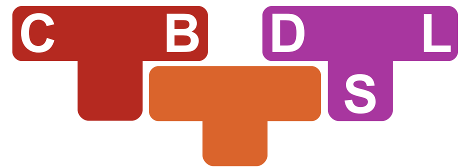
Compilerbau und Domänenspezifische Sprachen



Skriptum

Wintersemester 2019/2020

Andreas Schwenk

contact@compiler-construction.com

17. Januar 2020

Abkürzungen und Akronyme

ADD Addition.
API Application Programming Interface.
AST Abstract Syntax Tree.

BFS Breadth-first search; Breitensuche.
BNF Backus Naur Form.
BUP Bottom-Up Parsing.

CISC Complex Instruction Set Computer.

DDT Delimiter Directed Translation.
DEA Deterministischer endlicher Automat.
DFS Depth-first search; Tiefensuche.
DIV Division.
DKF Deterministisch kontextfrei.
DSL Domain-Specific Language.

EBNF Erweiterte Backus Naur Form.

FP Framepointer.

GNU GNU's not Unix.

IDE Integrated Development Environment.
ISA Instruction Set Architecture.

LBA Linear beschränkter Automat.

LL Top-Down Parsertyp.
LLVM Low Level Virtual Machine.
LR Bottom-Up Parsertyp.

MC Maschinencode.
MUL Multiplikation.

NEA Nichtdetermin. endlicher Automat.
NP Nominalphrase.

PDA Push-Down Automaton; Kellerautomat.

REG Register.
RISC Reduced Instruction Set Computer.
RR Returnregister.

SDT Syntax-Directed Translation.
SP Stackpointer.
ST Symboltabelle.
SUB Subtraktion.

TDP Top-Down Parsing.
TM Turingmaschine.

VM Virtuelle Maschine.
VP Verbalphrase.

Inhaltsverzeichnis

Vorwort 9

1 Grundlagen 11
1.1 Komplexitätstheorie 11
1.2 Graphentheorie 12
1.3 Syntax und Semantik 13
1.4 Universalsprachen 14
1.5 Domänenspezifische Sprachen 14

2 Formale Sprachen 15
2.1 Sprache 16
2.2 Grammatik 17
2.3 Chomsky Hierarchie 18
2.4 Reguläre Sprachen (Typ 3) 19
2.4.1 Endliche Automaten 19
2.4.2 Grenzen regulärer Sprachen 22
2.5 Kontextfreie Sprachen (Typ 2) 23
2.5.1 Kellerautomaten 23
2.5.2 Deterministisch kontextfreie Sprachen 25
2.6 Metasprachen zur Darstellung von Grammatiken 26
2.6.1 Backus Naur Form (BNF) 27
2.6.2 Erweiterte Backus Naur Form (EBNF) 27
2.7 Attribute und Aktionen 28
2.7.1 Attributierte Grammatiken 28
2.7.2 Emittieren von Aktionscode 28
2.8 Transduktoren 29

3 Compilerbau 31
3.1 Einführung 31
3.2 Datenstrukturen 33
3.2.1 Abstrakter Syntaxbaum (AST) 34
3.2.2 Symboltabelle 34
3.3 Lexikalische Analyse / Scanning 34
3.4 Syntaktische Analyse / Parsing 36
3.4.1 Definitionen 36
3.4.2 Top-Down Parsing 37
3.4.2.1 Methode des rekursiven Abstiegs 37

3.4.2.2 Beseitigung von Linksrekursionen . . . 39
3.4.2.3 LL Parser . . . 40
3.4.2.4 LL(k) Parser . . . 41
3.4.2.5 Komplexität . . . 42
3.4.3 Bottom-Up Parsing . . . 42
3.4.3.1 LR(k) Parser . . . 44
3.4.3.2 Shift-Reduce Konflikte . . . 47
3.4.3.3 Reduce-Reduce Konflikte . . . 47
3.5 Semantische Analyse . . . 48
3.5.1 Attributierte Grammatiken . . . 48
3.5.1.1 Typregeln . . . 48
3.5.2 Deklarationen . . . 49
3.5.2.1 Symboltabelle . . . 49
3.5.2.2 Kontext und Geltungsbereiche . . . 50
3.5.2.3 Mehrdimensionale Datentypen . . . 51
3.6 Codegenerierung . . . 52
3.6.1 Interpretation . . . 52
3.6.2 Kompilation . . . 52
3.6.3 Registermanagement . . . 53
3.6.4 Datenrepräsentation . . . 54
3.6.5 Befehlssatzarchitektur . . . 55
3.6.6 Mathematische Ausdrücke . . . 57
3.6.7 Bedingte Anweisungen . . . 57
3.6.8 Funktionen . . . 58
3.7 Codeoptimierung . . . 59
3.7.1 Onlineoptimierung . . . 60
3.7.1.1 Verzögerte Codegenerierung . . . 60
3.7.1.2 Konstantenfaltung . . . 60
3.7.2 Offlineoptimierung . . . 61
3.7.2.1 Peephole-Optimierung . . . 61
3.7.2.2 Schleifenoptimierung . . . 64
3.7.2.3 Registereinsatz . . . 65
3.7.3 Aktuelle Entwicklungen . . . 67
3.8 Beispielcompiler . . . 67
3.8.1 Iom . . . 67
3.8.2 Clang und LLVM . . . 67

4 Sprachentwurf 69
4.1 Motivation . . . 69
4.2 Arten von DSLs . . . 70
4.3 Definition . . . 71
4.4 Abgrenzungen . . . 71
4.5 Integration von DSLs . . . 72
4.6 Domänenmodellierung . . . 73
4.6.1 Domänenanalyse . . . 73
4.7 Sprachen und Programme . . . 75

4.7.1 Größe und Umfang einer Sprache . . . 75
4.7.2 Übereinstimmung von Domäne und Sprache . . . 76
4.8 Entwurfsdimensionen . . . 76
4.8.1 Ausdrucksstärke . . . 76
4.8.2 Abdeckung . . . 77
4.8.3 Semantik und Ausführung . . . 77
4.8.4 Trennung der Angelegenheiten . . . 78
4.8.5 Vollständigkeit . . . 78
4.8.6 Sprachmodularität . . . 78
4.8.7 Konkrete Syntax . . . 78
4.9 Verhaltensparadigmen . . . 79
4.10 Strukturierung . . . 79
4.11 Beispiele . . . 80

5 Eingebettete Domänenspezifische Sprachen 81
5.1 Anforderungen an die Hostsprache . . . 82
5.2 Entwurfsmuster . . . 82
5.2.1 Methodenverkettung . . . 82
5.2.2 Funktionenfolge . . . 83
5.2.3 Geschachtelte Funktionen . . . 83
5.2.4 Wörterbücher . . . 84
5.2.5 Wahl der Entwurfsmuster . . . 84
5.3 Lisp . . . 84
5.3.1 Grundlagen . . . 85
5.3.2 Homoikonizität . . . 86

6 Externe Domänenspezifische Sprachen 87
6.1 Modellierung . . . 88
6.2 Codegenerierung . . . 90
6.2.1 Trennzeichengesteuerte Übersetzung . . . 91
6.2.2 Syntaxgesteuerte Übersetzung . . . 91
6.2.3 Eingebettete Übersetzung . . . 91
6.2.4 Baumkonstruktion . . . 91
6.2.5 Eingebettete Interpretation . . . 92
6.2.6 Anreicherung um Allzweckcode . . . 92
6.3 Softwaretests . . . 93
6.4 Dokumentation . . . 93
6.5 Sprachwerkbänke . . . 93

7 Forschungsgebiete 95
7.1 Forschungslandschaft . . . 95
7.2 Codeoptimierung . . . 95
7.3 Quantum Computing . . . 96
7.3.1 Konferenzen . . . 96
7.4 Natürliche Sprachverarbeitung . . . 96

Anhang 97

A Werkzeuge 97
A.1 Flex und GNU Bison . . . 97
A.1.1 Fast Lexical Analyzer Generator (FLEX) . . . 97
A.1.2 GNU Bison . . . 98
A.2 ANTLR . . . 99
A.3 Eclipse Xtext . . . 99
A.3.1 Einführung . . . 99
A.3.2 Xtend-Tutorial . . . 100
A.3.3 Xtext-Tutorial . . . 103
A.3.4 Parser . . . 105
A.3.4.1 Grammatik . . . 105
A.3.5 Verschiedenes . . . 107
A.4 JetBrains MPS . . . 108

B Der Iom-Compiler 109
B.1 Lexer . . . 110
B.2 Parser . . . 111
B.3 Aufrufkonventionen im Iom-Compiler . . . 114

C DSL Beispiele 119
C.1 Hausautomatisierung . . . 119
C.2 Conway's Game of Life . . . 121
C.3 Graphenbeschreibung . . . 121
C.4 Schriftsatzsystem . . . 121
C.5 Künstliche Neuronale Netze . . . 122
C.6 Fuzzy-Logik . . . 122
C.7 KI für Echtzeitstrategiespiele . . . 122
C.8 Energieverbundsysteme . . . 123
C.9 Zustandsautomaten . . . 123
C.10 Musiknotation . . . 124

Vorwort

"Science is what we understand well enough to explain to a computer.
Art is everything else we do."

— Donald Knuth

Dieses zusammenfassende Skript zur Lehrveranstaltung **Compilerbau und domänenspezifische Sprachen (CBDSL)** enthält alle Themen der Vorlesung. Während die Vortragsfolien nach didaktischen Gesichtspunkten angeordnet sind, stellt das vorliegende Dokument jeweils alle Aspekte zu einem Themengebiet konkateniert dar. Die Übungen mit Lösungen, Quizzes und Biografien sind in dieses Skript bisher nicht integriert worden.

Für das Melden von syntaktischen und semantischen Fehlern, sowie Anregungen jeglicher Art, ist der Autor sehr dankbar. Senden Sie gerne eine formlose Mail an errata@compiler-construction.com. Zur Belohnung erhalten Sie einen **Scheck** der "Bank of Compiler-Construction and Domain-Specific Languages".

Windeck und Köln im Dezember 2019,

Andreas Schwenk

Technische Hochschule Köln
Universität zu Köln

Folienkapitel	Skriptkapitel
1	1, 2.4.1, 3.2
2	4, Anhang C
3	2
4	3.3
5a	Anhang A.3
5b	6
6	5
7	3.4
8	3.5, 3.6, 3.8
9	3.7

Tab. 1: Zuordnung der Kapitelnummern: Foliensatz vs. Skript.

Die Lehrveranstaltung CBDSL

"Die Grenzen meiner Sprache bedeuten die Grenzen meiner Welt."

— Ludwig Wittgenstein

Im Folgenden werden die **Learning Outcomes** des Kurses kurz umrissen.

WAS Diese Lehrveranstaltung vermittelt Kompetenzen für die Erstellung von neuen **domänenspezifischen Sprachen.** Dabei werden alle Arbeitsschritte vom **Entwurf** bis zur **Implementierung** einbezogen. Die Studierenden können auf Basis gegebener **Anforderungen** entscheiden, ob sie neue domänenspezifische Sprachen entwickeln oder eine bestehende Sprache und Standards analysieren, erweitern und konsolidieren sollten. Weiterhin werden Kompetenzen zur Erstellung neuer **Allzwecksprachen** und deren Weiterverarbeitung vermittelt. Sowohl praktische **Übungen** als auch ein parallel laufendes **Projekt** fördern die außerfachlichen Kompetenzen, wie zum Beispiel Teamarbeit und Projektorganisation. Die Studierenden evaluieren ihre selbst erstellten **Prototypen** kritisch und präsentieren und verteidigen sie vor den Kursmitgliedern. Sie überblicken das Angebot an aktuell verfügbaren, modernen **Toolchains** und greifen kontextsensitiv auf geeignete Werkzeuge für die Umsetzung zurück. Dabei arbeiten sich die Studierenden teils selbstständig in neu aufkommende Werkzeuge, wie zum Beispiel neuartige **Sprachwerkbänke**, ein.

WOMIT Der Dozent vermittelt Wissen und Basisfertigkeiten in einem **Vorlesungs**- und **Übungs**teil. Parallel dazu werden in Teams **Projekte** realisiert, in denen die Studierenden eine eigene domänenspezifische Sprache **modellieren** und in ein bestehendes System integrieren, oder wahlweise selbst eine Testumgebung schaffen.

WOZU Der **Compilerbau** gehört zu den **zentralen** und ältesten Gebieten der Informatik. Im Wesentlichen geht es darum, eine syntaktisch komplexe **Quellsprache** in eine andere, meist einfachere **Zielsprache** zu **übersetzen**.

Domänenspezifische Sprachen stellen ein essentielles Werkzeug für die **Kommunikation** zwischen InformatikerInnen und DomänenexpertInnen anderer Fachdisziplinen dar. Sie definieren präzise Kommunikationsschnittstellen, die eine erfolgreiche Zusammenarbeit in **interdisziplinär**en Projekten fördern. Beispielsweise kann eine sonst manuell nur sehr aufwändige Ansteuerung von Simulations- und Optimierungswerkzeugen (zum Beispiel im Bereich der erneuerbaren Energien) durch eine domänenspezifische Sprache eingängiger beschrieben werden.

Die Erschaffung neuer **Allzwecksprachen** ist nach wie vor von hoher Relevanz. Immer komplexer werdende Systeme verlangen aus **Abstraktions**gründen nach immer ausdrucksstärker werdenden Sprachen. Auch die Generierung von **energieeffizient** ausführbarem Maschinencode für **massiv parallelverarbeitende Computersysteme** wird wichtiger, um dem steigenden Bedarf an rechenintensiven Verfahren z.B. aus dem Bereich des Data-Mining standzuhalten. Auch gänzlich neue Architekturparadigmen, wie z.B. **Quantencomputer**, erfordern die Erschaffung und Weiterentwicklung entsprechender Programmierwerkzeuge.

Kapitel 1

Grundlagen

"Patience you must have, my young Padawan."

— *Yoda*

Wir werden in diesem Kapitel einige wichtige theoretische Grundlagen wiederholen.

1.1 Komplexitätstheorie

Die **$\mathcal{O}$-Notation** klassifiziert Algorithmen hinsichtlich ihrer **Laufzeit**. Die Komplexität eines Algorithmus beschreibt den Ressourcenbedarf der benötigt wird, um diesen auszuführen. Wir interessieren uns im Folgenden für die **asymptotische Komplexität.**

Definition 1 ($\mathcal{O}$-Notation) *Seien f und g zwei Funktionen $f : \mathbb{N} \to [0, \infty)$ und $g : \mathbb{N} \to [0, \infty)$. Dann ist f* ***von der Ordnung*** *g, falls für alle großen $n \geq n_0 \in \mathbb{N}$ die folgende Ungleichung gilt:*

$$f(n) \leq c \cdot g(n)$$

Für die Konstante gilt $c > 0$. Wir schreiben dann $f \in \mathcal{O}(g)$. Die Menge an Funktionen der Ordnung g ist $\mathcal{O}(g)$.

Theoretische Beispiele:

- $f_1(n) = 3n^3 + 5n + 7 \leq c \cdot n^3 \ \in \mathcal{O}(n^3)$
- $f_2(n) = 11n^2 \cdot 13 \log(n) \leq c \cdot n^2 \log(n) \ \in \mathcal{O}(n^2 \cdot \log(n))$

Praktische Beispiele:

- Lineare Suche: $\mathcal{O}(n)$
- Diskrete Fourier Transformation (DFT): $\mathcal{O}(n^2)$

Die Konstante c hängt in der Praxis maßgeblich von der Implementierung (Programmiersprache, Kodierung, Laufzeitumgebung usw.) ab.

Anmerkung: Die Aussage $f(n) = 2n + 1 \in \mathcal{O}(n^5)$ ist wahr, da die $\mathcal{O}$-Notation lediglich eine ***obere Schranke*** *angibt.*

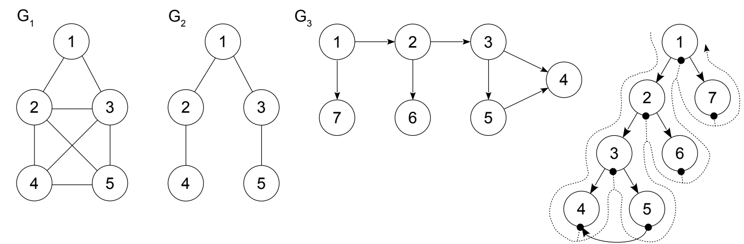


Abb. 1.1: Diagramme zu den Beispielgraphen G_1, G_2 und G_3.

Definition 2 (Die Komplexitätsklasse P) *Die **Zeitkomplexität** $T(n) \in \mathcal{O}(n^k)$ mit konstantem $k \in \mathbb{R}$ ist **polynomiell**. Die Menge aller Entscheidungsprobleme mit polynomieller Komplexität bezeichnet man mit **P**.*

Definition 3 (Die Komplexitätsklasse NP) *Die Klasse aller Entscheidungsprobleme, die von einer **nichtdeterministischen** Turingsmaschine bezüglich der Eingabelänge des Problems in Polynomialzeit gelöst werden bezeichnet man mit **NP**.*

Es gilt $P \subseteq NP$. Bis heute ist ungeklärt, ob $P = NP$, oder $P \neq NP$ wahr ist. Das P-NP Problem gehört zu den ungelösten *Millenium-Problemen.*

1.2 Graphentheorie

Die Grundlagen der **Graphentheorie** liefern ein besseres Verständnis wichtiger im Compilerbau eingesetzter Datenstrukturen. Die Graphentheorie ist ein Teilgebiet der Mathematik, welches die Eigenschaften von Graphen untersucht. Wir werden uns hier auf Grundlagen beschränken.

Definition 4 (Graph) *Ein Tupel $G = (V, E)$ bezeichnen wir als **Graph**. Dabei ist $V = \{v_1, v_2, \dots, v_n\}$ eine endliche Menge von **Knoten** und $E = \{e_1, e_2, \dots, e_m\}$ eine endliche Menge von **Kanten**. Dabei gelte $E \subseteq [V]_2$, also $e = (v_i, v_j)$.*

Wir motivieren einige Grundbegriffe anhand der Beispielgraphen $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$. Beide Graphen sind als **Diagramm** in der Abb. 1.1 dargestellt.

- $V_1 = V_2 = \{1, 2, 3, 4, 5\}$.
- $E_1 = \{(1, 2), (1, 3), (2, 3), (2, 4), (2, 5), (3, 4), (3, 5), (4, 5)\}$.
- $E_2 = \{(1, 2), (1, 3), (2, 4), (3, 5)\}$.
- **Ordnung** = Anzahl an Knoten: $n(G_1) = n(G_2) = 5$.
- **Größe** = Anzahl an Kanten: $m(G_1) = 8$, $m(G_2) = 4$.
- $P(1, 3) = \{1, 2, 4, 3\}$ ist ein **Pfad** von G_1; jedoch **kein kürzester Pfad**.
- G_2 ist ein **Baum**: a) $m(G_2) = n(G_2) - 1$. b) G_2 enthält **keine Kreise**. Beide Aussagen sind äquivalent.
- G_2 ist ein **Untergraph** von G_1; aber **kein induzierter Untergraph**.

Nun betrachten wir den **gerichteten azyklischen Graphen (DAG)** aus Abb. 1.1. Er enthält selbst keine gerichteten Kreise. Jedoch enthält der ungerichtete induzierte Subgraph einen Kreis mit den Knoten $\{v_3, v_4, v_5\}$. Wir möchten nun diesen Graphen, beginnend mit dem Knoten v_1 **durchlaufen**:

- **Breitensuche** (Breadth-first search, BFS):
  Wir nehmen an, dass jeder Knoten nur einmal durchlaufen wird. Wenn es mehr als eine Möglichkeit gibt einen Knoten zu wählen, dann wählen wir den Knoten mit dem kleinsten Index zuerst. Ergebnis für G_3: $1, 2, 7, 3, 6, 4, 5$.

- **Tiefensuche** (Depth-first search, DFS). Wir unterscheiden drei Verfahren:
  a) Durchlauf von G_3 in **Pre-order**: 1, 2, 3, 4, 5, 6, 7.
  b) Durchlauf von G_3 in **In-order:** 4, 3, 5, 2, 6, 1, 7.
  c) Durchlauf von G_3 in **Post-order**: 4, 5, 3, 6, 2, 7, 1.
  Man beachte zu In-Order auch die rechte Diagramm in Abb. 1.1

Graphfärbung

Für die optimale Vergabe von Registern werden wir in einem späteren Kapitel die Färbung von Graphen nutzen:

Definition 5 (Graphfärbung) *Sei $G = (V, E)$ ein einfacher Graph und*

$$f : V \to C$$

*eine Abbildung der Knotenmenge V auf eine Menge von Farben $C \subseteq \mathbb{N}_0$. Man fordert, dass alle benachbarten (adjazenten) Knoten eine unterschiedliche Farbe besitzen. Dann nennt man die Elemente aus C **Farben**. Eine Färbung, die höchstens k Farben nutzt, wird k-**Färbung** genannt.*

Definition 6 (Chromatische Zahl) *Die **chromatische Zahl** $\chi(G)$ beschreibt die kleinst mögliche Anzahl an Farben, die einen Graphen einfärben.*

Beispiele: Die chromatische Zahl für den vollständigen Graph K_n (jeder Knoten hat eine Kante zu jedem anderen Knoten) ist n. Für die Beispielgraphen von oben gilt: $\chi(G_1) = 3$, $\chi(G_2) = 2$.

Anmerkung: Das Färben von Graphen ist NP-vollständig; gehört also zu den hartnäckigen Problemen.

1.3 Syntax und Semantik

Bevor wir im nächsten Kapitel auf **formale Sprachen** zu sprechen kommen, geben wir nun eine Definition von **Syntax** und **Semantik** an.

Definition 7 (Syntax) *Als **Syntax** bezeichnet man im Allgemeinen die Anordnung von Worten, so dass wohlgeformte Sätze entstehen.*

Domäne	*Beispiele*
Businessprogrammierung	COBOL, Fortran
Datenbanken	SQL
Autonome Systeme	RobotML
Hardwaredesign	VHDL, Verilog
Parsergeneratoren	Yacc, GNU Bison
Textsatz	TeX, LaTeX
Numerik	Octave, Matlab
Grafische Programmiersysteme	LabVIEW, Simulink
Computergrafik	OpenGL Shading Language
Computerspiele	Game Maker Language

Tab. 1.1: Beispiele für domänenspezifische Sprachen.

Beispiel: Der Satz *"Colorless green ideas sleep furiously"* [6] ist ein korrekter Satz der englischen Sprache.

Definition 8 (Semantik) *Die **Semantik** ist ein Zweig der Linguistik und Logik, der sich mit der Bedeutung befasst.*

Eine konkrete Bedeutung der Phrase aus dem letzten Beispiel kann nicht angegeben werden.

1.4 Universalsprachen

Universalsprachen bzw. **Allzwecksprachen** (englisch General Purpose Language, GPL) können für **alle Anwendungsgebiete** eingesetzt werden. Sie unterstützen eine Vielzahl an Daten-, Kontroll- und Abstraktionsstrukturen. Sie sind **Turing-vollständig** und ermöglichen daher **universelle Berechenbarkeit.** Allerdings sind sie (für NichtinformatikerInnen), relativ **schwierig** zu erlernen und anzuwenden.

Beispiele für Allzwecksprachen sind C, C++, Lisp, Java, Python.

1.5 Domänenspezifische Sprachen

Eine **domänenspezifische Sprache** (englisch Domain-Specific Language (DSL) bezieht sich i.d.R. hingegen nur auf eine bestimmte **Anwendungsdomäne**, also auf ein Fachgebiet. Die Syntax ist so ausgerichtet, dass damit auch **Nichtinformatiker** Programme auf ihrem Gebiet implementieren können. Die Programme sind entsprechend **kurz** und **präzise**; weisen also eine gute **Ausdrucksstärke** auf.

Beispiele für domänenspezifische Sprachen sind in Tab. 1.1 aufgelistet.

"Eine Programmiersprache ist Lowlevel,
wenn die in ihr geschriebenen Programme Aufmerksamkeit
für das Irrelevante erfordern"

— Alan J. Perlis. Epigrams on Programming.
SIGPLAN Notices, 17(9):7-13,1982

Kapitel 2

Formale Sprachen

"Colorless green ideas sleep furiously"

— Noam Chomsky

Eine jede Sprache muss gewissen **Regeln** unterliegen. Wir betrachten zunächst den folgenden Satz der **natürlichen** Sprache.

The	student	attends	the	lecture	.
A	N	V	A	N	I

Eine erste Gliederung nach **Lexemen (Token)** erfolgt implizit durch die Leerzeichen. Weiterhin kann die **Wortart** für jedes **Lexem** bestimmt werden: { A := Artikel, N := Nomen, V := Verb, I := Interpunktion }. Offensichtlich lassen sich syntaktische Beziehungen zwischen den Wortarten finden: Zum einen besteht eine **Nominalphrase (NP)** aus je einem Artikel und einem Nomen. Zum anderen kann eine **Verbalphrase (VP)** als Konkatenation von Verb und Nominalphrase angegeben werden. Es lässt sich nun die folgende Baumstruktur gewinnen, welche in der Gesamtphrase P mündet. Diese setzt sich aus NP, VP und I zusammen. Man spricht von einem **Syntaxbaum**:

<table>
<tr><td>The</td><td>student</td><td>attends</td><td>the</td><td>lecture</td><td>.</td></tr>
<tr><td>A</td><td>N</td><td>V</td><td>A</td><td>N</td><td>I</td></tr>
<tr><td colspan="2">NP</td><td>V</td><td colspan="2">NP</td><td>I</td></tr>
<tr><td colspan="2">NP</td><td colspan="3">VP</td><td>I</td></tr>
<tr><td colspan="6">P</td></tr>
</table>

Anhand des Syntaxbaums[1] kann nun, beginnend von der Wurzel, eine endliche Menge an formalen Regeln definiert werden:

$$\begin{aligned} \{ \quad & P \to \text{NP VP I}, \quad \text{NP} \to \text{A} \quad \text{N}, \quad \text{VP} \to \text{V} \quad \text{NP}, \quad \text{V} \to \text{attends}, \\ & \text{N} \to \text{student}, \quad \text{N} \to \text{lecture}, \quad \text{A} \to \text{the}, \quad \text{I} \to . \quad \} \end{aligned} \tag{2.1}$$

Dabei werden { the, student, attends, the, lecture, . } als **Terminalsymbole** und { P, NP, VP, V, A, N, I } als **Nichtterminalsymbole** bezeichnet.

[1] Inder **Computerlinguistik** wird auch der Begriff **Parse Tree** verwendet.

2.1 Sprache

*Erste **Sprachfähigkeiten** des Menschen entwickelten sich ab ca. 300.000 v.Chr. Höhlenmalereien sind ab ca. 28.000 v.Chr. entstanden. Im Laufe der Jahrtausende entwickelte sich bereits eine implizite Benutzung von **Grammatik** in der gesprochenen Sprache. Ägyptische Hieroglyphen entstanden ab ca. 3.200 v.Chr. und waren zunächst eine reine **Bilderschrift**. Über die Jahrhunderte sind dann auch **Phonogramme**[2] und **Determinative**[3] hinzugekommen. **Übersetzungen** von Hand sind spätestens 196 v.Chr. durch den Stein von Rosetta bekannt, der den gleichen Text dreisprachig in Hieroglyphen, demotischem und altgriechischem Schriftsystem enthält. Während die **Verwirrung der Sprache** bereits im alten Testament über den Turmbau zu Babel metaphorisch dargelegt worden ist, kann man die Sprache des aus dem 15. Jahrhundert stammenden Voynich Manuskript bis heute nicht entschlüsseln. In den 1930er Jahren entwickelten Alonzo Church und Stephen Cole Kleene mit dem Lambda-Kalkül eine frühe **formale Sprache** für die Untersuchung von mathematischen Funktionen. Strukturelle Untersuchungen von formalen Sprachen gingen in den 1950er Jahren seitens Noam Chomsky so weit, dass die erarbeitete **Chomsky-Hierarchie** bis heute Bestand hat. Grace Hopper konzipierte schließlich 1949 den ersten **Übersetzer (Compiler).***

Zunächst arbeiten wir auf den Begriff der **formalen Sprache** hin.

Definition 9 (Alphabet) *Ein **Alphabet** Σ ist eine endliche, nicht-leere Menge von **Symbolen**.*

Zwei Beispiele sind $\Sigma_1 = \{0, 1\}$ und $\Sigma_2 = \{a, b, \ldots, z\}$.

Definition 10 (Wort) *Ein **Wort** w ist eine Folge von Symbolen eines Alphabets.*

Beispiele über die oben definierten Alphabete sind $w_1 = 0111000 \subseteq \Sigma_1^*$ und $w_2 = abcdeaef \subseteq \Sigma_2^*$.

Definition 11 (Länge eines Worts) *Die Länge eines Worts w ist definiert durch die Anzahl der Symbole aus denen das Wort besteht. Wir schreiben $|w|$.*

Beispiele sind $|w_1| = 7$ und $|w_2| = 8$.

Definition 12 (Potenzierung eines Alphabets) *Die Potenzierung eines Alphabets ist die Menge aller Symbolfolgen, die für eine gegebene Wortlänge k gebildet werden können.*

Beispiele: $k = 3 : \Sigma_1^3 = \{000, 001, 010, \ldots, 111\}$ und $k = 2 : \Sigma_2^2 = \{aa, ab, ac, \ldots, zz\}$.

Weiterhin definieren wir Σ^* als die **Menge aller Worte** über dem Alphabet Σ. Für die Menge aller Worte über Σ **ohne das leere Wort** ϵ, schreiben wir Σ^+. Also gilt $\Sigma^* = \Sigma^+ \cup \{\epsilon\}$.

[2] Schriftzeichen, das einen Sprachlaut repräsentiert.
[3] Stummes Zusatzzeichen in antiken Sprachen.

Definition 13 (Formale Sprache) *Sei Σ ein Alphabet. Dann wird $L \subseteq \Sigma^*$ formale Sprache über dem Alphabet Σ genannt. Wir definieren in Mengenschreibweise:*

$$L = \{w \in \Sigma^* |\ \textit{Aussagen über } w\ \}$$

Ohne weitere Einschränkungen können bisher nur Worte einer randomisierten Sprache ("Monkey Coding") generiert werden. Um Regeln anwenden zu können, wird im nächsten Abschnitt nun der Begriff der **formalen Grammatik** definiert.

