VII FUNKTIONSWEISE EINES COMPILERS

S. BARTELS, 10.7.2018

1. Umwandlung lesbarer Programme

Programmiersprachen wie MATLAB und C++ erlauben eine hohe Problemabstraktion durch die Verwendung von Funktionen, Schleifen und Rekursionen. Diese Konstrukte müssen vom Compiler in für den Prozessor verarbeitbare Instruktionen übersetzt werden. Dieser kann jedoch kaum mehr als Informationen aus Registern laden und einfache arithmetische und logische Verknüpfungen sowie Fallunterscheidungen durchführen. Entsprechende Prozessor-Anweisungen werden durch Folgen von Nullen und Einsen codiert, die für Menschen nur mit hohem Aufwand verständlich und zudem sehr fehleranfällig sind. Die Aufgabe von Compilern ist es, verständliche, lesbare Anweisungen in solche Codes zu übersetzen. Dies ist jedoch mit einigen Schwierigkeiten verbunden, da sichergestellt werden muss, dass Befehle eindeutig definiert sind. Wie schwierig korrekte Spracherkennung im Allgemeinen sein kann, zeigt bereits der mehrdeutige Satz *Der Fahrer muss das Hindernis umfahren*, bei dem die Bedeutung nur aus dem Kontext oder einer geeigneten Betonung hervorgeht.

2. Beispiel eines Maschinencodes

Um die Struktur eines Maschinencodes zu illustrieren, folgen wir einem Beispiel aus dem Buch *Computer* von R. Drechsler, A. Fink und J. Stoppe (Springer, 2017) und betrachten einen 8-Bit Modellprozessor, das heißt, ein Prozessor, der aus 8 Bits bestehende Anweisungen verarbeiten kann. Das betrachtete Maschinencode-Programm besteht aus den folgenden 64 Bits:

00010110 00100011 00010010 00100100
00000011 01000100 00100101 11000000

Das Programm definiert eine Folge von acht Anweisungen und jede davon hat die Struktur:

$$\underbrace{b_0\, b_1\, b_2}_{\text{Befehl}} \quad \underbrace{b_3}_{\text{Nummer}} \quad \underbrace{b_4\, b_5\, b_6\, b_7}_{\text{Operand}}$$

Dabei wählen die Bits b_0, b_1, b_2 einen von acht Befehlen aus und Bit b_3 legt fest, ob die Bits $b_4 b_5 b_6 b_7$ als Zahl oder Adresse eines Registers interpretiert werden sollen. Wir nehmen an, dass die von 0 bis 7 numerierten Befehle gegeben sind durch folgende Operationen:

`LOAD, STORE, ADD, SUB, COMP, JUMP, HALT, NOOP`

Der Befehl `LOAD` kann beispielsweise eine Zahl aus einem bestimmten Register in das Arbeitsregister laden. Anschließend kann mittels `ADD` eine konkrete Zahl zum Wert des Arbeitsregister addiert werden. Die resultierenden Bedeutungen der Befehle des obigen Maschinencodes sind in Tabelle 1 erklärt. Es stellt sich heraus, dass die Rechnung $6 + 2$ durchgeführt wird, die sich auch mit weniger Befehlen realisieren ließe.

8-Bit-Wort	Befehl	Nr.	Op.	Interpretation
00010110	`LOAD`	1	6	Lade Zehl 6 ins Arbeitsregister (AR)
00100011	`STORE`	0	3	Speichere Wert des AR in Register 3
00010010	`LOAD`	1	2	Lade Zahl 2 ins AR
00100100	`STORE`	0	4	Speichere Wert des AR in Register 4
00000011	`LOAD`	0	3	Lade Wert des Registers 3 ins AR
01000100	`ADD`	0	4	Addiere Wert des Registers 4 zum Wert des AR
00100101	`STORE`	0	5	Speichere Wert des AR in Register 5
11000000	`HALT`	0	0	Stoppe

TABELLE 1. Interpretation eines Beispielmaschinencodes.

