1 Einleitung

Unter *Compilerbau* oder auch *Übersetzerbau* versteht man die Entwicklung von Compilern, d.h. von Übersetzern höherer Programmiersprachen in Maschinencode. Die Vorlesung behandelt *allgemeine sprachunabhängige Techniken und Methoden* des Compilerbaus und ihre theoretischen Grundlagen.

1.1 Grundlegende Begriffe

Zu Beginn legen wir zunächst fest, was wir unter einem Compiler verstehen wollen:

> Ein *Compiler* ist ein Programm zur Übersetzung von Programmen einer höheren Programmiersprache (Quellprogramme) in äquivalente Programme einer Zielsprache (Zielprogramme).

Unter einer höheren Programmiersprache versteht man dabei imperative und objektorientierte Programmiersprachen oder deklarative, d.h. funktionale oder Logik- Programmiersprachen. In den imperativen Programmiersprachen bestehen Programme aus Wertzuweisungen, Kontrollstrukturen, Datenstrukturen und Programmstrukturen wie Modulen oder Klassen. In deklarativen Sprachen bilden Ausdrücke, Funktionen und Prädikate, Rekursion und algebraische Strukturen zentrale Elemente. Der Sprachtyp der Quellsprache spielt vor allem in den "späten Phasen" der Übersetzung, d.h. bei der Codegenerierung eine zentrale Rolle, während die Techniken und Methoden der frühen Übersetzungsphasen, bei denen die Quellsprache erkannt und analysiert wird, weitgehend sprachunabhängig sind. Hier geht wesentlich die Theorie der formalen Sprachen (nach Chomsky) und die pragmatische Umsetzung von diesbezüglichen Aufgaben in Algorithmen ein. Dies erklärt auch, warum Compilertechniken in vielen Anwendungen von Bedeutung sind, in denen Beschreibungssprachen für spezielle Zwecke gebraucht werden, etwa Dokumentbeschreibungssprachen wie LaTeX oder HTML oder Datenbankanfragesprachen wie SQL, um nur zwei Beispiele zu nennen. Die Analysetechniken des Compilerbaus und die zugehörigen Werkzeuge können hier eine wichtige Unterstützung bieten.
Die Zielsprache eines Compilers kann prinzipiell eine beliebige Programmiersprache sein. Meist handelt es sich allerdings um eine Maschinensprache (native code), eine Assemblersprache oder eine systemnahe Sprache wie etwa C. Bei sogenannten Cross-Compilern ist die Zielsprache eine andere höhere Programmiersprache, wie etwa Java. Verschiedene Aspekte von Programmiersprachen spielen bei der Übersetzung eine Rolle:

Syntax: formaler hierarchischer Aufbau eines Programms aus strukturellen Komponenten (wie z.B. Deklarationen, Kontrollstrukturen, Ausdrücke...). Meistens wird die Syntax in erweiterter Backus-Naur-Form (EBNF) spezifiziert.

Semantik: Bedeutung eines Programms.

> Diese kann in verschiedener Weise angegeben werden, z.B. in Form der sogenannten *denotationellen Semantik*, bei der einem Programm die Ein-/Ausgabefunktion als Semantik zugeordnet wird:

$$\mathcal{M} : \text{Program} \rightarrow \text{Input} \longrightarrow \text{Ausgabe}$$

Pragmatik: alles, was den Umgang mit einer Programmiersprache erleichtert

Hierzu zählt etwa der sogenannte *syntaktische Zucker*, unter dem man benutzerfreundliche Formulierungen für Sprachkonstrukte versteht. Zum Beispiel vermittelt die Anweisung

```
if <bedingung> then <anweisung1> else <anweisung2>
```

eine intuitive Vorstellung über den Ablauf der Auswertung, während etwa die Formulierung

```
f_125 (<bedingung>, <anweisung1>, <anweisung2>)
```

keinerlei Rückschlüsse zulässt.

Zur Pragmatik gehören auch Hilfswerkzeuge wie etwa syntaxgesteuerte Editoren oder Debugger.

