

Einleitung

1.1 Höhere Programmiersprachen

Programme werden heute zumeist in sogenannten problemorientierten, höheren Programmiersprachen geschrieben. Diese Programmiersprachen abstrahieren (in verschiedenem Maße) von der Struktur und den Details der Rechner, auf denen die geschriebenen Programme ausgeführt werden sollen. Die vier wichtigsten Klassen von universell einsetzbaren Programmiersprachen sind

- die Klasse der *imperativen Sprachen* wie etwa Algol 60, Algol 68, Fortran, Cobol, Pascal, Ada, Modula-2 und C. Sie orientieren sich eng an der Struktur des sogenannten von-Neumann-Rechners, die fast allen kommerziell erwerbbaren Rechnern zugrunde liegt und aus (aktiver) Zentraleinheit (CPU), (passivem) Speicher und einem Bus für den Verkehr zwischen Zentraleinheit und Speicher besteht.
- die Klasse der *funktionalen Sprachen* wie etwa pure Lisp, SML, OCaml und Haskell. Charakteristisch für diese Klasse ist es,
 - dass es keine Trennung zwischen der Welt der Anweisungen und der Ausdrücke gibt,
 - dass Namen nur als Bezeichner für Ausdrücke und Funktionen aber nicht für Speicherzellen dienen und
 - dass Funktionen als Argumente und Ergebnisse von Funktionen auftreten dürfen.

Das Ausführungsprinzip ist *Reduktion*; d.h. ein funktionales Programm wird ausgewertet, indem in einzelnen Schritten so lange jeweils ein (Teil-) Ausdruck durch einen äquivalenten, einfacheren ersetzt wird, bis dieser Prozess mit Erreichen der Normalform endet.

- die Klasse der *logischen Programmiersprachen* wie Prolog und seine verschiedenen Dialekte. Diese Sprachen basieren auf einer operationellen Sicht der Prädikatenlogik. Der Ausführungsmechanismus ist *Resolution*, ein Verfahren, das für das Beweisen von Implikationen in der Prädikatenlogik der ersten Stufe entwickelt wurde.

- die Klasse der *objektorientierten Programmiersprachen* wie etwa Smalltalk, C++ oder Java. Ihre Vertreter sind im Kern imperativ, verfügen gegebenenfalls über Typsysteme, die Datenabstraktion unterstützen und eine „evolutionäre“ Art der Softwareentwicklung ermöglichen.

Neben diesen Klassen gibt es noch viele Sprachen für spezielle Anwendungen:

- Hardware-Beschreibungssprachen wie etwa VHDL. Sie dienen zur Spezifikation von Rechnern und Rechnerkomponenten. Solche Spezifikationen können das funktionale Verhalten, den hierarchischen Aufbau und die geometrische Platzierung von Komponenten beschreiben.
- Kommandosprachen von Betriebssystemen. Sie besitzen als primitive Konstrukte u.a. die Aktivierung von Systemfunktionen und Benutzerprogrammen und die Möglichkeit, mehrere solcher Programme und Systemfunktionen koordiniert zusammenwirken zu lassen, Prozesse zu erzeugen und zu beenden, Ausnahmesituationen zu entdecken und zu behandeln.
- Spezifikationssprachen für Druckseiten, Graphikobjekte oder Animationen. Ein Beispiel ist hier etwa die Programmiersprache Postscript von Adobe, die im Drucker das Erscheinungsbild der jeweiligen Seiten berechnet. Im Falle von Animationen müssen nicht nur die geometrischen Dimensionen der darzustellenden Objekte beschrieben werden, sondern auch zeitliche Abläufe und möglicherweise vorgesehene Reaktionen auf Ereignisse.
- Sprachen zur Bearbeitung strukturierter Dokumente. In den letzten Jahren hat sich XML als Standardformat zur Repräsentation strukturierter Daten durchgesetzt. Die Fülle der Anwendungen und die Weite der Verbreitung im Internet führte zu einer Vielzahl weiterer Standards, angefangen von XSchema zur sehr genauen Beschreibung von Dokumenten über die XML-Transformationssprache XSLT und die XML-Anfragesprache XQuery bis hin zu Formalismen im Zusammenhang mit Web Services oder Business Processes.