Deutlich übersichtlicher wird das Beispielprogramm in sogenannter Assemblersprache. Dabei können die Befehle als Begriffe angegeben und mit dem Symbol # Zahlen von Adressen unterschieden werden:

```
LOAD #6
STORE 3
LOAD #2
STORE 4
LOAD 3
ADD 4
STORE 5
HALT
```

Assemblersprachen definieren eine Zwischenstufe zwischen Maschinencodes und Programmen höherer Programmiersprachen wie MATLAB und C++.

3. Höhere Programmiersprachen

Unter *höheren Programmiersprachen* versteht man solche, die im Vergleich zum Maschinencode eine höhere Abstraktion durch Schleifen, Funktionen und Rekursionen erlauben. Insbesondere hängen sie nicht von den besonderen Eigenschaften des verwendeten Rechners ab, beispielsweise was die Speicherverwaltung betrifft. Übersetzer oder *Compiler* erzeugen aus einem Programm einer hóheren Programmiersprache wie C++ einen Assembler- beziehungsweise Maschinencode. Die Programmiersprache C++ ist dabei sehr maschinennah, da sie beispielsweise durch den Einsatz von Zeigern einen sehr direkten Zugriff auf den Speicher ermöglicht. Im Gegensatz dazu wird

MATLAB nicht als Compiler- sondern als Interpretersprache angesehen. Vereinfacht dargestellt werden dabei bereits übersetzte Programme aufgerufen und weitere Teile des Programms wie Schleifen erst bei ihrem Auftreten und abhängig von den Eingabewerten übersetzt. Durch den Einsatz sogenannter *virtueller Maschinen*, die eine zusätzliche Ebene zwischen Programmiersprache und Maschinencode schaffen, verschmelzen die Konzepte von Compiler- und Interpretersprachen zunehmend. Dadurch ist es insbesondere möglich, auch in Interpretersprachen Konzepte wie Rekursion anzuwenden. Noch etwas weiter geht die Methodik der *Just-In-Time*-Compiler, die ein Programm analysieren und abhängig von Eingaben teilweise übersetzen oder bereits übersetzten Code wiederverwenden, und bei Programmiersprachen wie JavaScript und neueren Versionen von MATLAB zum Einsatz kommt. Eine Gegenüberstellung der Vor- und Nachteile klassischer Compiler- und Interpretersprachen findet sich in Tabelle 2.

Compilersprache (C++)	Interpretersprache (MATLAB)
⊕ schnelle Programme ⊕ Flexibilität durch direkten Zugriff auf Speicheradressen ⊕ Übersetzung unabhängig von Eingabedaten ⊖ lange Übersetzungsphase ⊖ kompliziert und fehleranfällig	⊕ direkte Nutzung, keine Übersetzung ⊕ keine Deklaration von Variablen ⊕ einfache Programme ⊖ langsame Schleifen ⊖ keine explizite Speicherverwaltung ⊖ Übersetzung abhängig von Eingabe

TABELLE 2. Vor- und Nachteile von Compiler- und Interpretersprachen am Beispiel von C++ und MATLAB.

4. DER COMPILER

Zur Beschreibung der Arbeitsweise eines Compilers folgen wir in diesem Abschnitt der Darstellung des Vorlesungsskripts *Compilerbau* von U. Goltz, T. Gehrke und M. Lochau (TU Braunschweig, 2010) Ein Compiler, der aus einem Quellprogramm einen Maschinencode erzeugt, arbeitet zusammen mit einem *Präprozessor*, der aus dem rohen Quellcode Makros ersetzt und Kommentare entfernt, einem *Assembler*, der aus Assemblercode relokatiblen, das heißt verschiebbaren Maschinencode generiert, sowie einem *Binder*, der konkrete Sprungadressen einfügt und somit einen ausführbaren Machinencode erstellt. Diese Schritte sind in Abbildung 1 schematisch dargestellt.
Die Übersetzung eines Programms durch den Compiler erfolgt in einer Analyse- und einer Synthesephase, wobei die Analysephase aus der lexikalischen,

