Übersetzerbau

Prof. Dr. Michael Hanus

WS 2011/12

Vorwort

Die Übersetzung von Programmiersprachen ist eine wohldefinierte aber dennoch komplexe Aufgabe. Zur Beherrschung dieser Komplexität wurde eine Zerlegung in einzelne Übersetzungsaufgaben entwickelt, die heute in dieser oder ähnlicher Form in den meisten Übersetzern verwendet wird. In dieser Vorlesung werden die einzelnen Übersetzungsaufgaben und die bekannten Lösungsansätze dazu vorgestellt. Im Einzelnen werden behandelt:

- Programmiersprachen, Interpreter, Übersetzer
- Lexikalische Analyse
- Syntaktische Analyse
- Semantische Analyse
- Codeerzeugung

Dieses Skript ist eine überarbeitete Fassung einer Mitschrift, die ursprünglich von Klaas Ole Kürtz in LaTeX gesetzt wurde. Ich danke Herrn Kürtz für die LaTeX-Vorlage und seine Zustimmung, diese weiterzuverwenden.
Noch ein wichtiger Hinweis: Dieses Skript soll nur einen Überblick über das geben, was in der Vorlesung gemacht wird. Es ersetzt nicht die Teilnahme an der Vorlesung, die zum Verständnis der Konzepte und Techniken des Übersetzbaus wichtig ist. Ebenso wird für ein vertieftes Selbststudium empfohlen, sich die in der Vorlesung angegebenen Lehrbücher anzuschauen.

Kiel, Oktober 2011 Michael Hanus

P.S.: Wer in diesem Skript keine Fehler findet, hat sehr unaufmerksam gelesen. Ich bin für alle Hinweise auf Fehler dankbar, die mir persönlich, schriftlich oder per e-mail mitgeteilt werden.

Inhaltsverzeichnis

1 Einführung

Es existieren zahlreiche Lehrbücher zu dem Thema der Vorlesung, allerdings keines, was exakt die Vorlesungsinhalte abdeckt. Aus diesem Grund sind hier nur einige empfehlenswerte Bücher aufgeführt:

- A. Aho, R. Sethi, J.Ullman: Compilerbau (Teil 1 + 2), Oldenbourg, 1999
- A. Aho, M. Lam, R. Sethi, J.Ullman: Compilers: principles, techniques, and tools (2nd ed.), Pearson Education, 2007
- A. Appel: Modern compiler implementation in ML, Cambridge UP, 1997
- R.H. Güting, M. Erwig: Übersetzerbau, Springer 1999
- W. Waite, G. Goos: Compiler Construction, Springer, 1984
- R. Wilhelm, D. Maurer: Übersetzerbau: Theorie, Konstruktion, Generierung, Springer, 1992
- N. Wirth: Grundlagen und Techniken des Compilerbaus, Oldenbourg, 2008

Das Buch von Andrew Appel behandelt dabei am besten die Vorlesungsinhalte.

1.1 Was ist ein Übersetzer?

Ein **Übersetzer/Compiler** ist selbst ein Programm, das jedem PS-Programm (Quellprogramm) ein semantisch äquivalentes MS-Programm (Ziel-Programm) zuordnet. Hierbei ist die zu übersetzende Programmiersprache PS meist eine **höhere Programmiersprache**:

- **imperative** Programmiersprachen (Inhalt dieser Vorlesung): Wertzuweisung, Kontroll- und Datenstrukturen, Module, Klassen
- **deklarative** Programmiersprachen: funktionale, logische Sprachen
- **nebenläufige** Programmiersprachen: parallele, verteilte Prozesse

