Compiler - Interpreter - Linker

1.1 Aufgaben eines Compilers

Alle Anweisungen des Quellcodes werden zuerst interpretiert (d.h. es wird festgestellt, um welchen Quellcode es sich handelt, ist es z.B. eine Zuweisung, eine Schleife, ein mathematischer Ausdruck, usw.).
Sind die Anweisungen syntaktisch korrekt, dann werden die Anweisungen in die entsprechenden Maschinenbefehle übersetzt und als Programm (in einer Datei) gespeichert.. Erst dann wird das Programm gestartet.
Der Maschinencode, der durch die Übersetzung eines Quellcodes durch einen Compiler erzeugt wurde, nenne ich im Folgenden abgekürzt Compilercode.

1.1.1 Optimierung mit Inlining

Um die Rechenzeit eines Programms zu reduzieren, gibt es u.a. die Technik des Inlinings: Angenommen die Methode heißt "erhoehe" und macht nichts anderes als den Inhalt einer Variable (den sie als Parameter erhält) um den Wert 1 zu erhöhen, also:

```
int erhoehe (int a){
  a=a+1;
  return(a)
}
```

Dann wird beim Aufruf von z.B. erhoehe(17) folgendes gemacht:
- Alle Prozessorregister werden auf dem sogenannten Stack gesichert.
- Der aktuellen Parameter 17 (der konkret beim Aufruf verwendet wird), wird ebenfalls auf dem Stack gesichert.

Das Sichern auf dem Stack wird mit den sogenannten push-Befehlen gemacht.
Dies braucht mehr Rechenzeit, als wenn man den zu dieser Methode, also hier konkret den zu a=a+1 zugehörigen Maschinencode statt des Aufrufs erhoehe(17) verwenden würde (d.h. den Aufruf erhoehe(17) durch den zu a=a+1 zugehörigen Maschinencode ersetzen würde). Dies nennt man **Inlining**. Beim Inlining fallen also die push-Befehle (für das Sichern auf dem Stack) weg. Deswegen wird beim Inlining weniger Rechenzeit verbraucht.

Statt

```
x=17
erhoehe(x)   // enthält die push-Befehle
```

wird dann durch das sogenannte Inlining folgendes gemacht:

```
x=17
goto M1
M2:
...
...
M1:
a=x  //besser: den dazu zugehörigen Maschinencode
a=a+1; //besser: den dazu zugehörigen Maschinencode
x=a; //besser: den dazu zugehörigen Maschinencode
goto M2
...
```

1.2 Aufgaben eines Interpreters

Ein "klassischer" Interpreter führt den zu einem Interpreter zugehörigen Quellcode wie folgt aus:
1) Eine Anweisung des Quellcodes wird interpretiert (d.h. er stellt fest, um welchen Quellcode es sich handelt, ist es z.B. eine Zuweisung, eine Schleife, ein mathematischer Ausdruck, usw.).

2) Ist diese Anweisung syntaktisch korrekt, dann wird das zu dieser Anweisung sich schon im Interpreter (an einer bestimmten Stelle im Arbeitsspeicher) befindliche Unterprogramm (das natürlich aus Maschinenbefehlen besteht) aufgerufen und ausgeführt.
Beim klassischen Interpreter wird Maschinencode für keine einzige zu interpretierende Anweisung erzeugt.

3) Weiter mit 1)

1.3 Gemeinsamkeit Interpreter und Compiler

Beim Kompilieren und Ausführen eines Compilercodes bzw. beim Interpretieren eines Quellcodes gibt es zwei Anteile, die Zeit verbrauchen:
1) Interpretationsteil:
Interpretation des Quellcodes. Compiler bzw. Interpreter müssen feststellen, um welchen Quellcode es sich handelt

2) Ausführungsteil:
Ausführen des Maschinencodes der zum Quellcode gehört.

1.4 Vergleich: klassischer Interpreter und Compilercode

1.4.1 Zeitvergleich

Beim klassischen Interpreter ist die für einen Programmablauf benötigte Zeit - gegenüber dem reinen Ausführen des durch einen Compiler erzeugten Maschinencodes - aus dem Grund größer, weil der Compilercode keinen Interpretationsteil besitzt.
Nur beim Compilieren gibt es diesen Interpretationsteil. Da der Anwender aber nur den Compilercode ausgeliefert bekommt, fällt bei ihm der Zeitbedarf für den Interpretationsanteil weg. Dagegen fällt beim Programmierer beim Compilieren dieser Zeitaufwand an.

Beim Interpreter dagegen fällt bei jedem Programmlauf dieser Zeitanteil an.

1.4.2 Weitere Vergleiche

1) Ein Interpreter kostet weniger.

2) Der Aufwand für die Erstellung eines Interpreters ist wesentlich geringer (Interpreter muß z.B. nur den Teil des gesamten Quellcodes betrachten der tatsächlich ausgeführt wird).

