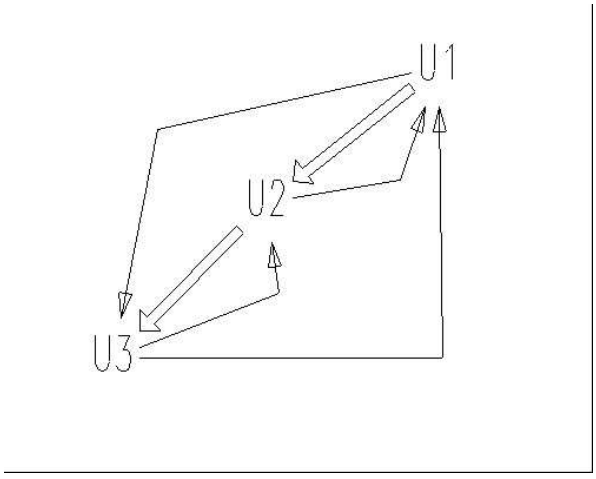
```
  Testen U11,U12,U13
 Testen U2
  Testen U21
 Testen U3
  Testen U31
```

Testen:

1. neu übersetzten, wenn Quelldatei jünger als TPU-Datei
2. wenn sich der Interfaceteil geändert hat, sind alle im Baum direkt darüber liegenden ungültig.

Zirkuläre Abhängigkeiten

```
unit u1;
interface
 uses u2;

implementation
 uses u3;

unit u2;
interface
 uses u3;

implementation
 uses u1;

unit u3;
interface

implementation
 uses u1,u2;
```




```
Übersetzten U1 Interface
 Übersetzen U2 Interface
  Übersetzen U3 Interface
  Übersetzen U3 Implematation
 Übersetzen U2 Implemantation
Übersetzen U1 Implementation
```

Zirkuläre Abhängigkeiten im Interfaceteil sind nicht möglich, da sonst keine Übersetzung möglich.

Zirkuläre Abhängigkeiten im Interface entstehen meist durch:

```
unit u1;
uses u2;
 procedure p1(x:t2);
 type t1=
...

unit u2;
uses u1;
 procedure p2(x:t1);
 type t2=
...
```

Dies kann aufgelöst werden durch eine Unit für die Typen:

```
unit udecl;
type t1=...
type t2=...
```

U1 und U2 benutzen dann jeweils Udecl.