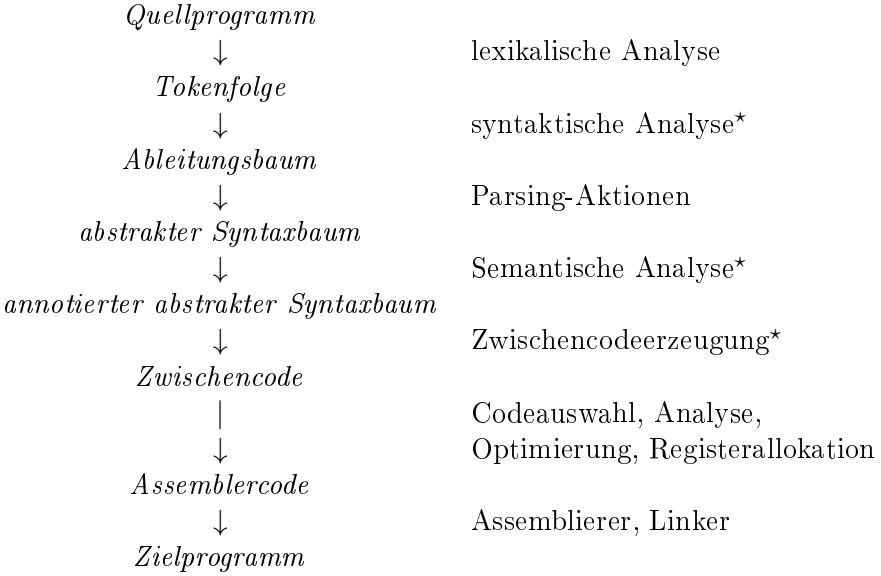
Die Maschinensprache MS (im allgemeinen für einen von-Neumann-Rechner) ist ausgelegt z.B. auf:

- **CISC**-Rechner (complex instruction set computers) stellen relativ viele Befehle zur Verfügung, wobei die einzelnen Befehle komplex sein können
- **RISC**-Architekturen (reduced instruction set computers) haben einen kleineren, einfacheren Befehlssatz, dafür werden die Befehle aber in der Regel schneller abgearbeitet (u.a. durch Pipelining)
- **Parallel-Rechner**, beispielsweise SIMD (single instruction, multiple data) oder MIMD (multiple instruction, multiple data)

Die **Aufgaben** eines Übersetzers bestehen aus dem Erkennen der Syntax eines Programms und dem Erkennen der Semantik (Übersetzung in semantisch äquivalentes Programm). Übersetzer sind komplexe Programme, die hohe **Korrektheitsanforderungen** erfüllen müssen (beispielsweise bei sicherheitskritischen Systemen). Daher werden Übersetzer realisiert durch eine geeignete Struktur und geeignete Spezifikationsmethoden.

1.2 Struktur von Übersetzern

Die typische Struktur eines Übersetzers sieht wie folgt aus:



Phasenübergreifend existiert bei * die **Symboltabelle**, eine zentrale Datenstruktur des Compilers. Die einzelnen **Phasen** des Übersetzungsvorgangs beinhalten folgende Aufgaben:

- **lexikalische Analyse**: Zweck ist die Erkennung von zusammengehörigen Symbolen, Kommentaren etc. Werkzeug hierfür sind: reguläre Ausdrücke ($\rightarrow$ endliche Automaten)
- **syntaktische Analyse**: dient der Erkennung der Programmstruktur, Werkzeug sind kontextfreie Grammatiken ($\rightarrow$ Kellerautomat)
- **Parsing-Aktionen**: zur Vereinfachung der Programmstruktur (alles entfernen, was semantisch egal ist)
- **semantische Analyse**: Erkennen von Kontextabhängigkeiten (Zuordnung zwischen Definition und Verwendung, Typanalyse)

  *Ab hier wird ein Programm als **korrekt** akzeptiert!*
- **Zwischencodeerzeugung**: vereinfachter, maschinenunabhängiger Code (Portabilität!)

  *Bis hier ist die Prozedur **maschinenunabhängig!***
- **Codeerzeugung**: Erzeugung konkreten Maschinencodes
- **Assembler:** Auflösung von Adressen

Als **Frontend** bezeichnet man den maschinenunabhängigen Teil des Compilers bis zum Zwischencode, alles weitere wird als **Backend** bezeichnet. Häufig existiert ein Übersetzer in Form eines Frontends mit mehreren Backends für unterschiedliche Architekturen.

Die Begriffe **One-Pass-** oder **n-Pass-Compiler** geben an, wie oft der Compiler über den Quellcode läuft (bzw. heute über den Ableitungsbaum), früher wurde One-Pass bevorzugt, was durchaus Einfluss auf die Syntax der Sprache hatte (z.B. bei Pascal mittels Vorwärtsreferenzen `FORWARD`).