2.2 Grammatik

Eine Grammatik **generiert** Worte einer Sprache. Das gegenläufige Modell sind **Automaten**. Ein Automat prüft, ob ein Wort w zur Sprache L gehört, also ob $w \in L$ wahr ist. Letzteres ist ein Entscheidungsproblem. Vgl. auch mit Abb. 2.1.

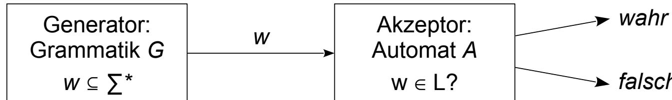


Abb. 2.1: Grammatiken erzeugen Worte w zu einer Sprache L. Automaten prüfen, ob ein gegebenes Wort zur Sprache gehört, also ob $w \in L$ wahr ist.

Definition 14 (Grammatik) *Eine* ***Grammatik*** *ist ein Quadrupel (4-Tupel) in der Form $G = (V, \Sigma, P, S)$ mit*

- einer endlichen Menge von **Variablen** V, mit der Einschränkung, dass diese disjunkt zu dem Alphabet benannt sein müssen: $V \cap \Sigma = \emptyset$.
- einem endlichen **Alphabet** Σ.
- einer endlichen Menge von **Produktionsregeln** $P \subseteq (V \cup \Sigma)^+ \times (V \cup \Sigma)^*$.
- einer **Startvariablen** $S \in V$.

Eine Regel $(u, v) \in P$ mit $u \in (V \cup E)^+$ und $v \in (V \cup \Sigma)^*$ kann auch kurz und einfach $u \to v$ geschrieben werden.

Beispiel: Formale Definition der Grammatik G_1 zu dem englischen Satz von oben:

$$\begin{aligned} G_1 = (V, \Sigma, P, S) = (&\{\mathrm{P}, \mathrm{NP}, \mathrm{VP}, \mathrm{A}, \mathrm{N}, \mathrm{I}\},\ \{\mathrm{the}, \mathrm{student}, \mathrm{attends}, \mathrm{lecture}, .\}, \\ &\{P \to \mathrm{NP\ VP\ I},\quad \mathrm{NP} \to \mathrm{A\ \ N},\quad \mathrm{VP} \to \mathrm{V\ \ NP}, \\ &\mathrm{V} \to \mathrm{attends},\quad \mathrm{N} \to \mathrm{student},\quad \mathrm{N} \to \mathrm{lecture},\quad \mathrm{A} \to \mathrm{the},\quad \mathrm{I} \to .\},\ P) \end{aligned}$$

2.3 Chomsky Hierarchie

Wir haben bisher beliebige Formen von Produktionsregeln betrachtet. Im Rahmen der natürlichen Sprachen sprechen wir von **Phrasenstrukturgrammatiken**. Formale Sprachen sind entgegen den natürlichen Sprachen eher präzise. Es stellt sich nun die Frage, ob es möglich ist, Grammatiken in die Typen $T_0, T_1, T_2, \ldots, T_n$ $(n \in \mathbb{N})$ nach absteigender Schwierigkeit zu kategorisieren. Können die Regeln einer Grammatik so eingeschränkt werden, dass disjunkte **Sprachklassen** gefunden werden können? Gibt es schwierige Klassen, die wir für unsere Zwecke einfach vernachlässigen können? Eine Antwort liefert die **Chomsky-Hierarchie**, welche Noam Chomsky 1956 in [5] beschrieben hat. Eine Sprache vom Typ $k+1$ hat dabei restriktivere Produktionsregeln als eine Sprache vom Typ k. Abbildung 2.2 stellt die Sprachklassen in einer Übersicht dar.

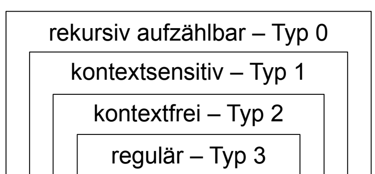


Abb. 2.2: Chomsky Hierarchie

Nun listen wir alle Sprachklassen zusammen mit ihren grammatikalischen Einschränkungen auf. Die Klasse $k+1$ erbt alle Einschränkungen von Klasse k:

- **Rekursiv aufzählbare Sprachen – Typ 0:**
  Alle Regeln haben die Form:
  $$\alpha \rightarrow \beta$$
  Dabei muss α aus mindestens einem Nichtterminalsymbol bestehen. Ansonsten sind für α, β beliebige Folgen von Terminalen und Nichtterminalen erlaubt.
  Zugehöriger Automat: Turingmaschine (TM).

- **Kontextsensitive Sprachen – Typ 1:**
  Alle Regeln haben die Form:
  $$\alpha A \beta \rightarrow \alpha \gamma \beta$$
  Das Nichtterminal A wird auf eine Folge aus Terminalen und Nichtterminalen γ abgeleitet. Man sieht leicht, dass der *Kontext*, also die konkrete Einbettung in α, β relevant ist.
  Zugehöriger Automat: Linear beschränkter Automat (LBA).

- **Kontextfreie Sprachen – Typ 2**:
  Alle Regeln haben die Form:
  $$A \rightarrow \gamma$$
  Die Einbettung ist also nicht mehr von Bedeutung.
  Zugehöriger Automat: Kellerautomat.
  Relevanz für den Compilerbau: Syntaktische Analyse / Parsing.

- **Reguläre Sprachen – Typ 3**:
  Alle Regeln haben die Form:
  $$A \to a$$
  oder:
  $$A \to aB$$
  Dabei sind A, B Nichtterminale und a Terminale.
  Zugehöriger Automat: Endlicher Automat.
  Relevanz für den Compilerbau: Lexikalische Analyse / Scanning.

Für den Compilerbau sind also die Sprachen vom Typ 3 und vom Typ 2 von besonderer Bedeutung.

2.4 Reguläre Sprachen (Typ 3)

Die einfachste Sprachklasse sind die **regulären Sprachen**. Sie werden von einem **endlichen Automat**en akzeptiert. Alle Produktionsregeln haben die Form $A \to a$ bzw. $A \to aB$, wobei A, B Nichtterminalsymbole und a Terminalsymbole sind.

Beispiel:

- Sprache L und Grammatik G:

$$L \subseteq \sum^* = \{0,1\}^*$$
$$L = \{\ 0^l\ 1\ 0^m\ 1\ 0^n \mid l,m,n \geq 0\ \wedge\ l,m,n \in \mathbb{N}\}$$
$$G = (V, \sum, P, S) = (\ \{q_0, q_1, q_2\},\ \{0,1\},\ P,\ q_0)$$
$$P = \{\ q_0 \to 0\ q_0,\ q_0 \to 1q_1,\ q_1 \to 0\ q_1,\ q_1 \to 1\ q_2,\ q_2 \to 0q_2,\ q_2 \to 0\ \}$$

- Akzeptierender endlicher Automat (siehe Abb. 2.3):

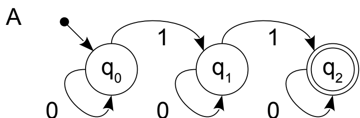


Abb. 2.3: Deterministischer endlicher Automat (DEA)

Satz 15 *Jede Sprache die durch einen **endlichen Automaten** akzeptiert wird ist eine **reguläre Sprache.***

2.4.1 Endliche Automaten

Als Akzeptoren für reguläre Sprachen unterscheiden wir die beiden Typen **Deterministische Endliche Automaten (DEA)** und **Nichtdeterministische Endliche Automaten (NEA)** [10].

Deterministische endliche Automaten

Definition 16 (Deterministischer Endlicher Automat) *Ein **deterministischer endlicher Automat (DEA)** wird formal als Quintupel*

$$A = (Q, \Sigma, \delta, q_0, F)$$

definiert mit

- einer endlichen **Zustandsmenge** Q .
- einer endlichen Menge an Eingabesymbole (**Alphabet**) Σ .
- einer **Übergangsfunktion** $\delta : Q \times \Sigma \to Q$.
- einem **Startzustand** $q_0 \in Q$ *(eingehender Pfeil).*
- einer Menge von **Endzuständen** $F \subseteq Q$ *(Doppelt umrandete Kreise).*

Beispiel: $A = (\{q_0, q_1, q_2, q_3\}, \{0, 1\}, \delta, \{q_3\})$. **Akzeptierte Worte** sind zum Beispiel $w_1 = 000010010111$, $w_2 = 101$, $w_3 = 011111$. Die **Übergangsfunktion** δ können wir entweder als Tabelle oder als Diagramm darstellen. Vgl. dazu Tab. 2.1 und Abb. 2.4.

δ	0	1
q_0	q_0	q_1
q_1	q_2	q_2
q_2	q_0	q_3
q_3	q_3	q_3

*Tab. 2.1: Tabellarische Übergangsfunktion des deterministischen endlichen Automaten aus Abb. 2.4. In der linken Spalte steht der aktuelle Zustand. In der ersten Zeile steht das Eingabesymbol. Das Beispiel ist **vollständig** definiert, d.h. es existiert für jede Kombination aus Zustand und Eingabesymbol ein Folgezustand.*

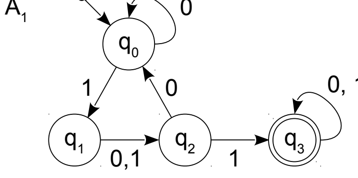


Abb. 2.4: Deterministischer endlicher Automat (DEA)

Anmerkung: Die Zustände dürfen auch anders als $q_0, q_1, \ldots$ benannt werden.

Nichtdeterministische endliche Automaten

Wir beobachten, dass die Zustandsfolge von deterministischen Automaten immer **eindeutig** ist. Zum Beispiel akzeptiert der Automat aus Abb. 2.4 das Wort $w = 111$ alternativlos mit der Zustandsfolge q_0, q_1, q_2, q_3. Nun betrachten wir den **nichtdeterministischen endlichen Automaten** in Abb. 2.5. Es gibt nun eine **Menge** an möglichen Folgezuständen, wenn der aktuelle Zustand q_2 und das Eingabesymbol 0 ist:

$$\delta(q_2, 0) = \{q_0, q_3\}$$

Die tabellarische Übergangsrelation findet man in Tab. 2.2.

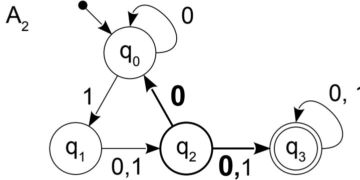


Abb. 2.5: Nichtdeterministischer endlicher Automat (NEA)

Δ	0	1
q_0	$\{q_0\}$	$\{q_1\}$
q_1	$\{q_2\}$	$\{q_2\}$
q_2	$\{q_0, q_3\}$	$\{q_3\}$
q_3	$\{q_3\}$	$\{q_3\}$

Tab. 2.2: Tabellarische Übergangsrelation des nichtdeterministischen endlichen Automaten aus Abb. 2.5

Definition 17 (Nichtdeterministischer Endlicher Automat) *Ein* ***nichtdeterministischer endlicher Automat (NEA)*** *wird formal als Quintupel*

$$A = (Q, \Sigma, \Delta, q_0, F)$$

definiert mit

- einer endlichen **Zustandsmenge** Q .
- einer endlichen Menge an Eingabesymbole (**Alphabet**) Σ .
- einer **Übergangsrelation** $\Delta : Q \times \Sigma \rightarrow \wp(Q)$.
- einem **Startzustand** $q_0 \in Q$ *(eingehender Pfeil).*
- einer Menge von **Endzuständen** $F \subseteq Q$ *(Doppelt umrandete Kreise).*

Es stellt sich nun die Frage, ob ein NEA mächtiger ist, als ein DEA. Dies ist **nicht** der Fall. Ein Beweis ist über die **Potenzmengenkonstruktion** möglich. Wir erweitern Q zu $\wp(Q)$. Beispiel mit 4 Zuständen:

$$\wp(Q) = \{\emptyset, q_0, q_1, q_2, q_3, q_{01}, q_{02}, \ldots, q_{0123}\}$$

Weiterhin werden auch die Übergänge vereinigt. Der neue Automat ist dann **deterministisch** und **äquivalent**.

Einfaches Beispiel: Abb. 2.6 zeigt auf der linken Seite einen NEA mit 2 Zuständen. Auf der rechten Seite ist ein Diagramm eines dazu äquivalenten DEA mit $|\wp(Q)| = 2^{|Q|} = 2^2 = 4$ Zuständen dargestellt.

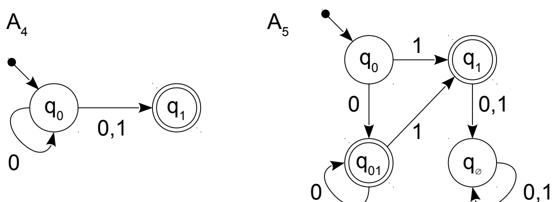


Abb. 2.6: Potenzmengenkonstruktion

2.4.2 Grenzen regulärer Sprachen

Welche **Grenzen** haben reguläre Sprachen? Wir betrachten das Beispiel der **geklammerten Ausdrücke** / Dyck-Sprache. Man beachte, dass die runden Klammern hier Terminale sind!

$$L = \{w = (^n\ )^n \mid n \geq 0\}$$
$$G = (V, \sum, P, S) = (\{A\},\ \{(,)\},\ P,\ A)$$
$$P = \{A \to (\ A\ ),\quad A \to \epsilon\}$$

Zum Beispiel ist das Wort $w = (((())))$ ein Wort dieser Sprache. Offensichtlich haben die Produktionsregeln nicht die Form $A \to a$ bzw. $A \to aB$. Ist es eventuell möglich, die Regeln so umzuschreiben, dass die Grammatik regulär ist? Alternativ ist auch die erfolgreiche Konstruktion eines endlichen Automaten für den Beweis dienlich. Hierzu betrachten wir die Abbildung 2.7. Die Konstruktion kann nur gelingen, wenn unser Automat **unendlich** viele Zustände hat. Ein Widerspruch zur Definition eines **endlichen** Automaten.

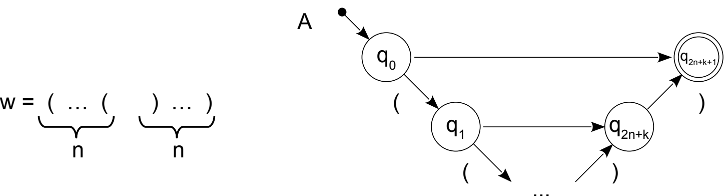


Abb. 2.7: Unendlicher Automat für die Sprache der geklammerten Ausdrücke.

□

Das Beispiel der geklammerten Ausdrücke impliziert, dass Programmiersprachen (und folglich auch domänenspezifische Sprachen) mindestens vom Typ 2 sein müssen. Tatsächlich sind **kontextfreie Sprachen** bereits für alle unsere Zwecke hinreichend!

2.5 Kontextfreie Sprachen (Typ 2)

Eine **Typ-2** Sprache wird auch **kontextfreie Sprache** genannt. Sie wird durch einen **Kellerautomaten** akzeptiert. Alle Produktionsregeln haben die Form $A \rightarrow \gamma$, mit einem Nichtterminalsymbol A und einer Folge von Terminal- und Nichtterminalsymbolen γ.

Satz 18 *Jede Sprache die durch einen **Kellerautomaten** akzeptiert werden kann, ist eine **kontextfreie Sprache.***

2.5.1 Kellerautomaten

Bei den geklammerten Ausdrücken im letzten Beispiel wurden unendlich viele Zustände benötigt. Stattdessen hätten wir jedoch auch in einem **Speicher** die Anzahl der geöffneten Klammern merken können. Dann wäre die Anzahl der Zustände wieder endlich. Ein **Kellerautomat (Push-Down Automaton)** bietet genau diese Möglichkeit. Er verfügt über einen **Stackspeicher** (Stack=Keller), der die beiden Operationen *push()* und *pop()* unterstützt:

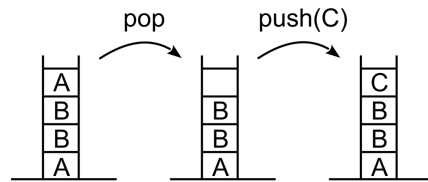


Abb. 2.8: Prinzip eines Stackspeichers.

Definition 19 (Kellerautomat) *Ein **Nichtdeterministischer Kellerautomat (NKA)** wird formal als Septupel (7-Tupel) $M = (Q, \Sigma, \Gamma, \delta, q_0, Z, F)$ definiert mit*

- einer endlichen Menge von **Zuständen** Q.
- einem endlichen **Eingabealphabet** Σ.
- einem endlichen **Kelleralphabet** Γ.
- einer **Übergangsrelation** $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \rightarrow \mathcal{P}(Z \times \Gamma^*)$.
- einem **Startzustand** $q_0 \in Q$.
- einem **initialen Kellersymbol** $z \in \Gamma$.
- einer Menge **akzeptierender Zustände** $F \subseteq Q$.

Ein **Übergang** wird wie folgt definiert:

$$(p, a, A, q, \alpha) \in \delta \tag{2.2}$$

mit

- dem **aktuellen Zustand** $p \in Q$.
- der **aktuellen Eingabe** $a \in \Sigma \cup \{\epsilon\}$.
- dem **obersten Kellersymbol** $A \in \Gamma$.
- den **auf den Stack zu legenden Symbolen** $\alpha \in \Gamma^*$.

Beispiel 1:

$$\begin{aligned}
&A_1 = (\{q_0, q_1\},\ \{0,1\},\ \{A, B\},\ \delta,\ q_0,\ \emptyset,\ \{q_1\}) \\
&\delta_0 = (q_0, 0, B, q_1, A), \quad \delta_0 \in \delta \\
&\delta_1 = \dots \\
&\dots
\end{aligned}$$

Angenommen die folgende **Konfiguration** liegt vor: Der aktuelle Zustand ist q_0, das nächste Eingabesymbol ist 0 und der Kellerspeicher enthält AB (linke Abbildung). Dann wenden wir die Regel δ_1 an. Der neue Zustand ist dann q_1. Vgl. mit Abbildung 2.10.

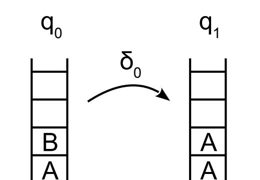


Abb. 2.9: Erstes Beispiel zu Kellerautomaten.

Beispiel 2: Nun betrachten wir ein weiteres Beispiel, welches gerade vom Typ 2, aber nicht vom Typ 3 ist:

$$L = \{0^n 1^n \mid n \geq 1\}$$

Beispielworte dieser Sprache sind

$$w \in L: \ w_1 = 01, w_2 = 0011, w_3 = 000111, \dots$$

Die zugehörige Grammatik sieht folgendermaßen aus:

$$\begin{aligned}
&G = (V, \Sigma, P, S) = (\{S\}, \{0,1\}, P, S) \\
&P = \{S \rightarrow 0\ S\ 1, \quad S \rightarrow 0\ 1\}
\end{aligned}$$

Im Rahmen der Modellierung des Kellerautomaten bedienen wir uns eines Mechanismus zum “Merken” des ersten Auftretens von n. Wir konstruieren:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z, F) = (\{q_0, q_1, q_2\},\ \{0, 1\}, \{a, b\}, \delta, q_0, b, \{q_2\})$$
$$\delta_1(q_0, 0, b) = (q_0, ab).$$
$$\delta_2(q_0, 0, a) = (q_0, aa).$$
$$\delta_3(q_0, 1, a) = (q_1, \epsilon).$$
$$\delta_4(q_1, 1, a) = (q_1, \epsilon).$$
$$\delta_5(q_1, 1, b) = (q_2, \epsilon).$$

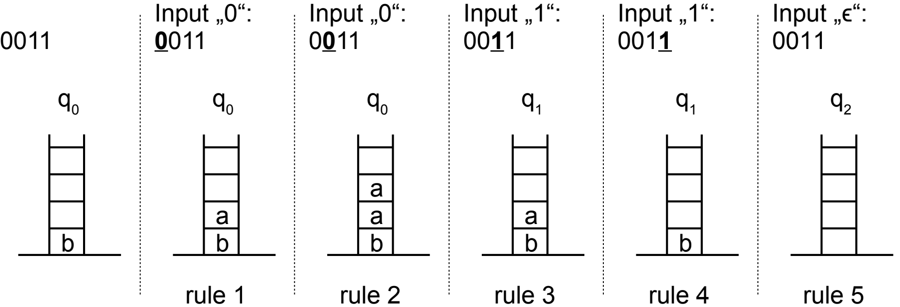


Abb. 2.10: Zweites Beispiel zu Kellerautomaten.

Wir betrachten nun die einzelnen Akzeptierungsschritte des Beispielwortes $w_1 = 0011$. Einfach ausgedrückt:

- Jedes Eingabesymbol 0 legt das Kellersymbol a auf den Keller. Jedes Eingabesymbol 1 löscht das oberste Kellerelement.
- Das Wort wird schließlich per leerem Speicher, sowie Endzustand q_2 **akzeptiert**.

2.5.2 Deterministisch kontextfreie Sprachen

Im Rahmen des **Parsens** (vgl. Kapitel 3.4) sind wir insbesondere an **deterministisch kontextfreien Sprachen (DKF)** interessiert. So können Parser möglichst einfach gehalten werden. Wir werden später beobachten, dass die Klasse DKF identisch zur Klasse $LR(k)$ ist. Nun muss noch der Begriff **Determinismus** für kontextfreie Sprachen definiert werden. Wir beginnen zunächst exemplarisch:

Definition 20 (Palindrom) *Ein **Palindrom** ist ein Wort w, welches identisch mit seiner Umkehrung ist, d.h. $w = w^R$.*

Dabei steht R für **revers** oder rückwärts. Beispiele für Palindrome sind – abgesehen von der Groß- und Kleinschreibung – *Rentner*, *level*, *racecar* oder die Binärdarstellung der Zahl $73_{10} = 1001001_2$ [4]. Um die Modellierung zu vereinfachen, beschränken

[4] “73 is the 21st prime number. Its mirror, 37, is the 12th and its mirror, 21, is the product of multiplying 7 and 3 ... and in binary 73 is a palindrome, 1001001, which backwards is 1001001.” [S.C. aus TBBT]

wir unser Alphabet auf $\Sigma = \{0,1\}$. Die Sprache der binären Palindrome wird definiert durch:

$$L_{Palin.} = \{v\ v^R \mid v \in \{0,1\}^n,\ n \in \mathbb{N}\}$$

Was ist nun der Unterschied zwischen den folgenden beiden Sprachen?

$$L = \{0^n 1^n \mid n \in \mathbb{N}\}$$
$$L_{Palin.} = \{v\ v^R \mid v \in \{0,1\}^n,\ n \in \mathbb{N}\}$$

Wir müssen annehmen, dass die Wortlänge $|w|$ nicht *à priori* bekannt ist! Entsprechend ist nicht bekannt, zu welchem **Zeitpunkt** der erste Teil des Palindroms bereits akzeptiert wurde, und die Mechanismen des Kellerautomaten für die Akzeptanz des zweiten Teilwortes umgestellt sein müssen. Da dies nicht bekannt ist, ist $L_{Palin.}$ **nicht deterministisch.**

- Der erste Teil **füllt** den Kellerspeicher
- Der zweite Teil **leert** den Kellerspeicher

Satz 21 *Die Menge der* ***deterministisch kontextfreien Sprachen (DKF)*** *ist eine* ***echte Teilmenge*** *($\subset \neq \subseteq$) der Menge der kontextfreien Sprachen.*

Eine DKF-Sprache wird durch einen deterministischen Kellerautomaten akzeptiert, dessen Anzahl an Nachfolgezuständen immer genau eins ist. Abb. 2.11 zeigt nun die Chomsky-Hierarchie in erweiterter Form.

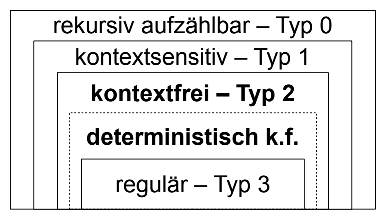


Abb. 2.11: Erweiterte Chomsky Hierarchie

Ausblick: Wir stellen fest, dass Variationen von Palindromen sehr wichtig für die Implementierung von Programmiersprachen sind; insbesondere für das *Matching* sich öffnender und schließender Geltungsbereiche.

2.6 Metasprachen zur Darstellung von Grammatiken

Wir verlangen nach einer **kompakten** und präzisen **Schreibweise** für kontextfreie Sprachen. Idealerweise soll diese Notation später ohne großen Aufwand in eine für den Compilerbau geeignete Darstellung überführt werden können. Die nachfolgenden Abschnitte führen die **Backus Naur Form (BNF)** und die **Erweiterte Backus Naur Form (EBNF)** ein.

2.6.1 Backus Naur Form (BNF)

Die **Backus Naur Form (BNF)** ist eine **Metasprache** um Typ 2 Sprachen zu beschreiben. Tabelle 2.3 listet die syntaktischen Elemente auf.

<A>	Nichtterminal
a	Terminal
|	Alternative
::=	Definition
.	Regelende

Tab. 2.3: Syntaktische Elemente der Backus Naur Form (BNF).

Das nachfolgende **Beispiel** listet eine BNF Grammatik für die Beschreibung von Ganzzahlen auf:

```
1 <digit>    ::= 1 | 2 | ... | 9.
2 <digit0>   ::= 0 | <digit>.
3 <positive> ::= <digit> | <positive> <digit0>.
4 <integer>  ::= 0 | <positive> | - <positive>.
```

2.6.2 Erweiterte Backus Naur Form (EBNF)

Die **Erweiterte Backus Naur Form (EBNF)** wurde von Niklaus Wirth entwickelt, um die Syntax der Programmiersprache *Pascal* zu beschreiben. Tabelle 2.4 listet wieder die syntaktischen Elemente auf.

a	Nichtterminal
“a”	Terminal
|	Alternative
=	Definition
;	Regelende
,	Aufzählung
[...]	Optionaler Teilausdruck
{ ... }	Sich beliebig oft wiederholender Teilausdruck
(...)	Gruppierung eines Teilausdrucks
(* ... *)	Kommentierung

Tab. 2.4: Syntaktische Elemente der Erweiterten Backus Naur Form (EBNF).

Das nachfolgende **Beispiel** liefert eine EBNF Grammatik für die Beschreibung von Ganzzahlen:

```
1 digit = "1" | "2" | ... | "9";
2 digit0 = "0" | digit;
3 integer = "0" | [ "-" ], digit, { digit0 };
```

Man sieht leicht, dass die Formulierungen in EBNF kompakter sind, als in BNF.

Satz 22 *“Eine Sprache ist regulär genau dann, wenn dessen Syntax durch einen einzelnen EBNF Ausdruck dargestellt werden kann” [23]*

2.7 Attribute und Aktionen

Als Ausblick auf das nächste Kapitel betrachten wir nun zwei Aspekte, die einen einfachen **Parser** zu einem echten **Übersetzer** ausbauen.

2.7.1 Attributierte Grammatiken

Die meisten Programmiersprachen unterstützen eine große Menge an Datentypen. Würde man nun für jeden Datentyp eigene Produktionsregeln definieren, so würde die Grammatik schnell unübersichtlich und nicht mehr wartbar. **Attributierte Grammatiken** erlauben es, die Regeln **generisch** zu halten. Das Hinzufügen von **Typregeln** für jede Produktionsregel ist eine Möglichkeit, eine Grammatik zu attributieren. In EBNF kann dies wie in Tab. 2.5 gezeigt aussehen. In dem Beispiel

Syntax	Attributregel	Bedingung
`exp(`T_0`) =`		
`term(`T_1`)`	T_0 `:=` T_1	
`| exp(`T_1`), "+", term(`T_2`)`	T_0 `:=` T_1	T_1 `==` T_2
`| exp(`T_1`), "-", term(`T_2`)`	T_0 `:=` T_1	T_1 `==` T_2

Tab. 2.5: Beispiel für eine attributierte Grammatik mit Typregeln

besteht die rechte Seite der Syntax-Regel aus 3 Alternativen:

- `term(`T_1`)`:
  Der Rückgabetyp T_0 ist in diesem Fall gleich dem Typ T_1 des Terms und kann so zurückgegeben werden.

- `exp(`T_1`) "+" term(`T_2`)`:
  Wir setzen für eine Addition die **Bedingung** voraus, dass beide Operanden vom gleichen Typ sind, also T_1==T_2 (z.B. vom Typ *Integer*). Der Rückgabetyp T_0 ist hier T_1. Alternativ dürfte die Attributregel auch T_0 `:=` T_2 lauten.

  In der Praxis treten meist komplexere Bedingungen auf. Schließlich erlauben es die meisten Programmiersprachen[5] zum Beispiel auch Operanden zu addieren, die von verschiedenen Datentypen sind, wie zum Beispiel Integer und Double.

- `exp(T1) "-" term(T2)`:
  (äquivalent zur vorherigen Zeile)

2.7.2 Emittieren von Aktionscode

Um nachfolgende Kompilationsschritte anzusteuern, wird einer Grammtik **Aktionscode** hinzugefügt. Dies ist das gängige Procedere im Parsergenerator *GNU Bison*. Parsergeneratoren wie z.B. *ANTLR* setzen dafür einen **Listener** ein. Aus theoretischer Sicht werden wir dazu die bisher kennengelernten Automaten zu **Transduktoren** erweitern.

[5] Ausnahmen sind sehr stark typisierte Sprachen wie z.B. VHDL.

2.8 Transduktoren

Wir erinnern wir uns an die endlichen Automaten für die Akzeptanz von Worten einer regulären Sprache. Es wird nun nach einem Automatentyp gesucht, der nicht nur Eingabeworte verarbeitet kann, sondern auch **Ausgabenworte** erzeugt.

Das Konzept der **Transduktoren** korrespondiert zu den Mealy-Automaten, die z.B. im Bereich *Embedded Systems* häufig eingesetzt werden. Mealy-Automaten sind immer **deterministisch.**

Definition 23 (Endlicher Transduktor) *Ein* ***endlicher Transduktor*** *ist ein* ***erweiterter endlicher Automat*** *in Form eines Septuples*

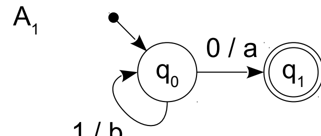
$$A = (Q, \Sigma, \Gamma, q_0, \delta, F, \omega)$$

, welchem die folgenden (neuen) Komponenten zugefügt werden:

1. Ein endliches **Ausgabealphabet** Γ.
2. Eine **Ausgabefunktion** $\omega : \ Q \ \times \ \Sigma \cup \{\epsilon\} \ \times \ Q \ \rightarrow \ \Gamma^*$.