3) Um z.B. ein Java-Programm auf einer Website runterzuladen, muss dort nur der Byecode als Datei bereitgestellt werden. Wenn man das Programm dagegen als exe-Datei bereitstellen würde, müsste man auf der Website für jeden Prozessor (+ Betriebssystem) eine andere exe-Datei bereitstellen. Satt einer Datei (Bytecode) also eine exe-Datei für den x386 Prozessor, eine exe-Datei für den xyz-Prozessor, usw.

1.5 Zeitvergleich: JIT-Interpreter und Compilercode

Man geht davon aus, dass nur ca. 5% eines Programms "oft" und ca. 95% "selten" durchlaufen werden. Diese ca. 5% eines Programms nennt man auch Hot Spots (deutsch: heiße Flecken) Der sogenannte Interpreter mit Hot Spot JIT-Compiler (Just-in-Time-Compiler) kann dies nur während der Laufzeit (**nachdem** nämlich diese Bereiche mehrfach durchlaufen wurden) feststellen.
Diese 5% Hot Spots kompiliert nun (während der Laufzeit !) der JIT-Compiler des Interpreters (fortan müssen diese also nicht mehr interpretiert werden).

1) Der Compilercode wird nicht interpretiert. Also braucht er weniger Zeit als ein Interpreter. Da der Java-Interpreter (Virtual Machine) aber nicht menschenlesbaren Bytecode interpretieren muss, ist dieser zeitliche Nachteil des Javainterpreters (gegenüber dem Compilercode) nicht so groß wie der zeitliche Nachteil des klassischen Interpreters (der menschenlesbaren Quellcode interpretieren muss) gegenüber dem Compilercode.

2) Weil der Interpreter mit Hot Spot JIT-Compiler aber die 5% Hot Spots des Programms während der Laufzeit optimieren kann, kann er mehr optimieren, als der (klassische) Compiler (der vor der Laufzeit optimiert). Denn bei der Laufzeit fallen zusätzlich noch Informationen an, die der Compiler (der vor der Laufzeit optimiert), nicht hat.

Ergebnis:
Punkt 1) ergibt einen zeitlichen Nachteil für den Interpreter, Punkt 2) einen zeitlichen Vorteil für den Interpreter so dass man die Frage nicht so einfach beantworten kann, wer zeitlich schneller ist.

1.5.1 Informationen vor und während der Laufzeit

Im folgenden werden Beispiele für Informationen angegeben, die nur während, aber auf keinen Fall vor der Laufzeit eines Programms anfallen und die der JIT-Compiler ausnutzt um den Maschinencode zu optimieren.
(Diese Optimierung kann von einem klassischen Compiler nicht gemacht werden, da er diese Informationen nur vor der Laufzeit erhält).

1.5.1.1 Beispiel (virtuelle / nicht virtuelle Methoden)

Der JIT-Compiler kann spekulativ optimieren und diese Optimierungen zurück nehmen, falls die Bedingungen für eine Optimierung nicht mehr vorhanden sind.
Das geschieht z.B. im Zusammenhang mit virtuellen Methoden, die nur unter bestimmten Umständen virtuell sein müssen, nämlich wenn das Überschreiben einer Methode tatsächlich genutzt wird. Ein Vorteil für nicht-virtuelle Methoden ist ja die (Begründung siehe unten) höhere Ausführgeschwindigkeit, da nicht zur Laufzeit überprüft wird zu welcher Methode gebunden werden muss. Der JIT-Compiler kann also eine Methode als nicht-virtuelle Methode kompilieren (und außerdem noch inlinen), egal ob sie überschrieben wurde oder nicht. Wird nun festgestellt, dass eine andere Methode, die diese Methode überschreibt, genutzt wird, kann der JIT-Compiler diese Optimierung zurücknehmen. Das geht bei C#, wo von dem CLI Just-in-Time vor dem Lauf kompiliert wird, nicht! Deswegen gibt es unter C# auch die Kennzeichnungen "virtual", "override" und "new" für Methoden."

1.5.1.1.1 Ausführungszeit virtuelle – nichtvirtuelle Methoden

Eine virtuelle Methode hat gegenüber einer nichtvirtuellen Methode eine niedere Ausführgeschwindigkeit (braucht mehr Rechenzeit).
Begründung:
Bei einer nichtvirtuellen Methode wird bei einem Aufruf der Compiler eine feste Adresse codieren und so was ähnliches wie z.B. "springe zu Adresse 0800" in Maschinencode übersetzen. Bei einer virtuellen Methode dagegen steht diese Adresse erst zur Laufzeit (nicht vor der Laufzeit) fest.
Zum Beispiel steht im Inhalt der Adresse 0123 die Adresse zu der gesprungen werden soll.
D.h. man braucht dann vor dem Sprung noch einen weiteren Befehl. also insgesamt:
Lade den Inhalt von 0123 in das Register BX
Springe zu Adresse BX.