Unter der Äquivalenz von Programmen versteht man ihre semantische Gleichheit. Ein korrekter Compiler sollte natürlich Quellprogramme in äquivalente Zielprogramme übersetzen, d.h. es muss für einen Compiler $comp : Q \longrightarrow Z$, wobei Q und Z die Menge der Programme der Quell- bzw. Zielsprache seien, und beliebige Eingaben *inp* gelten:

$$\text{Compilerkorrektheit: } \mathcal{M}_Q[\![prog]\!]inp = \mathcal{M}_Z[\![compiler(prog)]\!]inp$$

Unter der *Übersetzungszeit* versteht man die Zeit, während der der Übersetzungsvorgang abläuft. Alle Informationen und Größen, die zu dieser Zeit bestimmt werden können, heißen *statisch*. Entsprechend ist die *Laufzeit* die Zeit, in der das Zielprogramm auf einer realen oder abstrakten Maschine ausgeführt wird. Größen, die dem Compiler noch nicht bekannt sind, nennt man *dynamisch*.
An Compiler werden häufig die folgenden Anforderungen gestellt:

Korrektheit Das Programm in Zielsprache (Ausgabe des Compilers) soll zu dem Programm der Quellsprache *semantisch äquivalent* sein (siehe oben).

Effizienz Die Programme in Zielsprache (Ausgabe) sollen effizient in Zeit- und Platzbedarf sein. Dies ist *keine* Forderung an die Effizienz des Compilers!

Fehlerbehandlung Der Compiler soll Fehler im Quellprogramm erkennen und in verständlicher Form mitteilen. Daneben kann er auch Debugging unterstützen, etwa durch interaktive Ausführung und Breakpoints.

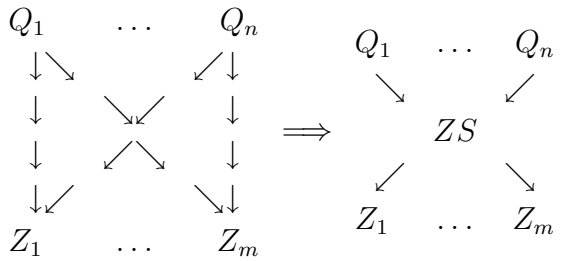
verschiedene Optimierungsstufen Je mehr Zeit für Optimierung investiert wird, desto länger kann die Übersetzung dauern. Daher soll die „Anzahl“ der Optimierungen einstellbar sein.

inkrementelle Übersetzung Bei einem erneuten Aufruf des Compilers werden nur die Programmteile übersetzt, die von Programmänderungen betroffen sind.

1.2 Struktur eines Compilers bzw. Übersetzungsvorgangs

Grundsätzlich geschieht die Übersetzung in verschiedenen Stufen, die im allgemeinen stark verzahnt ablaufen. Dabei werden zwei *Phasen* der Übersetzung unterschieden: die *Analyse-* und die *Synthesephase*, deren Bedeutung begrifflich klar ist. In der Analysephase wird die Struktur eines Programms bestimmt. Hier steht die Fehlererkennung im Vordergrund. In der Synthesephase (Synthese = Erzeugung) wird Maschinencode aus attributierten Syntaxbäumen generiert.
Daneben kann man auch das sog. *Frontend* und das *Backend* des Compilers unterscheiden, wobei das Frontend im Gegensatz zum Backend *maschinenunabhängig* ist. Es umfasst also die Analysephase zusammen mit der Zwischencodegenerierung und maschinenunabhängigen Optimierungen, während das Backend aus der eigentlichen Maschinencodegenerierung und -optimierung besteht.