Die Ziele der Vorlesung sind:

- Grundlegende Techniken/Methoden des Übersetzerbaus kennenlernen
- Anwendung von Theorie (z.B. Theoretische Informatik) in der Praxis
- Übersetzer als Beispiel eines guten Softwareentwurfs

Die Implementierungssprache in dieser Vorlesung ist die funktionale Sprache Haskell – dabei wird der Haskell98-Standard verwendet, denn funktionale Sprachen sind besonders gut für Übersetzungsaufgaben geeignet. Als Haskell-Implementierung kann z.B. der Interpreter Hugs oder die interaktive Umgebung Ghci des Glasgow Haskell Compilers benutzt werden, die für zahlreiche Systeme frei verfügbar sind (siehe auch die Einführung zu Haskell im Anhang B).

1.3 Eine einfache Beispielsprache: Simple

Als Beispiel für eine einfach zu übersetzende Beispielsprache verwenden wir Simple, die nur Zuweisungen, einfache Ausdrücke (mit Seiteneffekten) und Anweisungssequenzen erlaubt und definiert ist durch eine kontextfreie Grammatik (hierbei wird die konkrete Syntax von Zahlen und Bezeichner weggelassen, d.h. diese sind Terminalsymbole für diese Grammatik):

Regel			Name
Stm	$\longrightarrow$	*Stm*; *Stm*	CompoundStm
Stm	$\longrightarrow$	*Id* `:=` *Exp*	AssignStm
Stm	$\longrightarrow$	`print(`*ExpList*`)`	PrintStm
Exp	$\longrightarrow$	*Num*	NumExp
Exp	$\longrightarrow$	*Id*	IdExp
Exp	$\longrightarrow$	*Exp BinOp Exp*	OpExp
Exp	$\longrightarrow$	`(`*Stm*`,`*Exp*`)`	EseqExp
ExpList	$\longrightarrow$	*Exp*`,` *Explist*	PairExpList
ExpList	$\longrightarrow$	*Exp*	LastExpList
BinOp	$\longrightarrow$	`+`	Plus
BinOp	$\longrightarrow$	`*`	Times
BinOp	$\longrightarrow$	`-`	Minus
BinOp	$\longrightarrow$	`/`	Div

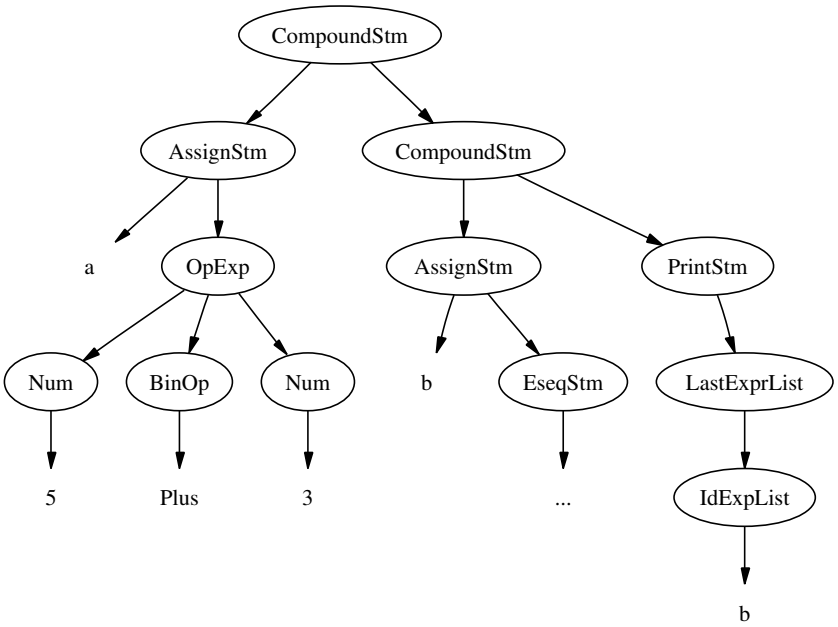
Die Semantik von Simple sollte intuitiv klar sein.
BEISPIEL: Folgendes ist ein einfaches Simple-Programm:

```
a := 5+3; b := (print(a, a-1), 10*a); print(b)
```