1.2 Die Implementierung von Programmiersprachen

Damit Programme einer bestimmten Programmiersprache L auf einem Rechner ausgeführt werden können, muss diese Programmiersprache auf diesem Rechnertyp verfügbar gemacht, man sagt, *implementiert* werden. Dies kann auf verschiedene Weise geschehen. Man teilt die Implementierungen in interpretierende und übersetzende Verfahren ein.

1.2.1 Interpreter

Wir betrachten eine Programmiersprache L . Ein Interpreter I_L bekommt als Eingabe ein Programm p_L aus L und eine Eingabefolge e und berechnet daraus eine Ausgabefolge a . Eventuell führt die Interpretation von p_L auch zu einem Fehler. Also hat I_L die Funktionalität

$$I_L : L \times D^* \rightarrow D^* \times \{\text{error}\},$$

wenn sowohl Eingabe- wie Ausgabedaten aus einem Bereich D stammen. Die oben geschilderte Ausführung des Programms p_L mit Eingabefolge e und Ergebnisfolge a ist dann durch die Gleichung

$$I_L(p_L, e) = a$$

beschrieben.

Was ist kennzeichnend für die Arbeitsweise eines Interpreters? Er bearbeitet das Programm p_L und die Eingabe e zur gleichen Zeit. Jedem Konstrukt steht er – auch bei erneuter Ausführung – unvorbereitet gegenüber. Er nutzt keine von den Eingabedaten unabhängige Informationen aus, die er durch Inspektion des Programmtextes gewinnen könnte, etwa die Zahl der deklarierten Variablen in einem Block, einer Funktion oder einer Klausel. Diese Information könnte er für eine Speicherzuteilung mit effizientem Zugriff auf die Variablenwerte benutzen.

1.2.2 Übersetzer

Es gilt die mit der Interpretation verbundenen Ineffizienzen zu vermeiden. Dazu benutzt man ein in der Informatik häufig nützliches Prinzip, meist *Vorberechnung*, manchmal auch *partielle Auswertung* (Partial Evaluation) oder *gemischte Berechnung* (Mixed Computation) genannt.

Während der Interpreter I seine beiden Argumente, das Programm p_L und die Eingabefolge e , zur gleichen Zeit bekommt und verarbeitet, wird jetzt die Verarbeitung des Programms und der Eingabefolge auf zwei verschiedene Zeiten aufgeteilt. Erst wird das Programm p_L „vorverarbeitet“, d.h. unabhängig von Eingabedaten analysiert und in eine Form überführt, welche die effizientere Ausführung des Programms mit beliebigen Eingabefolgen erlaubt. Man nimmt dabei an, dass sich der zusätzliche Aufwand für die Vorverarbeitung des Programms bei der Ausführung auf einer oder mehreren Eingabefolgen amortisiert.

Wie sieht die Vorverarbeitung von Programmen aus? Meist besteht sie in der Übersetzung des Programms p_L der Sprache L , ab jetzt *Quellsprache* genannt, in die Maschinen- oder Assemblersprache M eines konkreten oder virtuellen Rechners. Die Zeit, zu der diese Übersetzung geschieht, heißt folglich *Übersetzungszeit*, das sich ergebende Programm p_M *Zielfprogramm* zu p_L .

Bei der Übersetzung von Quellprogrammen in Zielfprogramme werden insbesondere die beiden oben aufgeführten Quellen von Ineffizienz bei der Interpretation beseitigt. Jedes Programm wird einmal zur Übersetzungszeit analysiert; das erzeugte Zielfprogramm – nehmen wir an, es sei ein Programm in der Maschinensprache eines realen Rechners – erfährt an Analyse nur die Decodierung des Befehlscodes durch die Befehlseinheit des Rechners. Der effiziente, teilweise direkte Zugriff auf Variablenwerte bzw. Zellen wird durch ein Speicherverwaltungsschema ermöglicht, das allen Variablen des Programms feste (Relativ-) Adressen zuordnet. Diese Adressen stehen dann auch im erzeugten Zielfprogramm.

Das erzeugte Zielfprogramm p_M wird zu einer auf die Übersetzungszeit folgenden Zeit, genannt *Laufzeit*, mit der Eingabefolge e ausgeführt. Natürlich verlangen

wir von der Übersetzung, dass das Zielprogramm p_M bei der Ausführung mit Eingabe e nach Möglichkeit genau die Ergebnisfolge produziert, die der Interpreter auf p_L und e liefert.

