Von Assembler zu Java

Prof. Dr.-Ing. Thomas Schwotzer

1 Einführung

Die erste imperativen Programme wurden in den Urzeiten der IT tatsächlich direkt auf der Hardware der Maschinen geschrieben. Die verfügbaren Befehle der jeweiligen Prozessoren wurden direkt angesprochen.

Die Prozessoren hatten (und haben) unterschiedliche Befehlssätze. Sie gibt es einen grundsätzlichen Unterschied zwischen RISC (*reduced instruction set computer*) und CISC (*complex ..*) Prozessoren.

RISC Prozessoren verfügen über einen möglichst kleinen Satz an Befehlen. Das macht die Prozessoren billiger und schneller.

CISC Prozessoren bieten deutlich mehr Befehle an als RISC. Sie sind teurer Die einzelnen Befehle mögen zwar teilweise deutlich länger dauern aber dafür lösen sie komplexere Probleme. Fakt ist aber auch: Jedes Programm was man auf einem CISC Rechner schreiben kann, kann man auch auf einem RISC Rechner implementieren – macht braucht nur mehr Befehle. Wir wollen das Thema CISC/RISC hier nicht vertiefen.

In den Urzeiten der IT, d.h. bis in die 1960er hinein hat man eher daran gedacht, die Hardware für ein Problem zu bauen, d.h. spezielle Prozessoren für spezielle Aufgaben zu bauen.

Ab den 1970er änderte sich das grundsätzlich. Rechner fanden eine stark wachsende Verbreitung und es eine rasant wachsende Zahl von Problemen wurden durch IT bearbeitet. Gesucht waren Prozessoren, die möglichst viele Probleme lösen können.

Gesucht waren aber auch Entwickler/innen, die in der Lage waren möglichst schnell Programme zu schreiben. Schnell stellt man fest: Es ist enorm ineffektiv, wenn das gleiche Programm für jeden Prozessor neu geschrieben wird. Besser wäre es, wenn ein Programm einmal geschrieben wird und dann läuft es auf verschiedenen (möglichst allen) Prozessoren die am Markt verfügbar sind.

2 Maschinen- und Hochsprache

Eine Maschinensprache ist die Menge aller möglichen Befehle, die ein Prozessor ausführen kann. Alle Prozessoren können Speicher schreiben und lesen und addieren. Manche können komplexere Funktionen direkt auf der Hardware durchführen.

Programme in Maschinensprache funktionieren nur auf dem jeweils passenden Prozessor. Die Idee einer Hochsprache besteht darin: Es wird eine Sprache entwickelt, die – für Menschen[1] – lesbarer sein soll.

C war die erste Sprache, die weite Verbreitung fand (und noch heute hat), die nicht für einen konkreten Prozessor entwickelt worden war. Das folgende ist ein C Programm:

```
int main(char** args) {
    int x, y, z;
    x = 1;
    y = 2;
    z = x + y;
}
```

Es wird eine Funktion definiert. Dann werden drei Variablen definiert (a,b,c). Den Variablen a und b wird ein Wert zugewiesen. Am Ende wird der Variablen c die Summe der beiden Werte zugewiesen.

Nettes kleines Programm. Das Problem ist nur: Dieses Programm läuft auf keinem Prozessor. Wie auch? Es ist nicht in der Maschinensprache des Prozessors geschrieben.

Für jede Hochsprache werden *Compiler* benötigt.

Compiler sind Programme. Diese Programm nehmen das Programm der Hochsprache als Eingabe und produzieren ein Programm in Maschinensprache. Und wenn der Compiler keinen Fehler hat, dann macht das Maschinenprogramm das, was sich die Entwickler in der Hochsprache beschrieben haben.

Abbildung 1 stellt die Situation dar.

Das obige Programm wird in ein Maschinenprogramm übersetzt[2].

Und wenn Sie sich fragen, in welcher Sprache der erste Compiler geschrieben wurden, dann liegt die Antwort auf der Hand: In Maschinensprache natürlich. Aber bereits der zweite Compiler kann in C geschrieben werden. Das muss man mal durchdenken: Wenn man einen Compiler hat, dann kann man mit einer Hochsprache einen neuen Compiler programmieren, den man mit dem alten übersetzt. Ja, das bereitet etwas Kopfschmerzen – und wir vertiefen das auch nicht weiter.