Ein Ablauf dieses Programms erzeugt die Ausgabe

```
8 7
80
```

Wenn dieses Programm in einem Übersetzer oder Interpreter verarbeitet werden soll, wird dessen Struktur als Ableitungsbaum dargestellt. In unserem Fall könnte dieser wie folgt aussehen (wobei wir in den inneren Knoten den Regelbezeichner schreiben):



Diese Baumstrukturen kann man in einer funktionalen Programmiersprache wie Haskell mittels folgender Datentypen darstellen:

```
type Id = String

data BinOp = Plus | Minus | Times | Div
  deriving (Eq,Show)

data Stm = CompoundStm Stm Stm
         | AssignStm Id Exp
         | PrintStm [Exp]
  deriving (Eq,Show)

data Exp = NumExp Int
         | IdExp Id
         | OpExp Exp BinOp Exp
         | EseqExp Stm Exp
  deriving (Eq,Show)
```

Das obige Beispielprogramm bzw. sein Ableitungsbaum könnte dann durch den folgenden Ausdruck/Datenterm dargestellt werden:

```
CompoundStm
  (AssignStm "a" (OpExp (NumExp 5) Plus (NumExp 3)))
```

```
(CompoundStm
   (AssignStm
      "b"
      (EseqExp (PrintStm [IdExp "a",
                          OpExp (IdExp "a") Minus (NumExp 1)])
               (OpExp (NumExp 10) Times (IdExp "a"))))
   (PrintStm [IdExp "b"]))
```

Damit ist die Funktionalität der lexikalischen und syntaktischen Analyse leicht erklärbar: es ist eine Funktion vom Typ

```
String → Stm
```

Bevor wir zeigen, wie man eine solche Funktion implementieren kann, diskutieren wir die Begriffe von Programmiersprache, Interpreter und Übersetzer und deren Zusammenhänge.

2 Programmiersprachen, Interpreter, Übersetzer

2.1 Interpreter

Programmiersprachen basieren auf einem Zeichenvorrat wie ASCII, wir definieren daher A als die Menge aller Zeichenketten über diesem Vorrat, z.B. $A = \text{ASCII}^*$ oder $A = \text{UNICODE}^*$.

DEFINITION: Eine *Programmiersprache* L ist eine partielle Funktion[1] $L\colon A \dashrightarrow (A^* \dashrightarrow A)$. Der Definitionsbereich einer Programmiersprache ist die *Menge der syntaktisch zulässigen L-Programme* L-Prog. Sei $l \in L$-Prog. Dann ist $L(l)$ (im folgenden einfach Ll geschrieben) eine Funktion $Ll\colon A^* \dashrightarrow A$, eine *Ein-/Ausgabefunktion*, die *Semantik* von L.

Die syntaktische bzw. semantische Analyse dient dann der Erkennung, ob $l \in L$-Prog gilt. Üblich sind folgende Forderungen an eine Programmiersprache:

1. L-Prog $\subseteq A$ ist eine entscheidbare Teilmenge (d.h. die syntaktische und die semantische Analyse ist berechenbar)
2. $Ll\colon A^* \dashrightarrow A$ ist partiell rekursiv (d.h. die Semantik von l implementierbar)

(Falls L eine Datenbankanfragesprache (z.B. $L = \text{SQL}$), dann muss meist auch Ll entscheidbar sein für alle $l \in L$-Prog).

Beachte: $l \in A$ alleine reine Syntax ohne Bedeutung, d.h. $L_1 l \neq L_2 l$ für verschiedene Sprachen L_1, L_2.