Die einzelnen Compilerphasen sind in Abbildung 1 gezeigt. Die Analysephase besteht aus drei Stufen. Die erste Stufe ist die *lexikalische Analyse*, die von einem sogenannten *Scanner* durchgeführt wird. Der Scanner versucht in der Zeichenfolge, die er als Eingabe erhält, elementare lexikalische Komponenten wie Bezeichner, Zahlen und Schlüsselworte, die sogenannte *Mikrosyntax*, zu erkennen. Er produziert als Ausgabe eine Folge von sogenannten *Token* oder *Symbolen*. In der folgenden Phase wird aus dieser Folge die (i.allg. kontextfreie) *Syntax* des Programms, d.h. der hierarchische Programmaufbau durch den sogenannten *Parser* analysiert und als Ableitungs- oder Syntaxbaum dargestellt. Die anschließende *semantische Analyse* durch den *Analyser* behandelt die kontextsensitiven Anteile der Syntax. Sie versieht die Syntax mit sog. *Attributen*, etwa den Typen der Bezeichner. Außerdem werden Kontextabhängigkeiten wie etwa die Deklariertheit von Bezeichnern überprüft. Auf dem entstehenden attributieren Syntaxbaum können bereits erste Optimierungen durchgeführt werden. Während der gesamten Analysephase wird die *Symboltabelle* aufgebaut und nach und nach mit Inhalten gefüllt, bevor sie in der Codeerzeugung benutzt werden kann. Hier werden die Bezeichner (Speicheradressen), Zahlen und symbolischen Konstanten verwaltet.
In der Synthesephase wird (evtl. über bestimmten Zwischencode) der Code in der Zielsprache erstellt und evtl. noch optimiert (Entfernung von Redundanzen, Umordnung). Die Erzeugung von Zwischencode hat verschiedene Vorteile. Es können maschinenunabhängig eine Reihe von Optimierungen zur Verbesserung der Laufzeit und des Speicherbedarfs des erzeugten Maschinenprogramms durchgeführt werden. Die Portabilität wird erhöht. Statt bei n Quell- und m-Zielsprachen $n \times m$ separate Compiler zu entwickeln kann man mit $n + m$ Übersetzern auskommen, indem man eine geeignete Zwischensprache ZS verwendet:



Beispiel: Zur Illustration der grundsätzlichen Arbeitsweise eines Compilers zeigen wir, wie das folgende Pascal-Programmfragment übersetzt wird:

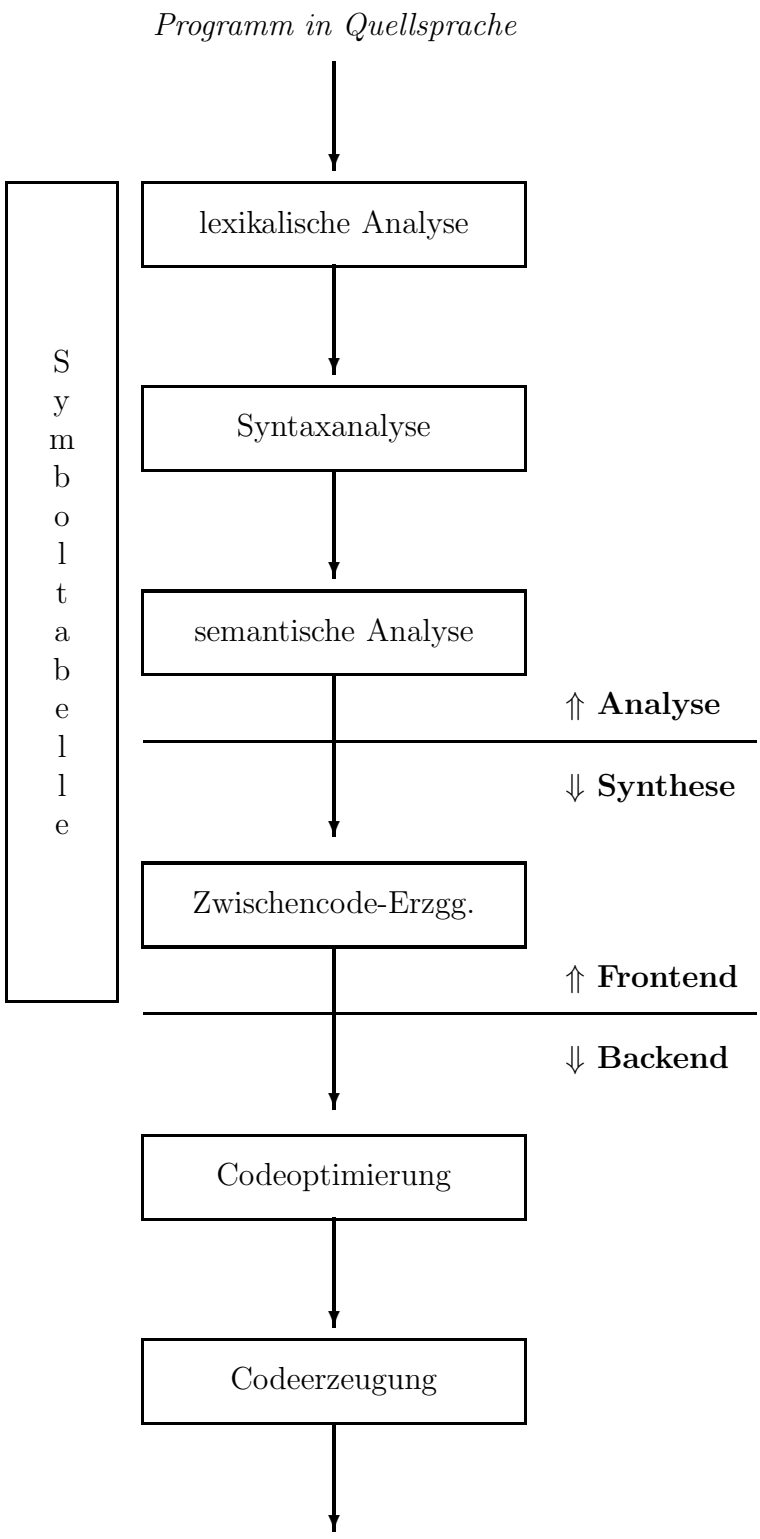


Abbildung 1: Phasen eines Compilers

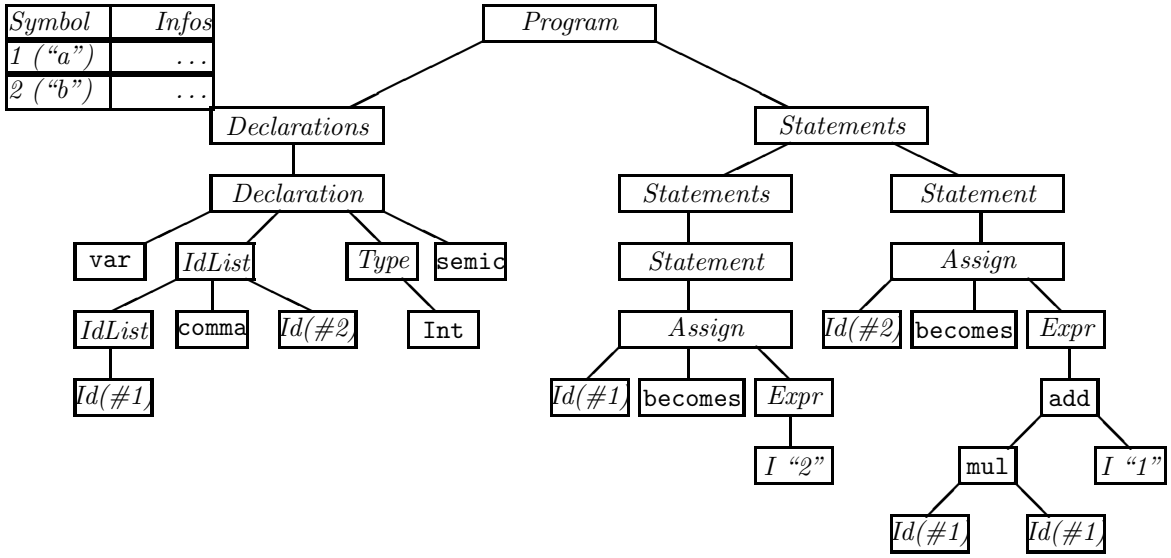


Abbildung 2: Syntaxbaum

```
var a,b: int;

a := 2;
b := a*a+1;
```

Der Scanner (lexikalische Analyse) erkennt aus dieser Eingabe (mit den hier dargestellten Zeilenwechseln) zunächst die folgenden lexikalischen Einheiten, die sogenannten Symbole:

```
id(‘‘var’’) sep id(‘‘a’’) comma id(‘‘b’’) colon sep id(‘‘int’’) sem sep sep
id(‘‘a’’) sep becomes sep num(‘‘2’’) sem sep
id(‘‘b’’) sep becomes sep id(‘‘a’’) mul id(‘‘a’’) add num(‘‘1’’) sem sep
```

Außerdem trägt der Scanner in die Symboltabelle die Bezeichner `a` und `b` etwa an den Positionen 1 und 2 ein und kodiert die Bezeichner durch Verweise auf die Symboltabelle. Reservierte Symbole und eventuelle Compiler-Direktiven werden erkannt sowie Leerzeichen und Kommentare eliminiert. Der für diese Aufgaben zuständige Teil des Scanners wird *Sieber* genannt. Als Ausgabe des Scanners ergibt sich etwa die Symbolfolge:

```
var id(1) comma id(2) colon int sem id(1) becomes num(‘‘2’’) sem
id(2) becomes id(1) mul id(1) add num(‘‘1’’) sem
```

Der Parser (Syntaxanalyse) analysiert die syntaktische Struktur, indem er Deklarationen und Anweisungen erkennt. Er erzeugt etwa den in Abbildung 2 gezeigten Syntaxbaum.