Der Compiler muss die Konzepte der Hochsprache in die Maschinenbefehle übersetzen. Es gibt keine Variablen in einem Rechner. Der Compiler schreibt also die Werte 1 und 2, die eigentlich x und y zugewiesen werden sollen in den Speicher. Der Compiler hat sich offenbar entschieden, dass die Werte von x im Speicher auf der Adresse 1 stehen und der Wert von y in der Zelle 2. Was in

[1] konkret ITler, die auch Menschen sind, auch wenn sie manchmal scheinbar wirr reden!

[2] Das ist nicht ganz exakt, was die Grafik darstellt. Maschinenbefehle sind immer als Zahlen gespeichert. Ein Maschinenprogramm ist daher immer eine Abfolge von Zahlen, die die CPU als Befehl interpretieren kann. Das ist leider denkbar schlecht lesbar für Menschen. Daher hat man sich Assember ausgedacht und genau das ist, was in der Grafik symbolisiert ist. Für jeden Maschinenbefehl definiert man einen Assemblerbefehl, der etwas besser für Menschen lesbar ist. Da wir aber (leider) keine Rechnerorganisation oder Compilerbau machen, ignorieren wir diese wichtigen Unterschied und behandeln Assembler und Maschinencode gleich.

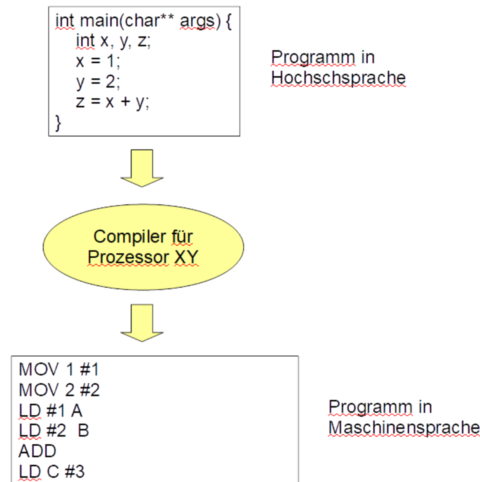

Abbildung 1: Compiler: Hochsprache zu Maschinensprache

der Hochsprache x heißt, ist im Maschinenprogramm Speicherzelle Nummer 1. Kann man so machen.

Dann soll addiert werden. Es werden als erst einmal die beiden Werte aus dem Speicher in die CPU geladen. Dann werden sie addiert und das Ergebnis wird in den Speicher zurück geschrieben. Konkret wird das Ergebnis auf die Adresse 3 im Speicher geschrieben.

Die Kunst der Entwicklung einer Hochsprache besteht darin: Man braucht eine Sprache, die leichter zu programmieren ist als Maschinencode. Diese Sprache muss aber auch ohne Probleme durch ein Programm in Maschinencode übersetzt werden können. Das geht. C ist der Beweis.

3 Betriebssysteme und Compiler

Schauen wir uns aber das Programm an:

```
int main(char** args) {
    printf("Hallo Welt");
}
```

Dieses C Programm wird die Zeichenkette `Hallo Welt` auf dem Bildschirm ausgeben. Wie macht der Prozessor das? Die Antwort ist: Macht er nicht - macht das Betriebssystem. Wir wollen hier dem Modul Betriebssysteme nur ganz wenig vorgreifen: Ein Betriebssystem bietet u.a. eine Fülle von Funktionen an, die die viele Programm benötigen.

Die Ausgabe auf den Bildschirm gehört dazu. Anstatt dass sich alle Entwickler damit herumschlagen, welche elektrischen Signale man auf ein Röhrengerät

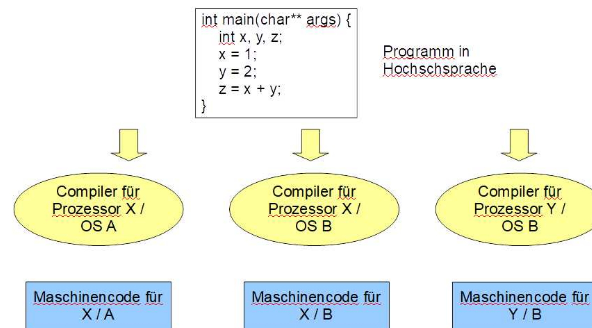

Abbildung 2: Compiler Abhängigkeit vom Betriebssystem

schickt, hat das einmal jemand gemacht und es zum Teil eines Betriebssystems gemacht.