Beispiel:

$$A_1 = (\{q_0, q_1\},\ \{0, 1\},\ \{a, b\},\ q_0,\ \delta,\ \{q_1\},\ \omega)$$



Das Scannen des Eingabewortes $w_1 = 1110$ **generiert** das Ausgabewort *bbba*.

Transduktoren werden zum Beispiel im Scannergenerator *Flex* eingesetzt. Im Rahmen der lexikalischen Analyse werden aus dem Eingabecode Token generiert, die später beim Parsen weiterverarbeit werden können.

Kapitel 3

Compilerbau

"Developing a compiler was a logical move."

— Grace Hopper

Definition 24 (Compiler) *Ein Compiler ist eine **Softwarekomponente**, die ein in einer **Eingabesprache** A geschriebenes Programm in eine **Zielsprache** B übersetzt, und dabei die **Semantik nicht verändert.** Für gewöhnlich ist die Zielsprache B näher an eine Rechnerarchitektur angelehnt, als die Eingabesprache A.*

Ein Compiler analysiert **Syntax** und **Semantik** von Ausdrücken der Eingabesprache und übersetzt diese in die Zielsprache. Ein Compiler besteht i.d.R. aus einem **Frontend**, einem **Middleend** und einem **Backend** (vgl. mit Abb. 3.1).

Achtung: Frontend und Backend dürfen hier nicht mit dem Client-/Servermodell verwechselt werden.

3.1 Einführung

Im Folgenden werden die einzelnen Phasen kurz umrissen. Ausführliche Informationen findet man ab Kapitel 3.2.

Frontend

- Die **Lexikalische Analyse** übersetzt eine Folge von Eingabezeichen in eine Sequenz von Terminalsymbolen.

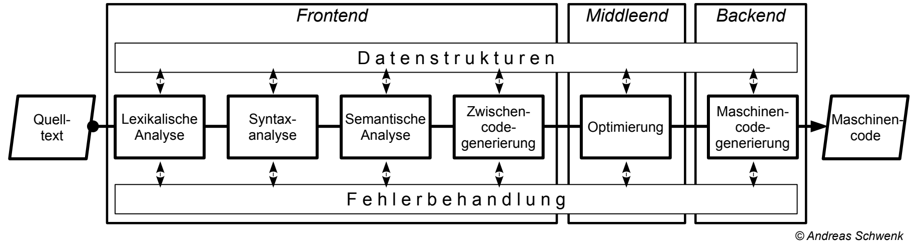


Abb. 3.1: Phasen eines Compilers

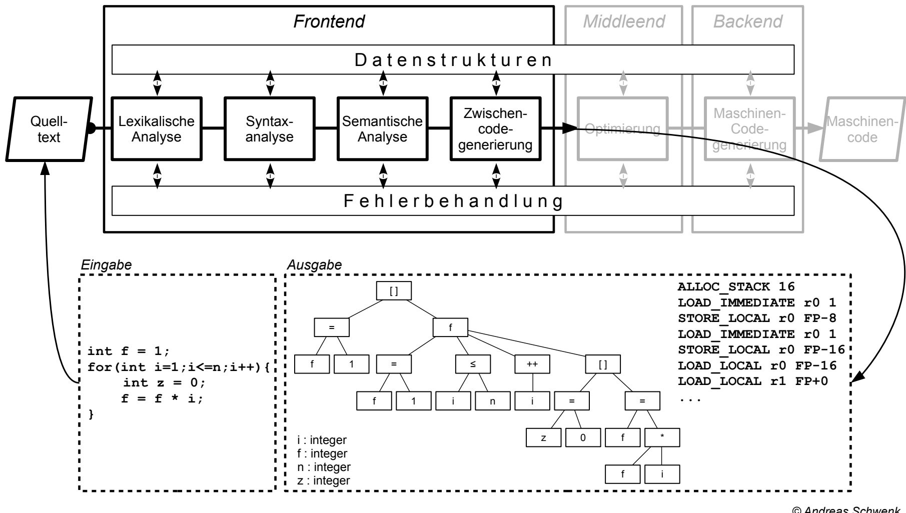


Abb. 3.2: Compiler Frontend

- In der Phase der **syntaktischen Analyse** werden die Symbole unter Anwendung einer **Grammatik** geparst. Dabei wird der **abstrakte Syntaxbaum** (Abstract Syntax Tree, AST) erstellt. Beispiel: siehe unten rechts in Abb. 3.2.

- Die **semantische Analyse** überprüft zum Beispiel Variablen auf ihre Deklaration und führt eine Typüberprüfung durch. Eine **Symboltabelle** verwaltet die zu den Bezeichnern bekannte Datenbasis.

- Schließlich wird ein von der Zielarchitektur unabhängiger **Zwischencode** (Intermediate Code, IC) generiert.

Middleend

- Der Zwischencode wird im Rahmen der **Zwischencodeoptimierung** zum Beispiel hinsichtlich einer schnellen Ausführbarkeit optimiert. Diese Phase ist bei sehr einfachen Compilern ggf. nicht vorhanden.

Eine Übersicht über das Compiler **Frontend** findet man in Abb. 3.2.

Backend

- Aus dem Zwischencode wird schließlich **Maschinencode** erzeugt, der auf der Zielhardware (real oder virtuell) lauffähig ist. Der Maschinencode wird nochmals auf eine hohe Ausführungsgeschwindigkeit und/oder eine kleine Programmgröße hin optimiert. Zum Beispiel kann eine Parallelisierung vorgenommen werden.

Eine Übersicht über das Compiler **Backend** findet man in Abb. 3.3.

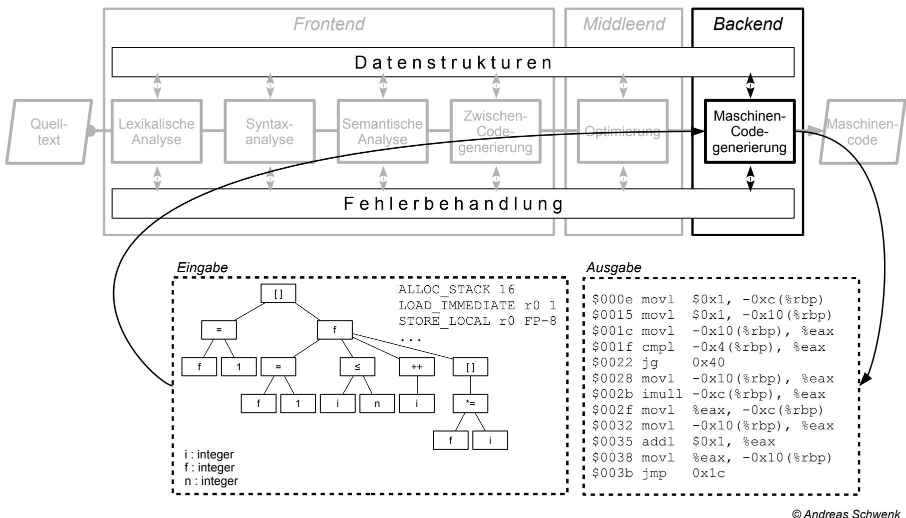


Abb. 3.3: Compiler Backend

Tombstone Diagramme

T-Diagramme (vgl. mit Abb. 3.4) beschreiben die Übersetzung einer **Quellensprache** (links) in eine **Zielsprache** (rechts). Die Sprache in welcher der Compiler geschrieben ist (unten), wird **Implementierungssprache** genannt. Der rechte Teil

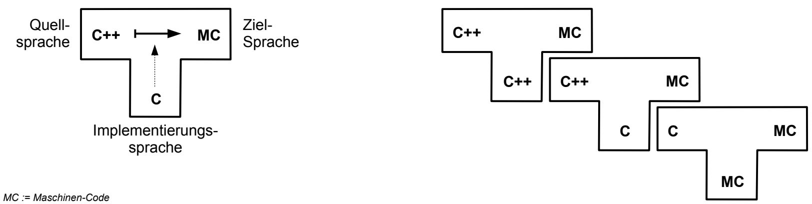


Abb. 3.4: Beispiele für T-Diagramme

aus Abb. 3.4 veranschaulicht den Prozess des **Bootstrapping**: Die **Selbstkompilierung eines Compilers** (also folglich die Übersetzung einer Eingabesprache, die auch gleichzeitig die Implementierungssprache darstellt), ist für neue Sprachen zunächst nicht möglich, sodass weitere "Schichten" nötig sind.

Zum Beispiel benötigte man für die Programmiersprache C++ in den 80er Jahren zunächst einen in der Sprache C geschriebenen Compiler. Zudem hatte man den C++ Code zunächst in C Code übersetzt.

3.2 Datenstrukturen

Die folgenden beiden **Datenstrukturen** sind wesentlich für den Compilerbau und werden sehr stark in den Phasen der syntaktischen Analyse, semantischen Analyse und Zwischencodegenerierung eingesetzt.

3.2.1 Abstrakter Syntaxbaum (AST)

Definition 25 *Ein* ***Abstrakter Syntaxbaum*** *(Abstract Syntax Tree, AST) ist eine Datenstruktur in Form einer graphischen Baumstruktur, die alle syntaktischen Elemente des Eingabeprogramms einbezieht. Die unterliegende Grammatik ist meist kontextfrei.*

Häufig erkennt man ein eingängiges Mapping von Produktionsregeln auf die Elemente des AST. Vgl. dazu Abb. 3.2 unten rechts. Der Begriff **abstrakt** rührt daher, dass Teile der Grammatik nicht direkt, sondern implizit einbezogen werden. Beispielsweise werden im AST oft Klammern weggelassen. Letztere können in der weiteren Verarbeitungskette anhand der gelabelten Knoten zur Laufzeit leicht rekonstruiert werden. Ein Syntaxbaum kann auch **textuell** angegeben werden. In Abb. 3.5 findet man dazu ein durch den *Iom*-Compiler erzeugtes Beispiel.

```
SYM_FUNCTION: id='factorial' : TYPE_INT
    SYM_FUNCTION_PARAM: id='x' : TYPE_INT
    SYM_FUNCTION_BEGIN:
    SYM_ASSIGNMENT: id='='
        SYM_LOCAL_VARIABLE: id='res' : TYPE_INT
        SYM_CONST_INTEGER: id='1' value='1' : TYPE_INT
    SYM_FOR_LOOP: id='for'
        SYM_LOCAL_VARIABLE: id='i' : TYPE_INT
        SYM_CONST_INTEGER: id='1' value='1' : TYPE_INT
        SYM_FUNCTION_PARAM: id='x' : TYPE_INT
        SYM_ASSIGNMENT: id='='
            SYM_LOCAL_VARIABLE: id='res' : TYPE_INT
            SYM_MULTIPLY: id='*'
                SYM_LOCAL_VARIABLE: id='res' : TYPE_INT
                SYM_LOCAL_VARIABLE: id='i' : TYPE_INT
    SYM_RETURN: id='return'
        SYM_LOCAL_VARIABLE: id='res' : TYPE_INT
```

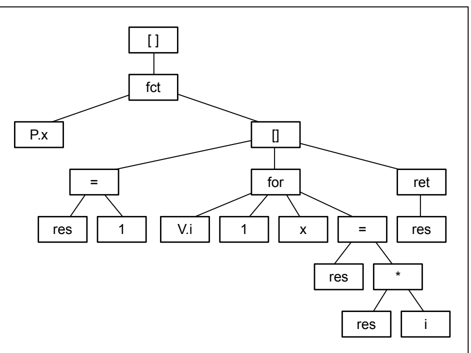


Abb. 3.5: Textuelle und graphische Darstellung eines abstrakten Syntaxbaums.

3.2.2 Symboltabelle

Wir werden **Symboltabellen** in Kapitel 3.5.2.1 ausführlich behandeln. Im wesentlichen ist eine Symboltabelle eine Datenstruktur, die zu jedem im Eingabeprogramm auftretenden **Bezeichner** die Gesamtheit der dazu bekannten Informationen ablegt. Beispiele für Bezeichner sind Klassennamen, Funktionsnamen, Variablennamen usw. Zum Beispiel enthält die Symboltabelle die Adressen von Variablen oder den Datentyp für eine semantische Prüfung.

3.3 Lexikalische Analyse / Scanning

Die **lexikalische Analyse** unterteilt eine Folge von Eingabezeichen in logische Einheiten, den so genannten **Token**. Die Zerlegung erfolgt nach den Regeln einer **regulären Grammatik** (Typ 3). Den hierzu eingesetzten Softwarepart nennt man **Scanner, Lexer** oder **Tokenizer.** Dieser implementiert eine Menge von **endlichen Automaten**. Das folgende Beispiel zeigt einen **Transduktor**, welcher die folgende in EBNF angegebene reguläre Sprache akzeptiert:

```
1 statement = "for" | "int" | "if";
```

$$\begin{aligned} A_1 &= (Q, \Sigma, \Gamma, q_0, \delta, F, \omega) \\ &= (\{q_0, q_1, \ldots, q_7\}, \{f, i, n, o, r, t\}, \{\text{TK_FOR}, \text{TK_IF}, \text{TK_INT}\}, \\ &\quad q_0, \delta, \{q_3, q_6, q_7\}, \omega) \end{aligned}$$

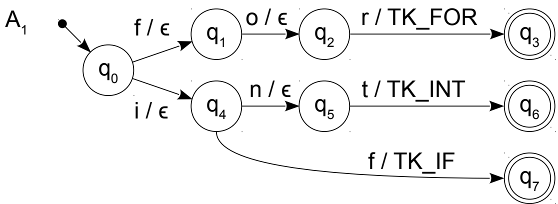


Abb. 3.6: Transduktor

Lexikalische Analyse mit dem Iom-Compiler Nun betrachten wir ein praktisches Beispiel, das mit dem *Iom*-Compiler (vgl. Anhang B) erzeugt wurde. Als Eingabeprogramm sei das folgende Listing gegeben:

```
1 + factorial(x : int) : int {
2   res = 1;
3   for i in [1,x]
4     res = res * i;
5   return res;
6 }
```

Der Compiler erzeugt nun die folgende Sequenz aus Token:

- Zeile 1: { PLUS, ID, LEFT_PARENTHESIS, ID, COLON, INT, RIGHT_PARENTHESIS, COLON, INT, LEFT_BRACE }.
- Zeile 2: { ID, ASSIGN, CONST_INT, SEMICOLON }.
- Zeile 3: { FOR, ID, IN, LEFT_BRACK. CONST_INT, COMMA, ID, RIGHT_BR. }.
- Zeile 4: { ID, ASSIGN, ID, ASTERISK, ID, SEMICOLON }.
- Zeile 5: { RETURN, ID, SEMICOLON }.
- Zeile 6: { RIGHT_BRACE }.

Anmerkung: Manchmal ist es auch sinnvoll, das Eingabeprogramm vor dem Scanning vorzuverarbeiten. Zum Beispiel nutzt ein C-Compiler einen ***Präprozessor****, um Include-Direktiven aufzulösen. Weiterhin ist es denkbar, die* ***Einrückungen*** *einer Python-ähnlichen Sprache zu “glätten”, oder diese intern in eine Darstellung mit geschweiften Klammern zu ersetzen.*

3.4 Syntaktische Analyse / Parsing

Wir werden in den nächsten Abschnitten verschiedene Techniken des **Parsing** kennenlernen. Insbesondere werden wir uns mit dem **Top-Down Parsing (TDP)** und dem **Bottom-Up Parsing (BUP)** befassen. Wir untersuchen die Grenzen der konkreten Parsertypen und lernen Phänomene wie Linksrekursion, Shift/Reduce Konflikte und Reduce/Reduce Konflikte kennen. Der Leser wird dazu befähigt, Grammatiken so umzuschreiben, dass diese durch einen vorgegebenen Parsertyp verarbeitet werden können.

Definition 26 (Parsen / Parser) *Ein **Parser** ist eine **technische Instanz** eines theoretischen Automaten und führt die **syntaktische Analyse** durch. Dabei werden **Regeln** einer formalen Grammatik einbezogen.*

Ein Parser entscheidet nicht nur, **ob** ein Eingabewort von einem Startsymbol aus abgeleitet werden kann, sondern gibt auch eine Antwort darauf, **wie** das Wort abgeleitet wird.

3.4.1 Definitionen

Als Voraussetzung für das weitere Verständnis müssen nun einige Definitionen aufgestellt werden.

Definition 27 (Look-ahead) *Als **Look-ahead** bezeichnet man die Anzahl an Token, die ein Parser dazu benutzen kann, zu entscheiden welche Regel als nächstes angewandt wird.*

Definition 28 (First-Menge) *Die Funktion **first(A)** gibt die Menge der ersten Terminalsymbole einer Produktionsregel A zurück.*

Für das folgende Beispiel ist $\text{first}(A) := \{a, c\}$ und $\text{first}(B) := \{a, c, d\}$:

```
1 A = "a", "b" | "c";
2 B = A | "d";
```

*Hinweis: Eine **Follow-Menge** kommt dann zum tragen, wenn das betrachtete Nichtterminal auch auf das leere Wort ϵ abgeleitet werden kann.*

Definition 29 (Linksrekursion) *Eine Produktionsregel $A \to \gamma_1|\gamma_2|\ldots$ wird als **linksrekursiv** bezeichnet, wenn mindestens eine Alternative γ_i auf der rechten Seite ebenfalls mit A beginnt.*

Beispiel für eine linksrekursive Regel:

```
1 A = "a", "b" | A, "c", "d";
```

3.4.2 Top-Down Parsing

Top-Down Parser (TDP) beginnen mit dem Wurzelknoten des Syntaxbaums und arbeiten sich durch das Umschreiben von Regeln zum Eingabewort hinab.

- Ein häufig genutzter Typ von Top-Down Parsern ist der LL-Parser.
- Top-Down Parsing bildet immer die **linksseitige Ableitung:** Es wird also immer das am weitest links stehende Nichtterminal einer Regel als erstes ersetzt.

Beispiel: Wir betrachten die folgende in EBNF angegebene Grammatik:

```
A = "a", B, C;
B = "c" | "c", d;
C = "d", "f" | "e", "g";
```

Das Eingabewort $w_1 = acdf$ wird von der Wurzelregel A aus, von links nach rechts, wie folgt richtig abgeleitet:

$$A \Rightarrow aBC \Rightarrow acC \Rightarrow acdf =: w_1$$

Bei einem Look-Ahead von eins ist auch die folgende Ableitung möglich. Diese führt jedoch zu einem anderen (nicht gesuchten) Wort w_2:

$$A \Rightarrow aBC \Rightarrow acdC \Rightarrow acddf =: w_2 \quad \text{(falsch)}$$

Das Beispiel ist also **mehrdeutig**. Wird ein zu kleines Look-ahead gewählt, so kann eine **falsche Ableitungsentscheidung** getroffen werden. Wenn wir B durch c substituieren, dann gibt es keine zulässige Wahl mehr, um C abzuleiten.

- Die erste Produktionsregel führt zu einem doppelten Auftreten von d.
- Die zweite Produktionsregel wendet die Folge $e\ g$ an, welche in der Tat nicht Teil des Wortes ist.

3.4.2.1 Methode des rekursiven Abstiegs

Bevor State-of-the-Art Techniken des Parsings vorgestellt werden, betrachten wir die zu den Top-Down Parsern gehörende **Methode des rekursiven Abstiegs** (vgl. [23]). Wir werden später feststellen, dass die zugehörige Sprache vom Typ LL(1) sein muss. Gegeben sei die folgende Grammatik in EBNF Definition:

```
A = "a" | "b", A, "d" | B;
B = "c", B | "e";
```

Für das Parsen **eines jeden Nichtterminals** können wir nun je eine separate Funktion definieren (z.B. in Python). Terminale werden einfach mit dem aktuellen Eingabesymbol verglichen:

```python
def A():
  if sym=="a":
    next_sym()
  elif sym=="b":
```

```
5     next_sym()
6     A()
7     if sym=="d":
8       next_sym()
9     else:
10      error()
11  elif sym=="c" or sym=="e":
12    B()
13  else:
14    error()
15 def B():
16  if sym=="c":
17    next_sym()
18    B()
19  elif sym=="e":
20    next_sym()
21  else:
22    error()
```

Wir gehen davon aus, dass wir das Eingabewort von links nach rechts abarbeiten. Zum Beispiel ist $w_1 = bbadd$.

- Die globale Variable `sym` enthält stets das **aktuelle Symbol**.
- Rufen wir die Funktion `next_sym()` auf, so wird das **nächste Symbol** gelesen und nach `sym` geschrieben.
- Erkennen wir, dass wir in einen undefinierten Zustand laufen, rufen wir die **Fehlerfunktion** `error()` auf. Dies ist zum Beispiel der Fall, wenn wir ein unerwartetes Eingabesymbol einlesen.

Tabelle 3.1 zeigt ein Regelwerk, um EBNF-Konstrukte in Produktionsregeln zu übersetzen: Das Verfahren funktioniert für Nichtterminale sowohl im rekursiven als auch im nichtrekursiven Kontext. Optionen dürfen jedoch kein gemeinsames Startterminal besitzen. Negativbeispiel:

```
1 A = "a", "b" | "a", "c";
```

Wir beobachten also, dass die Methode des rekursiven Abstiegs eine **deterministische Grammatik** erfordert. Bei einem Look-ahead von eins ist nicht entscheidbar, ob entweder die erste, oder die zweite Teilregel angewandt werden muss. Weiterhin ist auch **Linksrekursion** verboten. Hierauf wird im nächsten Abschnitt eingegangen.

Zwischenfazit: Die konkrete Form einer Grammatik kann einen großen Einfluss auf den Prozess des Parsens haben. Während es manchmal eher einfach ist, eine andere Formulierung für die Grammatik zu finden, gibt es jedoch auch Fälle, wo kein gewinnbringenderes Grammatikäquivalent existiert.

ENBF Konstrukt K	Produktionen Pr(K) in Python-Code

" x "	`if sym == "x":`
	`  next_sym()`
	`else:`
	`  error()`

(exp)	`Pr( exp )`
[exp]	`if sym == first( exp ):` `  Pr( exp )`
{ exp }	`while sym == first( exp ):` `  Pr( exp )`

a, b, c	`Pr( a )`
	`Pr( b )`
	`Pr( c )`

a | b	`if sym == first( a ):`
	`  Pr( a );`
	`elif sym == first ( b ):`
	`  Pr( b );`

Tab. 3.1: Übersetzung von EBNF Konstrukten in Produktionsregeln [23]

3.4.2.2 Beseitigung von Linksrekursionen

Wir betrachten die folgende Beispielregel in EBNF:

```
A = A, "a" | "c";
```

Es ist möglich, diese Regel so umzuformulieren, dass die Linksrekursion aufgehoben wird. Mit Hilfe der **First-Menge** einer Regel erhalten wir die Menge der möglichen Startterminale. Im Beispiel gilt first(A) = $\{ c \}$. Durch Substitution erhalten wir bei Ableitung einen Ausdruck der Form $L(A) := \{w = c\ a^k \mid k \in \mathbb{N}_0 \wedge k \geq 0\}$. Also ersetzten wir die Linksrekursion durch die folgende neue EBNF-Grammatik:

```
A = "c", { "a" };
```

In diesem Fall verbleibt auf der rechten Seite kein Nichtterminal.

In dem Beispiel wurde die Linksrekursion einer Grammatik manuell beseitigt. Wir definieren nun einen **Algorithmus**, der diesen Prozess automatisiert.

Eingabe:

1. (I.) Sei $A \rightarrow A\ \alpha_1 \mid A\ \alpha_2 \mid \ldots \mid A\ \alpha_k$ die Menge der **linksrekursiven Regeln**.
2. (II.) Sei weiterhin $A \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_l$ die Menge der **nicht linksrekursiven Regeln** .

Dabei sind $\{A, B, \ldots\}$ Nichtterminale und α_i, β_i Folgen von Terminal- und Nichtterminalsymbolen.

Achtung: Der angezeigte Algorithmus funktioniert *nicht* für die **indirekte Linksrekursion**!

Algorithmus:

1. Ersetze alle Regeln von (I.) durch:
   $B \rightarrow\ \alpha_1\ B \mid \alpha_2\ B \mid\ \dots\ \mid \alpha_k\ B.$
2. Ersetze alle Regeln von (II.) durch:
   $A \rightarrow\ \beta_1\ B \mid\ \beta_2\ B \mid\ \dots\ \mid \beta_l\ B.$
3. Zum Schluss wird die folgende neue Regel zugefügt:
   $B \rightarrow \epsilon$

Beispiel: Gegeben sei die Grammatik:

```
1 x = x, "a" | x, "b", "c";
2 x = "c" | "d", "e";
```

Anwenden der Substitutionsregeln ergibt:

```
1 y = "a", y | "b", "c", y;
2 x = "c", y | "d", "e", y;
3 y = ε;
```

Schließlich kann die Regel `y = ε` durch weitere Substitution beseitigt werden. Wir tauschen auch die beiden Zeilen um aufzuzeigen, dass `x` die Wurzelregel ist:

```
1 x = "c" | "c", y | "d", "e" | "d", "e", y;
2 y = "a" | "a", y | "b", "c" | "b", "c", y;
```

Der folgende **Algorithmus für die Eliminierung der indirekten Linksrekursion** wird nur der Vollständigkeit wegen aufgezeigt und hier nicht weiter vertieft. Seien $A = \{A_1, A_2, \dots A_n\}$ Nichtterminale und $n := |A|$.

Algorithmus:

for $j \in [1, n]$ **do**
  for $i \in [1, j-1]$ **do**
    Ersetze alle Regeln vom Typ
      $A_i \rightarrow A_j\ \gamma$
    durch
      $A_i \rightarrow \delta_1\ \gamma \mid \delta_2\ \gamma \mid\ \dots\ \mid \delta_k\ \gamma$
    wobei
      $A_j \rightarrow \delta_1 \mid \delta_2 \mid\ \dots\ \mid \delta_k$
    Dann löse die **direkte** Linksrekursion für die Produktionen A_i auf.
  end
end

3.4.2.3 LL Parser

Definition 30 *Ein* ***LL-Parser*** *ist ein Top-Down Parser. Er arbeitet die Eingabe von* ***L****inks nach rechts ab, um eine* ***L****inksableitung der Eingabe zu berechnen*

Ein gewöhnlicher **LL-Parser** besteht aus den folgenden Teilen:

- Einem **Parse Stack:** Eine stackbasierte Datenstruktur, welche Terminale und Nichtterminale enthält.
- Einer **Parsingtabelle:** spezifiziert Aktionen, wie z.B. **Match**, **Predict**, **Accept** und **Error**.
- Einer **Steuerungsfunktion:** Diese interagiert als Steuerungseinheit mit allen anderen Teilen.

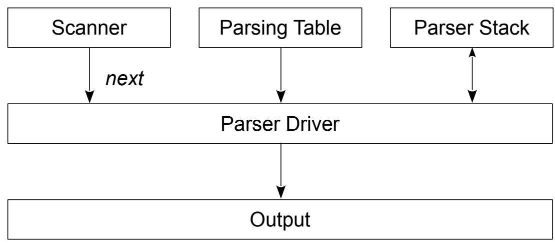


Abb. 3.7: LL-Parser

3.4.2.4 LL(k) Parser

Unsere Definitionen der letzten Seiten implizieren, dass **mehrdeutige Grammatiken** eine große Hürde für Top-Down Parser darstellen. Wie kann dieser Nachteil überwunden werden?

Definition 31 (LL(k) Parser) *Ein **LL(k)** Parser ist ein TDP, der ausschließlich **deterministische** Entscheidungen trifft. Wir nutzen einen Puffer aus k Symbolen (look-ahead). LL ist ein Akronym für "Left to right" und "Left most derivation". Die unterliegende Grammatik wird $LL(k)$-**Grammatik** genannt. Die assoziierte Sprache heißt $LL(k)$-**Sprache**.*

Eine $LL(k)$-Grammatik ist eine spezielle Form einer **kontextfreien Grammatik**. Es gilt (LR-Grammatiken werden wir später kennenlernen):

$$\mathcal{L}(LL(1)) \subset \mathcal{L}(LL(2)) \subset \cdots \subset \mathcal{L}(LL(k)) \subset \mathcal{L}(LR(1)) = \mathcal{L}(DKF)$$

$LL(1)$ **Grammatiken** haben eine Look-Ahead von eins und sind weder **mehrdeutig** noch **linksrekursiv**. Eine kontextfreie Grammatik ist vom Typ $LL(1)$ genau dann, wenn die folgende Eigenschaft für jede Produktionsregel $A \to \alpha_1 \mid \alpha_1 \mid \ldots \mid \alpha_n$ mit $\alpha_i \neq \alpha_j$ gilt:

- für alle $1 \leq i, j \leq n$ mit $i \neq j$ gilt: $f(\alpha_i) \cap f(\alpha_j) = \emptyset$.

Dabei definieren wir noch die Funktion f wie folgt:

$$f(\alpha_i) = \begin{cases} \text{first}(\alpha_i) \cup \text{follow}(A) & \text{falls } \alpha_i \Rightarrow^* \varepsilon \\ \text{first}(\alpha_i) & \text{sonst} \end{cases}$$

Beispiele:

1. Ist die folgende Grammatik vom Typ LL(1)?
   $A \to aB$
   $B \to cd$
   Antwort: **Ja,** da kein Nichtterminal mit mehr als einer Option existiert.

2. Ist die folgende Grammatik vom Typ LL(1)?
   $A \to aB \mid ad$
   $B \to cd$
   Antwort: **Nein,** da $\text{first}(\alpha_1 = aB) \cap \text{first}(\alpha_2 = ad) = \{a\} \neq \emptyset$.

Ist es möglich jede Grammatik die **nicht** vom Typ LL(1) ist, in eine äquivalente Grammatik zu transformieren, die vom Typ LL(1) ist? Nein, dies ist leider **nicht** der Fall. Die folgenden Ansätze können bei der Umformung helfen:

- **Faktorisierung:** Ersetze zum Beispiel $A \to BC \mid BD$ durch die beiden Regeln $A \to BX$ und $X \to C \mid D$.

- **Substitution:** Ersetze zum Beispiel $A \to BC$ durch die beiden Regeln $A \to BD \mid B$ und $C \to D \mid \varepsilon$.

3.4.2.5 Komplexität

Top-Down Parsing einer mehrdeutigen Grammatik kann eine **exponentielle** Anzahl an Berechnungsschritten zur Folge haben, sowie auch exponentiellen Speicher für die Syntaxbäume veranschlagen. Für **kontextfreie Grammatiken** wurde vor einiger Zeit in [8] gezeigt, dass dies auch in **Polynomzeit** möglich ist, sofern berechnete Zwischenergebnisse wiederverwendet werden:

a) $\mathcal{O}(n^4)$ für linksrekursive Grammatiken.

b) $\mathcal{O}(n^3)$ für nicht linksrekursive Grammatiken.