DEFINITION: Sei $l \in L$-Prog. Dann sei der Definitionsbereich

$$L\text{-Eingabe}(l) = \{(x_1, \ldots, x_n) \in A^* \mid Ll(x_1, \ldots, x_n) \in A\}$$

Weiterhin ist die Ausgabe definiert durch

$$\begin{aligned} L\text{-Ausgabe}(l) &= Ll(L\text{-Eingabe}(l)) \\ &= \{x \in A \mid \exists\, (x_1, \ldots, x_n) \in A^*\colon Ll(x_1, \ldots, x_n) = x\} \end{aligned}$$

Interpreter sind selbst Programme, die in einer Sprache I geschrieben sind:
DEFINITION: Ein *Interpreter* i für eine Sprache L, geschrieben in der Sprache I, ist ein Programm $i \in I$-Prog, das für alle $l \in L$-Prog und alle $(x_1, \ldots, x_n) \in L$-Eingabe(l) folgendes erfüllt:

$$Ii(l, x_1, \ldots, x_n) = Ll(x_1, \ldots, x_n)$$

[1] partielle Funktionen sind hier notiert mit „$\dashrightarrow$“

Als Symbol[2] bezeichnet $[{}^{L}_{I}]$ die Menge aller Interpreter für L geschrieben in I. Somit haben Interpreter L-Programme und deren Eingabe als Eingabe und berechnen das Verhalten gemäß der L-Semantik.

Die obige Gleichung ist strikt, d.h. es ist nichts ausgesagt über die Fälle, bei denen Ll undefiniert oder Eingaben unzulässig sind. Im Beispiel aus Kapitel 1.3 ist $L = \mathsf{Simple}$ und $I = \mathsf{Haskell}$ (siehe Übung).

Generelle **Eigenschaften von Interpretern**:

- Funktionale Sprachen sind gut geeignet zur Implementierung von Interpretern
- Interpreter sind einfacher als Übersetzer, damit ist auch die Korrektheit einfacher nachweisbar (wir benötigen keine Zielsprache und deren Semantik!)
- Interpreter meistens langsam, dafür aber geringe Übersetzungs-/Startzeit – daher gut zum Testen

Wünschenswert sind daher für eine Programmiersprache sowohl Interpreter als auch Übersetzer.

2.2 Übersetzer

DEFINITION: Gegeben sei eine Quellsprache Q und eine Zielsprache Z. Ein C-Programm c heißt *Übersetzer von Q nach Z*, falls:

1. C-Eingabe$(c) = Q$-Prog
2. C-Ausgabe$(c) \subseteq Z$-Prog
3. Für alle $q \in Q$-Prog, $(x_1, \ldots, x_n) \in Q$-Eingabe(q)

$$Qq(x_1, \ldots, x_n) = Z(Cc(q))(x_1, \ldots, x_n)$$

Als Symbol[3] wird hier $\left[{}^{Q \to Z}_{C}\right]$ verwendet (auch „T-Diagramm“ genannt).

Beispiele: $Q = \mathsf{Simple}$, $Z = \mathsf{C}$ (oder Maschinencode), $C = \mathsf{Haskell}$.
Oder ein C-Compiler: $Q = \mathsf{C}$, $Z = \mathsf{8086}$ und $C = \mathsf{8086}$.
Es ist auch eine **Komposition** möglich, d.h. beispielsweise

$$\left[\begin{array}{c|c} \mathsf{Simple} \to \mathsf{C} & \mathsf{C} \to \mathsf{8086} \\ \mathsf{Haskell} & \mathsf{8086} \end{array}\right]$$

Zum **Unterschied** zwischen Interpretern und Übersetzern:

[2] Original-Symbol siehe Anhang A
[3] Original-Symbol siehe Anhang A

- Der Übersetzer „kennt“ das Quellprogramm, beispielsweise ist einem Simple-Übersetzer die Menge aller Identifier bekannt und kann ein Feld statt einer Liste für die Umgebung verwenden, er übersetzt dann die Zuweisung durch indizierte Feldmodifikation (konstante Laufzeit!).
- Ein Compiler hat i.allg. durch die Kenntnis des Quellprogramms die Möglichkeit, viele Optimierungen im späteren Programmablauf vorzunehmen.

2.3 Konstruktion von Übersetzern aus Interpretern

Eine Methode zur Konstruktion von Compilern aus gegebenen Interpretern ist die **partielle Auswertung** des Interpreters. Damit hätte man Interpreter und Compiler für eine zur Verfügung, wobei man nur die Korrektheitsnachweis für den Interpreter führen muss, wenn der Übersetzer per Konstruktion korrekt ist.