Eine Mindestanforderung für die Übersetzung ist darum die folgende: Sei p_L ein Programm, welches syntaktisch korrekt ist und außerdem den Kontextbedingungen von L genügt. Der Übersetzer erzeuge für p_L das Zielprogramm p_M . Stößt I_L bei der Ausführung von p_L mit e auf keinen Fehler und produziert die Ausgabe a , so stößt auch p_M bei der Ausführung mit Eingabefolge e auf keinen Fehler, und die Interpretation und die Ausführung von p_M auf e liefern das gleiche Ergebnis.

Fassen wir die Maschine M als einen Interpreter I_M für ihre Maschinensprache auf, so muss für solche Kombinationen (p_L, e) also gelten:

$$I_L(p_L, e) = a \implies I_M(p_M, e) = a$$

Dazu gibt es mehrere Fehlersituationen. Einmal kann p_L syntaktische Fehler oder Verletzungen der Kontextbedingungen in Programmteilen enthalten, die der Interpreter bei der Ausführung mit e gar nicht berührt. Dann könnte die Interpretation erfolgreich ablaufen, während der Übersetzer, der das ganze Programm analysiert, die Übersetzung in ein Zielprogramm wegen entdeckter Fehler ablehnt. Zum anderen kann I_L , obwohl p_L syntaktisch und gemäß der Kontextbedingungen korrekt ist, bei der Ausführung mit Eingabe e auf einen (Laufzeit-) Fehler stoßen. Wenn wir I_L als die Definition der Semantik von L betrachten, muss dann auch das erzeugte Zielprogramm p_M bei der Ausführung mit e auf einen Fehler stoßen.

1.2.3 Reale und virtuelle Maschinen

In der Regel wird dem Programmierer ein als *real* bezeichneter Rechner zur Verfügung stehen; reale Rechner sind in großer Vielfalt käuflich zu erwerben und zwar in Form von Hardware, d.h. Platinen bestückt mit irgendeinem Prozessor, Speicherchips, und was sonst noch erforderlich ist. Die Zielsprache der Übersetzung ist in diesem Fall durch den verwendeten Prozessortyp definiert.

Will man für eine höhere Programmiersprache Code erzeugen, wird man jedoch schnell feststellen, dass man bei der Übersetzung gerne Befehle verwenden würde, die so von einer gegebenen realen Maschine nicht bereit gestellt werden. Andererseits ändern sich die Instruktionssätze moderner Rechner so schnell, dass es nicht sinnvoll erscheint, den Compiler zu sehr auf zufälligerweise bereitgestellte Operationen festzulegen. Eine solche Festlegung könnte nämlich bedeuten, dass man nach wenigen Jahren den Compiler für die nächste Generation von Rechnern neu schreiben müsste.

Bereits bei der Implementierung des ersten Pascal-Übersetzers kam man darum auf die Idee, zuerst Code für eine leicht idealisierte *virtuelle* Maschine zu erzeugen, deren Befehle dann jeweils nur noch auf den verschiedenen konkreten Zielrechnern zu implementieren waren. Auch die Übersetzung moderner Programmiersprachen wie Prolog, Haskell oder Java basieren auf diesem Prinzip. Einerseits erleichtert dieses Vorgehen die Portierbarkeit des Übersetzers. Andererseits vereinfacht dies die

Übersetzung selbst, da man den Befehlssatz entsprechend der jeweiligen zu übersetzenden Programmiersprache geeignet wählen kann.

Neue Anwendungen im Bereich des Internet haben die Idee virtueller Maschinen seit einiger Zeit zusätzlich attraktiv gemacht. Die Portierbarkeit, die man durch eine Implementierung einer Programmiersprache auf einer virtuellen Maschine gewinnt, kann man ausnutzen, um Systeme *plattformunabhängig* zu realisieren, d.h. so, dass sie unter verschiedenen Betriebssystemen lauffähig sind. Noch etwas weiter gedacht, kann Code so sogar *mobil* gemacht und über das Internet verbreitet werden. Das ist unter anderem die Idee hinter der Programmiersprache Java.