3.4.3 Bottom-Up Parsing

Wir haben gelernt, dass ein LL(1)-Parser nur bestimmte Grammatiken handhaben kann. Vor allem ist Determinismus eine wichtige Voraussetzung. Wir werden nun ein anderes Parserparadigma betrachten, welches diese Nachteile kompensiert. Das sogenannte **Bottom-Up Parsing (BUP)** kann als "Gegenmodell" zum Top-Down Parsing aufgefasst werden. BUP wird in der Praxis oft bevorzugt[1]

Definition 32 (Bottom-Up Parsing (BUP)) *Ein **Bottom-Up Parser reduziert** ein gegebenes Wort einer Sprache durch Anwendung umgekehrter Produktionen **auf das Startsymbol**.*

Ein weit verbreiteter Parsertyp für Bottom-Up Parsing sind **LR-Parser**: Die Token werden von **L**inks nach rechts gelesen. Dabei wird eine umgekehrte **R**echtsseitige Ableitung erstellt. In Abb. 3.8 werden die gängigen Parsertypen dargestellt.

[1] Hinweis: nicht in Xtext / ANTLR eingesetzt.

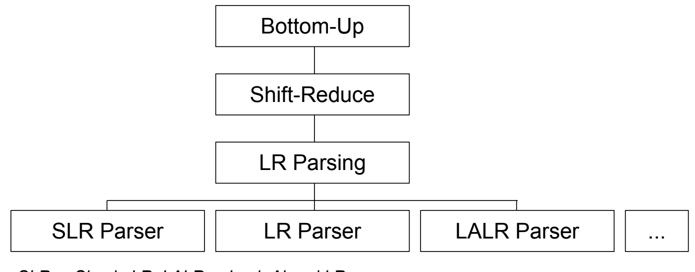


Abb. 3.8: Hierarchische Auflistung von Bottom-Up Parsertypen

Vorteile BUP:

- Grammatiken müssen **nicht** zwangsläufig **links-faktorisiert** sein.
- Grammatiken dürfen **links-rekursiv** sein.

Ein Bottom-Up Parser beinhaltet einen **Stack-Speicher** für Zwischenergebnisse. Die folgenden zwei Arten von **Aktionen** spielen bei allen Bottom-Up Parsern eine große Rolle. Entsprechend spricht man bei Bottom-Up Parsern auch häufig von **Shift**/**Reduce**-Parsern.

- **Shift:** Shiftet das nächste Symbol des Eingabewortes in den Stack.
- **Reduce:** Nimmt eine Sequenz von Symbolen vom Stack herunter und legt danach ein äquivalentes Nichtterminalsymbol auf den Stack.

Beispiel: Das folgende Beispiel ist aus [23] entnommen. Zunächst definieren wir eine einfache Grammatik für geklammerte, mathematische Ausdrücke. Die Addition und die Multiplikation sind erlaubt:

```
1 E = T | E, "+", T;
2 T = F | T, "*", F;
3 F = id | "(", E, ")";
```

Die Abkürzungen für die Nichtterminale sind wie folgt definiert: { E:= Expression/ Ausdruck, T := Term, F := Faktor }. Wir parsen nun das folgende Eingabewort:

$$w_1 = x * (y + z)$$

Der zugehörige AST wird in Abb. 3.9 darstellt. Tabelle 3.2 listet alle **Shift** und

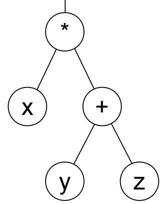


Abb. 3.9: AST zum Beispielwort

No	Verbl. Eingabe	Shift/Reduce	Stack	Kommentar
1	x * (y + z)	∅	∅	
2	* (y + z)	S	x	Shifte erstes Terminalsymbol x.
3	* (y + z)	R (3)	F	id wird auf F reduziert.
4	* (y + z)	R (2)	T	F wird auf T reduziert.
5	(y + z)	S (!!!)	T *	
6	y + z)	S	T * (	
7	+ z)	S	T * (y	
8	+ z)	R (3)	T * (F	
9	+ z)	R (2)	T * (T	
10	+ z)	R (1)	T * (E	
11	z)	S	T * (E +	
12	)	S	T * (E + z	
13	)	R (3)	T * (E + F	
14	)	R (2)	T * (E + T	
15	)	R (1)	T * (E	
16	∅	S	T * (E)	
17	∅	R (3)	T * F	
18	∅	R (2)	T	
19	∅	R (1)	E	ACCEPT

Tab. 3.2: Shift-Reduce Tabelle zum Eingabewort $w_1 = x * (y + z)$.

Reduce Schritte auf. Für die **Reduce** Aktionen wird auch die Regelnummer, also die entsprechende Zeile aus der EBNF Grammatik mit angegeben. **In Zeile 5** stellt man sich vielleicht die Frage, wieso keine weitere Reduktion vorgenommen wird. Wieso wird `T` nicht durch `E` ersetzt? **Antwort:**

- Dies würde bedeuten, dass der Stack dann als Inhalt `[ E ]` hat.
- Danach würde dann keine weitere Reduktion mehr möglich sein und es wären nur noch Shifts erlaubt.
- Das Look-ahead Symbol ist dann `*`. Der Parser betrachtet das Symbol vorausschauend, **bevor** eine weitere Entscheidung getroffen wird.
- Die einzige Option ist dann das Symbol `*` zu shiften, sodass der Stack dann den Inhalt `[ E * ]` hat.
- Betrachtet man nun die Gesamtheit aller Produktionsregeln, so gibt es keine Möglichkeit eine spätere Reduktion mehr durchzuführen. Denn es gibt keine Produktionsregel, die eine Symbolfolge `[ E * ]` beinhaltet.
- Der Parser ist dann nicht mehr in der Lage das Wort zu akzeptieren.

Entsprechend **muss** in Zeile 5 der Beispieltabelle eine **Shift**operation durchgeführt werden.

3.4.3.1 LR(k) Parser

LR(k) Parser gehören zur Klasse der **Bottom-up Parser.** Der Parameter k gibt die Größe des Look-Ahead an. Wird k mit 1 gewählt, was für gewöhnlich der Fall

ist, dann wird k meist nicht explizit mit angegeben. In der Abhandlung [15] von Donald E. Knuth, wurde der Begriff **LR** Parsing erstmals eingeführt.

Vorteile:

- LR Parser verarbeiten **deterministisch-kontextfreie Sprachen** (DFK) **effizient** in Linearzeit.
- LR Parser setzen **nicht auf Backtracking**.

Es gibt einige Varianten von LR Parsern. Beispiele sind, SLR := Simple LR Parser, LALR := Look-ahead LR Parser, Canonical LR(1) Parser, Minimal LR(1) Parser und GLR := Generalized LR Parser. Tabelle 3.3 ordnet die Parsertypen konkreten **Parsergeneratoren** zu.

Parsertyp	Parsergenerator(en)
LL(k)	JavaCC, Coco/R
LL(*)	ANTLR 3/4
LR(1)	GNU Bison
LALR(1)	SableCC, yacc, GNU Bison
GLR(k)	GNU Bison

Tab. 3.3: Unvollständige Liste von Parsergeneratoren

Man beachte, dass GNU Bison mehrere Sprachtypen unterstützt. Per Default ist dies LALR(1).

Moderne **Parsergeneratoren** bedienen sich einer Vielzahl an "Tricks", um den Vorgang zu beschleunigen. Einige technische Details werden nun aufgelistet.

Reduce-Operationen sind nicht auf das oberste Kellersymbol beschränkt. Oft werden viele (oder sogar alle) Symbole des Stacks gleichzeitig betrachtet. Um dennoch eine **konstante Laufzeit** zu garantieren, werden alle relevante Informationen in einer einzigen Zahl gebündelt; dem **Parserzustand.** Die Anzahl dieser Zustände ist immer **endlich.** Jeder Stackeintrag besteht aus zwei Attributen:

- dem **Symbol** selbst.
- dem **Parserzustand**.

Ein (LR-) **Parsergenerator** wird als Zustandsautomat umgesetzt. Eine Zustandsüberführung basiert auf:

- dem aktuellen **Parserzustand** und
- den **Look-ahead** Symbolen.

Es gibt hauptsächlich zwei Ansätze zur Implementierung von Parsergeneratoren:

1. Implementierung einer **generischen Parser**-Schleife, die wiederum intern auf Tabellen setzt. Der Vorteil ist hier, dass immer nur die Menge der Tabellen ersetzt werden muss, sofern die Grammatik substituiert wird.

2. Bereitstellung von **konkretem Programmcode** für die Zustände. Der resultierende Parser ist dann ein statisches Programm (z.B. ein C Programm).

Offensichtlich erzeugt die zweite Methode schnellere Parser.

Aktions- und Goto-Tabellen: Die Implementierung von Parsergeneratoren basiert hauptsächlich auf den beiden folgenden Datenstrukturen:

- **Aktionstabelle:**
  Die Aktionstabelle beschreibt anhand a) des aktuellen Zustandes und b) des aktuellen Eingabesymbols, was als nächstes zu tun ist. Meist gibt es die folgenden Zustände:
  - **Shift(S):**
    1. Lege das nächste Eingabesymbol auf den Stack.
    2. Der neue Zustand ist nun S.
  - **Reduce(n):**
    1. Wende die Produktionsregel n der Grammatik an.
    2. Nehme die Anzahl an Elementen vom Stack, die der Anzahl an Symbolen der rechten Seite der Produktionsregel entspricht.
    3. Lese den nächsten Zustand aus der Goto-Tabelle und lege es auf den Kellerspeicher.
- **Goto-Tabelle:**
  Die Goto-Tabelle gibt an, in welchen Zustand nach Abarbeiten des aktuellen Zustands gewechselt wird.
  - **Accept:** Akzeptiert das Eingabewort und terminiert mit Erfolg.
  - **Error:** Wenn die Kombination aus aktuellem Zustand und aktuellem Eingabesymbol nicht zulässig ist (d.h. nicht in der Tabelle vermerkt ist), dann wird die Eingabe verworfen. Der Parser terminiert mit einem Fehler.
  - *Keine Angabe:* Wechsle in den folgenden Zustand.

Beispiel: Das folgende Beispiel zeigt eine konkrete Action-Goto Tabelle. Als EBNF Grammatik sei gegeben:

```
1 E = E, "*", B | E, "+", B | B;
2 B = "0" | "1";
```

In einem ersten Schritt identifizieren wir die Reduktionsregeln:

1. $E \rightarrow E * B$
2. $E \rightarrow E + B$
3. $E \rightarrow B$
4. $B \rightarrow 0$
5. $B \rightarrow 1$

Die vom Parsergenerator erzeugte Action-Goto Tabelle sieht wie folgt aus:
In einer Übung werden wir mit Hilfe einer durch GNU Bison erzeugten Action-Goto-Tabelle ein Wort manuell parsen.

	Action					Goto	
State	$*$	$+$	0	1	\$	E	B
0			S(1)	S(2)		3	4
1	R(4)	R(4)	R(4)	R(4)	R(4)		
2	R(5)	R(5)	R(5)	R(5)	R(5)		
3	S(5)	S(6)			ACC		
4	R(3)	R(3)	R(3)	R(3)	R(3)		
5			S(1)	S(2)			7
6			S(1)	S(2)			8
7	R(1)	R(1)	R(1)	R(1)	R(1)		
8	R(2)	R(2)	R(2)	R(2)	R(2)		

Tab. 3.4: Action-Goto Tabelle

3.4.3.2 Shift-Reduce Konflikte

Ein **Shift-Reduce Konflikt** tritt auf, wenn nicht klar ist, ob eine **Shift**-Operation oder eine **Reduktions**-Operation durchgeführt werden soll. Beispielgrammatik in EBNF Notation:

```
1 statement = if;
2 if = "if", exp, "then", statement
3    | "if", exp, "then", statement, "else", statement;
```

Wenn das nächste Terminalsymbol `else` ist, dann gibt es die folgenden beiden Möglichkeiten:

- Gemäß der ersten Teilregel von `if` kann der Kellerinhalt auf das Nichtterminal `if` reduziert werden.
- Ebenso könnte aber auch das Terminalsymbol `else` (gemäß der zweiten Teilregel von `if`) geshiftet werden.

Anmerkung: Der Parsergenerator GNU Bison betrachtet Shift/Reduce Konflikte als Warnung. Wird die Warnung ignoriert, dann wird das Shiften bevorzugt.

3.4.3.3 Reduce-Reduce Konflikte

Reduce-Reduce Konflikte treten auf, wenn gleichzeitig mehr als eine Regel für die Reduktion angewandt werden kann. Einfache Beispielgrammatik in EBNF Notation:

```
1 x = "a" | "a";
```

Eine weitere Grammatik in EBNF Notation aus [1]:

```
1 s = "a", x, "d" | "b", y, "d" | "a", y, "e" | "b", x, "e";
2 x = "c";
3 y = "c";
```

GNU Bison betrachtet Reduce/Reduce Konflikte als Warnung. Wenn die Warnung ignoriert wird, dann wird immer diejenige Definition genutzt, die zuerst definiert worden ist.

3.5 Semantische Analyse

Im Rahmen der syntaktischen Analyse haben wir der **Semantik**, also der Bedeutung des Eingabeprogramms bisher keine Beachtung geschenkt. Wir erweitern in den folgenden Abschnitten unsere Grammatiken nun um Datentypen und Deklarationen. Diese müssen persistent gepflegt werden. Weiterhin müssen auch potenzielle Fehler abgefangen werden.

3.5.1 Attributierte Grammatiken

In Kapitel 2.7.1 haben wir **attributierte Grammatiken** bereits kurz eingeführt. Eine attributierte Grammatik reichert die Regeln einer Grammatik mit gewissen **Eigenschaften** an.

3.5.1.1 Typregeln

Die meisten imperativen Programmiersprachen unterstützen eine große Menge an Datentypen. Dazu gehörten zum Beispiel *int*, *double*, *struct*, *enum*, *class* usw. Es ist meist hinreichend, **einmalig** eine **generische Syntax** für alle Datentypen zu deklarieren, anstelle für jeden Datentyp neue Regeln zu schreiben. Wir fügen der Grammatik das **Datentyp-Attribut** T hinzu. Wir betrachten zunächst die folgende in EBNF spezifizierte Grammatik zur Umsetzung von geklammerten mathematischen Ausdrücken:

```
1 exp = term | exp, ("+"|"-"), term;
2 term = factor | term, ("*"|"/"), factor;
3 factor = id | "(", exp, ")";
```

Nun soll die Grammatik um das Datentypattribut T erweitert werden. Wir gehen zunächst davon aus, dass die Operatoren $\{+, -, *, /\}$ nur Operanden eines einzigen Datentyps verarbeiten können, also z.B. “Integer + Integer”. Die attributierte Grammatik ist in Tabelle 3.5 aufgezeigt.

Syntax	Attributregel	Bedingung
exp(T_0) =		
term(T_1)	T_0 := T_1	
| exp(T_1), "+", term(T_2)	T_0 := T_1	T_1 == T_2
| exp(T_1), "-", term(T_2);	T_0 := T_1	T_1 == T_2
term(T_0) =		
factor(T_1)	T_0 := T_1	
| term(T_1), "*", factor(T_2)	T_0 := T_1	T_1 == T_2
| term(T_1), "/", factor(T_2);	T_0 := T_1	T_1 == T_2
factor(T_0) =		
id	$\emptyset$	
| "(", exp(T_1), ")";	T_0 := T_1	

Tab. 3.5: Beispiel für eine attributierte Grammatik mit Typregeln

Für Grammatiken mit **Repetitionen** ist die Definition der Bedingung nicht mehr so trivial anzugeben. Meist ist es einfach, die Repetition durch **Rekursion** zu **ersetzen**. Zum Beispiel betrachten wir die folgende Regel:

```
term(T0) = factor(T1), { "*", factor(Ti) };
```

Wir können die Repetition wie folgt auflösen, sodass i konkret angegeben werden kann:

```
term(T0) = factor(T1)
  | term(T1), "*", factor(T2);
```

Die **Implementierung** von Typregeln in manuell programmierte Top-Down Parser (TDP) ist eingängig, wenn die Typen als Rückgabewert zurückgegeben werden werden. Für die **Methode des rekursiven Abstiegs** ist folgende Umsetzung (hier in Python) möglich:

```python
def term():
  t1 = factor()
  while sym == "*":
    next_sym()
    t2 = factor()
    if t1 != t2:  # semantical analysis
      error()
  t0 = t1
  return t0
```

Wir nehmen an, dass die globale Variable `sym` das aktuelle Token beinhaltet. Die Funktion `next_sym()` liest das nächste Token in die Variable `sym`. Auf die Angabe expliziter Fehlermeldungen verzichten wir im Moment.

3.5.2 Deklarationen

Das Konzept der **Deklaration** ist essentiell für jeden Allzwecksprachen-Compiler. Während zwischen **statischer** und **dynamischer Typisierung** unterschieden wird, konzentrieren sich die folgenden Abschnitte der Einfachheit wegen auf die statische Typisierung; wie es z.B. in der Programmiersprache C üblich ist.

Definition 33 (Kontext) *Ein **Kontext** ist ein logischer Abschnitt innerhalb einer formalen Sprache: Beispiele sind Klassen, Methoden, Funktionen usw.*

Eine **Deklaration** wird innerhalb eines **Kontexts** getätigt. Wir ordnen jedem Kontext eine **Symboltabelle** zu. Diese speichert die Menge der **lokalen Deklarationen**.

3.5.2.1 Symboltabelle

Wenn in einem Programm **Bezeichner** auftreten, so sind diese entweder bereits bekannt, oder werden erstmalig eingeführt:

1. Eine **Deklaration** fügt der Symboltabelle einen neuen Eintrag hinzu.

2. Bei einer **Verwendung** wird die Symboltabelle um den Bezeichner durchsucht. So ist es zum Beispiel möglich, den zugehörigen **Datentyp** oder die **Speicheradresse** zu erhalten.

Definition 34 (Symbol) *Ein **Symbol** stellt die Gesamtheit der zu einem **Bezeichner** vorliegenden Informationen dar.*

Beispiele für Symbole in Allzwecksprachen sind Variablen, Enumerationen, Funktionen, Klassen usw.

Definition 35 (Symboltabelle) *Eine **Symboltabelle** ist eine **Datenstruktur**, die zu jedem Symbol einen zugehörigen Datensatz speichert. Dazu gehören Attribute wie zum Beispiel die **Klasse**, der **Typ**, der **Wert** und die **Adresse** des Eintrags.*

Die Menge der Attribute ist abhängig von dem konkreten Compiler. Mindestens werden die folgenden Einträge hinterlegt:

- Die **Klasse**[2] gibt die Art des Eintrags an. Also ob es sich um eine Konstante, Variable, Funktion, Enumeration usw. handelt.
- Der **Typ** bezeichnet den unterliegenden Datentyp. Zum Beispiel Integer, Long, Klassenname usw.
- Ein **Wert** kann z.B. dazu genutzt werden, um Konstanten abzulegen.
- Eine **Adresse** gibt an, wo sich das Symbol im Speicher befindet. Die angegebene Adresse ist immer **relativ** bezogen auf eine **Basisadresse**. Die Anzahl der benötigten Bytes hängt von der Zielarchitektur ab.

In der Praxis gibt es meist eine Vielzahl an zusätzlichen Attributen. Nicht alle Attribute müssen explizit gespeichert werden, sondern können aus anderen Attributen *on demand* abgeleitet werden. Die Menge aller Symboltabellen ist in einer **hierarchischen Struktur** angeordnet. Neben den **Nutzdaten** müssen auch die **Relationen** zu anderen Symboltabellen gespeichert werden.

3.5.2.2 Kontext und Geltungsbereiche

Der Code von imperativen Programmiersprachen besteht aus einer Menge von **Kontexten** (siehe oben). Tabelle 3.6 zeigt ein dazu ein Beispiel:
Vom Begriff des Kontexts unterscheiden wir den **Geltungsbereich**:

Definition 36 (Geltungsbereich) *Ein **Geltungsbereich** (Scope) beschreibt die Gültigkeit eines Bezeichners.*

Die Gültigkeit geht oft über den Kontext hinaus. Zum Beispiel wird eine globale Variable im Kontext 0 definiert und ist auch in allen weiteren Kontexten $1, 2, \ldots$ gültig.

Beispiel: Für den Code aus Tab. 3.6 geben wir nun in Tab. 3.7 die zugehörigen Symboltabellen an. Man beachte, dass die relativen Adressen von lokalen Variablen

Beispielcode	Kontext
`#include <stdio.h>`	0
	0
`int factorial(int x) {`	1
`  int y = 1;`	1
`  for(int i=0; i<=x; i++) {`	2
`    y = y * i;`	2
`  }`	2
`  return y;`	1
`}`	1

Tab. 3.6: Beispielcode und Kontext-IDs

Tabelle	P	C	ID	Klasse	Typ	Adresse
0	-1	{1}	factorial	Funktion	Integer	$2000
			x	Parameter	Integer	0
			y	lokale Variable	Integer	-1
1	0	∅	i	lokale Variable	Integer	-2

Tab. 3.7: Symboltabellen für das Beispielprogramm aus Tabelle 3.6

i.d.R. negativ sind.
Die Spalten *P* und *C* beschreiben die relationalen Beziehungen zwischen den Symboltabellen:

- **P := Parent** ist der Index der Elterntabelle.
- **C := Children** ist die Menge der Indizes der Kindtabellen.

*Anmerkungen zur Implementierung: Es ist alternativ auch möglich anstelle einer **hierarchischen Tabellenstruktur** auch eine **globale Symboltabelle** einzusetzen. Diese enthält dann weitere Spalten für die Verwaltung. Die Tabellen selbst sind bei einfachen Compilern meist als **verkettete Liste** umgesetzt.*

3.5.2.3 Mehrdimensionale Datentypen

Bisher haben wir nur **skalare** Datentypen betrachtet. Die Implementierung von **Arrays** verlangt nach zusätzlichen Einträgen in der Symboltabelle. Oft wird auch auf dedizierte Datenstrukturen referenziert.

Beispiel: Arrays in der Programmiersprache C haben die folgende Form:

```
1 int a[10];
```

Es wird keine dynamische Grenzüberprüfung ("bounds checking") zur Laufzeit unterstützt. Entsprechend kann die Speicherung der Elementanzahl[3] in der Symboltabelle weggelassen werden. Im einfachsten Fall wird das Array intern als **Zeiger** `int *a` betrachtet. Die Generierung von Maschinencode lässt sich so erheblich vereinfachen:

[2]Nicht zu verwechseln mit objektorientierter Programmierung.
[3]Abgesehen von der Unterstützung des `sizeof()` Operators

Man ersetzt z.B. den Zugriff `a[5]` durch den (ebenfalls in der Quellsprache gültigen) Ausdruck `*(a+5)`.

3.6 Codegenerierung

3.6.1 Interpretation

Bisher haben wir Parser zu Zwecken des Einlesens verwendet. Nun erweitern wir unsere Grammatiken um Attributregeln für die **semantische Auswertung**. Man beachte, dass das folgende Beispiel keine Typregeln enthält. Wir nehmen an, dass alle Datentypen vom Typ Integer sind. Das Attribut v beschreibt einen berechneten Wert. Wir definieren damit nun einen **Interpreter**, aber noch keinen Compiler.

Produktion	Semantische Attributregel
term(v_0) =	
factor(v_1)	v_0 := v_1
| term(v_1), "*", factor(v_2)	v_0 := v_1 * v_2
| term(v_1), "/", factor(v_2);	v_0 := v_1 / v_2

Tab. 3.8: Beispiel einer attributierten Grammatik mit semantischer Auswertung

3.6.2 Kompilation

Um einen echten **Compiler** zu konstruieren, führen wir nun die Definition von **Ausgaberegeln** ein. Man betrachte dazu Tabelle 3.9. Wir nehmen an, dass die Funktionen zunächst einfach den zugehörigen Operator bzw. Operanden ausgeben. Bei Kompilierung des Eingabeworts $w = 2 + 5 * 6$ erhalten wir dann die folgende

Produktion	Aktionscode
exp =	
term	∅
| exp, "+", term	addition()
| exp, "-", term;	subtraction()
term =	
factor	∅
| term, "*", factor	multiplication()
| term, "/", factor;	division()
factor =	
number	constant(number)
| "(", exp, ")";	∅

Tab. 3.9: Beispiel einer Grammatik mit Aktionscode

Ausgabe in **Postfix-Notation:**

```
1  2, 5, 6, *, +
```

Wie können wir diese Ausgabe auf einer echten oder virtuellen **Maschine** berechnen?

- Zunächst werden die Konstanten auf **Prozessorregister** abgebildet.
- Wenn unsere Zielarchitektur N Register bereitstellt, dann können wir die auftretenden Konstanten einfach aufsteigenden Registerindizes zuordnen; beginnend mit 0. Für das Beispiel von oben setzen wir $N \geq 3$ voraus.

Registerindex	Konstante
0	2
1	5
2	6

Tab. 3.10: Mapping der Konstanten auf Register

Unser **Maschinencode** hat nun die folgende Form. Das Pflegen der Registerindizes diskutieren wir unten.

Adresse	Pseudocode	Maschinencode
1	Lade den Wert 2 in das Register R_0.	`LDI R0 ← 2`
2	Lade den Wert 5 in das Register R_1.	`LDI R1 ← 5`
3	Lade den Wert 6 in das Register R_2.	`LDI R2 ← 6`
4	Multipliziere die Register R_1 und R_2. Schreibe das Ergebnis in Register R_1.	`MUL R1 ← R1,R2`
5	Addiere die Register R_0 und R_1. Schreibe das Ergebnis in Register R_0.	`ADD R0 ← R0,R1`

Tab. 3.11: Maschinencode

Die **Instruktion** `LDI` steht für Load-Immediate und lädt einen konstanten Wert in ein CPU-Register. Nach **Ausführung** des Maschinenprogramms steht das Ergebnis in dem Register mit dem kleinsten Index; also in Register R_0. Die Registerinhalte sind $\{R_0 := 32, R_1 := 30, R_2 := 6\}$.

*Anmerkung: Oftmals wird sogenannter **Dreiaddresscode** eingesetzt. Jede Instruktion hat dann maximal drei Operanden.*

3.6.3 Registermanagement

Für die Codegenerierung muss noch das **Registermanagement** definiert werden, also die Frage geklärt werden, wie man die Indizes **systematisch** bestimmen kann. Hierzu bedienen wir uns des Prinzips des **Kellerspeichers**. Wir definieren im **Codegenerator** eine **globale Variable** `reg_idx`, die den aktuellen Registerindex speichert.

Der folgende Codeausschnitt zeigt eine beispielhafte Python-**Implementierung** für die Funktionen des **Aktionscodes** aus Tabelle 3.9. Die Umsetzung der Funktionen für die anderen Grundrechenarten ist äquivalent zur Addition. Lediglich das **Mnemonic** für die Instruktion ist dort nicht `ADD`, sondern `SUB`, `MUL` bzw. `DIV`.

```
reg_idx = 0

def constant(number):
  global reg_idx
  print("LDI␣R" + reg_idx + ",␣" + number)
  reg_idx += 1

def addition():
  global reg_idx
  reg_idx -= 1
  print("ADD␣R" + (reg_idx-1) + ",␣" + (reg_idx-1) + ",␣" + reg_idx)
```

In diesem Beispiel wird der Maschinencode einfach auf der Kommandozeile ausgegeben. Man beachte, dass wir nicht prüfen, ob mehr **Register** benötigt werden, als **physikalisch vorhanden** sind. In der Praxis müsste in diesem Fall auf den **Hauptspeicher** zurückgegriffen werden.

3.6.4 Datenrepräsentation

Die **Datenrepräsentation** hängt maßgeblich von der konkreten **Zielarchitektur** ab:

- Die **Speichergröße** von *intrinsischen* Datentypen, wie z.B. Integer und Float, ist überlicherweise ein Vielfaches von zwei. Diese Größen können sehr einfach durch eine Lookup-Tabelle abgerufen werden. Für *komplexe* (strukturierte) Datentypen ist die Gesamtspeichergröße schwieriger zu berechnen:
  - Bei **Arrays** wird die skalare Datengröße mit der Anzahl an Elementen multipliziert.
  - Bei **Strukturen** summiert man die einzelnen Datenfelder.

  Allerdings werden die Daten im Speicher in der Praxis **ausgerichtet (aligniert)**, um den Overhead bei Speicherzugriffen gering zu halten: Die Adresse einer Variablen wird dann auf die nächste freie Speicheradresse festgelegt, die ein Vielfaches der Speichergröße für diese Variable darstellt.

  Ein **Beispiel** in der Programmiersprache C:

  ```
  char a;
  int b;
  long c;
  ```

  Wir nehmen an, dass hier gilt: sizeof({ char, int, long }) := { 1, 4, 8 }: Setzt man voraus, dass ein Prozessor 8 Bytes / 64 Bit in einem Zyklus lesen bzw. schreiben kann[4], so dauert ein nicht alignierter Zugriff auf die Variable `c` genau 2 Zyklen, während ein alignierter Zugriff aus nur einem Zyklus besteht.

- Die **Speicheradresse** ist eine lineare Adresse, die entweder auf den **Stackspeicher** oder auf den **Heapspeicher** verweist.

[4] Man beachte, dass heutige Wortbreiten größer sind.

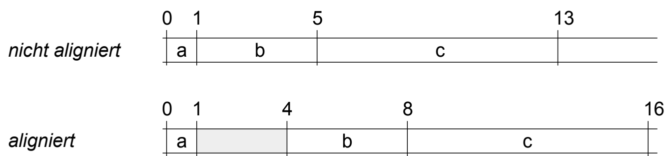


Abb. 3.10: Beispiel zur Alignierung.

3.6.5 Befehlssatzarchitektur

Der Befehlssatz kann entweder als **Zwischencode** oder als **Maschinencode** vorliegen.

Definition 37 (Zwischencode) *Als **Zwischencode (Intermediate Code)** bezeichnet man eine **abstrakte Kodierung**, die von der Komplexität her zwischen der Quellsprache und der Zielsprache liegt (meist näher an der Zielsprache).*

Die Verwendung von Zwischencode hat den Vorteil, dass man das **Compilerfrontend** vom **Compilerbackend entkoppeln** kann.

Definition 38 (Befehlssatzarchitektur) *Eine **Instruction Set Architecture, ISA** spezifiziert den Befehlssatz eines Prozessors durch die Angabe einer Menge von Instruktionen mit ihren Parametern. Weiterhin ist die binäre Kodierung festgelegt.*

Die Namen von Instruktionen werden meist durch **Mnemonics** abgekürzt.