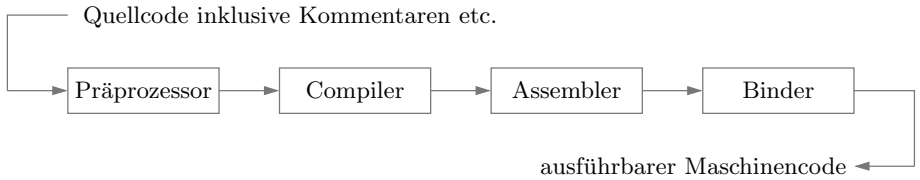

ABBILDUNG 1. Schritte der Übersetzung eines Programms.

der syntaktischen und der semantischen Analyse besteht. In der lexikalischen Analyse, die durch den *Scanner* durchgeführt wird, wird die gegebene Zeichenkette in Bestandteilte zerlegt und eine Symboltabelle angelegt. Variablen, die in diesem Kontext oft als Bezeicher betitelt werden, werden dabei numeriert. In der syntaktischen Analyse, die vom *Parser* durchgeführt wird, wird analysiert, ob beispielsweise Terme korrekt sind. Dazu wird ein Strukturbaum, auch als Syntaxbaum bezeichnet, erstellt, der die verwendeten Operationen widergibt. In der semantischen Analyse werden die auftretenden Bezeichner mit Attributen wie Variablentyp und Gültigkeitsbereich versehen und die Wohldefiniertheit der verwendeten Operationen geprüft. Dies garantiert die statische semantische Korrektheit, das heißt die formale Wohlgestelltheit der Berechnungen unabhängig von konkreten Eingabedaten, die möglicherweise zu dynamischen Semantikfehlern wie Division durch Null führen können. Die anschließende Synthese-Phase besteht aus verschiedenen Schritten wie der Zwischencode-Erzeugung, einer Code-Optimierung und der Maschinencode-Generierung. Im Zwischencode werden beispielsweise Formeln mit Hilfsvariablen in ihre Bestandteile zerlegt und als Folgen einfacher Dreiadressbefehle der Form

```
tmp_l = id_j op id_k;
```

dargestellt. Die darauf folgende Codeoptimierung entfernt überflüssige Zuweisungen. Bei der finalen Maschinencode-Erzeugung werden die einfachen Befehle des optimierten Assemblercodes in Maschinenbefehle übersetzt. Diese letzte Phase der Übersetzung ist von dem verwendeten Rechner abhängig und damit Teil der *Back-End-Phase*, die im Gegensatz zur *Front-End-Phase* den maschinenabhängigen Teil der Übersetzung bezeichnet. Die schrittweise Übersetzung der Zuweisung $position := initial + rate * 60$ in Assemblercode ist in Abbildung 2 dargestellt.

5. Methodik des Scanners

Die lexikalische Analyse eines Quellprogramms basiert auf der Theorie regulärer Sprachen. Eine Sprache ist dabei eine Menge L von Wörtern über einem Alphabet Σ beispielsweise