Mit der Ausführung fremden Codes auf dem eigenen Rechner handelt man sich jedoch nicht nur Vergnügen ein, sondern setzt sich auch den Angriffen böswilliger Angreifer aus. Hier bietet eine virtuelle Maschine ebenfalls eine Lösung: da der Code nicht direkt auf der eigenen Hardware ausgeführt wird, kann man das Verhalten des auszuführenden Codes genau beobachten bzw. seine Zugriffsrechte auf die Ressourcen des Rechners entsprechend einschränken. Diese Idee nennt man auch das *Sandkasten-Prinzip* (Sand Boxing).

In diesem Buch stellen wir virtuellen Maschinen für imperative, funktionale, logische und objektorientierte Programmiersprachen vor. Insbesondere sind wir dabei natürlich an den Übersetzungsschemata interessiert, wie man für die jeweiligen konkreten Programmkonstrukte der Programmiersprache die zugehörigen Befehlsfolgen der virtuellen Maschinen konstruiert.

1.2.4 Kombinationen von Übersetzung und Interpretation

Verschiedene Kombinationen von Übersetzung und Interpretation sind möglich. Werden Quellprogramme in die Maschinensprache einer virtuellen Maschine übersetzt, so wird diese meist durch einen Interpreter, einem Stück Software interpretiert. Allerdings kann man auch die Sprache der virtuellen Maschine weiter übersetzen, z.B. in die Sprache eines realen Rechners, heute *nativer Code* genannt. Interessante Unterschiede ergeben sich, je nachdem zu welcher Zeit und auf welcher Maschine dieser zweite Übersetzungsschritt erfolgt. Er kann nämlich auch zur Ausführungszeit erfolgen und wird dann *just-in-time Übersetzung (JIT)* genannt. Meist werden die beiden Schritte auf zwei verschiedenen Rechnern ausgeführt. Der erste Schritt passiert auf dem Rechner des Programmierers. Hier wird ein portabler Zwischencode in der Sprache einer virtuellen Maschine erzeugt. Dieser wird eventuell auf einen anderen Rechner heruntergeladen. Der zweite Übersetzungsschritt wird auf diesem Rechner durchgeführt und erzeugt möglichst effizienten nativen Code für diesen Rechner. Es werden also Übersetzungszeit und Ausführungszeit kombiniert; der JIT-Übersetzer wird zur Ausführungszeit für ein Stück des Programms, z.B. eine Funktion aufgerufen, meist zu dem Zeitpunkt, wenn diese Funktion gebraucht wird.

Das Ziel der JIT-Übersetzung ist die Kombination von Effizienz und Portierbarkeit. Der erste Übersetzungsschritt, der im Allgemeinen sehr aufwendig ist, wird nur einmal gemacht. Der zweite Schritt, die Übersetzung des Zwischencodes, ist einfacher. Der erzeugte native Code ist (hoffentlich) effizient. Er wird in einem Speicher,

meist Cache genannt, aufgesammelt, so dass jede Funktion nur einmal übersetzt werden muss.

1.3 Allgemeine Literaturhinweise

Eine Übersicht über die Geschichte der Programmiersprachen und verschiedene Programmiersprachenkonzepte bieten u.a. [Seb05, Sco05, TN06]. Den Einsatz der Hardwarebeschreibungssprache VHDL beschreibt [Ped04]. Die Seitenbeschreibungssprache Postscript findet man in [Inc99]. Eine Übersicht über ActionScript zur Realisierung von Flash-Animationen enthält etwa [Hau06]. Die XML-Transformationssprache XSLT-2.0 wird ausführlich in [Kay04] erläutert. Für die XML-Anfragesprache XQuery kann man [KCD⁺03, Bru04] konsultieren. Einen Einstieg in gängige W3C-Standards rund um Web Services und Business Processes findet man etwa über [ACKM03]. Unterschiedliche Anwendungen für virtuelle Maschinen insbesondere im Bereich von Betriebssystemen behandelt [SN05].



<http://www.springer.com/978-3-540-49596-3>

Übersetzerbau

Virtuelle Maschinen

Wilhelm, R.; Seidl, H.

2007, XIII, 192 S., Softcover

ISBN: 978-3-540-49596-3