*Anmerkung: Weiterhin ist es auch möglich, mit **Bytecode** zu arbeiten. Dies ist insbesondere bei der Programmiersprache Java der Fall. Zur Ausführung wird dann eine **Virtuelle Maschine** auf dem Zielrechner benötigt. Im Falle von Java ist dies die JVM := Java Virtual Machine.*

Die Iom-VM ISA

Eine sehr **einfache ISA** wurde vom Autor für den Mini-Compiler *Iom* definiert. Link: https://gitlab.com/andreas.schwenk/iom/. Die Intention war es, eine Befehlsarchitektur zu entwickeln, die didaktisch möglichst gut in eine Lehrveranstaltung integriert werden kann. Alle weiteren Beispiele beziehen sich auf die *Iom*-Architektur.

*Achtung: Wir unterschieden hier nicht weiter Zwischencode von Maschinencode. In der Praxis sollte selbstverständlich ein weit verbreitetes und standardisiertes Format eingesetzt werden; wie z.B. die **LLVM Intermediate Representation, IR**. So ist es möglich, dass man auf das sehr ausgereifte Compiler Backend LLVM zurückgreifen kann. Darüber hinaus müssen übrigens auch Compiler Frontends für Allzwecksprachen nicht manuell umgesetzt werden. Anstelle von Parsergeneratoren kann eine neue Sprache auch mittels **Clang** umgesetzt werden.*

Den *Iom*-Befehlssatz findet man in Tabelle 3.12. Die Maschine verfügt über 32 General-Purpose Register, sowie über 4 weitere Special-Purpose Register:

- **Stack Pointer (SP):** Zeigt auf das oberste Element des Stapelspeichers.

Instruktion	Parameter	Semantik
MOVE	a, b	R[a] := R[b]
ADD	a, b, c	R[a] := R[b] + R[c]
SUB	a, b, c	R[a] := R[b] - R[c]
MUL	a, b, c	R[a] := R[b] · R[c]
DIV	a, b, c	R[a] := R[b] / R[c]
CMP_GREATER	a, b, c	R[a] := R[b] > R[c]
CMP_LESS	a, b, c	R[a] := R[b] < R[c]
CMP_GREATER_EQUAL	a, b, c	R[a] := R[b] ≥ R[c]
CMP_LESS_EQUAL	a, b, c	R[a] := R[b] ≤ R[c]
LOAD	a, b	R[a] := MEM[b];
STORE	a, b	MEM[b] := R[a];
LOAD_IMM	a, b	R[a] := b;
LOAD_LOCAL	a, b	R[a] := MEM[FP + b];
STORE_LOCAL	a, b	MEM[FP+b] := R[a];
BRANCH	a	PC += a;
BRANCH_EQUAL	a, b, c	if R[a] == R[b] then PC += c;
BRANCH_NOT_EQUAL	a, b, c	if R[a] != R[b] then PC += c;
BRANCH_IF_ONE	a, b	if R[a] == 1 then PC += b;
BRANCH_IF_ZERO	a, b	if R[a] == 0 then PC += b;
CALL	a	call function at address a;
RETURN	∅	return;
RETURN_WITH_VALUE	a	RR := [a]; return;
PUSH	a	MEM[SP] := R[a]; SP ++;
POP	a	SPP --; R[a] := MEM[SP];
NOP	∅	no operation

Tab. 3.12: Iom-Befehlssatzarchitektur. Das Ergebnis der Vergleichsinstruktionen `CMP_*` *ist 0 (false) oder 1 (true).*

- **Frame Pointer (FP):** Kopie des Stack Pointers zum Zeitpunkt eines Funktionsaufrufs.
- **Return Register (RR):** Rückgabewert oder -zeiger.
- **Program Counter (PC):** Aktuelle Programmadresse des Maschinen-Eingabeprogramms während der Ausführung.

Die beiden Instruktionen CALL und RETURN sind **komplexe Instruktionen**, d.h. diese werden im Rahmen der Übersetzung durch eine Sequenz einfacher Instruktionen ersetzt:

```
CALL(addr):
  PUSH PC;         # Erstelle eine Kopie des Program Counters
  PUSH FP;         # Erstelle eine Kopie es Frame Pointers
  MOVE FP, SP;     # Setze den neuen Frame Pointer
  JUMP addr;       # Springe zur Zieladresse

```

```
 7  RETURN:
 8    MOVE SP, FP;
 9    FP := [SP - 1];  # Stelle den Framepointer wieder her
10    SP := SP + 2;    # Stelle den Stackpointer wieder her
11    JUMP [SP-0];     # Springe zurueck
```

Eine exemplarische Übersetzung von zwei Funktionen sieht man in Abbildung 3.11.

Abb. 3.11: Beispielübersetzung in die Iom-ISA. Man beachte, dass hier keine echten Instruktionen, sondern Pseudocode angegeben wird. Zum Beispiel müsste man statt `SP := SP - 1` *korrekter die beiden konsekutiven Maschinenbefehle* `LOAD_IMM R[0],1` *und* `ADD R[SP], R[SP], R[0]` *schreiben. Einige Architekturen bieten auch direkt eine Instruktion für die Dekrementierung an.*

In den folgenden Abschnitten generieren wir Code für Ausdrücke, Bedingungen und Funktionen.

3.6.6 Mathematische Ausdrücke

Sei `R` eine **Generatorvariable** zur Verwaltung der Registerindizes. Dann können wir Maschinencode für **mathematische Ausdrücke** so wie in Abb. 3.12 gezeigt erzeugen. Man beachte in dem Beispiel den **Operatorvorrang**. Während die Addition und Subtraktion am nächsten zum Wurzelelement `exp` liegt, sind Multiplikation und Division in einer "darunter" liegenden Regel enthalten. Als **Faustregel gilt**: Je stärker ein Operator bindet, um so weiter ist er von der Wurzelregel entfernt.

Die Programmiersprache C hat fünfzehn (15) ***Prioritäten*** *von Operatoren. Im Anhang B findet man als Beispiel die Regeln des Iom-Compilers.*

3.6.7 Bedingte Anweisungen

Bedingte Anweisungen können so wie in Abb. 3.13 dargestellt erzeugt werden. Die **Sprungzieladresse** δ muss nach der Generierung von `block` angepasst werden. Erst zu diesem Zeitpunkt ist die konkrete Adresse bekannt.

Syntax	Generated Code and Side Effects *(gen := generate)*
`exp =`	
`term`	*gen(term)*
`| exp, "+", term`	`gen(term₁); gen(term₂); R --;` **`INSTR__ADD R-1, R-1, R;`**
`| exp, "-", term;`	`gen(term₁); gen(term₂); R --;` **`INSTR__SUB R-1, R-1, R;`**
`term =`	
`factor`	*gen(factor)*
`| term, "*", factor`	`gen(factor₁); gen(factor₂); R --;` **`INSTR__MUL R-1, R-1, R;`**
`| term, "/", factor;`	`gen(factor₁); gen(factor₂); R --;` **`INSTR__DIV R-1, R-1, R;`**
`factor =`	
`number`	**`INSTR__LOAD_IMMEDIATE R ;`** `R ++;`
`id`	**`INSTR__LOAD_LOCAL symTable.get(id).address;`** `R ++;`
`| "(", exp, ")";`	*gen(exp)*

Abb. 3.12: Codegenerierung mathematischer Ausdrücke.

Syntax	Address	Generated Code and Side Effects
`if = "if",`		
`"(", exp, ")",`	α	gen(exp)
	β	**INSTR__BRANCH_IF_ZERO R, δ**
`block;`	γ	gen(block)
	δ	

gen(...) := generate(...)

Abb. 3.13: Codegenerierung bedingter Ausdrücke.

Anmerkung: Adressen können entweder sofort aufgelöst werden, wenn dies möglich ist. Alternativ erledigt man dies in einer zusätzlichen Kompilationsphase.

Der folgende Code demonstriert die Implementierung innerhalb des *Iom*-Compilers:

```
1  case SYM_IF_STATEMENT:
2    iterator = iterate_list(sym->ast_children);
3    i = 0;
4    instr = NULL;
5    while ((child = get_next_list_item(&iterator)) != NULL) {
6      generate_intermediate_code_rec(root, child);
7      if (i == 0) {
8        instr = push_ic_instruction(root->intermediate_code, sym,
9          INSTR_BRANCH_IF_ZERO, current_reg, 0,
10         /* address must be fixed later */ 0);
11     }
12     i++;
13   }
14   link_instructions(instr, push_ic_instruction(
15     root->intermediate_code, sym, INSTR_NOP, 0, 0, 0));
16   break;
```

3.6.8 Funktionen

Maschinencode für **Funktionen** muss so erzeugt werden, dass **Rekursion**, also der Selbstaufruf einer Funktion, möglich ist. Hierzu ist es nötig, dass bei Aufruf einer

Funktion der aktuelle Kontext gesichert wird. Dazu gehören z.B. der Stackzeiger, und damit verbunden die lokalen Variablen, die auf dem Stack liegen. Im Anhang B.3 findet man ein ausführliches Beispiel.

3.7 Codeoptimierung

“We should forget about small efficiencies, say about 97 % of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3 %.”

— Donald Knuth

Die **Optimierung** von generiertem Code verfolgt zwei Arten von Optimierungszielen.

- **Kurze Programme:** Die geschriebene Anzahl an Instruktionen ist minimal.
- **Schnell ausführbarer Code**: Die Anzahl der ausgeführten Instruktionen ist minimal. *Achtung: Entscheidend sind die Zyklen pro Instruktion. Während bei bei einem Reduced Instruction Set Computer (RISC) die Anzahl an benötigten Zyklen pro Instruktion eher gering und konstant ist, variiert die Anzahl an Zyklen für Complex Instruction Set Computern (CISC) je nach Befehlsart teilweise erheblich.*

Es ist jedoch nicht möglich, ein **globales** Minimum für beide Zielfunktionen zu erreichen. Wir unterscheiden zwei grundlegende **Klassen von Optimierungsverfahren**:

- **Onlineoptimierung:** Die Folge von Maschinencodeinstruktionen wird während der laufenden Codegenerierung *on-the-fly* optimiert. Online-Algorithmen verarbeiten ihre Eingabe bereits, bevor die Gesamtheit der Informationen verfügbar sind. Ein Beispiel ist die *Methode der verzögerten Codegenerierung.*
- **Offlineoptimierung:** Zunächst wird der gesamte Maschinencode generiert. Nachdem der Maschinencode für mindestens eine vollständige Funktion verfügbar ist, können darauf Optimierungsalgorithmen angewandt werden. Zumindest theoretisch kann ein globales Optimum gefunden werden.

In der Praxis muss ein Kompromiss zwischen beiden Strategien gemacht werden: Während die Onlineoptimierung meist aus Sicht der Laufzeit sehr effizient ist, erzeugt eine Offlineoptimierung hingegen bessere Resultate; auf Kosten der Rechenzeit.

Instruktion	Parameter	Semantik
ADDI	a, b, c	R[a] := R[b] + c

Tab. 3.13: Erweiterung der Iom-Befehlssatzarchitektur

3.7.1 Onlineoptimierung

Wir betrachten zunächst einige Verfahren der Onlineoptimierung.

3.7.1.1 Verzögerte Codegenerierung

Es liegt der folgende Maschinencode vor:

```
1 LOAD_IMM     R[1], 5             # R[1] := 5
2 ADD          R[0], R[0], R[1]    # R[0] := R[0] + R[1]
```

Der Inhalt des Registers R_0 wird also um die Konstante 5 erhöht. Nehmen wir nun an, dass unser Befehlssatz um die neue Instruktion `ADDI` (Add Immediate) erweitert wird. Dessen Definition ist in Tabelle 3.13 dargestellt. Der zweite Operand in dieser neuen Intruktion ist nun kein Register mehr, sondern eine Konstante. Man spart sich also die Instruktion `LOAD_IMM`. Nun ist es möglich, den Maschinencode von oben in eine einzelne Zeile zu schreiben:

```
1 ADDI         R[0], R[0], 5       # R[0] := R[0] + 5
```

Wir definieren nun einen **Onlinealgorithmus**, der diese Ersetzung systematisch vornimmt. Die Methode der **verzögerten Codegenerierung** schreibt erst dann einen Ausgabecode, wenn eindeutig entschieden werden kann, welche **optimale Instruktion** gewählt werden kann:

1. Wird durch den Parser eine **Konstante** verarbeitet, so verzichten wir zunächst auf die Generierung der Instruktion `LOAD_IMM`. Stattdessen wird die Konstante temporär gemerkt.

2. Wird durch den Parser ein **Operator** verarbeitet, so hängt die dafür nötige Codegenerierung von den **zuvor** verarbeiteten Operanden ab:

   - Ist einer der beiden Operanden eine Konstante, so wird die hier effiziente Instruktion vom Typ `ADDI` erzeugt.
   - Sind beide Operanden konstant, so wird die Methode der **Konstantenfaltung** angewandt (siehe nächster Abschnitt).
   - Andernfalls wird die konventionelle Instruktion `ADD` erzeugt.

3.7.1.2 Konstantenfaltung

Mathematische Terme deren Operanden alle konstant sind, müssen nicht in Maschinencode übersetzt werden. Stattdessen ist es hinreichend und performanter nur das Endresultat in Form einer **berechneten Konstante** auszugeben. Dies nennt man **Konstantenfaltung (Constant Folding)**. Das Verfahren gehört ebenfalls zur verzögerten Codegenerierung. **Beispiel:** Der folgende Code ...

```
$00000020 ALLOC_STACK 8                     # alloc(8)
$00000021 LOAD_IMMEDIATE @0 5 [8]           # @0 := 5 [8]
$00000022 PUSH @0 [8]                       # push(@0) [8]
$00000023 LOAD_IMMEDIATE @0 7 [8]           # @0 := 7 [8]
$00000024 LOAD_IMMEDIATE @1 3 [8]           # @1 := 3 [8]
$00000025 LOAD_IMMEDIATE @2 2 [8]           # @2 := 2 [8]
$00000026 MUL @1 @1 @2 [8]                  # @1 := @1 * @2 [8]
$00000027 ADD @0 @0 @1 [8]                  # @0 := @0 + @1 [8]
$00000028 PUSH @0 [8]                       # push(@0) [8]
$00000029 LOAD_IMMEDIATE @0 8 [8]           # @0 := 8 [8]
$00000030 PUSH @0 [8]                       # push(@0) [8]
$00000031 LOAD_IMMEDIATE @0 9 [8]           # @0 := 9 [8]
$00000032 PUSH @0 [8]                       # push(@0) [8]
$00000033 CALL $00000000                    # call factorial
$00000034 POP 4                             # pop(4)
$00000035 STORE_LOCAL @0 FP+-8 [8]          # f := @0 [8]
$00000036 LOAD_LOCAL @0 FP+-8 [8]           # @0 := f [8]
$00000037 LOAD_IMMEDIATE @1 7 [8]           # @1 := 7 [8]
$00000038 CMP_GREATER @0 @0 @1              # @0 := @0 > @1
$00000039 BRANCH_IF_ZERO @0 $00000044       # if(@0==0) { jump $00000044; }
$00000040 LOAD_LOCAL @0 FP+-8 [8]           # @0 := f [8]
$00000041 LOAD_IMMEDIATE @1 1 [8]           # @1 := 1 [8]
$00000042 ADD @0 @0 @1 [8]                  # @0 := @0 + @1 [8]
$00000043 STORE_LOCAL @0 FP+-8 [8]          # f := @0 [8]
$00000044 LOAD_IMMEDIATE @0 0 [8]           # @0 := 0 [8]
$00000045 RETURN_WITH_VALUE                 # return @0
```

Abb. 3.14: Veranschaulichung des Peephole Ansatzes zur Codeoptimierung.

```
1 LOAD_IMM      R[1], 3             # R[1] := 3
2 LOAD_IMM      R[1], 5             # R[1] := 5
3 ADD           R[0], R[0], R[1]    # R[0] := R[0] + R[1]
```

kann durch eine einzelne Instruktion ersetzt werden

```
1 LOAD_IMM      R[0], 8             # R[0] := 8
```

3.7.2 Offlineoptimierung

Offlinealgorithmen haben Zugang zum gesamten Maschinencode, oder wenigstens zu einer vollständig übersetzten Funktion. Zunächst wird auf den Ansatz der **Peephole-Optimierung** eingegangen.

3.7.2.1 Peephole-Optimierung

Ein eingegrenztes **Peephole** (Fenster) wird über den generierten Maschinencode geschoben. Nur der jeweils aktuell darunter fokussierte Code wird für die weitere Optimierung herangezogen. Vgl. dazu Abb. 3.14. Die folgenden Beispiele demonstrieren den Ansatz exemplarisch:

- Langsame Instruktionen können ähnlich wie bei der verzögerten Codegenerierung durch **schnellere Instruktionen** ersetzt werden. Die Multiplikation von $x \cdot 2 \quad \ldots$

```
1 LOAD_LOCAL     R[0], -8            # R[0] := MEM[FP-8]
2 LOAD_IMM       R[1], 2             # R[1] := 2
3 MUL            R[0], R[0], R[1]    # R[0] := R[0] * R[1]
```

  kann auch durch eine Shift-Operation der Form $x << 1$ ersetzt werden:

```
1 LOAD_LOCAL     R[0], -8            # R[0] := MEM[FP-8]
2 SHIFT_LEFT     R[0], R[0], 1       # R[0] := R[0] << 1
```

- Entfernen **redundanter Speicher- und Ladeinstruktionen:**

```
1 LOAD_LOCAL     R[0], -8          # R[0] := MEM[FP-8]
2 ADDI           R[0], R[0], 5     # R[0] := R[0] + 5
3 SAVE_LOCAL     R[0], -8          # MEM[FP-8] := R[0]
4 LOAD_LOCAL     R[0], -8          # R[0] := MEM[FP-8]
5 SUBI           ...               # ...
6 SAVE_LOCAL     R[0], -8          # MEM[FP-8] := R[0]
7 ...
```

Kürzerer Code:

```
1 LOAD_LOCAL     R[0], -8          # R[0] := MEM[FP-8]
2 ADDI           R[0], R[0], 5     # R[0] := R[0] + 5
3 SUBI           ...               # ...
4 SAVE_LOCAL     R[0], -8          # MEM[FP-8] := R[0]
5 ...
```

Die Speicher- und Ladeinstruktionen in den Zeilen 3 und 4 des ersten Listings heben sich gegenseitig auf.

- **Optimierung des Kontrollflusses:**

```
0:  LOAD_LOCAL        R[0], -8          # R[0] := MEM[FP-8]
1:  LOAD_LOCAL        R[1], 1           # R[1] := 1
2:  BRANCH_EQUAL      R[0], R[1], A     # if R[0] == R[1] goto A
3:  BRANCH            B                 # goto B
A:  ... block ...
A1: BRANCH            C
B:  ... block ...
C:  ... block ...
```

Kürzerer Code:

```
0:  LOAD_LOCAL        R[0], -8          # R[0] := MEM[FP-8]
1:  LOAD_LOCAL        R[1], 1           # R[1] := 1
2:  BRANCH_NOT_EQUAL  R[0], R[1], B     # if R[0] != R[1] goto B
A:  ... block ...
A1: BRANCH            C
B:  ... block ...
C:  ... block ...
```

Die Invertierung des Gleichheitsoperators, sowie der Austausch der *goto*-Zieladressen erlaubt es, eine Sprung-Instruktion des originalen Listings einzusparen.

- **Entfernen von unerreichbarem Code:**

```
A: BRANCH            C                  # goto C
B: ...
C: ...
```

kann ersetzt werden durch:

```
1 A: BRANCH            C                  # goto C
2 C: ...
```

Achtung: Es muss sichergestellt werden, dass B nicht in anderen Codebereichen als Sprungziel verwendet wird.

- **Fortpflanzung von Kopien (Copy Propagation)**:

```
1 int x = y;
2 int z = x + 314;
```

kann ersetzt werden durch:

```
1 int z = y + 314;
```

Achtung: Es muss sichergestellt werden, dass `x` nicht auch anderswo verwendet wird!

- **Algebraische Vereinfachungen.** Erstes Beispiel:

```
1 LOAD_LOCAL          R[0], -8            # R[0] := MEM[FP-8]
2 LOAD_IMMEDIATE      R[1], 0             # R[1] := 0
3 ADD                 R[0], R[0], R[1]    # R[0] := R[0] + R[1]
```

Kann ersetzt werden durch:

```
1 LOAD_LOCAL          R[0], -8            # R[0] := MEM[FP-8]
```

Zusätzliches Beispiel für eine **Identität**: Man kann `x*1` durch `x` ersetzen. In der Praxis kommen solche Identitäten direkt im Code nur sehr selten vor. Betrachten wir aber nun das folgende Beispiel, und gehen davon aus, dass der Speicherbedarf für `char` genau eins ist:

```
1 char a[10];
2 int x = a[i];    /* sei  i  vorab definiert worden */
```

Unter Verwendung von Zeigern kann der Code auch wie folgt geschrieben werden:

```
1 char a[10];
2 int x = *(a+i);
```

Dann wird die Adresse des i-ten Elements des Arrays wie folgt berechnet:

$$\text{address}(a) + i \cdot \text{sizeof(char)} = \text{address}(a) + i \cdot 1$$

Wird jetzt hierfür Maschinencode generiert, so findet man die Identität $i \cdot 1$ bei der Multiplikation, und kann sie durch i ersetzen.

- **Algebraische Vereinfachungen.** Zweites Beispiel: Seien a, b, c, d und e skalare Variablen:

```
1 a =  b + c;
2 d = (b + c) * e;
```

Durch Einfügen einer weiteren skalaren Variable f können **gemeinsame Ausdrücke eliminiert** werden.

```
f = b + c;
a = f;
d = f * e;
```

Nun kann eine Operation für die Addition eingespart werden. Die neu entstandene Kopierfortpflanzung kann nun, wie oben erklärt, noch beseitigt werden.

3.7.2.2 Schleifenoptimierung

Die Optimierung von Schleifen hat insbesondere für numerische Verfahren eine essentielle Bedeutung. Ein Großteil der Ausführungszeit eines Programms wird in **inneren Schleifen** verbracht. Unter Betrachtung der Laufzeitkomplexität (siehe Kapitel 1.1) hat die Schleifenoptimierung einen Einfluss auf den Faktor c aus der Definition der $\mathcal{O}$-Notation. In der Literatur sind sehr viele Verfahren beschreiben. Hier betrachten wir nur einen dieser Ansätze: das **Ausrollen von Schleifen**.

Moderne CPUs beinhalten große **Pipelines**, die effizient ausgenutzt werden sollten. Sprung-Instruktionen erfordern es, dass diese Pipeline (zumindest zum Teil) geleert und wieder befüllt werden muss. Während eine direkt in die CPU implementierte Sprungvorhersage das Problem reduzieren kann, ist der schnellste Maschinencode so strukturiert, dass **möglichst wenige Sprünge** durchgeführt werden. Indem Schleifen **ausgerollt** werden, entfallen einige der Sprünge, die zurück auf den Schleifenkopf verweisen.

Beispiel:

```
double a = 1.0;
for(int i=2; i<6; i++)
  a *= (double)i;
```

Die Schleife kann durch den folgenden Code substituiert werden:

```
double a = 1.0;
a *= 2.0;
a *= 3.0;
a *= 4.0;
a *= 5.0;
```

Als Nachteil ist vor allem die **wachsende Programmgröße** zu nennen. Es ist aber durchaus auch sinnvoll, Zwischenlösungen so zu definieren, sodass nur wenige Iterationen mehrfach geschrieben werden müssen:

```
double a = 1.0;
for(int i=0; i<2; i++) {
  a *= 2.0 + 2.0*(double)i;
  a *= 3.0 + 2.0*(double)i;
}
```

Eine optimale Lösung kann über eine **mathematische Optimierung** gefunden werden.

Anmerkung: Das Problem liegt in der Klasse NP und kann zum Beispiel als Integer Linear Program (ILP) in der Form

$$\left\{\max_x \mid Ax \leq b \ \wedge \ x \in \mathbb{Z}^n\right\}$$

modelliert werden.

3.7.2.3 Registereinsatz

Reduced Instruction Set Computer (RISC) -Architekturen verfügen meist eine **große Zahl an Prozessor-Registern** (zum Beispiel 32). Die Zugriffszeiten auf Register sind sehr kurz; insbesondere im Vergleich zu Zugriffen auf den Hauptspeicher (RAM). In den vergangenen Diskussionen haben wir Register stets nur zur Pufferung von Zwischenergebnissen bei der Auswertung von mathematischen Ausdrücken eingesetzt. Betrachten wir z.B. den zunächst eher lang erscheinenden Ausdruck

$$(((a + b) * c + 1) * d + 3) * e$$

, so sind dennoch nur zwei Register als Zwischenregister erforderlich. Es ist zudem meist so, dass die durchschnittliche Länge von Ausdrücken in Programmen eher gering ist. Folglich bleibt mit unseren bisherigen Ansätzen die Mehrzahl an Registern **ungenutzt**.

Wir suchen nun nach eine Strategie, welche die **meistgenutzten lokalen Variablen** in Registern hält. Statistisch gesehen kann so die Anzahl an Zugriffen auf den Hauptspeicher drastisch reduziert werden. Ein möglicher Ansatz ist, die Problemstellung der Registerallokation auf das Problem der **Graphfärbung** zu reduzieren. Die Theorie dazu haben wir bereits in Kapitel 1.2 kennengelernt. Der Einsatz von Graphfärbung für die Registervergabe wurde zuerst in [4] beschrieben.

Zur Erinnerung: Die chromatische Zahl beschreibt die kleinst mögliche Anzahl an Farben, mit denen ein Graph eingefärbt werden kann. Benachbarten Knoten haben eine unterschiedliche Farbe.

Modellierung:

- Die **Knoten** des Graphen beschreiben den **Lebensbereich** der Variablen.
- Die **Kanten** des Graphen verbinden zwei Lebensbereiche, wenn beide an mindestens einer Stelle im Programm interferieren.

Man nennt dies einen **Inferenzgraphen**. Der Graph wird durch eine **Lebendigkeitsanalyse** aufgestellt:

- Eine Variable **lebt** in einem bestimmten Programmpunkt, wenn ihr Wert in der Zukunft noch einmal verwendet wird.
- Andernfalls ist eine Variable **tot.**

Letztendlich wird das Problem der Registervergabe durch das Lösen des **Graphfärbungsproblems** gelöst.

Beispiel: Gegeben ist der folgende Codeausschnitt[5] in der Programmiersprache C:

```
1 int x = 0;
2 int y = 5;
3 while(x < 8) {
4   int z = x * 2;
5   x = z + 1;
6   y = y + x;
7 }
8 y = y - 1;
```

Zunächst wird die Lebendigkeit aller Variablen untersucht. Dazu notieren wir die Verkettung der Zeilennummern des Programms:

$$x = \{1 \to 2 \to 3 \to 4, \quad 5 \to 6 \to 3 \to 4\}$$
$$y = \{2 \to 3 \to 4 \to 5 \to 6, \quad 6 \to 7 \to 8\}$$
$$z = \{4 \to 5\}$$

Zum Beispiel wird die Variable x in Zeile 1 definiert und lebt dann bis Zeile 4 weiter. Dort wird sie letztmalig verwendet, bevor sie in Zeile 5 wieder überschrieben wird. Von dort aus verfolgen wir sie weiter über Zeile 5 und springen zurück an den Kopf der Schleife.

Wir interessieren uns nun für die Schnittmengen zwischen den oben erstellten Lebendigkeiten der Variablen. Dazu bildet man **alle Permutation**[6], also $v_i \cap v_j$ mit $i \neq j$. Hier sind das $x \cap y, x \cap z, y \cap z$. Nun erhält man:

$$x \cap y = \{2 \to 3 \to 4, \quad 5 \to 6\}$$
$$x \cap z = \emptyset$$
$$y \cap z = \{4 \to 5\}$$

Nun zeichnet man einen Graphen mit der Knotenmenge $V(G) = \{x, y, z\}$ und der Kantenmenge $E(G) = \{(x, y), (y, z)\}$. Die Kante (x, z) gehört nicht zum Graphen, da $x \cap z$ der leeren Menge entspricht.

Nun wählen wir die Farbe 1 für den Knoten x. Dann muss der Nachbar y eine andere Farbe erhalten: wir wählen 2. Der Knoten z hat den Knoten y als Nachbarn, also darf die Farbe nicht 2 sein; aber 1 ist erlaubt.

Nun liegt eine 2-Färbung für den Graphen vor. Dies ist auch die chromatische Zahl, also $\chi(G) = 2$. Antwort: Wir können die Menge der Programmvariablen $\{x, y, z\}$ in genau zwei Registern unterbringen:

[5] Ähnlich zu einem Beispiel aus http://www.cs.colostate.edu/~mstrout/CS553/slides/lecture03.pdf

[6] Sei n die Anzahl der Variablen. Dann ist die Anzahl der Permutationen $n!/2$.

- Die Variablen x und z teilen sich ein Register, da beide die gleiche Färbung 1 haben.
- Die Variable y benötigt ein separates Register.

3.7.3 Aktuelle Entwicklungen

Moderne Compiler nutzen zusätzlich u.a. die folgenden Ansätze:

- **Automatische Parallelisierung:** Die folgende Beispielfunktion hat Potenzial effektiv auf Vektorprozessoren ausgeführt zu werden, denn die Anweisung in Zeile 3 hängt weder von vor- noch von nachgeschalteten Schleifeniterationen ab:

```
1 int add(double *a, double *b, double *c, int n) {
2   for(int i=0; i<n; i++)
3     c[i] = a[i] + b[i];
4 }
```

- **Interprozedurale Optimierung:** Hier bezieht man nicht nur in sich geschlossene Funktionen, sondern gleichzeitig das gesamte Eingabeprogramm in die Optimierung ein.

Wenn Sie Interesse an einer Vertiefung haben, sprechen Sie den Dozenten gerne an. Eine Forschungs- oder Abschlussarbeit zu diesem Thema wird sehr gerne durch Mitbetreuung unterstützt.

3.8 Beispielcompiler

3.8.1 Iom

Begleitend zur Lehrveranstaltung wird ein einfacher Beispielcompiler bereitgestellt. Dieser übersetzt die imperative Allzwecksprache **Iom**[7] in einen einfachen Maschinencode. Der Compiler ist selbst in der Sprache C implementiert und nutzt **Flex** für die lexikalische Analyse und **GNU Bison** für das Parsing. Verfügbar ist er unter https://gitlab.com/andreas.schwenk/iom.git. Nähere Informationen findet man in Anhang B.