Gegeben sei also $i \in [{}^{L}_{I}]$ (d.h. ein L-Interpreter in I). Gesucht ist $c \in [{}^{L}_{C}\!\!\rightarrow\!{}^{I}]$, d.h. ein L-Übersetzer nach I, der in C geschrieben ist.

Idee: da i ein Interpreter ist, gilt:

$$Ii\colon (l, x_1, \ldots, x_n) \mapsto Ll(x_1, \ldots, x_n)$$

Nun transformieren wir für jedes $l \in L$-Prog das Programm i in ein Programm $\mathrm{Restprog}(i, l) \in I$-Prog mit der Eigenschaft

$$I(\mathrm{Restprog}(i, l))(x_1, \ldots, x_n) = Ii(l, x_1, \ldots, x_n)$$

Transformationsmethode: Setze in i konkreten Parameter l ein und werte dann i überall da aus, wo es möglich ist (d.h. wo es von l abhängt), d.h. man wertet das Programm i an einigen Stellen aus (daher nennt man dies **partielle Auswertung** von i angewendet auf l). Dann ist

$$Cc\colon l \mapsto \mathrm{Restprog}(i, l)$$

ein Übersetzer (wobei der partielle Auswerter c in der Sprache C geschrieben ist).

Beispiele für mögliche partielle Auswertungstransformationen:

- Ausrechnen von bekannten Werten: `3+2` $\rightarrow$ 5, `3==3` $\rightarrow$ `True`
- Bedingungen vereinfachen:

$$\texttt{if True then } e_1 \texttt{ else } e_2 \;\rightarrow\; e_1$$
$$\texttt{if False then } e_1 \texttt{ else } e_2 \;\rightarrow\; e_2$$

- „Auffalten“: Falls `f` $\texttt{x}_1$ `...`$\texttt{x}_n$ `= b` ist:

$$\texttt{f e}_1 \texttt{ ...e}_n \rightarrow \texttt{b[x}_1 \mapsto \texttt{ e}_1\texttt{, ..., x}_n \mapsto \texttt{ e}_n\texttt{]}$$

- Spezialisierung:

$$\texttt{f c}_1 \texttt{ ...c}_k \texttt{ e}_1 \texttt{ ...e}_n \rightarrow \texttt{f-c1-...-ck e}_1 \texttt{ ...e}_n$$

  mit einer neuen Funktion `f-c1-...-ck` $\texttt{x}_1$ `...`$\texttt{x}_n$ `= b[`$\texttt{y}_1 \mapsto \texttt{c}_1$`, ...,` $\texttt{y}_k \mapsto \texttt{c}_k$`]` falls `f` definiert ist durch `f` $\texttt{y}_1$ `...`$\texttt{y}_k$ $\texttt{x}_1$ `...`$\texttt{x}_n$ `= b`.

Beispiel: Seien folgende Funktionen definiert:

```
f x y = if x > 0 then 3*x+y else y
g z = f 4 z
```

Dann ergibt eine partielle Auswertung (hier durch Auffalten):

```
g z = if 4 > 0 then 3*4+z else z
    = if True then 3*4+z else z
    = 3*4+z
    = 12+z
```

Definition: Ein Programm $PEVAL \in E$-Prog
heißt *partieller Auswerter* für I-Programme, falls

1. E-Eingabe$(PEVAL) = I$-Prog $\times A$
2. $EPEVAL(i, x_1) = i_1 \in I$-Prog
3. $Ii_1(x_2, \ldots, x_n) = Ii(x_1, x_2, \ldots, x_n)$ für alle $(x_2, \ldots, x_n)$ mit $(x_1, \ldots, x_n) \in I$-Eingabe(i)

Damit gilt: $PEVAL$ angesetzt auf einen Interpreter $i \in [{}^{L}_{I}]$ und ein Quellprogramm $l \in l$-Prog liefert ein Zielprogramm in I:

$$\begin{aligned} I(EPEVAL(i,l))(x_1, \ldots, x_n) &= Ii(l, x_1, \ldots, x_n) \\ &= Ll(x_1, \ldots, x_n) \end{aligned}$$

Somit ist $EPEVAL(i)$ ein Übersetzer von L nach I in E.