Die semantische Analyse überprüft die statischen semantischen Eigenschaften des Programms wie Typkorrektheit, Deklariertheit der Bezeichner. Gegebenenfalls werden verschiedene Analysen durchgeführt, etwa eine Abhängigkeitsanalyse oder eine Typinferenz.

Im Beispiel liefert der *Declarations*-Unterbaum Informationen über deklarierte Bezeichner und deren Typ:

$$\begin{aligned} id(1) &\rightarrow (var, int) \\ id(2) &\rightarrow (var, int) \end{aligned}$$

In *Statements* wird etwa jeweils überprüft, ob in Wertzuweisungen links ein Variablenbezeichner und rechts ein Ausdruck steht, dessen Typ dem des Variablenbezeichners entspricht. Hier ist gegebenenfalls auf Typüberladungen (overloading) zu achten.

Auf dem attributierten Syntaxbaum sind eventuell bereits erste optimierende Transformationen möglich, beispielsweise:

- Konstantenpropagation, Konstantenfaltung
  im Beispiel: In der Wertzuweisung `b := a*a+1` hat `a` den konstanten Wert 2, d.h. der Ausdruck `a*a+1` kann schon zur Übersetzungszeit zu 5 ausgewertet werden.
- Herausziehen von schleifeninvarianten Berechnungen aus Schleifen
- Elimination redundanter Berechnungen
- Elimination von nicht-erreichbarem Code

Bei der Zwischencodererzeugung werden den Bezeichnern Speicher- oder Registeradressen zugeordnet und es werden die Anweisungen in Maschinenbefehlssequenzen übersetzt. Für eine einfache Registermaschine mit den Befehlen

```
LOAD   reg adr
STORE  adr reg
LOADI  reg int
ADDI   reg int
MUL    reg adr
...
```

und der Adresszuordnung $id(1) \mapsto 0, id(2) \mapsto 1$ könnte der Zwischencode dann etwa wie folgt aussehen:

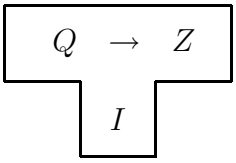
```
LOADI R1 2  ; Zahl 2
STORE 0  R1 ; a := 2
LOAD  R1 0  ;         unnoetig! (Optimierung)
MUL   R1 0  ; a*a
ADDI  R1 1  ; a*a+1   eventuell billiger: INC R1
STORE 1  R1 ; b := a*a+1
```

◁

Compiler bezeichnet man je nach Anzahl der Läufe, die das Programm durch das Quellprogramm vornimmt, als *One-Pass-* bzw. *n-Pass-Compiler*.

1.3 Bootstrapping

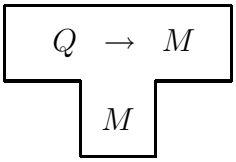
Unter Bootstrapping versteht man die Benutzung eines Compilers zur eigenen Übersetzung, also selbstreferenziell. Voraussetzung ist, dass der Compiler in der Quellsprache geschrieben ist (Implementierungssprache = Quellsprache). Bootstrapping dient zum einen der Erweiterung von eingeschränkten Compilern für Kernsprachen auf den vollen Sprachumfang, zum anderen dem Erzeugen effizienterer Compiler oder neuer Compilerversionen. Zur Darstellung werden sog. T-Diagramme benutzt (Q=Quellsprache, I = Implementierungssprache, Z = Zielsprache).



Wir zeigen den Einsatz von Bootstrapping in zwei typischen Anwendungsfällen.

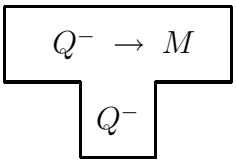
1.3.1 Hochziehen eines Compilers

Ziel ist ein ausführbarer Compiler mit I = Z = M (Maschinencode), der die Quellsprache in Maschinencode übersetzt:



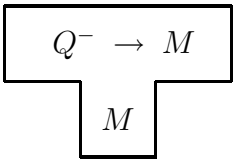
Die folgenden Zwischencompiler müssen zur Vorbereitung des Bootstrapping-Schritts entwickelt werden:

1. Compiler für eine Kernsprache Q^-, der auch in Q^- geschrieben ist

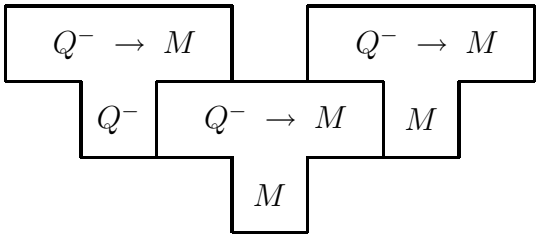
   

   Q^- ist eine einfache Teilsprache von Q, in der aber der Compiler ausgedrückt werden kann.