3.8.2 Clang und LLVM

Anhand des Beispiels von **Clang** und **LLVM** kann die Sinnhaftigkeit der **Entkopplung von Front- und Backend** motiviert werden. Vgl. dazu auch Abb. 3.15.

- **Clang** ist ein Compiler Frontend, welches ursprünglich von *Apple Inc.* entwickelt wurde. Clang nutzt LLVM als Backend.
- **LLVM** ist eine Compiler-Infrastruktur, welche ab dem Jahr 2000 an der Universität von Illinois (USA) entwickelt wurde. LLVM besteht aus Werkzeugen für die Entwicklung von Compiler Frontends und -backends.

[7] Esperanto für "wenig"

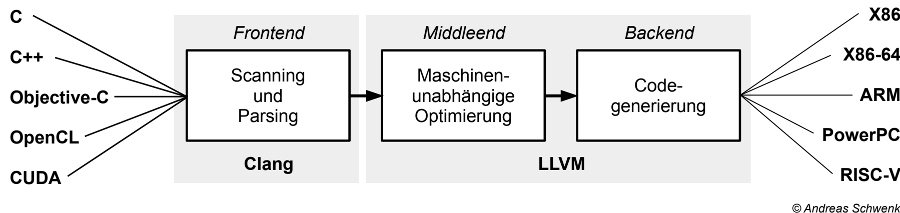


Abb. 3.15: Clang und LLVM

Kapitel 4

Sprachentwurf

"Make everything as simple as possible, but not simpler."

— Albert Einstein

In diesem Kapitel betrachten wir Methodiken zum **Entwurf** neuer domänenspezifischer Sprachen. Dazu zählen zum Beispiel Techniken aus der **Domänenanalyse**, sowie der **Domänenmodellierung**. Sowohl **technische DSLs** als auch **DSLs aus Anwendungsdomänen** können mit Hilfe einer Menge von **Spracheigenschaften** kategorisiert werden. Ein Satz von **Entwurfsdimensionen** evaluiert die DSL Prototypen. Verschiedene **Verhaltensparadigmen** und **Sprachkonzepte** runden die Darstellungen ab.

*Anmerkung: Während es auch **grafische** und **symbolische** domänenspezifische Sprachen gibt, werden wir in diesem Kapitel ausschließlich **textuelle** DSLs betrachten. Sollte man in eigenen Projekten selbst eine grafische DSLs erstellen, so hat sich die folgende Verfahrensweise bewährt:*

1. *Zunächst entwirft und entwickelt man eine textuelle DSL.*
2. *Danach fügt man eine grafische Erweiterung hinzu, welche die textuelle DSL als Abhängigkeit verwendet.*

Alternativ nutzt man Sprachwerkbänke mit entsprechender Unterstützung. Ein Beispiel ist JetBrains MPS.

4.1 Motivation

Zunächst sollte hinterfragt werden, wieso es überhaupt sinnvoll ist, neue domänenspezifische Sprachen zu implementieren. Allem voran besteht die Gefahr dass der *"Language Zoo"* in Zukunft immer größer werden wird.

Betrachten wir zunächst die Anzahl an ExpertInnen aus dem Bereich Informatik und fachfremden Domänen, so stellen wir in Abb. 4.1 fest, dass es nur eine kleine Anzahl an ExpertInnen gibt, die auf beiden Gebieten über ein großes Wissen verfügen. DSLs

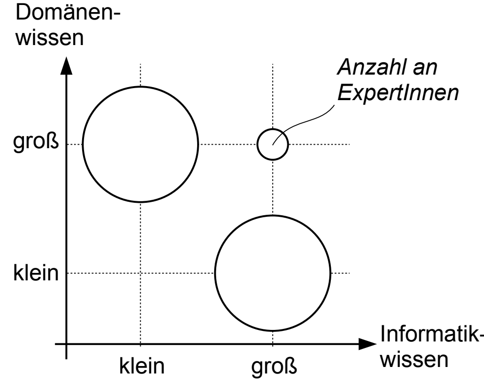


Abb. 4.1: Expertenwissen

erleichtern also **interdisziplinäres Arbeiten**, indem **formale Kommunikationsschnittstellen** zwischen den Stakeholdern geschaffen werden. Weiterhin versuchen domänenspezifische Sprachen den Quellcode wieder in eine verständlichere Form zu bringen:

> *"Any fool can write code that a computer can understand.*
> *Good programmers write code that humans can understand"*
>
> — *Martin Fowler*

4.2 Arten von DSLs

Das Anwendungsgebiet domänenspezifischer Sprachen ist sehr heterogen. Wir nehmen hier die folgende, grundlegende Kategorisierung vor:

- Eine **technische** DSL wird primär von InformatikerInnen und IngenieurInnen eingesetzt. Für technische DSLs ist es praktikabel auf bereits **existierende Frameworks** zurückgreifen zu können. **Vorhandenes Wissen** wird in eine neue DSL transformiert. Der Domänenrahmen ist *à priori* bekannt. Die Domäne kann anhand einer Menge von Programmen, die bereits in einer Allzwecksprache (GPL) geschrieben wurden, abgesteckt werden. Ein Beispiel ist eine DSL, welche Zustandsautomaten für die Hardwareentwicklung modelliert.

- Eine DSL für eine **Anwendungsdomäne** wird in einem Kontext außerhalb der Informatik eingesetzt. Für DSLs in Anwendungsbereichen ist das Domänenwissen oft nicht für den Sprachersteller von Beginn an verfügbar. Eine **Domänenanalyse** muss zunächst durchgeführt werden. Ein Beispiel ist eine DSL, die EnergieexpertInnen dazu in die Lage versetzt, Vorhersagen über zukünftige Technologien aufzustellen.

In der Quelle [21] wird hingegen eine feinere Subunterteilung vorgenommen: Z.B. werden Hilfs-DSLs, Architektur-DSLs, vollständig technische DSLs, DSLs für das Anforderungsengineering, Produktionsengineering usw. genannt.

4.3 Definition

Gemäß [7] besteht die **Definition** einer domänenspezifischen Sprache aus drei Teilen:

1. Definition eines **Schemas / Metamodells.** Zum Beispiel wird zunächst ein abstrakter Syntaxbaum (AST) entwickelt.
2. Erstellung und Integration von **Editoren**, inklusive der Sprachgrammatik.
3. Implementierung eines **Generators**.

Anmerkung: Die Quelle [21] stimmt nicht damit überein, dass die Reihenfolge der Punkte 1.) und 2.) für alle Fälle geeignet ist. Es wird beschrieben, dass es auch sinnvoll sein kann, mit dem Aufstellen einer formalen Grammatik zu beginnen.

Definition 39 (Programmfragment) *Wir definieren ein (Programm-)**Fragment** als einen Sourcecodeausschnitt, der in der Syntax einer DSL geschrieben ist. Ein Fragment muss weder vollständig noch in sich geschlossen sein.*

Oftmals muss initial von einem nur rudimentären Verständnis der unterliegenden **Domäne** ausgegangen werden. Entsprechend können zunächst nur einfache **Programmfragmente** geschrieben werden. Die DSL wächst mit der Zeit sukzessive heran.

4.4 Abgrenzungen

Es gilt zu prüfen, ob die zu entwickelnde DSL nicht eventuell auch **eingebettet** sein kein. Hierzu findet man in Kapitel 5 weiterführende Informationen.
Die folgende Liste grenzt beide Konzepte voneinander ab:

Eingebettete DSLs:

- Eine **eingebettete** DSL nutzt eine vorhandene **Hostsprache**.
- Die Toolchain von eingebetteten DSLs ist **einfacher zu implementieren,** da die zu Grunde liegende Hostsprache bereits verfügbar ist.
- Meist gibt es **keinen dedizierten IDE Support.**
- Domänenspezifische semantische Fehler werden nicht aufgedeckt.
- Es gibt **keine Portabilität** im Rahmen des Wechsels zu einer anderen Hostsprache (außer z.B. C → C++).

Externe DSLs:

- Externe DSLs repräsentieren **eigenständige Sprachen** und werden entlang ihrer Domäne entworfen.

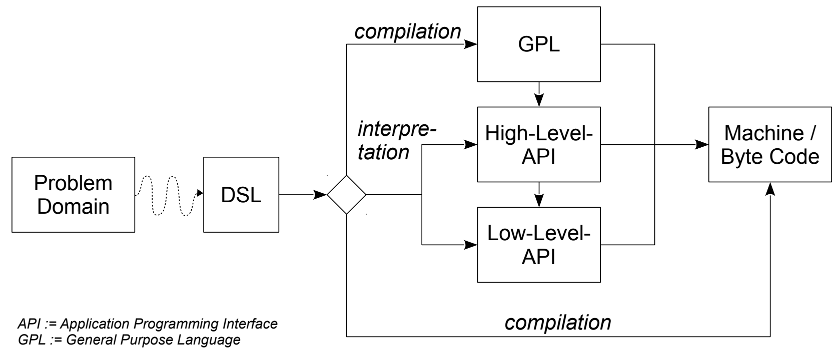


Abb. 4.2: Integrationsmöglichkeiten von DSLs

- Domänenbezogene, semantische Fehler werden teilweise aufgedeckt.
- Die DSL-Nutzer müssen eine neue Sprache **lernen**.
- Die Sprache muss **gewartet** werden.

Weiterhin soll nun noch der Unterschied zwischen der DSL-Entwicklung und dem **allgemeinen Compilerbau** aufgezeigt werden:

- DSLs befassen sich oft mit der zeitgleichen Definition einer zugehörigen **Integrierten Entwicklungsumgebung (IDE).**
- GPL-Compiler generieren für gewöhnlich Maschinencode, während DSLs oft in Allzwecksprachencode übersetzt werden.
- Compilerbau für GPL befasst sich in großem Maßstab mit der Optimierung von Zwischen- und Maschinencode; DSLs befassen sich meist nicht damit.

4.5 Integration von DSLs

Abbildung 4.2 veranschaulicht die verschiedenen Integrationsmöglichkeiten von DSLs.

- Oftmals wird eine DSL in eine Allzwecksprache übersetzt. Die **Allzwecksprache** selbst kann auf existierende Programmierschnittstellen (API) zurückgreifen. Gegebenenfalls sind eine Reihe von Zwischenschritten nötig. Beispiele:
  1. Energiesprache → Gleichungen → Solver → C++
  2. Hausautomatisierung → Sensoren → Zustandsautomaten → C
- Weiterhin ist aber auch die **Interpretation** von DSL Programmen möglich. In diesem Fall ruft die Ausführungsumgebung High-Level oder Low-Level Funktionen auf.
- Nur sehr selten werden DSL-Programme direkt in **Maschinencode** oder **Bytecode** übersetzt. *Beispiel: Eine High-Level Shader Langauge (HLSL) zur Ausführung auf Grafikprozessoren.*

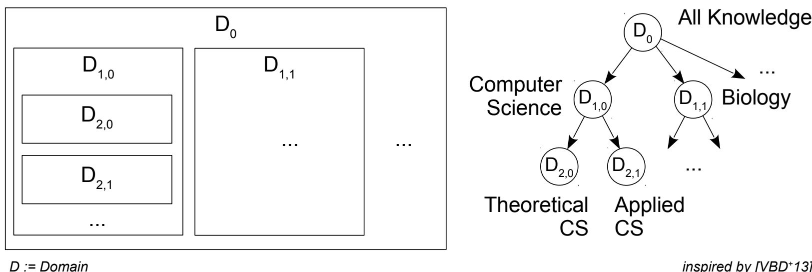


Abb. 4.3: Domänenhierarchie

4.6 Domänenmodellierung

Ein **Modell** ist eine vereinfachte **Abstraktion** der realen Welt. Eine der grundlegenden Herausforderungen ist es, eine neue DSL so zu definieren, dass die gewählte Abstraktion **hinreichend** ist. Weiterhin darf diese Abstraktion die **Anforderungen nicht übersteigen,** um den Implementierungsaufwand und somit Kosten gering zu halten.

Definition 40 (Domäne) *Eine **Domäne** ist eine Teilmenge des gesamten Wissens; sozusagen eine “Sphäre” von Wissen mit gemeinsamer Terminologie und gemeinsamen Eigenschaften.*

Um eine unterliegende **Anwendungsdomäne** in eine DSL zu überführen, muss die Domäne zunächst systematisch analysiert werden. Gemäß Abb. 4.3 können Domänen ein einer hierarchischen Struktur aufgefasst werden:

- Die Top-Domäne D_0 stellt die **Wurzel** aller Domänen dar. Eine zugehörige Programmiersprache ist eine Allzwecksprache.
- Jeder Kindknoten repräsentiert eine **Subdomäne** D_i, welche spezialisierter als der jeweilige Elternknoten ist.

4.6.1 Domänenanalyse

Die **Analyse** einer Domäne kann zum Beispiel wie folgt in den Entwicklungsprozess integriert werden:

1. **Interview** mit Domänenexperten.
2. **Strukturierung** des gewonnenen Wissens. *Achtung: Auch ExpertenInnen tendieren dazu, unpräzise und/oder inkonsistente Anforderungen zu definieren!*
3. Erzeugen von **Beispielen**.
4. **Erstellen** der Sprache.
5. **Feedback** erfragen und einbeziehen.

Dieser Prozess wird **iterativ** durchgeführt:

- Zu Beginn wird lediglich ein kleiner Teil der Domäne betrachtet.
- Ein **Wasserfallmodell** sollte **vermieden** werden: Stattdessen sollten Beispiele iterativ erweitert werden.
- Eine besondere Herausforderung ist, dass **Anforderungen** sich manchmal im Laufe der Zeit **ändern**. Domänenwissen ist nur selten statisch. Auch die Zielplattform oder die syntaktischen Anforderungen an eine Sprache können sich ändern.

Die Gruppe der **Stakeholder** aus dem Domänenbereich ist in **ExperteInnen** und **BenutzerInnen** unterteilt:

- **DomänenexpertInnen** können dabei behilflich sein, eine neue Sprache zu **entwerfen**. Sie verfügen meist sogar über ein formales Verständnis ihrer Domäne.
- **DomänennutzerInnen** sind weniger erfahren als ExpertInnen. Sie eignen sich ideal, um eine neue Sprache hinsichtlich ihrer Alltagstauglichkeit zu **testen**.

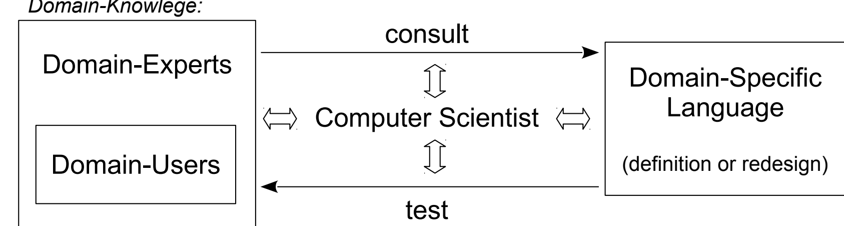


Abb. 4.4: Einbeziehung der DomänennutzerInnen und -expertenInnen

Bei der **DSL-Entwicklung** können zunächst Codefragmente für Subdomänen entworfen werden. Diese Herangehensweise ist mit einem Brainstorming vergleichbar (vgl. mit Abb. 4.5):

1. Zunächst wird mit einer Menge von Fragmenten $f_{i,j}$ begonnen.
2. In einem weiteren Schritt wird eine **Menge von Fragmenten** $f_k := \cup f_{i,j}$ als **zusammenhängende Einheit** identifiziert.
3. Die Syntax aller betroffen Fragmente wird dann **angepasst** bzw. **angeglichen**.
4. Letztendlich mündet dieser iterative Ansatz in einem Ergebnisfragment f_0, welches als erste **implementierbare Sprachrevision** angesehen werden kann.

Beispiel: Für die Energiesprache CEMPL konnten zunächst unabhängige Bausteine für Energiespeicher ($f_{1,1}$) und Energiekonverter ($f_{1,2}$) entwickelt werden. Im Rahmen der Definition von Energienetzen ($f_{1,3}$) konnten alle drei Konzepte syntaktisch vereint und in das übergeordnete Fragment f_1 überführt werden.

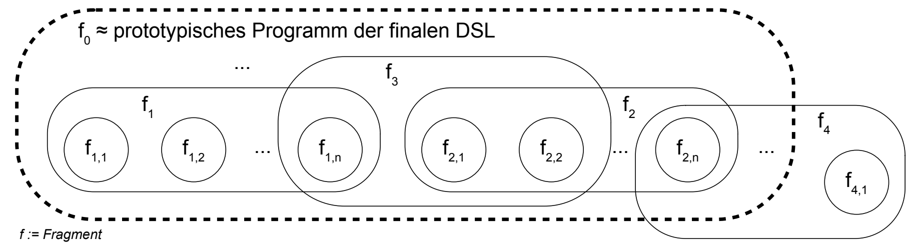


Abb. 4.5: Zusammenschluss von DSL-Fragmenten

4.7 Sprachen und Programme

Bevor wir die Eigenschaften von Sprachen vertiefen, entwickeln wir zunächst einige Definitionen:

Definition 41 (Programm) *Ein **Programm** setzt einen Algorithmus in einer formalen Sprache um.*

Definition 42 (Sprachkonzept) *Ein **Konzept** einer Sprache ist ein **logischer Teil** der Sprache.*

Zum Beispiel ist die *for-Schleife* ein Konzept einer Allzwecksprache. Die Syntax eines deskriptiv beschriebenen technischen Geräts ist ein beispielhaftes Konzept einer DSL.

Definition 43 (DSL) *Sei L eine **Sprache**, die eine **Konzeptmenge** einer **Domäne** umsetzt. Dann ist L eine **DSL**.*

Jedoch ist z.B. noch nicht garantiert, dass eine **gut gewählte Abdeckung** vorliegt. Zur genaueren Analyse werden nun die Eigenschaften einer Sprache näher analysiert.

4.7.1 Größe und Umfang einer Sprache

Definition 44 (Sprachgröße) *Die **Größe** einer Sprache beschreibt die **Anzahl** ihrer unterstützten **Konzepte**.*

Der Begriff ist nicht messbar. Wir nehmen nun eine grobe Unterteilung vor. Vgl. dazu auch Abb. 4.6.

- **Große Sprachen** weisen viele Sprachkonzepte auf. Diese sind meist sehr speziell, und damit nicht notwendigerweise wiederverwendbar. Ein Beispiel für eine große Sprache ist SAP/ABAP aus dem wirtschaftlichen Bereich.
- **Kleine Sprachen** haben wenige, aber **orthogonale Sprachkonzepte.** Sie sind hochgradig **abstrahierbar**. Nachteilig ist die schwere Erlernbarkeit. Es erfordert für den NutzerInnen Erfahrung, um komplexere Systeme zu erschaffen. Ein Beispiel für eine kleine Sprache ist C.
- **Modulare Sprachen** versuchen die Vor- und Nachteile von großen und kleinen Sprachen zu vereinen. Sie sind **modular erweiterbar**.

Definition 45 (Sprachumfang) *Der **Umfang** einer Sprache beschreibt die **Größe** ihrer abgebildeten **Domäne**.*

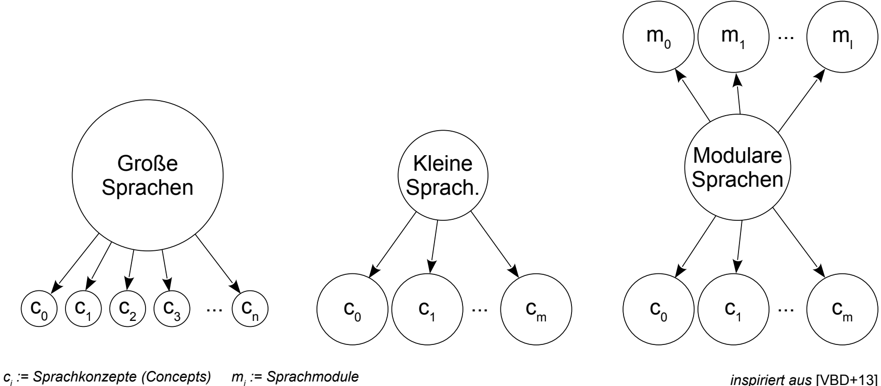


Abb. 4.6: Verschiedene Größen von Sprachen

4.7.2 Übereinstimmung von Domäne und Sprache

Wir definieren nun $P(L)$ als die **Menge aller Programme** $\{p_1, p_2, \dots\}$, die mit der **Sprache** L ausgedrückt werden können. I.d.R. sind dies unendlich viele. Sei nun $P(D)$ die **Menge aller Programme**, die bezüglich der Domäne D ausgedrückt werden können. Wir interessieren uns für die **Maximierung der Schnittmenge von** $P(L) \cap P(D)$. Herausforderung:

- Eine konkrete Sprache $L(D)$ für eine Domäne D sollte **ausdrucksstärker** sein, als alle anderen Sprachen. Insbesondere sollte die Sprache ausdrucksstärker sein als Allzwecksprachen.

- In einem iterativen Prozess sollte die DSL so spezifiziert werden, dass sie die Domäne weder **unvollständig**, noch **zu generisch** abbildet.

Der Begriff der **Ausdrucksstärke** drückt sich vor allem durch **kurzen Programmcode** aus. Die Thematik wird im nächsten Abschnitt weiter vertieft.

4.8 Entwurfsdimensionen

Die Quelle [21] gibt sieben **Entwurfsdimensionen** für domänenspezifische Sprachen an. Diese werden nun im Detail erläutert.

4.8.1 Ausdrucksstärke

DSL Programme sind für gewöhnlich **kürzer** und **präziser**, als eine äquivalente Implementierung in einer Allzwecksprache. Wir nennen dies eine gute **Ausdrucksstärke (Expressivity)**.

- Die Semantik kann schnell aufgefasst werden.

- Irrelevante Details werden, z.B. durch API-Aufrufe, in die Ausführungsumgebung verlagert.

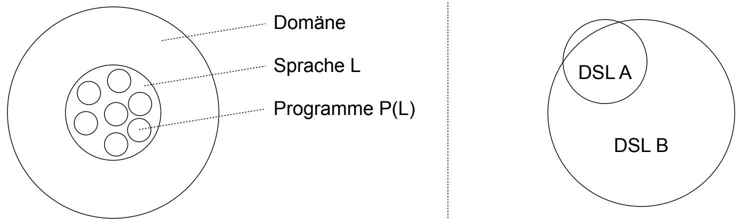


Abb. 4.7: Links: Eine Sprache, die ihre Domäne nur unvollständig abdeckt. Rechts: Sprache B kann große Teile von Sprache A wiederverwenden (Modularität).

Vergleicht man zwei Sprachen L_1 und L_2, dann schreiben wir $L_1 \prec L_2$, wenn die Menge der Programme $P(L_1)$ ausdrucksstärker ist, als die Menge der Programme $P(L_2)$. Das **deklarative** Programmierparadigma (siehe unten) begünstigt die Ausdrucksstärke.

4.8.2 Abdeckung

Die **Abdeckung (Coverage)** einer Sprache beschreibt, inwiefern alle erdenklichen Unterprobleme einer Domäne abgedeckt werden können. Gibt es keine Möglichkeit, ein bestimmtes Problem einer Domäne als Programm zu formalisieren, dann ist die Sprache nicht von maximaler **Reichweite.** Die Sprache in Abb. 4.7 (links) ist zu eingeschränkt, um die Domäne vollständig abzubilden.

4.8.3 Semantik und Ausführung

Wir unterscheiden zwei Formen von **Semantik**:

- **Statische Semantik (Static Semantics):**
  Zum Beispiel hat der folgende C-Code **statische Typen,** die sich nicht während der Ausführung ändern:

```
1 int a = 42;
2 int b = 1337;
3 int c = a + b;
```

- **Ausführungssemantik (Execution Semantics):**
  Die Typen im folgenden Codeausschnitt sind erst zur Laufzeit bekannt; man beachte, dass Python eine interpretierte Sprache ist.

```
1 def add(a, b):
2   return a + b
3
4 c = add(3, 4)
5 s = add("Hello,␣", "World!")
```

  Der Typ der Parameter a und b ist erst während der Ausführung bekannt.

4.8.4 Trennung der Angelegenheiten

Eine **Angelegenheit (Concern)** ist ein Aspekt einer DSL, der eine **Teilmenge einer Domäne beschreibt**, **modelliert** oder **implementiert**. Für eine DSL aus dem Energiekontext, könnten z.B. Netzwerke, Speicher, Konverter, Szenarien genannt werden.

Man sollte sich immer die Frage stellen, ob es eine "weiche Integration" bei der Zusammenführung der einzelnen Angelegenheiten gibt.

4.8.5 Vollständigkeit

Die **Vollständigkeit (Completeness)** beschreibt, welcher Anteil an nötigen **Ausführungsinformationen** in der DSL selbst enthalten ist. Nicht direkt in die DSL eingebettete Funktionalität muss über zusätzliche Konfigurationsdateien oder Module ausgedrückt werden. Diese sind oft in einer Low-Level Sprache gekapselt. Eine DSL ist vollständig genau dann, wenn in der Zielsprache keine Blackbox-Funktionalität aufgerufen wird.

Achtung: Nicht mit der Turing-Vollständigkeit verwechseln!

4.8.6 Sprachmodularität

Eine domänenspezifische Sprache sollte nach Möglichkeit so entworfen werden, dass Teile oder sogar die gesamte DSL in einem neuen Kontext wiederverwendet werden kann. **Modularität (Modularity)** ist ein Gegenpol zur "DSL-Hölle"; zum "Sprachzoo". In dem Beispiel in Abb. 4.7 kann beim Entwurf der Sprache B ein großer Teil der bereits bestehenden Sprache A wiederverwendet werden.

4.8.7 Konkrete Syntax

Die **konkrete Syntax** sollte entlang der folgenden Fragen entworfen werden:

- **Lesbarkeit:** Kann ein DSL-Programm einfach verstanden werden; evtl. sogar von NichtprogrammiererInnen?

- **Schreibbarkeit:** Wie schwierig ist es, ein Programm in einer DSL zu schreiben? Ist die Syntax zugänglich?

- **Lernbarkeit:** Wie lange benötigt einE durchschnittlicheR DomänennutzerIn, sich in die neue DSL einzuarbeiten?

- **Effektivität:** Wie viel Zeit wird *nach* der Lernphase benötigt, um ein DSL-Programm umzusetzen?

4.9 Verhaltensparadigmen

Wir interessieren uns für domänenspezifische Sprachen, die nicht einfach nur deskriptiv sind, d.h. welche nicht ohne Einbußen direkt in JSON, XML oder anderen Markupsprachen umgesetzt werden können. Um die **Verhaltensaspekte** zu implementieren, werden nun einige Programmierparadigmen aufgezählt:

- Eine **imperative** Sprache verarbeitet eine Folge von Anweisungen. *Beispiele: C, C++, Java, Python.*
- Eine **deklarative** Sprache spezifiziert nicht explizit den Kontrollablauf. Es wird stattdessen z.B. eine Menge von Eigenschaften, Gleichungen und Nebenbedingungen deklariert. Zu den Untergruppen deklarativer Sprachen zählt z.B. die logische Programmierung. *Beispiele: Prolog, CEMPL*[1].
- Eine **funktionale** Sprache hat **keine Seiteneffekte:** Jede Funktion gibt einen Wert zurück, der **ausschließlich** von den Funktionsargumenten abhängt.
- Bekannte Beispiele für das **Datenflussparadigma** sind Tabellenkalkulationen *(z.B. LibreOffice Calc)*, oder Hardware Gatter.
- Im **ereignisgesteuerten** Ansatz wird der Kontrollfluss durch eingehende Ereignisse gesteuert. *Beispiele für Ereignisse: Sensoreingaben, Signale von Hardwaregeräten oder Nachrichten in der Interprozesskommunikation.*

DSLs folgen oft dem **deklarativen** Paradigma. Das gewählte Paradigma hängt maßgeblich von der konkreten Domäne ab.

4.10 Strukturierung

Eine Domäne ist meist **hierarchisch** strukturiert. Entsprechend ist ein in einer domänenspezifischen Sprache geschriebenes Programm derart **strukturiert**, dass auch große Programme **verwaltbar** bleiben und mit vertretbarem Aufwand verändert werden können. Die folgenden Methoden können die Strukturierung verbessern (vgl. mit[21]):

- **Modularisierung:** Programme können z.B. in **Namensräume** und **Funktionen / Methoden** unterteilt werden.
- **Partitionierung:** Die Definition eines Programms ist nicht auf eine Datei limitiert. Referenzierende **Importausdrücke** können weitere Dateien einbinden.
- **Sichtbarkeit:** Wie in Allzwecksprachen, können Variablen **lokal** oder **global** definiert sein. **Methoden** und **Attribute** können **öffentlich (public)** oder **nicht-öffentlich (private)** sein. Jeder Bezeichner gehört einem **Geltungsbereich (Scope)** an.

[1] An der TH-Köln entstand eine Sprache zur Formulierung von Optimierungsproblemen aus dem Energiebereich.

- **Typen, Objekte und Instanzen:** Manchmal ist es vorteilhaft, eigene Typen zu deklarieren. Als Beispiel sei auf den Typ "Energiespeicher" in einer Energiemanagementsprache verwiesen. Ein Typ kann **instanziiert** werden.
- **Spezialisierung:** Auch die **Vererbung** liefert ein mächtiges Werkzeug. Allerdings müssen auch aus Allzwecksprachen bekannte Probleme berücksichtigt werden, z.B. das **Diamond-Problem**[2].

4.11 Beispiele

Im Anhang C findet man Beispiele für DSLs in Form von Codeausschnitten.

[2]Sofern zwei Klassen B und C beide von einer Klasse A erben und Klasse D sowohl von Klasse B, als auch von Klasse C erbt, dann entsteht eine Mehrdeutigkeit: Überschreiben B und C beide eine Methode aus A, welche aber in D nicht überschrieben wird, dann stellt sich die Frage: Erbt D diese Methode von B oder von C?

Kapitel 5

Eingebettete Domänenspezifische Sprachen

“I’m not a great programmer;
I’m just a good programmer with great habits.”

— Martin Fowler

Anstelle von **eingebetteten** domänenspezifischen Sprachen spricht man auch häufig von **internen** domänenspezifischen Sprachen.