$$\Sigma = \{a, b, c\}, \quad L = \{a, b, ab, ac, cba\}.$$

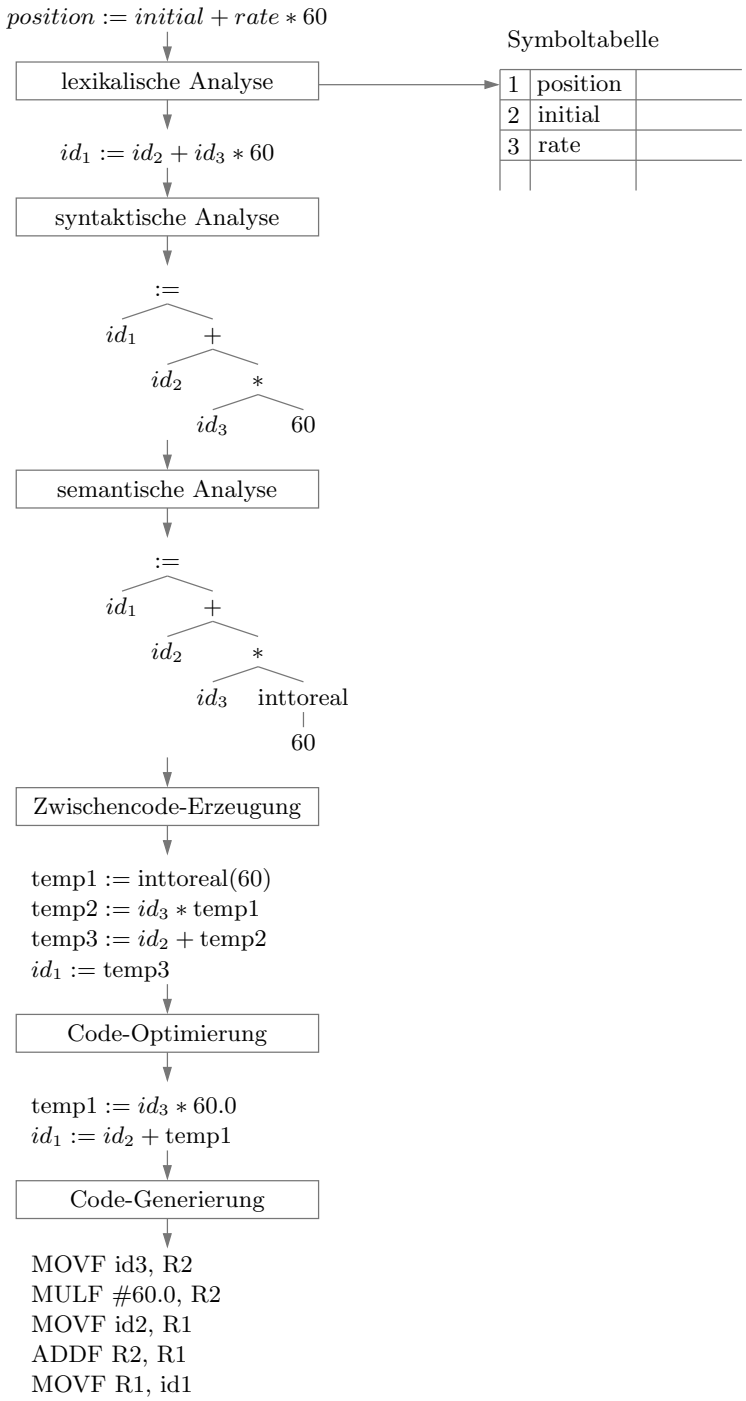

ABBILDUNG 2. Schrittweise Übersetzung einer Zuweisung.

Reguläre Sprachen sind solche, die sich aus aus der leeren Menge und den einelementigen Sprachen durch Vereinigung, Konkatenation und Exponenzieren bilden lassen. Mit regulären Ausdrücken können reguläre Sprachen durch Formeln definiert werden. Als Operationen sind dabei Auswahl, Konkatenation, und Exponenzieren mit den im Alphabet vorhandenen Buchstaben erlaubt. Zusätzlich steht das leere Wort ε zur Verfügung, das beispielsweise neutral bezüglich Konkatenation ist.

Beispiele 5.1. *(i) Sei $\Sigma = \{a, b\}$. Die Auswahl $a|b$ erzeugt $L = \{a, b\}$, die Konkatenation ab erzeugt $L = \{ab\}$, das Exponenzieren a^* erzeugt $L = \{\varepsilon, a, aa, aaa, \dots\} = \{a^n : n \geq 0\}$.*
(ii) Mit $\Sigma = \{a, b, \dots, z, 0, 1, \dots, 9\}$ und dem regulären Ausdruck

$$(a|b|\dots|z)(a|b|\dots|z|0|1|\dots|9)^*$$

wird eine Menge zulässiger Variablennamen definiert, die mit einem Buchstaben beginnen.

Durch reguläre Ausdrücke definierte reguläre Sprachen lassen sich algorithmisch erkennen.