Was macht nun aber der Compiler? Der Compiler übersetzt den Befehl `printf("Hallo Welt")` in einen Aufruf einer Betriebssystem-Funktion. Wir werden das hier nicht vertiefen - spitzen Sie die Ohren in dem anderen Modul!

Entscheidend ist aber: Ein Compiler übersetzt ein Programm für einen Prozessor *und* ein Betriebssystem. Der Maschinencode des gleichen Programms auf dem gleichen Prozessor unterscheidet sich vom genutzten Betriebssystem.

Das ist an sich nicht weiter schlimm, man muss es nur wissen. Die Welt sieht also eigentlich so aus, siehe Abbildung 2.

Für jede Prozessor/Betriebssystem-Kombination muss es einen Compiler geben. Das war lange Jahre auch kein Problem. Das Problem ist etwas anderes.

4 Virtuelle Maschine

Wenn man eine IT-Firma ist, dann verkauft man Software. Software sind Programme. Die konnte man auch im Jahre 1970 schon klauen und oft will man auch nicht, dass die liebe Konkurrenz den Code zu Gesicht bekommt. Daher liefert man gern compilierte Programm aus und nicht den Quellcode.

Was aber, wenn das Unternehmen das Programm nicht in alle denkbaren Kombinationen von Maschinencode übersetzt hat? Was wenn es neue Prozessoren gibt, aber das Unternehmen nicht mehr existiert? Und überhaupt ist es schon anstrengend sich einen Zoo von unterschiedlichen Compilern zu halten.

Ende der 1990er Jahre kamen Leute die bei der Firma SUN gearbeitet haben auf folgende Idee: Wie wäre es, wenn man eine Sprache baut und dafür nur einen einzigen Compiler bereit stellt. Der erzeugt Maschinencode. Dieser Maschinencode soll aber gar nicht auf einer existierenden Hardware laufen. Dieser Code soll auf einer virtuellen Hardware, einer virtuellen Maschine laufen.

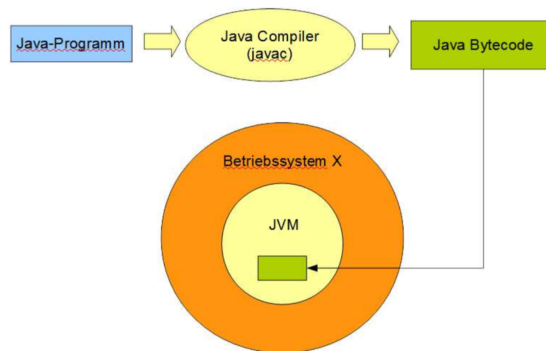

Abbildung 3: Virtuelle Maschine

Hä? Auf die Idee konnte man auch erst in den 1990ern kommen als die Prozessoren sehr schnell waren. Die Idee ist in vielen Systemen umgesetzt; sie ist das Konzept der virtuellen Maschine:

Man beschreibt eine solche virtuelle Maschine. Diese ist vergleichbar mit einem realen Prozessor. Wie ein Prozessor bietet sie eine Satz von Befehlen an und sie kann Programme abarbeiten.

Man schreibt Compiler, die die Hochsprache (es ist Java!) in die Sprache dieser Java Virtual Machine (JVM) übersetzt. Das kennen wir an sich schon.

Die JVM wiederum ist ein Programm! Für jede Betriebssystemversion muss eine eigene JVM gebaut werden und das Ganze sieht dann so aus wie in Abbildung 3.

Das Java Programm wird übersetzt in den Java Bytecode (so nennt sich die Java Maschinensprache). Dieser Bytecode wird einem Programm übergeben, das auf einem Betriebssystem läuft und diesen Bytecode abarbeitet. Das ist die Java Virtual Machine (JVM).