Tatsächlich wurde eine Prolog-Implementierung auf dieser Basis erstellt (Kahn/Carlsson 1984) mit $L = Prolog$ und $E = I = Lisp$, d.h. das Ergebnis ist $\left[{}^{\text{Prolog}\rightarrow\text{Lisp}}_{\text{Lisp}}\right]$. Allerdings ergab dies einen sehr langsamen Übersetzer (etwa Faktor 100 langsamer als ein handgeschriebener Übersetzer).

Verbesserung: Falls $E = I$ ist (d.h. der partielle Auswerter ist in gleicher Sprache wie der Interpreter geschrieben), so setze *PEVAL* auf sich selbst und den Interpreter i an, um einen schnelleren Übersetzer zu erhalten:

$$c := IPEVAL(PEVAL, i)$$

Dann ist $c \in I$-Prog, es sei $Ic(l) =: z \in I$-Prog mit

$$\begin{aligned} Iz(x_1, \ldots, x_n) &= I(Ic(l))(x_1, \ldots, x_n) \\ &= I(I(IPEVAL(PEVAL, i)(l)))(x_1, \ldots, x_n) \\ &= I(IPEVAL(i, l))(x_1, \ldots, x_n) \\ &= Ii(l, x_1, \ldots, x_n) \\ &= Ll(x_1, \ldots, x_n) \end{aligned}$$

Damit ist $c \in [{}^{L}_{I}\!\!\rightarrow^{I}]$ tatsächlich ein Übersetzer. Nun wird der recht aufwändige partielle Auswerter nur noch für den Interpreter aufgerufen, woraus man dann einen Übersetzer erhält.

In der Übung wird gezeigt, dass man einen *Compilergenerator* erhält durch:

$$cc := IPEVAL(PEVAL, PEVAL)$$

2.4 Kombination von Interpretern und Übersetzern

Häufig geschieht keine direkte Übersetzung in Maschinencode, sondern eine Kombination aus Interpretern und Übersetzern. Vorteile sind:

- Übersetzungskomplexität geringer
- Wiederverwendung
- mehr Portabilität

BEISPIEL: Haskell-Implementierung auf einer Maschine M durch Übersetzung nach C, dann:

$$\left[\begin{array}{c|c} \text{Haskell} \rightarrow \text{C} & \text{C} \rightarrow \text{M} \\ \text{M} & \text{M} \end{array}\right]$$

Das Problem ist es hier, einen Haskell-Compiler in M zu schreiben. Daher wird zum Beispiel ein Haskell-Complier in C geschrieben und dann der Compiler nach M übersetzt.

Eine weitere Möglichkeit zur Implementierung höherer Programmiersprachen ist die Verwendung **abstrakter Maschinen**, die eine Zwischenschicht der Übersetzung darstellen. Beispiele hierfür sind:

- WAM: Warren Abstract Machine (Prolog)
- STGM: Spinless Tagless G-Machine (Haskell)
- JVM: Java Virtual Machine (Java)

Eigenschaften abstrakter Maschinen sind die Orientierung an der Quellsprache, aber auch die Effizienz der Implementierung auf existierenden Maschinen (etwa durch Interpretation/Emulation oder durch einen weiteren Übersetzer).

Der Ablauf beispielsweise bei Java: Der Code wird zu Byte-Code übersetzt auf JVM, danach dann interpretiert oder mit einem „Just-in-time"-Compiler übersetzt. Vorteil ist, dass der übersetzte JVM-Code portabel ist – Nachteil ist wiederum die Effizienz. Schema hier:

$$\left[\begin{smallmatrix}\mathsf{Java}\to\mathsf{JVM}\\ \mathsf{JVM}\end{smallmatrix}\right] \quad \left[\begin{smallmatrix}\mathsf{JVM}\\ M\end{smallmatrix}\right]$$
$$\left[\begin{smallmatrix}\mathsf{JVM}\\ M\end{smallmatrix}\right]$$

2.5 Bootstrapping

Ziel ist ein Übersetzer $\left[\begin{smallmatrix}Q\to M\\ M\end{smallmatrix}\right]$. Problem ist, dass Q eine Hochsprache ist und M in der Regel niedrig, d.h. ungeeignet für komplexe Software. Wünschenswert wäre es, den Übersetzer in Q zu schreiben, um die „Eleganz" von Q für die Implementierung zu nutzen und auch gleich die erste große Anwendung für den neuen Übersetzer zu haben. Unser Ziel ist also die Entwicklung eines Übersetzers $\left[\begin{smallmatrix}Q\to M\\ Q\end{smallmatrix}\right]$. Leider ist dieser so nicht lauffähig!