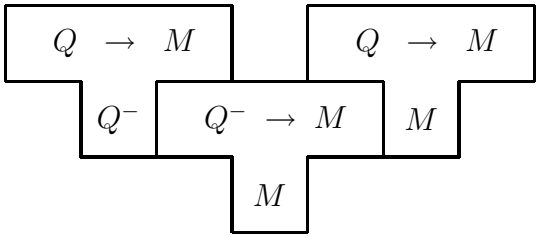
2. Ein erster Compiler von Q^- nach M wird von Hand oder mithilfe von 1) durch Übertragung in eine Sprache mit existierendem Compiler auf M (beispielsweise C) erzeugt werden.

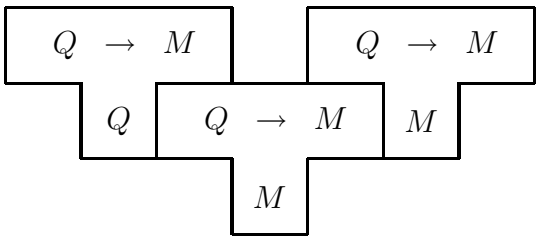
3. Das Ergebnis von 1 wird nun mit 2 übersetzt, es ergibt den ersten vollwertigen Q^--Compiler.



4. Der erste Q^--Compiler wird zu einem Q-Compiler in Implementierungssprache Q^- erweitert und mit 3 übersetzt.



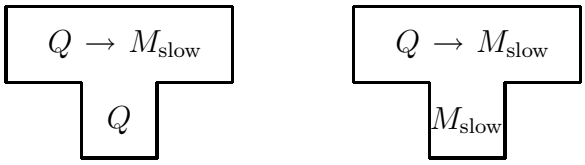
5. Das Ergebnis wird zu einem Q-Compiler in der Implementierungssprache Q erweitert und mit der vorigen Version übersetzt.



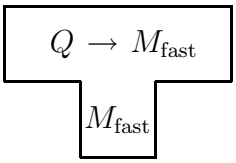
6. Der Compiler kann nun weiterentwickelt und anschließend mit der vorigen Version übersetzt werden

1.3.2 Verbesserung der Effizienz

Hier soll ein existierender langsamer Compiler mit einer Optimierungsphase versehen und beschleunigt werden. Der Compiler sei in der Quellsprache implementiert und liege auch in ausführbarer Form vor (s.o.).

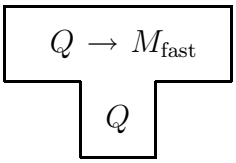


Allerdings sind sowohl die Compilation als auch der erzeugte Code langsam, was durch den Index slow angezeigt wird. Ziel ist die Entwicklung eines schnellen Compilers, der auch schnellen Code erzeugt:

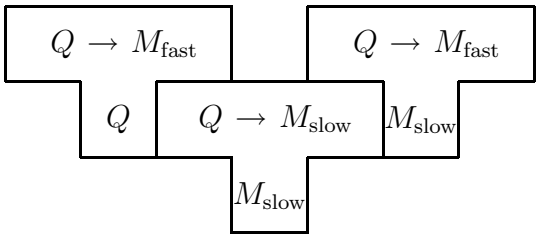


Hierzu ist folgende Vorgehensweise möglich:

1. Durch Modifikation des in der Quellsprache geschriebenen langsamen Compilers wird ein optimierender Compiler entwickelt, d.h. ein Compiler, der besseren Objectcode erzeugt:

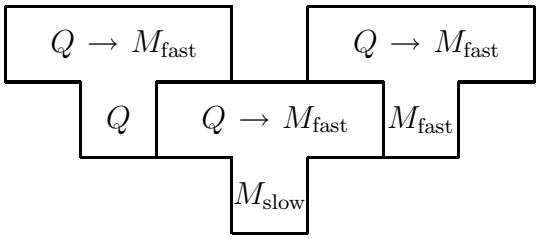


2. Dieser Compiler wird mit dem existierenden langsamen Compiler übersetzt.



   Dies ergibt einen langsamen Compiler, der schnellen Objektcode erzeugt.