Die Methode immer als virtuell anzunehmen ist daher schlechter, als sie als nicht virtuell zu betrachten - wann immer das geht.

1.5.1.1.2 Spekulieren

Angenommen, die Klasse Quadrat und die Klasse Kreis sind Unterklassen der Klasse GeoForm und angenommen, in der Klasse Quadrat wird die Methode printAllAttributes() überschrieben, aber in der Klasse Kreis nicht.
Dann kann es ja (aus welchen Gründen auch immer) sein, dass während der Laufzeit immer (oder vielleicht die ersten 100 000 mal) printAllAttributes() der Klasse Kreis aufgerufen wird.
Deswegen ist es sinnvoll, dass der JIT-Compiler so spekuliert, dass er die Methode printAllAttributes() zunächst als nicht virtuell kennzeichnet.
Wenn dann irgendwann einmal printAllAttributes() der Klasse Quadrat aufgerufen wird (oder ein Exemplar der Klasse Quadrat erzeugt wird), kann sich der JIT-Compiler korrigieren und printAllAttributes() als virtuell kennzeichnen.

Weitere Vorschläge zum Spekulieren:
1) Klasse nicht abstrakt, nur von der Klasse selbst (nicht von den Unterklassen) wird ein Objekt erzeugt:
Es ist sinnvoll, die Methode nicht virtuell zu machen

2) Klasse nicht abstrakt, Methode wird in keiner Unterklasse überschrieben:
Es ist sinnvoll, die Methode nicht virtuell zu machen

3) Klasse nicht abstrakt, auch von den Unterklassen wird ein Objekt erzeugt, Methode wird in allen Unterklassen überschrieben:
es ist sinnvoll, die Methode virtuell zu machen

4) Ausnahme (siehe 3):
Klasse nicht abstrakt, auch von den Unterklassen wird ein Objekterzeugt, Methode wird in allen Unterklassen überschrieben und Methode wird nur von Objekten der Oberklasse aufgerufen:
es ist sinnvoll Methode nicht virtuell zu machen

5) Klasse nicht abstrakt, auch von den Unterklassen wird ein Objekt erzeugt, Methode wird nicht in allen Unterklassen überschrieben:
wenn die nicht überschriebene Methode (der Oberklasse) häufig aufgerufen wird, ist es sinnvoll, die Methode nicht virtuell zu machen

6) Klasse abstrakt, Methode wird nicht in allen Unterklassen überschrieben:
wenn die nicht überschriebene Methode (der Oberklasse) häufig aufgerufen wird, ist es sinnvoll, die Methode nicht virtuell zu machen

7) Klasse abstrakt, Methode wird in allen Unterklassen überschrieben:
es ist sinnvoll, die Methode virtuell zu machen

8) Klasse abstrakt, Methode wird in keiner Unterklasse überschrieben:
es ist sinnvoll, die Methode nicht virtuell zu machen

1.5.1.2 Java-Compiler

Wenn während des Kompilierens Klassen benutzt werden, die sich außerhalb der gerade kompilierten Klasse befinden, ist das auf zwei Arten möglich:
a) importieren (mit dem Schlüsselwort import) des Packages in der sich diese Klasse befindet. Der Import eines Packages hat nur den Sinn, dass nicht immer jeder Klassennamen ausführlich (inklusive Packagenamen) angegeben werden muss, der sogenannte fully qualified name (fqn).
b) Angabe des sogenannten fully qualified name (fqn), das bedeutet die Angabe des Packages, gefolgt von einem Punkt und den Namen der sich dort befindlichen Klasse.

Die Importe sind nur für den Compiler gedacht (werden nur vom Compiler verwendet), zur Laufzeit sind die nicht mehr da.
Im Bytecode sind dann schon bei den Klassen die Packagenamen aufgeführt.

Der Java-Compiler (der Bytecode erzeugt) erstellt für jede Klasse eine eigene class-Datei, auch wenn mehrere Klassen in derselben java-Datei vorkommen.
Jede Klasse, die verwendet wird, muss irgendwo (z.B. im Quelltext einer anderen Klasse) erwähnt werden, oder eben erst zur Laufzeit (explizit durch den Programmierer, wenn er es will) mit Class.forName() geladen werden.
Der Java-Compiler setzt also während des Kompilierens Verweise auf Methoden, die sich z.B. auf andere Klassen (in anderen class-Dateien) beziehen. Der Compiler kann diese Verweise nicht in konkrete Adressen (im Arbeitsspeicher) übersetzen, da er nicht weiß, wo sich der Maschinencode dieser Klassen (beim dynamischen Laden dieser Klassen) im Arbeitspeicher befindet.