Anfangs wurde das Konzepte sehr skeptisch gesehen aber das Konzept *write once run everywhere* mit der VM hat sich durchgesetzt.

C# nutzt das gleiche Konzept. Und es gibt nun auch andere Sprachen (Scala z.B.) die Bytecode generieren und damit auf einer JVM laufen.

Androidprogramme sind ebenfalls Java. Allerdings wird nicht die JVM genutzt, sondern eine eigene Entwicklung. Das hat wiederum mit rechtlichen Streitigkeiten zu tun zwischen Google und Oracle. Oracle hat SUN vor Jahren gekauft und Google war nicht bereit die geforderten Lizenzkosten zu zahlen.

Android-Programme und andere Java-Programme sind aber dort wo man in den 1970 auch einmal war: Die Hochsprache ist Java, aber die Compiler erzeugen unterschiedlichen Bytecode für die unterschiedlichen Virtuellen Maschinen.

5 Ein Java Programm

Und hier ist endlich auch mal ein Java-Programm.

```
class X {
    public static void int main(String[] args) {
        int x, y, z;
        x = 1;
        y = 2;
        z = x + y;

        System.out.println("Hallo Welt");
    }
}
```

Die Elternschaft von C ist in Java nicht zu übersehen! Die Details dieses Programm diskutieren wir im SU und in den Übungen. Lesen Sie unbedingt in Ihrem bevorzugten Java-Buch nach.

5.1 Variable

Jede imperative Hochsprache benutzt Variablen. Eine Variable ist ein Speicherplatz. Es ist ein Ort an dem Daten abgelegt werden.

Wir können beliebig viele Variable definieren. Wir können aus dem Vollen schöpfen. Das ist gut. In obigem Beispiel wurden die Variable x, y und z definiert. Wir haben auch gesehen, was der Compiler daraus gemacht hat. Jeder Variablen wird ein Speicherplatz zugewiesen. Was in der Hochsprache x heißt ist im Maschinenprogramm die Speicherzelle mit der Nummer 1.

Das schöne an einer Hochsprache ist, dass wir uns nicht damit beschäftigen müssen an welche Adresse denn irgendwelche Daten liegen. Wir sagen einfach nur: Ich brauche Platz. Und diesem Platz gebe ich einen Namen. Fertig ist das. Gut!

In Java muss jede Variable vor ihrer Benutzung **deklariert** werden. Das geschieht in der Zeile

```
int x, y, z;
```

Damit wird dem Compiler klar gemacht, dass er in folgenden drei Speicherzellen suchen muss, in denen er Werte ablegt. Und der Compiler muss sich auch merken, welcher Variablen welcher Speicherplatz zugewiesen wurde. Das ist nicht schwer. Das ist eine einfache Tabelle, die der Compiler hält:

Variable	Speicherplatz
x	#1
y	#2
z	#3

Mit dieser Tabelle ist es dann auch nicht schwer die weiteren Befehle zu übersetzen.

```
x = 1;
y = 2;
```

Diese Java-Befehle sind Zuweisungen. Es bedeutet, dass einer Variablen ein Wert zugewiesen wird. Nachdem diese Befehle ausgeführt wurden, soll die Variable x den Wert 1 beinhalten und die Variable y den Wert 2. Das ist nicht schwer.

Dem Compiler fällt es auch nicht schwer, Maschinencode zu produzieren: Jede Variable entspricht einem Speicherplatz. Es gibt einen Maschinenbefehl der den Speicherplatz füllt. Dank der obigen Tabell kennt der Compiler auch die Adresse, die er jeder Variablen zugewiesen hat. Also kann er die beiden Java-Befehle übersetzen:

```
MOV 1 #1
MOV 2 #2
```

Merken wir uns: Variable sind Speicherzellen. Variable müssen vor Nutzung deklariert werden. Variablen können Werte zugeweisen werden.

Es gibt unterschiedliche Variablentypen. Darauf gehen wir bald ein. Das Programm nutzt `int`. Das ist eine vorzeichenbehaftete 32 Bit Zahl. Passen Sie in TGI schön auf. Dort erfahren Sie viel über die internen Darstellung von Binärzahlen.