3. Zum Abschluss wird mit dem erzeugten Compiler nochmals übersetzt, um einen schnellen Compiler zu bekommen.



   Dies führt zu einem schnellen Compiler, der zudem schnellen Code erzeugt.

Bootstrapping kann stufenweise durchgeführt werden, um Compiler immer weiter zu verbessern.

1.4 Vom Interpreter zum Compiler-Compiler mit partiellen Auswertern

Ein *Interpreter* erhält als Eingabe ein Programm und seine Eingabe. Das Programm wird für die spezielle Eingabe ausgeführt („interpretiert“), um die Ausgabe zu ermitteln. Ein Interpreter unterscheidet sich von einem Compiler also dadurch, dass ein Programm und seine Eingabe zur gleichen Zeit verarbeitet werden. Die Entwicklung eines Interpreters ist weniger aufwendig als die eines Compilers. Er besteht aus einem Analyse- und einem Simulationsteil. Beim Aufruf eines Interpreters werden im wesentlichen nur diejenigen Programmteile analysiert, die für die aktuelle Eingabe benötigt werden, und die entsprechenden Anweisungen oder Ausdrücke werden ausgeführt.

$$intp :: Q \times Input \longrightarrow Output$$

Da das Programm für verschiedene Eingaben unverändert bleibt, scheint eine Vorverarbeitung sinnvoll. Eine solche kann von einem sogenannten *partiellen Auswerter* vorgenommen werden.
Ein *partieller Auswerter* erhält als Eingabe ein Programm und einen Teil der Eingabe des Programms, den sogenannten *statischen Input*. Er produziert als Ausgabe ein für die gegebene Eingabe spezialisiertes Programm, in dem alle Anweisungen und Ausdrücke, die nur von dem bereits gegebenen Input abhängen, bereits ausgewertet wurden. Wesentlich ist hier eine Aufteilung der Eingabe in einen statischen und einen dynamischen Teil:

$$Input = Input_{st} \times Input_{d}$$

Die statische Eingabe stellt den bereits vorhandenen oder unveränderlichen Teil der Eingabe dar, während die dynamische Eingabe dem noch nicht bekannten oder veränderlichen Teil entspricht. Der partielle Auswerter erhält als Eingabe ein Programm und seine statische Eingabe und liefert ein spezialisiertes Programm:

$$pa :: \left\{ \begin{array}{ccccc} Q & \times & Input_{st} & \longrightarrow & Q \\ (p & , & inp_{st}) & \mapsto & p_{\text{spez}} \end{array} \right.$$

Dabei muss gelten:

$$\mathcal{M}_Q[\![p]\!](inp_{st}, inp_d) = \mathcal{M}_Q[\![p_{\text{spez}}]\!]inp_d = \mathcal{M}_Q[\![pa(p, inp_{st})]\!]inp_d.$$

1.4.1 Übersetzung durch partielle Auswertung

Ein Interpreter ist ein Programm, dessen Eingabe aus einem statischen Anteil, dem Programm, und einem dynamischen Anteil, der Eingabe des Programms, besteht. Sei I die Implementierungssprache des Interpreters, d.h. die Programmiersprache, in der der Interpreter geschrieben ist. Mit einem partiellen Auswerter für I-Programme können wir nun einen Q-Interpreter für ein gegebenes Q-Programm spezialisieren: $pa(intp, p) = intp_{\text{spez}}$. Es gilt:

$$\begin{array}{rcll} \mathcal{M}_I[\![intp_{\text{spez}}]\!]inp & = & \mathcal{M}_I[\![pa(intp, p)]\!]inp & \\ & = & \mathcal{M}_I[\![intp]\!](p, inp) & \text{(Korrektheit des partiellen Auswerters)} \\ & = & \mathcal{M}_Q[\![p]\!]inp & \text{(Korrektheit des Interpreters)} \end{array}$$

Damit folgt, dass der spezialisierte Interpreter $intp_{\text{spez}}$ ein zu dem Q-Programm p äquivalentes I-Programm ist. Die partielle Auswertung des Interpreters entspricht somit einem Compilationsvorgang von Q nach I.