Schon während des Kompilierens kann der Compiler Informationen in anderen schon kompilierten (class-) Dateien auswerten:
Wird z.B. eine Methode einer Klasse aufgerufen, die sich außerhalb der gerade kompilierten Datei befindet, dann kann er nachprüfen, ob diese Methode "passende Datentypen" hat (die Anzahl der Parameter und die jeweiligen Datentypen der Parameter müssen gleich sein).

Bemerkung:
Viele moderne Interpreter (Perl, php, ...) kompilieren das Script in Bytecode und führen es dann aus. Der Bytecode wird aber nicht gespeichert.

1.6 Linker

1.6.1 statischer Linker

Beim statischen Linken werden die Teile der Bibliothek, die benutzt werden (z.B. beim Auruf einer Funktion f) als fester Bestandteil in die exe-Datei reinkopiert.
Wenn eine neue Version der Bibliothek benutzt wird, muss der Programmierer nochmals neu linken. Dann werden eben die Teile der neuen Bibliothek, die benutzt werden (z.B. beim Auruf der Funktion f) als fester Bestandteil in die neue exe-Datei reinkopiert.

1.6.2 dynamischer Linker

Beim dynamischen Linken wird die Bibliothek (bzw. Teile davon) nicht in die exe-Datei reinkopiert. Stattdessen wird während der Laufzeit (z.B. beim Aufruf der Funktion f) die zu f gehörige Bibliothek in den Arbeitsspeicher geladen (falls sie sich nicht schon dort befindet) und der zu f gehörige Maschinencode angesprungen.
Wenn eine neue Version der Bibliothek benutzt wird, muss der Programmierer nicht nochmals neu linken. Zur Laufzeit wird der zu f gehörige Maschinencode angesprungen.

Beispiel:
In einer Datei wird f1 aufgerufen:
f1()

In der Bibliothek gibt es eine Tabelle, in der zu jeder Funktion f der Bibliothek die zu f zugehörige Adresse aufgelistet ist:

Tabelle (z.B.am Anfang) in der Bibliothek:

v_f1	& v_f1
v_f2	&v_f2
...	...

Der dynamische Linker trägt nun statt f1() einen Verweis auf v_f1 ein.
Während der Laufzeit wird dann die tatsächliche Adresse &v_f1von v_f1 geladen und dorthin gesprungen.

1.6.3 Vergleich statischer - dynamischer Linker

Vorteil dynamischer Bibliotheken
1) Der Vorteil (und das Problem) ist, dass man die sogenannte dynamische Bibliotheken (engl: Dynamic Link Library, kurz: DLL) , in denen sich die Methoden und Klassen von C++ - Programmen befinden, austauschen kann.
Damit kann man zentrale Teile wie MFC (Microsoft Foundation Class) austauschen ohne ein Programm neu kompilieren zu müssen. Für den, der kompiliert ist es zwar egal, wenn er neu kompilieren muss oder nicht, aber nicht für den Anwender.
Dieser will in ganz seltenen Fällen fast das komplette Windows in seinem Programm (es wird dann z.B. auch größer, vom Lizenzrechtlichen ganz abgesehen).

b) Weiterer Vorteil:
Bibliotheken, die gerade nicht gebraucht werden, verschwenden (im Gegensatz zum statischen Linken) keinen Arbeitsspeicher.

Nachteile (DLL Hell, DLL-Hölle)
Unter Windows können folgende Fälle auftreten:
a) Wenn man ein Programm A deinstalliert, wird man gefragt, ob man die zugehörige DLL löschen will. Wenn diese aber gelöscht wurde, konnte ein anderes Programm B diese DLL auch nicht mehr benutzen, so dass das Programm B nicht mehr lief.

b) Wenn Programm X eine neue Version einer DLL benötigte, wurde die alte DLL-Version überschrieben. Ein Programm Y, das die alte DLL-Verision benötigt, funktioniert dann nicht mehr. Wenn man also (für zwei verschieden Programme) zwei Versionen der selben DLL benötigt, gibt es Probleme.

1.6.3.1 Java-Linker

Der Java-Linker ist ein dynamischer Linker, d.h. während des Interpretierens werden die Bytecodes der verschiedenen class-Dateien verlinkt, d.h. wird im Bytecode der Class-Datei1 eine Methode der Class-Daiei2 verwendet, so lädt der Linker (ohne Zutun des Programmierers) den Bytecode der zugehörigen Klassen (mit den Methoden und Attributen) in den Arbeitsspeicher, so dass darauf zugegriffen werden kann.

Java hat zwar auch einen statischen Linker, doch dieser kann während des Kompilierens nur Konstanten aus java- und class-Dateien inlinen.

Plattform
Eine Plattform besteht aus den Komponenten:
Hardware (CPU), Betriebssystem und der dynamischen Bibliothek (DLLs bzw. Klassenbibliothek).