Definition 5.2. *Ein* nicht-deterministischer endlicher Automat *(NEA) ist ein Tupel $M = (\Sigma, Q, \Delta, q_0, F)$ mit den folgenden Eigenschaften:*
(i) Σ ist ein endliches Alphabet,
(ii) Q ist eine endliche Zustandsmenge,
(iii) $q_0 \in Q$ ist ein Anfangszustand,
(iv) $F \subset Q$ ist eine Menge von Endzuständen,
(v) $\Delta \subset Q \times \big(\{\varepsilon\} \cup \Sigma\big) \times Q$ ist eine Übergangsrelation.
Ein NEA heißt deterministischer endlicher Automat *(DEA), wenn die Übergangsrelation eine Funktion $\Delta : Q \times \big(\{\varepsilon\} \cup \Sigma\big) \to Q$ ist.*

Ein Übergang von einem Zustand q und einem Wort zw zu einem neuen Zustand q' mit Wort w ist zulässig, wenn $(q, z, q') \in \Delta$ gilt beziehungsweise im deterministischen Fall $\Delta(q, z) = q'$ gilt. Die von einem NEA akezptierte Sprache ist die Menge aller Wörter w_0, die sich in endlich vielen zulässigen Übergängen vom Anfangszustand q_0 auf das leere Wort ε und einen Endzustand $q_f \in F$ reduzieren lassen:

$$(q_0, w_0) \mapsto (q_1, w_1) \mapsto \dots \mapsto (q_f, \varepsilon),$$

wobei jeweils $w_{k-1} = zw_k$ mit $z \in \Sigma \cup \{\varepsilon\}$ gelte. Der folgende Satz stellt eine Beziehung zwischen regulären Sprachen und Automaten her.

Satz 5.3. *(i) Zu jedem regulären Ausdruck r existiert ein NEA, der die von r definierte reguläre Sprache $L(r)$ akzeptiert.*
(ii) Wird die reguläre Sprache L von einem NEA akzeptiert, so existiert ein DEA, der L akzeptiert.

Automaten werden am übersichtlichsten durch Graphen dargestellt. Die Menge der Variablennamen über dem Alphabet $\Sigma = \{a, b, \dots, z, 0, 1, \dots, 9\}$,

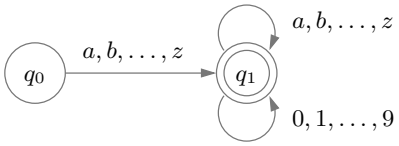

ABBILDUNG 3. Deterministischer endlicher Automat, der eine Menge zulässiger Variablennamen erkennt.

die mit einem Buchstaben beginnen, werden von dem in Abbildung 3 definierten Automaten mit $Q = \{q_0, q_1\}$ und $F = \{q_1\}$ akzeptiert.
Für die praktische Unterscheidung zwischen Bezeichnern für Variablen und Schlüsselwörtern wie `if`, `else`, `end`, `while`, `for`, `function` wird entweder ein geeigneter *lookahead* verwendet oder die Schlüsselwörter werden zunächst wie Variablennamen behandelt und anschließend extrahiert.

6. Methodik des Parsers

Das Konzept regulärer Ausdrücke und endlicher Automaten ist für eine Syntaxanalyse nicht ausreichend, da zum Beispiel Klammerstrukturen wie `begin ... end` nicht erkannt werden können. Ein Beispiel einer nicht-regulären Sprache sind die wohlgeformten Klammerterme, die auch als Dyck-Sprache bezeichnet werden,

$$\big\{w \in (a|b)^* : |w|_a = |w|_b, \forall u, v \in (a|b)^*,\, w = uv \implies |u|_a \geq |u|_b\big\},$$

wobei a und b öffnende beziehungsweise schließende Klammern repräsentieren und $|w|_a$ die Anzahl öffnender Klammern in w angibt. Endliche Automaten verfügen über keinen Mechanismus, der ein Zählen bestimmter Symbole realisiert. Das Konzept kontextfreier Grammatiken und damit verbundener Keller-Automaten erlaubt die Erkennung von Klammerausdrücken.

Definition 6.1. *Eine* kontextfreie Grammatik *ist ein Tupel* $G = (V_N, V_T, P, S)$ *mit den folgenden Eigenschaften:*
(i) V_N *ist eine endliche Menge von Nichtterminalsymbolen,*
(ii) V_T *ist eine endliche Menge von Terminalsymbolen,*
(iii) $P \subset V_N \times (V_N \cup V_T)^*$ *ist eine Menge von Produktionen,*
(iv) $S \in V_N$ *ist ein Startsymbol.*
Die von G *generierte Sprache* $L(G)$ *ist die Menge aller Wörter über* V_T*, die sich mit den Produktionen aus dem Startsymbol ableiten lassen, indem sukzessive Nichtterminalsymbole gemäß der Produktionsregeln ersetzt werden.*

Terminale stehen für Symbole, das heißt für Bezeichner, Schlüsselwörter, Operatorsymbole und Konstanten, die in der lexikalischen Analyse identifiziert worden sind. Eine Produktion $(A, \alpha) \in P$ schreibt man in der Form $A \to \alpha$ und zwei Produktionen $A \to \alpha$ und $A \to \beta$ werden zusammengefasst zu $A \to \alpha|\beta$.