1.4.2 Erzeugung eines Compilers

Da sich die Eingabe eines partiellen Auswerters ebenfalls in natürlicher Weise in einen statischen (das Programm bzw. im speziellen Kontext der Interpreter) und einen dynamischen Teil (der statische Input bzw. das zu interpretierende Programm) zerlegen lässt, ist auch hier eine partielle Auswertung sinnvoll. Hierzu muss der partielle Auswerter auf sich selbst angewendet werden können, also in der Lage sein, den eigenen Programmcode zu verarbeiten und zu spezialisieren. Daher muss der partielle Auswerter in der Sprache geschrieben sein, deren Programme er spezialisiert.
Anwendung des partiellen Auswerters auf sich selbst und einen Interpreter liefert einen spezialisierten partiellen Auswerter, der einem Q nach I-Compiler entspricht: $pa(pa, intp) = pa_{\text{spez}} : Q \to I$. Es gilt:

$$\begin{aligned}
\mathcal{M}_I[\![\mathcal{M}_I[\![pa_{\text{spez}}]\!]p]\!]inp &= \mathcal{M}_I[\![\mathcal{M}_I[\![pa(pa, intp)]\!]p]\!]inp \\
&= \mathcal{M}_I[\![\mathcal{M}_I[\![pa]\!](intp, p)]\!]inp \\
&= \mathcal{M}_I[\![intp]\!](p, inp) \\
&= \mathcal{M}_Q[\![p]\!]inp
\end{aligned}$$

Durch die Selbstanwendung des partiellen Auswerters gelingt es damit, einen Interpreter in einen Compiler umzuwandeln. Den Prozess der Selbstanwendung kann man noch weiter fortsetzen.

1.4.3 Erzeugung eines Compiler-Compilers

Ein Compiler-Compiler ist ein Programm, dass aus einer semantischen Beschreibung einer Programmiersprache, die etwa in Form eines Interpreters gegeben sein kann, automatisch einen Compiler erzeugt. Wir zeigen hier kurz, wie man prinzipiell einen Compiler-Compiler durch Anwendung eines partiellen Auswerters auf sich selbst herleiten kann. Wählt man als statischen Input des partiell auszuwertenden Auswerters den partiellen Auswerter selbst, so führt dies zu einem spezialisierten partiellen Auswerter, der bei Eingabe eines Interpreters einen Compiler als Ausgabe liefert. Sei $pa(pa, pa) = cc$. Was leistet das spezialisierte Programm cc? Bei Eingabe eines Interpreters wird gemäß dem vorherigen Abschnitt ein Compiler erzeugt, denn:

$$cc(intp) = pa(pa, intp) = pa_{\text{spez}} : Q \to I$$

Das Problem dabei ist, dass der partielle Auswerter I-Programme verarbeiten und gleichzeitig in I implementiert sein muss, ebenso wie der Interpreter. Man muss also entweder bereits einen funktionierenden I-Compiler besitzen, oder I muss sowohl ausführbar als auch für die Implementierung geeignet sein. Die meisten existierenden partiellen Auswerter erfüllen diese Anforderungen nicht. Insbesondere die Selbstapplikation ist schwierig zu realisieren.

1.5 Aufbau der Vorlesung

Die Vorlesung folgt dem Phasenaufbau von Compilern, der bereits in Abbildung 1 skizziert wurde. Im Zentrum stehen dabei die Wirkungsweise und grundlegende Techniken zur Konstruktion von Compilern. Vornehmlich in den Übungen wird die funktionale Sprache Haskell zur Entwicklung von Compilerteilen eingesetzt. Die funktionale Spezifikation ist kompakt und zeigt die grundsätzlichen Transformationen:

```
compiler  ::  String              ->  MachineCode
compiler  =   codeGen . analyser . parser . scanner

scanner   ::  String              ->  [Token]
parser    ::  [Token]             ->  AbstractSyntaxTree
analyser  ::  AbstractSyntaxTree  ->  DecoratedTree
codeGen   ::  DecoratedTree       ->  MachineCode
```

Die bedarfgesteuerte Auswertung von Haskell hat zudem den Vorteil, dass trotz unabhängiger Entwicklung und Programmierung der einzelnen Compilerteile automatisch eine verzahnte Abarbeitung der verschiedenen Compilerphasen erfolgt.