Definition 46 (Eingebettete DSL) *Eine* ***eingebettete DSL*** *verwendet eine zu Grunde liegende* ***Hostsprache*** *derart, dass der Eindruck einer eigenen Sprache entsteht. Die Hostsprache ist üblicherweise eine Allzwecksprache. Alle Eigenschaften der Hostsprache werden direkt an die eingebettete DSL vererbt.*

Die Hostsprache wird so angepasst, dass eine **höhere Abstraktion** möglich wird. Grundsätzlich können eingebettete DSLs so entworfen werden, dass ein **zusätzlicher Zwischenlayer** zu einer existierenden API geschaffen wird. Hierzu werden wir einige **Entwurfsmuster** kennenlernen. Eine eingebettete DSL benötigt (wie auch eine externe DSL) ein **semantisches Modell.**

Vorteile

- Meist **geringer Entwicklungsaufwand** im Vergleich zu externen DSLs.
- Bestehende Compiler und Toolchains können **wiederverwendet** werden. Insbesondere können auch existierende Softwarebibliotheken und APIs wiederverwendet werden.
- Die Menge der eingebetteten DSL pro Hostsprache kann ein **einheitliches Erscheinungsbild** haben. Dies wirkt sich positiv auf die **Lernrate** bei den DSL BenutzerInnen aus.
- Auch nicht-domänenspezifische Teile der Hostsprache können beerbt werden.

Nacheile

- DSL Programme werden unbenutzbar, sobald die Hostsprache gewechselt wird.

- Nachteile und **Einschränkungen der Hostsprache** werden kopiert.
- Es gibt **keine erweiterte, semantische Fehlerüberprüfung**.

5.1 Anforderungen an die Hostsprache

Nicht alle Allzwecksprachen eignen sich als Hostsprache. Die Syntax sollte eine gewisse **Flexibilität** gewährleisten. Weiterhin sollte die Syntax der Hostsprache **klein und orthogonal** sein. Zum Beispiel eignen sich die Sprachen **Lisp**, **Haskell** und **Smalltalk** als Hostsprache. Lisp unterstützt multiple Sprachparadigmen. Dazu gehören die **funktionale**, **prozedurale**, **reflexive** und **Metaprogrammierung**. Oft werden jedoch auch imperative Sprachen, wie z.B. Java als Hostsprache eingesetzt.

5.2 Entwurfsmuster

In den folgenden Abschnitten werden einige **Entwurfsmuster** für eingebettete DSLs vorgestellt. Als Quelle wird dazu vor allem [7] herangezogen.

5.2.1 Methodenverkettung

Das Muster der **Methodenverkettung (method chaining)** gehört zu den **fließenden Schnittstellen** (fluent interfaces). Eine objektorientierte API wird so entworfen oder umgestaltet, dass Methodenaufrufe in der Form `a().b().c().d();` verkettet werden können. Lokale Variablen zur Speicherung von Zwischenobjekten sind nun nach außen hin nicht mehr sichtbar. Das resultierende Programm ist eher **kurz**; die **Lesbarkeit wird verbessert**. Das folgende Beispiel erzeugt ein elektrisches Netz; bestehend aus einer Speicher- und einer Konverterkomponente:

```
1 network()
2   .storage()
3     .capacity(5)
4     .eta(0.8)
5   .converter()
6     .in(POWER)
7     .out(GAS)
8 .end();
```

Die folgenden **Nachteile / Herausforderungen** bestehen:

- **Haltepunkte** können nicht innerhalb der Methodenkette für Debugging-Zwecke gesetzt werden.
- Die **Einrückung** des obigen Beispiels hat keine semantische Bedeutung.
- Hierarchien müssen intern (ggf. aufwändig) verwaltet werden.
- Probleme der Art **Dangling-Else** können auftreten:

```
if(x==1)
  if(y==1)
    z=1;
else
  z=2;
```

Unabhängig von der konkreten Einrückung bezieht sich das Schlüsselwort `else` auf die **zweite** if-Anweisung.

5.2.2 Funktionenfolge

Das Entwurfsmuster **Funktionenfolge (function sequence)** ruft eine Reihe von **Funktionen separat** auf. Dabei ist die **Reihenfolge** der Funktionsaufrufe entscheidend. Intern ist ein **Zustandsautomat** erforderlich, der die Objekte verwaltet. Das folgende Beispiel ist inhaltlich äquivalent zu dem oben gezeigten Code für die Methodenverkettung:

```
network();
  storage();
    capacity(5);
    eta(0.8);
  converter();
    in(POWER);
    out(GAS);
end();
```

Als wesentlicher **Nachteil** zu sehen ist, dass alle **Funktionen global** deklariert werden müssen.

5.2.3 Geschachtelte Funktionen

Das Entwurfsmuster der **geschachtelten Funktionen (nested functions)** stellt eine **hierarchische Struktur** zur Verfügung. Im Gegensatz zur Funktionsfolge können auch **syntaktische Überprüfungen** durchgeführt werden. Es ist nicht erforderlich Kontextvariablen zu speichern. Das Entwurfsmuster ist vor allem für für High-Level Strukturen geeignet. Beispiel:

```
network(
  storage(
    capacity(5),
    eta(0.8);
  ),
  converter();
    in(POWER),
    out(GAS)
  )
);
```

Die erste Zeile des folgenden Codeausschnittes ist im Allgemeinen verständlicher, als die zweite Zeile; benötigt dafür aber mehr Zeichen:

```
storage( capacity(5), eta(0.8) );
storage( 5, 0.8 );
```

Die folgenden **Nachteile / Herausforderungen** bestehen:

- Die Hostsprache benötigt eine Unterstützung für eine dynamische Anzahl an Funktionsparametern (z.B. Lisp).
- Andernfalls ist die erstellte DSL sehr statisch.

5.2.4 Wörterbücher

Das folgende Beispiel für das Entwurfsmuster **Wörterbücher (literal maps)** ist in **Ruby** geschrieben. Weiterhin unterstützt z.B. auch Lisp Wörterbücher.

```
network(
  storage(
    :capacity => 5,
    :eta => 0.8
  ),
  converter(
    :input => POWER,
    :output => GAS
  )
);
```

Als wesentlicher **Nachteil** ist zu sehen, dass gültige Schlüsselnamen nicht erzwungen werden können.

5.2.5 Wahl der Entwurfsmuster

Für die Wahl des Entwurfsmusters können die **Regeln einer Grammatik** herangezogen werden [7]:

- $a_1\ a_2\ a_2$: Verschachtelte Funktionen
- $[a]$ oder $(a_1|a_2|\dots)^*$: Methodenverkettung
- a^*: Funktionenfolge

Anmerkung: Die Aufstellung ist weder vollständig, noch bindend, noch für alle Fälle geeignet.

5.3 Lisp

Lisp ist eine der ältesten Programmiersprachen, die heute noch in Verwendung sind. Sie erschien erstmals 1958. Streng genommen ist Lisp eigentlich eine Familie von Programmiersprachen. Dabei ist Common Lisp ein weit verbreitetes Derivat. Unsere Intention Lisp zu betrachten ist dadurch begründet, dass Lisp Eigenschaften vorweist, die im Rahmen von eingebetteten DSLs sehr nützlich sind. Entgegen vielen anderer Programmiersprachen gibt es in Lisp z.B. die Möglichkeit **Programmcode zu verändern.** In der Tat ist die praktische Handhabung jedoch nicht immer sehr eingängig. Soweit nicht anders referenziert, wurden die Quellen [9, 22] für dieses Kapitel herangezogen.

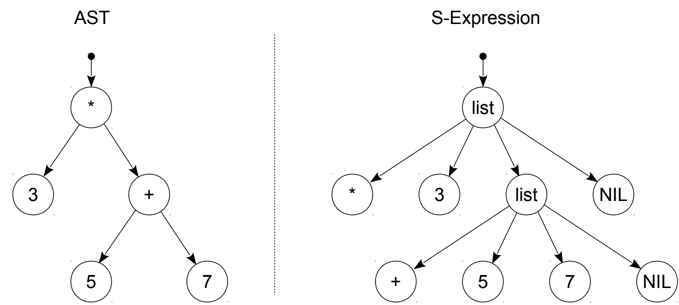


Abb. 5.1: Abstrakter Syntaxbaum mit zugehörigem s-Ausdruck

5.3.1 Grundlagen

Die **fundamentale Datenstruktur** in Lisp sind **Listen**. Diese speichern z.B. Symbole, Zahlen und Strings. Programme werden in Form von s-Ausdrücken (s-Expressions) ausgedrückt:

Definition 47 (s-Ausdruck) *Ein s-Ausdruck (symbolischer Ausdruck) ist eine Notation für geschachtelte Listen, also für baumstrukturierte Daten. Formal wird ein s-Ausdruck wie folgt definiert:*

- Durch ein **Atom**.
- Rekursiv durch einen Ausdruck in der Form $(x\ .\ y)$. Dabei sind x und y selbst wieder s-Ausdrücke.

Wir beschränken uns im Folgenden auf die Kurznotation $(x\ y)$, anstelle der formalen Langform $(x\ .\ (y\ .\ \text{NIL}))$. NIL ist die Abkürzung für Not-in-List. In Abb. 5.1 ist die Korrespondenz von einem **Abstrakten Syntaxbaum** zu einem **s-Ausdruck** dargestellt. Man beachte, dass Lisp Daten in **Präfixnotation** speichert. Zum Beispiel wird $3 \cdot (5 + 7)$ in Lisp in der Form `(* 3 (+ 5 7))` kodiert.

Beispiel: Als erstes erweitertes Beispiel wird unten stehend ein Programm in Lisp zur Berechnung der Fakultät einer Zahl gezeigt. Z.B. $5! = \prod_{i=1}^{5} i = 1 \cdot 2 \cdot \dots \cdot 5 = 120$:

```
1 (defun factorial(n)
2   (if(zerop n)
3     1
4     (* n factorial(- n 1))
5   )
6 )
7 (factorial 5)
```

Zum Vergleich ein äquivalentes Programm in der Programmiersprache C:

```
1 int factorial(int n) {
2   if(n == 0)
3     return 1;
4   else
5     return n * factorial(n - 1);
6 }
```

```
7  int main() {
8    factorial(5);
9    return 0;
10 }
```

5.3.2 Homoikonizität

Homoikonizität bedeutet wörtlich "in der selben Darstellung" (griechisch). Die Eigenschaft der Homoikonizität beschreibt, dass **Programme einer Sprache gleichzeitig auch Datenstrukturen der Sprache sind** ("Code as data"). Homoikonische Sprachen erlauben es, "Programme zu schreiben, die Programme schreiben". Als Beispiele von Sprachen mit Unterstützung für Homoikonizität können Lisp, Prolog und Julia genannt werden. In dem nachfolgenden Beispiel[1] erweitern wir Lisp um das Schlüsselwort **while** durch **Metaprogrammierung**. Man beachte, dass es möglich ist auf Ebene des Parse-Baums zu arbeiten.

```
1 (defmacro while (cond &body body)
2   (let ((name (gensym)))
3     `(labels ((,name ()
4       (if ,cond
5         (progn
6           ,@body
7           (,name)))))
8   (,name))))
```

Beispielnutzung:

```
1 (let ((x 0))
2   (while (<= x 5)
3     (print x)
4     (setq x (+ x 1))
5   )
6 )
```

Ein C-Äquivalent zum zweiten Teil sieht wie folgt aus:

```
1 int x = 0;
2 while(x <= 5) {
3   printf("%s\n", x);
4   x ++;
5 }
```

[1] entnommen aus https://de.wikipedia.org/wiki/Metaprogrammierung

Kapitel 6

Externe Domänenspezifische Sprachen

"The most dangerous phrase in the language is:
We've always done it this way."

— Grace Hopper

In diesem Kapitel diskutieren wir externe domänenspezifische Sprachen im Allgemeinen. Zunächst sollen die Vor- und Nachteile beleuchtet werden. Einige Aspekte zu externen DSLs werden in diesem Skript praxisbezogen für *Eclipse Xtext* beschreiben. Hier sollte man also parallel zu diesem Kapitel auch den Anhang A.3 lesen.

Vorteile:

- Großer **syntaktischer Freiraum.**
- Sehr hohe **Abstraktionsebene** der Zieldomäne.
- Fast **keine Einschränkungen** bei der Spracherstellung.
- Einfach erlernbar für **NichtinformatikerInnen**.
- **Kommunikationsschnittstelle**: DomänenexpertInnen werden interdisziplinär befähigt Programme zu verstehen, zu modifizieren und zu schreiben.
- **Ausdrucksstarke** Programme und damit verbunden eine **hohe Produktivität**.
- **Portabilität** und **Plattformunabhängigkeit**.
- **Selbstdokumentation**.
- Programme bleiben **wartbar** und **einfach testbar**.
- **Optimierung** auf der Domänenebene.
- **Langlebigkeit** von Daten.

Weiterhin können viele Techniken aus dem Parsing von Allzwecksprachen wiederverwendet werden.

Herausforderungen

- Hohe Implementierungs- und Wartungs**kosten**.
- **Ausbildung** von AnwenderInnen.
- Ggf. eingeschränkte **Verfügbarkeit**.
- Schwierigkeit den richtigen **Umfang** zu finden.
- Eingeschränkte **Berechenbarkeit**; neue Anforderungen können ohne Modifikation ggf. nicht umgesetzt werden.
- "Sprachzoo", "DSL-Hell".

Es gibt hauptsächlich zwei verschiedene Konzepte, um externe DSLs umzusetzen:

- **Parserbasierter Ansatz:** Die Struktur wird durch eine **Grammatik** spezifiziert. *Beispiel: Eclipse Xtext mit integriertem ANTLR Parser.*
- **Projektionseditoren:** Weder Grammatiken noch Parser werden explizit definiert. Diese Vorgehensweise ist vor allem von grafischen Editoren her bekannt. Als bekanntes Beispiel ist *JetBrains MPS* zu nennen. Man definiert hier zunächst den abstrakten Syntaxbaums (AST). Danach werden Definition von Projektionsregeln definiert, welche eine konkrete Syntax erzeugen.

In diesem Kapitel werden wir uns auf den parserbasierten Ansatz beschränken.

Historische Entwicklung

Aus historischer Sicht hat gem. [20] die folgende Evolution stattgefunden:

1. **Unterprogrammbibliotheken** kapseln wiederverwendbaren Code in einer **flachen Struktur**. Beispiele dafür sind Gleichungen, Graphen oder Geschäftsvorgänge.
2. **Objektorientierte Frameworks** bieten eine **hierarchische Struktur**. Komplexere Komponenten steuern einfachere Komponenten.
3. **Domänenspezifische Sprachen** sind klein und deklarativ. Sie rufen Unterprogrammbibliotheken auf und fokussieren sich ausdrucksstark auf eine bestimmte Domäne.

6.1 Modellierung

Definition 48 (Semantisches Modell) *Ein **semantisches Modell** ist ein **Speicherobjekt,** welches eine konkrete DSL beschreibt. Jedes **Konzept** der DSL wird auf eine **Klasse** abgebildet. Jeder konkrete Codeausschnitt der DSL wird auf eine **Objektinstanz** abgebildet.*

Es ist sinnvoll, das semantische Modell zu erstellen, **bevor** die DSL selbst entworfen wird. Man beachte, dass ein semantisches Modell nur eine **Teilmenge des Domänenmodells** abbildet. Die Quelle [7] definiert ein Domänenmodell als ein "verhaltensstarkes Objektmodell", während das semantische Modell hingegen auf Daten beschränkt ist, und nur eine **unterstützende Rolle** spielt.

Ein semantisches Modell hat für gewöhnlich zwei **Schnittstellen**:

- Die **Populationsschnittstelle** wird während des Parsens dazu genutzt, die **Objektinstanzen zu erzeugen**.
- Die **betriebliche Schnittstelle** erlaubt es den Clients, das Modell zu **nutzen**. Dies kann entweder eine weiterverarbeitende Software sein, oder aber auch der Codegenerator.

Neben den **Modellierungsaspekten** selbst hat das semantische Modell den Vorteil, dass man die abgebildete Domäne unabhängig von der DSL **testen** kann.

Beispiel: In Abb. 6.1 wird ein Ausschnitt aus einem Zustandsautomaten für eine Aufzugsteuerung gezeigt.

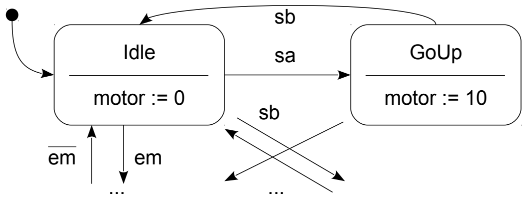


Abb. 6.1: Zustandsautomat für eine Aufzugsteuerung als Beispieldomäne

Ein zugehöriges **semantisches Modell** für die Transitionen (Übergänge) kann zum Beispiel wie in Abb. 6.2 dargestellt aussehen.

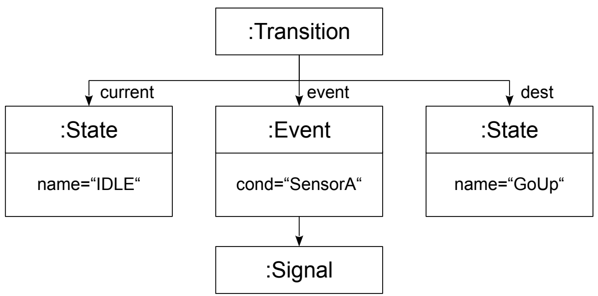


Abb. 6.2: Semantisches Modell für eine Transition eines Zustandsautomaten

Das Modell ist **zunächst unabhängig von einer konkreten Syntax.** Letztere könnte z.B. wie folgt – oder auch ganz anders – aussehen:

```
state Idle {
  GoUp <- sensorA
  ...
}
```

6.2 Codegenerierung

Für einige DSLs ist es nicht möglich oder nicht zielführend, Code zu generieren. Dann wird zunächst das semantische Modell aufgebaut und dann ausgeführt. Im Falle einer **Codegenerierung** unterscheidet [7] die folgenden beiden Formen der Ausgabe:

- **Modellbewusste Generierung:** Hier bleibt das semantische Modell im generierten Code weiterhin **rekonstruierbar**. Für das Beispiel des Zustandsautomaten von oben legt man auch im generierten Code Instanzen von Zuständen, Transitionen usw. an.
- **Modellignoranten Generierung:** Hier wird das semantische Modell selbst nicht weiter in den Ausgabecode geschrieben. Man beschränkt sich auf den Code, der für die Ausführung unbedingt notwendig ist.

Der **Prozess der Codegenerierung selbst** kann wie folgt gegliedert werden. Beide Verfahren können auch kombiniert eingesetzt werden:

- Bei der **Transformer Generation** navigiert man durch die Klassenobjekte des semantischen Modells und produziert den Ausgabecode. Ggf. sind mehrere Phasen nötig, um alle Querverweise aufzulösen.
- Bei der **Templated Generation** wird zunächst der statische Teil des Ausgabeprogramms in einen großen String geschrieben. Danach fügt man in genau diesen String die dynamischen Anteile durch einen eingebetteten **Templatecode** hinzu. Das nachfolgende Beispiel nutzt Templates in *Xtext/Xtend*:

```
1  '''
2    #include <stdio.h>
3    «FOR s : p.signals»
4        «s.datatype» «s.name»
5    «ENDFOR»
6    enum State {
7      «FOR s : p.states SEPARATOR ','»
8        «s.name»
9      «ENDFOR»
10   }
11 '''
```

Achtung: Wenn das semantische Modell insgesamt sehr dynamisch ist, sollte man die Transformer Generation bevorzugen.

Automatisch generierter Code sollte **nie** modifiziert werden. Es sollte eine klare **Trennung** von handgeschriebenem und generiertem Code vorgenommen werden. Weiterhin sollte generierter Code **gut lesbar**, **strukturiert**, **selbsterklärend** und **verständlich** sein. So ist auch später das Debuggen einfacher.

Es gibt verschiedene Strategien zur **Kompilierung** von externen DSLs. Die Wahl entscheidet sich zum einen durch die **Sprachkomplexität**. Zum anderen ist es oft sinnvoll, die Verfahren kombiniert einzusetzen.

6.2.1 Trennzeichengesteuerte Übersetzung

Die **Trennzeichengesteuerte Übersetzung (Delimiter-Directed Translation)** ist insbesondere für DSLs geeignet, die ähnlich wie Comma Separated Value (CSV) Datensätze strukturiert sind.

1. Zunächst wird der DSL Code in einzelne Teile **zerlegt**. Für gewöhnlich werden **Zeilenumbrüche** als Trennzeichen gewählt.

2. Nun kann jede Zeile einzeln **geparsed** werden.

Das Verfahren ist sehr **einfach** zu implementieren. Allerdings stößt man bei komplexeren Sprachen schnell an **Grenzen.**

6.2.2 Syntaxgesteuerte Übersetzung

Syntaxgesteuerte Übersetzung (Syntax-Directed Translation) parsed Code mit Hilfe einer Grammatik und schreibt in nachgeschalteten Phasen den Ausgabecode.

- Zunächst wird ein **abstrakter Syntaxbaum** konstruiert.

- Viele der technischen Aspekte können direkt aus dem konventionellen Compilerbau für Allzwecksprachen übernommen werden.

Detaillierte Informationen finden sich flächendeckend in diesem Skript.

6.2.3 Eingebettete Übersetzung

Bei der **Eingebetteten Übersetzung (Embedded Translation)** wird ein Aktionscode zum Befüllen des semantischen Modells **direkt in die Grammatik** eingebettet. Eine **Symboltabelle** unterstützt dabei einen "verzögerten" Aufbau von Teilen des semantischen Modells. Dies ist insbesondere dann nötig, wenn komplexe Strukturen verarbeitet werden. Dann sind Relationen nicht mehr unbedingt direkt auflösbar; zum Beispiel tritt dieser Fall ein, wenn **Vorwärtsdeklarationen** vorliegen.

6.2.4 Baumkonstruktion

Bei der **Baumkonstruktion (Tree Construction)** wird das semantische Modell nicht direkt aus dem Parser heraus befüllt.

1. Zunächst wird der vollständige **abstrakte Syntaxbaum** aufgestellt.

2. Daraufhin wird der abstrakte Syntaxbaum **durchlaufen** und das semantische Modell befüllt.

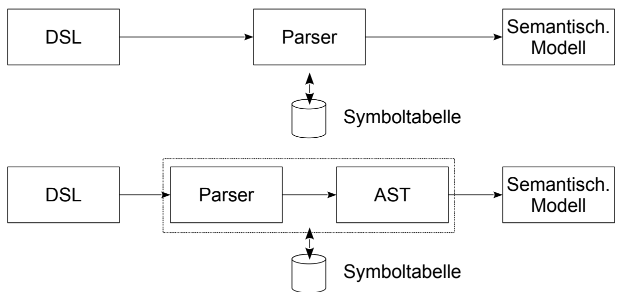


Abb. 6.3: Oben: Eingebettete Übersetzung. Unten: Baumkonstruktion.

6.2.5 Eingebettete Interpretation

Eingebettete Interpretation bedeutet, dass Ausdrücke der DSL **frühestmöglich**, während des Parsens, ausgewertet werden. Vor allem konstante mathematische Terme, wie z.B. $(2+3)*4$ können sofort ausgewertet und durch das Ergebnis ersetzt werden, ohne dass Informationen verloren gehen. Die eingebettete Interpretation kann für DSLs leider nur selten eingesetzt werden, da diese häufig deklarativ sind und Vorwärtsverweise erst später ausgewertet werden können.

6.2.6 Anreicherung um Allzweckcode

Aus ökonomischen Gründen sollten DSLs **nicht** so entworfen werden, dass alle möglichen, aber vielleicht nur von sehr wenigen BenutzernInnen benötigte, Sprachaspekte enthalten sind. In diesem Fall würde die DSL sehr komplex werden: die **Kosten** steigen und die **Wartbarkeit leidet**. Stattdessen kann manchmal auch **Code einer Allzwecksprache** als "Fremdcode" [7] in die DSL eingebettet werden. Ein bekanntes Beispiel ist die Einbettung von *JavaScript* in *HTML*. Man bedient sich meist ein- und ausleitenden **Tags**, wie z.B. `<script>` ... `</script>` oder `<?php` ... `?>`. Für gewöhnlich sollten nur kleine Teile von GPL Code in eine DSL eingebunden werden. Andernfalls könnte dies ein Zeichen dafür sein, dass die DSL nicht ordentlich entworfen wurde.

Anmerkung 1: Der Code der Allzwecksprache wird meist nicht durch den DSL Parser selbst verarbeitet. Entsprechend ist die Integration in die IDE oft nicht nahtlos. Die Kommunikationsschnittstellen zwischen DSL und GPL müssen weiterhin wohldefiniert sein. Syntaxfehler werden erst verzögert erkannt.

Anmerkung 2: Wie kann gezeigt werden, dass eine DSL selbst Turing-vollständig ist? Man versucht z.B. die Ackermann Funktion, oder jede andere total berechenbare Funktion, zu implementieren. Wenn dies möglich ist, kann davon ausgegangen werden, dass die DSL Turing-vollständig ist.

6.3 Softwaretests

DSLs sollten in zwei Teilen separat getestet werden:

- Testen des **semantischen Modells:** Unit-Tests erzeugen Instanzen und Relationen des semantischen Modells.

  Wichtig: Der Parsing-Prozess wird hier noch außen vor gelassen.

- Testen des **Parsers:** Man definiert kleine DSL-Codeausschnitte und testet, ob die korrekten Strukturen im semantischen Modell erzeugt werden. Dazu wird das semantische Modell sowohl manuell, als auch durch den Parser erzeugt. Dann ist ein Abgleich zwischen den beiden Erzeugnissen möglich.

Vor allem sollten auch Negativtests angewandt werden, also DSL Code erzeugt werden der **Syntaxfehler enthält**. Nur so kann geprüft werden, ob der Parser in einem stabilen Zustand verbleibt. Weiterhin können so die **Fehlerausgaben** hinsichtlich Korrektheit und Verständlichkeit **geprüft** werden.

Anmerkung: Man beachte, dass Unit-Tests alle Teilmengen und Relationen prüfen müssen, da ein semantisches Modell meist eine sehr komplexe Datenstruktur darstellt.

6.4 Dokumentation

Es gibt verschiedene Möglichkeiten DSLs zu dokumentieren:

- Domänenspezifische Sprachen können sehr agil durch exemplarische **Codeausschnitte** dokumentiert werden. DSLs sollten dazu auch immer eine Kommentarfunktion anbieten. Diese Dokumentationsart hat für den NutzerInnen gleichzeitig den **Charakter eines Tutorials.**

- Die **Grammatik** sollte im Detail dokumentiert werden. Jede Regel sollte separat beschrieben werden. Beispiele helfen dem Verständnis.

- Auch **generierter Code** sollte exemplarisch und in lesbarer Form mitgeliefert werden. So ist für den/die NutzerIn ersichtlich, wie konkrete Eingabeprogramme verarbeitet werden.

- Das **semantische Modell** kann am einfachsten graphisch dokumentiert werden. Details sollten für eine bessere Übersichtlichkeit weggelassen werden.

6.5 Sprachwerkbänke

Definition 49 (Sprachwerkbank) *Eine **Sprachwerkbank** ist eine Umgebung, die bei der Erstellung einer domänenspezifischen Sprache hilft.*

Folgende **Merkmale** sind für Sprachwerkbänke üblich und gehen über den Umfang schlichter **Parsergeneratoren** hinaus. Manchmal werden auch graphische DSLs unterstützt. Ein mächtiges Werkzeug ist die Möglichkeit zugeschnittene **integrierte Entwicklungsumgebungen (IDEs)** definieren zu können. Diese bieten meist die folgenden Möglichkeiten:

- **Syntaktische Merkmale:** u.a. Syntaxcheck, Syntaxhighlighting, Codefaltung, Abgleich von Klammerungen usw.
- **Semantische Merkmale:** u.a. Fehlerüberprüfungen jenseits der Syntax, Inhaltsvervollständigungen, Hilfeinformationen bei Überfahren von Text mit der Maus, Referenzverfolgung durch Klick usw.

Eine Hauptmotivation zur Nutzung von Sprachwerkbänken ist, **für möglichst viele Bereiche** DSLs eingängig definieren zu können, sodass NichtprogrammiererInnen dazu befähigt werden die Problemstellungen aus ihrer Domäne in formalen Text fassen zu können.

Beispiele für Sprachwerkbänke sind: Eclipse Xtext, MontiCore (RWTH Aachen), JetBrains MPS, Kermeta, ...

Kapitel 7

Forschungsgebiete

Dieses Kapitel soll einen kurzen Überblick über die jüngeren Forschungsgebiete rund um den Compilerbau geben.

Anmerkung: Das Kapitel befindet sich z.Zt. noch im Aufbau.

7.1 Forschungslandschaft

Der Compilerbau gehört zu den ältesten Gebieten der Informatik. Im Bereich des **Frontends** gibt es im herkömmlichen Compilerbau jenseits der syntaktischen Entwurfsaspekte neuartiger Sprachparadigmen für domänenspezifische Sprachen nur noch sehr wenige neue Publikationen, die sich mit den unterliegenden Techniken, wie z.B. dem Parsing befassen.

Im Bereich des **Backends** ist stets der Wandel der gebräuchlichen Computerarchitekturen zu verfolgen. Beispielsweise hat die Open-Source Instruction Set Architektur (ISA) *RISC-V* in der jüngsten Vergangenheit einige Versionssprünge zu vermelden. Compiler müssen stets neue Instruktionen in ihre Codegenerierung einbeziehen, um die Programmlaufzeit zu optimieren.

Auch der Bereich der IT-Sicherheit ist eng mit dem Übersetzerbau verzahnt, liefern doch die große Zahl existierender Programmen der auf Laufzeiteffizienz ausgerichteten Sprachen, wie z.B. C, sehr großes Potenzial unbedacht Pufferüberläufe usw. zu verursachen. Die im Jahre 2018 entdeckten hardwareseitigen Sicherheitslücken *Spectre* und *Meltdown* können z.B. auch über den Compilerbau abgefangen werden; auch wenn mit drastischen Performanzeinbußen einhergeht.

7.2 Codeoptimierung

Der Bereich des Compiler Backends ist eng verknüpft mit der unterliegenden Rechnerarchitektur. Da einer weitere Erhöhung der CPU-Taktrate gemäß den physikalischen Gesetzen nicht ohne weiteres möglich ist, wird man zukünftig immer mehr auf **Many-Core Architekturen** setzen müssen; auch um der Energieeffizienz mehr Aufmerksamkeit schenken zu können. Dazu vergleiche man die Effizienz des menschlichen Gehirns, welches trotz massiver Parallelverarbeitung nur vergleichsweise sehr

wenig Energie verbraucht. Auch wenn Grafikkarten mit x-tausend Kernen zwar bereits seit längerem verfügbar sind, gibt es nur wenige Klassen von Eingabeproblemen, die sehr eingängig durch den Programmierbar parallelisierbar sind. Dazu gehören vor allem die Bereiche des *Stream Processing*. Zukünftige Compiler sollten vor allem einen Schwerpunkt auf eine **automatisierte Parallelisierung** legen. Die Codeoptimierung muss vor allem **interprozedural** erfolgen, sodass sich ein global bemerkbarer Laufzeitgewinn einstellen kann.