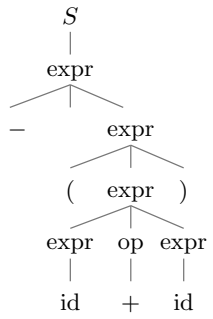

ABBILDUNG 4. Strukturbaum zur Formel $-(\mathrm{id}+\mathrm{id})$.

Beispiel 6.2. *Seien* $V_N = \{\mathrm{expr}, \mathrm{op}\}$ *und* $V_T = \{\mathrm{id}, +, -, *, /, \hat{}, (,)\}$*, wobei* id *für einen Bezeichner oder eine Konstante stehe, mit den Produktionen*

$$\mathrm{expr} \to \mathrm{expr}\ \mathrm{op}\ \mathrm{expr} \mid (\mathrm{expr}) \mid -\mathrm{expr} \mid \mathrm{id}$$
$$\mathrm{op} \to + \mid - \mid * \mid / \mid \hat{}$$

sowie dem Startsymbol $S = \mathrm{expr}$*. Mit der dadurch definierten kontextfreien Grammatik lassen sich zulässige arithmetische Ausdrücke definieren.*

Zu jedem Wort der Sprache einer kontextfreien Grammatik lässt sich ein Strukturbaum angeben. Für die aus der mit Beispiel 6.2 erzeugten Formel $-(\mathrm{id}+\mathrm{id})$ ist der zugehörige Strukturbaum in Abbildung 4 gezeigt. Die Erkennung der Sprache einer kontextfreien Grammatik erfolgt mit Kellerautomaten, die einen Hilfsspeicher besitzen und eine Ausgabe erzeugen können.

Definition 6.3. *Ein* Kellerautomat *ist ein Tupel* $M = (\Sigma, \Gamma, \Delta, z_0, \mathcal{O})$ *mit den Eigenschaften:*
(i) Σ *ist ein endliches Eingabealphabet,*
(ii) Γ *ist ein endliches Kelleralphabet,*
(iii) $z_0 \in \Gamma$ *ist ein Kellerstartsymbol,*
(iv) $\mathcal{O}$ *ist ein endliches Ausgabealphabet,*
(v) $\Delta \subset \big((\Sigma \cup \{\varepsilon\}) \times \Gamma\big)^* \times (\Gamma^* \times \mathcal{O}^*)$ *ist eine Übergangsrelation.*
Die Menge der akzeptierten Wörter sind alle Wörter über Σ*, die sich mit der Übergangsrelation auf das leere Wort und einen leeren Kellerinhalt reduzieren lassen.*

Ein Kellerautomat analysiert ein Eingabewort über dem Alphabet Σ, indem er schrittweise Zeichen einliest und in Abhängigkeit vom Kellerinhalt eine Ausgabe und einen neuen Kellerinhalt erzeugt. Dieses Vorgehen ist in Abbildung 5 skizziert. Der Kellerinhalt dient beispielsweise dem Zählen noch offener Klammern und als Ausgabe kann der Strukturbaum erzeugt werden.

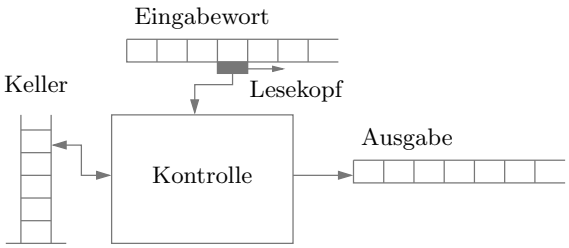

ABBILDUNG 5. Schematische Darstellung der Funktionsweise eines Kellerautomaten.