Realisierung durch **Bootstrapping** in mehreren Schritten:

1. Wähle eine **Teilsprache** T-$Q \subseteq Q$, die für einfachen Übersetzerbau geeignet ist und keine schwierig zu implementierenden Konstrukte enthält.

2. Schreibe einen Übersetzer $C_1 \in \left[\begin{smallmatrix}T\text{-}Q\to M\\ A\end{smallmatrix}\right]$ in einer beliebigen ausführbaren Sprache A (d.h. es existiert ein Übersetzer $\left[\begin{smallmatrix}A\to M\\ M\end{smallmatrix}\right]$ oder ein Interpreter $\left[\begin{smallmatrix}A\\ M\end{smallmatrix}\right]$). Beispiele für A sind Assembler oder C.
   C_1 muß nur korrekt sein, aber kann eine schlechte Fehlerbehandlung haben oder nur ineffizienten Code erzeugen.

3. Erhalte den T-Q-Übersetzer $\left[\begin{smallmatrix}T\text{-}Q\to M\\ M\end{smallmatrix}\right]$ durch Ansetzen von $\left[\begin{smallmatrix}A\to M\\ M\end{smallmatrix}\right]$ auf $\left[\begin{smallmatrix}T\text{-}Q\to M\\ A\end{smallmatrix}\right]$ (diese Übersetzer kann auch aus einem einfachen Interpreter für T-Q automatisch generiert werden, siehe oben)

4. Schreibe den „eigentlichen“ Übersetzer C_2 für Q, aber noch nicht in Q sondern in der Sprache T-Q. Dieser sollte den üblichen Qualitätsansprüchen bzgl. der Fehlerbehandlung und Codeeffizienz genügen.

5. Implementiere nun C_2 mittels (3), durch Ansetzen von $\left[{}^{T\text{-}Q\to M}_{M}\right]$ auf $\left[{}^{Q\to M}_{T\text{-}Q}\right]$. Dieser erzeugt **guten Code**, ist selbst jedoch **ineffizient**.

6. Wende nun C_2 auf den Sourcecode von C_2 selbst an und erhalte einen **effizienten Übersetzer** C_3, der auch effizienten Code erzeugt. Für diesen Bootstrap-Schritt ist es wichtig, dass T-Q eine Teilmenge von Q ist!

7. Eine weitere Selbstanwendung bringt nichts, ist aber trotzdem nützlich als **Praxistest**: Implementiere C_2 durch Anwendung von C_3 auf den Sourcecode von C_2 (bzw. C_3). Dieser erhaltene Übersetzer C_4 sollte **textuell identisch** zu C_3 sein.

Man benötigt also zwei Dinge zum Bootstrapping: Einen einfachen Übersetzer C_1 in einer existierenden Sprache und dann einen „guten“ Übersetzer C_2 für die Sprache Q in T-Q.

Vorteile:

1. Bei allen Verbesserungen und Erweiterungen von C_2 ist nun der volle Sprachumfang von Q verwendbar, d.h. C_2' ist aus $\left[{}^{Q\to M}_{Q}\right]$.

2. Jede Verbesserung in der Codeerzeugung von C_2 kommt C_2 selbst zugute: Ein verbesserter Compiler aus $\left[{}^{Q\to M}_{M}\right]$ ist erzeugbar durch Anwendung des alten implementierten Compilers auf den verbesserten Compiler.

3. Der Compiler ist leichter wartbar, da er selbst in der Hochsprache Q geschrieben ist.

4. Die Methoden erlauben *Querübersetzer (Crosscompiler)* etwa für neue Rechnertypen, in dem man nur das Backend anpasst.

Diese Vorgehen wurde von Wirth am Beispiel der Implementierung von Pascal umgesetzt und ist heute eine wichtige Implementierungstechnik im Übersetzerbau.