7.3 Quantum Computing

(folgt später)

7.3.1 Konferenzen

(folgt später)

7.4 Natürliche Sprachverarbeitung

(folgt später)

Anhang A

Werkzeuge

"Beware of bugs in the above code; I have only proved it correct, not tried it."

— Donald Knuth

A.1 Flex und GNU Bison

Der **Scannergenerator Flex** wird häufig in Kombination mit dem **Parsergenerator GNU Bison** eingesetzt. Üblicherweise nutzt man C oder C++ als Implementierungssprache. Detailliertere Informationen über GNU Bison findet man in [17]. Wir verwenden hier GNU Bison in der Version 3.

Unter https://gitlab.com/andreas.schwenk/cbdsl-bison findet man ein **erweitertes Beispiel** mit Erläuterungen. Der Code eignet sich auch als Grundlage für eigene Projekte. Weiterhin können die Quellen [1, 10, 11, 14] in Erwägung gezogen werden.

A.1.1 Fast Lexical Analyzer Generator (FLEX)

Der Scannergenerator **Flex** wurde 1987 in der Sprache C von Vern Paxon als Open-Source Alternative zu dem älteren Tool **Lex** entwickelt. Eine Flex-Datei hat die Dateierweiterung `*.l` und wird von Flex in eine C-Datei mit der Endung `*.yy.c` übersetzt. Sie besteht aus drei Teilen:

1. **Benutzerdefinitionen**
2. **Regeln** in Form von regulären Ausdrücken.
3. **Benutzercode**

Die einzelnen Teile werden per Doppel-Prozent Zeichen getrennt (%%). **Aktionscode** wird in geschweiften Klammern `{}` implementiert. Der folgende Quellcode zeigt die Umsetzung des Transduktors aus Abb. 3.6. Neben den Schlüsselwörtern `for`, `int` und `if` sind auch das Scannen von einzeiligen Kommentaren via `//`, sowie auch das Scannen von Bezeichnern (Identifiern) dargestellt.

```
/* ----- 1. Definitionen ----- */
%{
  #include <stdio.h>
  #include <string.h>
  #include "iom.tab.h"  /* Ausgabedatei von GNU Bison */
%}

/* ----- 2. Regeln ----- */

"//".*   { }

"if"    { return IF; }
"int"   { return INT; }
"for"   { return FOR; }

[a-zA-Z_][a-zA-Z0-9_]* {
          yylval.string_val = strdup(yytext);
          return IDENTIFIER;
        }

%%
/* ----- 3. Optionaler Nutzercode ----- */

/* hier weggelassen */
```

Flex wird meist zusammen mit dem Parsergenerator `GNU Bison` eingesetzt. In der Grammatikdatei `*.y` werden die Token deklariert:

```
...
%token TK_IF TK_INT TK_LESS TK_IDENTIFIER
...
```

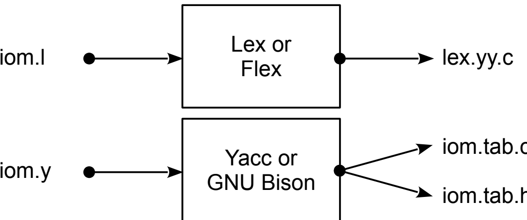


Abb. A.1: Flex und GNU Bison

A.1.2 GNU Bison

Ein ausführliches **Grammatikbeispiel** findet man im Anhang B.

Einige Details in loser Reihenfolge:

- GNU Bison reduziert nicht automatisch, sobald eine Reduktionsregel gefunden werden kann. Diese Strategie wäre für die meisten Grammatiken zu einfach.
- Ein gelesenes Token wird nicht sofort geshiftet, sondern als **Look-Ahead** aufgefasst.

- GNU Bison ist per Default ein LALR(1) Parser, also eine vereinfachte Version eines LR(1) Parsers.

Das folgende Beispiel in GNU Bison verdeutlicht den Einsatz des Look-Ahead:

```
expr: term, '+', expr | term;
term: '(', expr, ')' | term, '!' | NUMBER;
```

Betrachten wir die Eingabeworte `"... 2+3)"` und `"... 2+3!"`:

- Angenommen, `2 + 3` wird in den Stack geshiftet.
- Wenn das nächste Symbol `)` ist, dann *muss* GNU Bison zunächst reduzieren, da keine Regel für `2+3)` anwendbar ist.
- Wenn das nächste Symbol nach `2 + 3` das Symbol `!` (Fakultätsoperator) ist, so muss `!` zunächst geshiftet werden. Reduktionen können erst danach durchgeführt werden.

A.2 ANTLR

ANTLR (ANother Tool for Language Recognition) wird aktiv seit 1992 entwickelt. Die aktuelle Version ist 4.x. ANTLR generiert **Parser** in den Sprachen Java, C++, JavaScript, Python, Swift und Go. Die **Grammatikspezifikation** erfolgt ähnlich zu EBNF. Unter dem folgenden Link kann ein in Python geschriebenes **Beispielprojekt** gefunden werden:

https://gitlab.com/andreas.schwenk/cbdsl-antlr

ANTLR kann auch integrativ in *Eclipse Xtext* eingesetzt werden. Hierauf wird im nächsten Abschnitt eingegangen. Die **Vorteile** beim Einsatz von ANTLR außerhalb von *Eclipse Xtext* sind die **Leichtgewichtigkeit**, sowie die **freie Wahl der Implementierungssprache.** Allerdings werden nicht, wie bei der *Eclipse Xtext*-Integration, automatisch Datenstrukturen für den AST erzeugt. Stattdessen muss ein **Listener** umgesetzt werden, welcher nach Eintritt und/oder Austritt einer Nichtterminalregel vom Parser aufgerufen wird.

A.3 Eclipse Xtext

Eclipse Xtext ist eine **Sprachwerkbank** für die Erstellung von **externen DSLs.** Wir werden in diesem Kapitel sowohl auf Xtext selbst, als auch auf die Sprache **Xtend** eingehen. Letztere eignet sich insbesondere für die **Codegenerierung.**

A.3.1 Einführung

Xtext ist ein Open Source Framework für die Entwicklung von domänenspezifischen Sprachen. Viele der im Rahmen der Sprachentwicklung anfallenden Prozesse

werden **automatisiert.** Zum Beispiel wird ein **Klassenmodell** für den **abstrakten Syntaxbaum** automatisch erzeugt. Xtext integriert sich weiterhin nahtlos in die **Entwicklungsumgebung Eclipse** und wird ebenfalls von dem *Eclipse Project* entwickelt. Insgesamt stellt Xtext alle Bereiche einer **Sprachwerkbank** bereit [7]: **Metamodelle**, **Editierungsumgebungen** und **Codegenerierung**.

Wir beschränken uns hier nur auf einige wesentliche Aspekte. Mit Hilfe der Dokumentation sollte man sich in den Workflow selbstständig einarbeiten.

A.3.2 Xtend-Tutorial

Xtend ist eine Allzwecksprache für die Java Virtual Machine (JVM). Die Syntax von *Xtend* ist jedoch **kompakter**, als die Syntax von Java. Auch die **Erweiterbarkeit** eine wichtige Spracheigenschaft von *Xtend*. Zum Beispiel sind **Typinferenz**, **Extension Methods** und **Operatorüberladung** in *Xtend* vorhanden; aber *nicht* in Java. In Xtend geschriebener Code wird in Java Code übersetzt. Entsprechend können alle bestehenden Java Bibliotheken, sowie das gesamte **Java-Ökosystem** wiederverwendet werden. Eine vollständige Bedienungsanleitung für *Xtend* kann unter [18] gefunden werden.

Hello World

Ein Hello World Programm sieht in Xtend wie folgt aus:

```
1 package xtendtutorial
2 class Helloworld {
3   def static void main(String[] args) {
4     println("Hello, World!")
5   }
6 }
```

Im **Package Explorer** findet man einen Ordner “xtend-gen”. Hierin befindet sich die Java-Datei “*.java”, welche eine kompilierte Version der Eingabeprogramms in der Zielsprache Java darstellt.

Typen

Variablen haben keine Typdefinition auf der linken Seite. Der Typ wird stattdessen vom Part rechts des Zuweisungsoperators **abgeleitet**.

```
1 // Konstante:
2 val test = "Hello"
3
4 // Veraenderliche Variable:
5 var test2 = 5
6 test2 += 2
```

Listen

Das folgende Beispiel zur **Arrayliste** demonstriert die Erstellung einer Liste, sowie dessen Iteration. Achten Sie auf die im Vergleich zu Java kompaktere Notation:

```
val list = newArrayList("a", "b", "c")
for(item : list) {
  println(item)
}
```

Klassen

Im Gegensatz zu Java ist es erlaubt, verschiedene Klassen in eine einzige Quellcodedatei zu schreiben.

```
class Arithmetics {
  def add(int a, int b)
    return a + b
  }
}
```

Ähnlich wie in Python müssen Methoden den Rückgabetyp nicht angeben. Die Typen der Parameter müssen hingegen angegeben werden. Beispiel für das Erstellen von **Objekten**:

```
val arith = new Arithmetics()
var sum = arith.add(3, 4)
println(sum)
```

Datenklassen

Eine Klasse mit `@Data`-**Annotation** deklariert ein **unveränderliches Objekt (immutable object)**. Die folgenden Punkte werden dann automatisch erzeugt:

- Eine **Getter**-Methode für jedes Attribut.
- Eine `hashCode()` Implementierung.
- Eine `toString()` Implementierung.
- Ein **Konstruktor** für alle Attribute.

Beispiel:

```
@Data class Publication {
  String author
  String title
  int numberOfCitations
}
```

Instanziierung:

```
val publication0 = new Publication("Wolfram", "The Alpha Theory", 1337)
```

Lambda Ausdrücke

Achtung: Der Begriff ***Lambda Ausdruck*** *ist ein Polysem und wird auch im Rahmen des* ***Lambda Kalküls***[1] *verwendet. Hier ist eine* ***anonyme Funktion*** *gemeint.*

Eine **anonyme Funktion** ist eine Funktionsdefiniton, die nicht an einen Bezeichner gebunden ist. Das folgende Beispiel ist aus `https://www.eclipse.org/xtend/documentation/203_xtend_expressions.html#lambdas` entnommen:

```
val textField = new JTextField
textField.addActionListener([ ActionEvent e |
  textField.text = "my text"
])
```

Dazu äquivalenter Java-Code sieht wie folgt aus:

```
final JTextField textField = new JTextField();
textField.addActionListener(new ActionListener() {
  @Override
  public void actionPerformed(ActionEvent e) {
    textField.setText("my text");
  }
});
```

Das folgende Beispiel erzeugt eine Arrayliste vom Typ `Publication` (siehe oben).

```
val publications = newArrayList("AA B 11", "AB E 123", "G H 1234")
.map[ line |
  val iter = line.split(' ').iterator
  return new Publication(
    iter.next,
    iter.next,
    Integer.parseInt(iter.next)
  )
]
println(publications)
```

Nun sollen die Publikationen aus der Liste gefiltert werden, deren Autoren mit `A` beginnen:

```
val p0 = publications.filter[ publication
                              | publication.author.startsWith('A') ]
```

Kurzform:

```
val p0 = publications.filter[ author.startsWith('A') ]
```

Die nächste Abfrage filtert alle Publikationen mit minimal 10 und maximal 100 Zitierungen:

```
val p1 = publications.filter[ (10..100).contains(numberOfCitations) ]
```

Beispiel für das Sortieren von Listen:

[1] ein formales System in der mathematischen Logik

```
1 val p2 = publications.sortBy[ numberOfCitations ]
2 val p3 = publications.sortBy[ -numberOfCitations ] // reverse order
```

Sie können den Beispielcode von der folgenden URL beziehen:
`www.compiler-construction.com/lab/xtend0.zip`.

A.3.3 Xtext-Tutorial

Ein einfaches Xtext-Programm kann wie folgt erzeugt werden:

- Öffnen Sie Eclipse[2].
- Geben Sie einen **Workspace** an, oder nutzen Sie den standardmäßigen Workflow.
- Erzeugen Sie ein **neues Projekt** via [File] → [New] → [Project] → [Xtext] → [Xtext Project]
- Wählen Sie "helloworld" als **Projektnamen** und "helloworld.Dsl" als Sprachnamen.
- Wählen Sie "mfd' ("my first DSL") als **Spracherweiterung (language extension)**.
- Aktivieren Sie die Checkboxen "Testing Support" und "JUnit Version 5". Klicken Sie dann auf den Button [finish].

Der **Paketeditor** sollte nun zumindest die folgenden Punkte auflisten:

- `helloworld`, `helloworld.ide`
- `helloworld.tests`
- `helloworld.ui`, `helloworld.ui.tests`

Wir konzentrieren uns zunächst auf die **Grammatik**.

Öffnen Sie [helloworld] → [src] → [helloworld] → [Dsl.xtext].
Man findet die folgende Beispielgrammatik vor:

```
1 grammar helloworld.Dsl with org.eclipse.xtext.common.Terminals
2 generate dsl "http://www.Dsl.helloworld"
3 Model:
4   greetings += Greeting*;
5 Greeting:
6   'Hello' name=ID '!';
```

`Model` und `Greeting` sind **Nichtterminale**. Durch Selektion und Drücken auf [F3] kann man zur Definition eines Nichtterminals navigieren. Eine zugehörige EBNF-Grammatik sieht wie folgt aus:

[2] Sollten Sie Eclipse selbst installieren, wählen Sie "Eclipse IDE for Java and DSL Developers".

```
model = { greeting };
greeting = "Hello" ID "!";
ID = { "a" | "b" | ... | "Z" };
```

Die **Sprachartefakte** können wie folgt erzeugt werden:

- Rechtsklicken Sie in der Grammatikdatei `Dsl.xtext` und wählen Sie [Run as] → [Generate Xtext Artifacts] aus.

  Achtung: Überprüfen Sie, dass keine Fehler im Tab "Problems" im unteren Bereich auftreten.

- Klicken Sie auf "helloworld" im Paketexplorer. Dann wählen Sie [Run] → [Run As] → [Eclipse Application] im Hauptmenü.

Nun soll eine **eigene IDE** zu der konfigurierten Sprache erzeugt werden:

- Erzeugen Sie ein neues Projekt per [File] → [New] → [Project] → [Java Project] und wählen Sie den Projektnamen (z.B. "myproject"). Klicken Sie auf [finish].
- Fügen Sie eine neue Quellcodedatei "test.mfd" hinzu und klicken Sie auf [Finish].
- Das sich öffnende Dialogfenster mit der Nachricht "Do you want to convert 'myproject' to an Xtext project?" bestätigen Sie mit "yes".
- Nutzen Sie die in der letzten Aufgabe erstellen Worte der Sprache und schreiben Sie diese in die neue Datei.
- Drücken Sie [Steuerung] + [Leertaste], um die Autovervollständigung zu nutzen.

Sie können optional unter [19] erfahren, wie **IntelliJ IDEA-** und **Webeditoren** erstellt werden können.

*Hinweis: Einige Optionen erfordern bei Projekterzeugung **Gradle** zu aktivieren.*

EMF und Xcore

Das **EMF := Eclipse Modeling Framework** repräsentiert den abstrakten Syntaxbaum (AST) in Xtext. EMF stellt Werkzeuge und Laufzeitsupport bereit, um automatisch eine Menge von Javaklassen für das Modell (:= eine Menge von Adapterklassen) zu erzeugen. Dies nennt man dann **Model Driven Architecture (MDA)**. Man beachte, dass der abstrakte Syntaxbaum in Xtext noch kein vollständiges semantisches Modell darstellt! Entsprechend muss zusätzlicher Code hinzugefügt werden. **Xcore** ist eine Möglichkeit, um ein SM innerhalb der Xtext-Umgebung zu erstellen. *Achtung: Manchmal werden die Begriffe "AST" und "Modell" in der Xtext-Dokumentationen synonym verwendet.*

Terminal	*Erläuterung*
ID	Bezeichner. Z.B. `leetspeek`.
INT	Positive Ganzzahl. Z.B. `1337`.
STRING	Ein in Anführungszeichen eingebetteter String.
SL	Single-Line Kommentar der Form `// Kommentar`
ML	Multi-Line Kommentar der Form `/* Kommentar */`
WS	Leerzeichenfolge (Spaces, Tabs, Linefeeds)

Tab. A.1: Vordefinierte Terminalsymbole in Xtext

A.3.4 Parser

Xtext verwendet unterliegend den Top-Down Parser Generator **ANTLR** (vgl. dazu Anhang A.2). Ein **abstrakter Syntaxbaum** wird in *Xtext* automatisch erzeugt.

A.3.4.1 Grammatik

Einige Hinweise bei der Erstellung von Xtext-Grammatiken:

- Grammatikregeln werden von **oben nach unten** abgearbeitet. Dies ist insbesondere für Terminale wichtig. Denn Schlüsselwörter, wie z.B. `if` müssen von der Reihenfolge her *vor* allgemeinen Identifiern (Bezeichnern) definiert werden. Der Grund ist, dass Schlüsselwörter selbst eine Teilmenge der allgemeinen Bezeichner sind.

- Es ist besser eher eine größere Menge an **kurzen Regeln** zu schreiben, als wenige und lange. Dies verbessert die Wiederverwendbarkeit und Wartbarkeit.

- ANTLR ist ein **Top-Down Parser.** Entsprechend kann Linksrekursion nicht verarbeitet werden.

- Xtext enthält eine Menge von **vordefinierte Terminalsymbolen**. Diese sind in Tabelle A.1 dargestellt.

EMF Ecore Xtext erstellt während des Parsens eine graphische Struktur, die ähnlich zu einem **abstrakten Syntaxbaum ist.** Diese Datenstrukturen werden **EMF Ecore Modell** genannt. Ein **Ecore Modell** besteht aus einem **Epackage**, welches wiederum aus **Eclasses**, **EDataTypes** und **EEnums** besteht.

Der folgende Code **generiert** ein **Ecore Modell** `domainmodel` aus der Grammatik der gleichen Datei.

```
1 generate domainmodel "http://www.example.org/domainmodel/Domainmodel"
```

Form	*Erläuterung*
`( x )?`	Option
`x | y`	Alternative
`( x )`	Gruppierung
`( x )*`	Beliebige Wiederholung
`( x )+`	Mindestens eine Wiederholung

Tab. A.2: Xtext Grammatikregeln

Terminalregeln werden in der folgenden Form definiert:

```
1 import "http://www.eclipse.org/emf/2002/Ecore" as ecore
2
3 terminal INTEGER returns ecore::EInt:
4   ('-')? INT;
5
6 terminal BOOL:
7   'true' | 'false';
```

Wird kein Rückgabetyp deklariert, dann wird `ecore::EString` angenommen. Alle anderen Regeln werden ähnlich wie EBNF ausgedrückt. Vgl. mit A.2. **Beispiel:**

```
1 IO_Statement:
2   name=ID (':' datatype=Datatype)?;
```

Zur **Benennung der graphischen Datenstrukturen** (s.o.) **muss** eine Grammatik **Features** enthalten. Ein Feature wird für gewöhnlich per **Zuweisungsoperator** gesetzt:

- `x = X` definiert ein **skalares Feature** für Terminal- und Nichter.symbole.
- `x += X` fügt das Element x zu einer **Featureliste** X hinzu.
- `x ?= X` fügt ein **optionales Feature** zu einem optionalen Symbol hinzu. Dann ist x bool'sch.

Weiterhin kann die Grammatik um **Querverweise (Cross References)** auf **Namensterminale** anderer Regeln angereichert werden. Beispiel:

```
1 Author:
2   'author' name=ID ...;
3
4 Book:
5   'book' name=ID 'was' 'written' 'by' [ Author ];
```

Der referenzierte Name wird durch das Feature `name` deklariert. Der Verweis erfolgt durch eckige Klammern `[]`.

A.3.5 Verschiedenes

- Allgemeine **Einstellungen** können in Dateien mit der Endung **.mwe2* vorgenommen werden. Dazu gehören zum Beispiel der DSL-Name, die Schriftenkodierung, Unit-Tests.

- Die Ausführung kann **Interpretation** oder *Codegenerierung* erfolgen. Defaultmäßig findet man eine von `AbstractGenerator` abgeleitete Klasse in der Datei `*generator.xtend`. Die Methode `doGenerate(..)` wird überschrieben.

- Die folgenden Validierungsarten werden unterstützt:

  - **Syntaktische Validierung:** Die Syntax wird immer automatisch überprüft. Fehlermeldungen können per `ISyntaxErrorMessageProvider` API geändert werden. Aufgetretene Fehler können durch `getErrors()` und `getWarnings()` abgefragt werden.
  - **Validierung von Querverweisen:** "Cross-Reference Validation".
  - **Benutzerdefinierte Validierung:** Zusätzliche Validierungen können durch den `AbstractDeclarativeValidator` implementiert werden. Zum Beispiel kann geprüft werden, ob ein Name mit einem Großbuchstaben beginnt.

- **Scoping** bezeichnet den Bereich eines Programms, in dem ein deklarierter Bezeichner gültig ist. *Xtend* erlaubt es zu definieren, welche Elemente durch gegebene Referenzen zugreifbar sind. Der `IScopeProvider` gibt den Zugriff aller sichtbaren Elemente zurück.

- **Unit-Tests:** Der Xtext Projektwizard erzeugt automatisch ein Projekt für Unit-Tests per *JUnit 4* und *JUnit 5*. Dieses Projekt hat den Namenspostfix `.tests`. Es wird Beispielcode für die Hello-World Beispielsprache bereitgestellt. Der Import von `org.junit.Assert.*` stellt `assert*` Funktionen bereit. Eine Klasse `ParseHelper` stellt eine Methode `parse(..)` als Einsprungpunkt bereit.

- **Javaintegration:** Man hat Zugriff auf die Datentypen der Java Virtual Machine (JVM). Beispielsweise kann der folgende Code einer Grammatikdatei hinzugefügt werden:

```
1 import "http://www.eclipse.org/xtext/common/JavaVMTypes"
2   as jvmTypes datatype Date mapped-to java.util.Date
```

  Javatypen können weiterhin auch per **Xbase** referenziert werden; was bevorzugt werden sollte. *Xbase* ist eine partielle Programmiersprache, die in Xtext implementiert ist. Man bindet *Xbase* wie folgt in die Grammatikdatei ein:

```
1 grammar org.eclipse.xtext.example.Domainmodel
2   with org.eclipse.xtext.xbase.Xbase
```

  Beispiel:

```
name = ValidID;
param = FullJvmFormalParameter;
```

Eclipse als Custom-IDE kann wie folgt konfiguriert werden:

- **Label Provider:** Zeigt ein Label oder Bild bei einem Mouseover-Event.
- **Content Provider:** Die Codevervollständigung kann angepasst werden.
- **Quick Fixes:** Benutzerdefinierte Warnungen, Fehler und Vorschläge für den DSL Nutzer.
- **Template Proposals:** Code Templates, die als DSL Fragmente eingefügt werden können.
- **Outline View:** Graphische baumartige Darstellung des Modells.
- **Hyperlinking:** Intrinsische Hyperlinks sind im Rahmen der Grammatik gegeben. Das standardmäßige Verhalten kann angepasst werden.
- **Syntax Coloring:** Syntaxhighlighting zur Verbesserung der Lesbarkeit.
- **Rename Refactoring:** Die standardgemäße Umbenennung kann angepasst werden.
- **Project Wizard:** Benutzerdefinierter Wizard für die Projekterzeugung.
- **File Wizard:** Benutzerdefinierter Wizard für die Dateierzeugung.
- **Code Mining:** Fügt Inline-Annotationen hinzu.

Weitere Informationen entnimmt man der Xtext-Dokumentation [19].

A.4 JetBrains MPS

JetBrains MPS ist eine von JetBrains entwickelte **Sprachwerkbank**. Im Unterschied zu den meist üblichen syntaxgerichteten Ansätzen, verfolgt MPS den Ansatz des **projektionalen Editierens**. JetBrains MPS wendet sich vor allem an Domänenexperten, die ihre DSLs eigenständig entwickeln möchten; aber nicht unbedingt selbst über viel Programmiererfahrung verfügen. MPS bietet vor allem auch Möglichkeiten um **graphische DSLs** zu erzeugen. Da im Netz viele Tutorials zu finden sind und sich dieser Kurs auf grammatikbasierte Ansätze fokussiert, werden wir hier nicht weiter ins Detail gehen.

Anhang B

Der Iom-Compiler

"Programming is usually taught by examples."

— Niklaus Wirth

Begleitend zur Lehrveranstaltung CBDSL wird ein einfacher Beispielcompiler bereitgestellt. Dieser übersetzt die imperative Universalsprache **Iom** (Esperanto für "wenig") in einen einfachen Maschinencode. Der Compiler ist selbst in der Sprache C implementiert und nutzt den Scannergenerator **Flex** für die lexikalische Analyse und den Parsergenerator **GNU Bison** für das Parsen. Der Sourcecode ist unter der Adresse https://gitlab.com/andreas.schwenk/iom frei erhältlich.

Das folgende Beispielprogramm berechnet die Fakultät einer Zahl in der Sprache *Iom*:

```
1 + factorial(x : int) : int {
2   res = 1;
3   for i in [1,x]
4     res = res * i;
5   return res;
6 }
```

Dazu äquivalenter C Code sieht wie folgt aus:

```
1 int factorial(int x) {
2   int res = 1;
3   for(int i=1; i<=x; i++)
4     res *= i;
5   return res;
6 }
```

Der Compiler emittiert durch Übersetzung den folgenden Maschinencode:

```
1   $00000000 ALLOC_STACK 16                  # alloc(16)
2   $00000001 LOAD_IMMEDIATE r0 1             # r0 := 1
3   $00000002 STORE_LOCAL r0 FP-8             # res := r0
4   $00000003 LOAD_IMMEDIATE r0 1             # r0 := 1
5   $00000004 STORE_LOCAL r0 FP-16            # i := r0
6   $00000005 LOAD_LOCAL r0 FP-16             # r0 := i
7   $00000006 LOAD_LOCAL r1 FP+0              # r1 := x
8   $00000007 CMP_GREATER_EQUAL r1 r0 r1      # r1 := r0 >= r1
9   $00000008 BRANCH_IF_ZERO r0 0 $00000018   # if(r0==0) { jump $00000018 }
```

```
10    $00000009 LOAD_LOCAL r0 FP-8           # r0 := res
11    $00000010 LOAD_LOCAL r1 FP-16          # r1 := i
12    $00000011 MUL r0 r0 r1                 # r0 := r0 * r1
13    $00000012 STORE_LOCAL r0 FP-8          # res := r0
14    $00000013 LOAD_LOCAL r0 FP-16          # r0 := i
15    $00000014 LOAD_IMMEDIATE r1 0          # r1 := 0
16    $00000015 ADD r0 r0 r1                 # r0 := r0 + r1
17    $00000016 STORE_LOCAL r0 FP-16         # i := r0
18    $00000017 BRANCH $00000005             # jump $00000005
19    $00000018 LOAD_LOCAL r0 FP-8           # r0 := res
20    $00000019 RETURN_WITH_VALUE            # return r0
```

B.1 Lexer

Im Folgenden sind die regulären Ausdrücke für den Scannergenerator Flex gelistet.

```
1    /* ----- comments ----- */
2
3  "//".*    { }
4
5    /* ----- keywords ----- */
6
7  "module" { return TK_MODULE; }
8  "return" { return TK_RETURN; }
9  "for"    { return TK_FOR; }
10 "in"     { return TK_IN; }
11 "if"     { return TK_IF; }
12 "int"    { return TK_INT; }
13 "real"   { return TK_REAL; }
14
15   /* ----- constants ----- */
16
17 [a-zA-Z_][a-zA-Z0-9_]* {
18   yylval.string_val = strdup_(yytext);
19   return TK_IDENTIFIER; }
20 [1-9][0-9]*\.[0-9]* {
21   yylval.real_val = atof(yytext);
22   return TK_CONST_REAL; }
23 [1-9][0-9]* {
24   yylval.int_val = atoi(yytext);
25   return TK_CONST_INT; }
26 "0" {
27   yylval.int_val = 0;
28   return TK_CONST_INT; }
29
30   /* ----- symbols ----- */
31
32 "<=" { return TK_LEQ; }
33 ">=" { return TK_GEQ; }
34 "==" { return TK_EQUAL; }
35 "+=" { return TK_ADD_ASSIGN; }
36 "-=" { return TK_SUB_ASSIGN; }
37 "*=" { return TK_MUL_ASSIGN; }
38 "/=" { return TK_DIV_ASSIGN; }
```

```
39 "<"  { return TK_LESS; }
40 ">"  { return TK_GREATER; }
41 "="  { return TK_ASSIGN; }
42 "("  { return TK_LEFT_PARENTHESIS; }
43 ")"  { return TK_RIGHT_PARENTHESIS; }
44 "["  { return TK_LEFT_BRACKET; }
45 "]"  { return TK_RIGHT_BRACKET; }
46 "{"  { return TK_LEFT_BRACE; }
47 "}"  { return TK_RIGHT_BRACE; }
48 "+"  { return TK_PLUS; }
49 "-"  { return TK_MINUS; }
50 "*"  { return TK_ASTERISK; }
51 "/"  { return TK_DIVIDE; }
52 ","  { return TK_COMMA; }
53 ":"  { return TK_COLON; }
54 ";"  { return TK_SEMICOLON; }
55
56   /* ----- white spaces ----- */
57
58 " "  { }
59 "\t" { }
60 "\n" { yycolumn = 1; }
61
62   /* ----- unknown tokens ----- */
63
64 .    { return TK_UNEXPECTED; }
```

B.2 Parser

Die nachfolgenden kontextfreien **Produktionsregeln** geben den Kern der Sprache **Iom** wieder. Man beachte auch den **Aktionscode**, der die nachgeschalteten Phasen aufruft. Der **Operatorvorrang** ist ähnlich wie in der Sprache C umgesetzt.

```
1  program:
2    program_part
3    | program program_part
4    ;
5
6  program_part:
7    module
8    | function
9    ;
10
11 module:
12   TK_MODULE
13   TK_IDENTIFIER                { push_module($2); }
14   TK_SEMICOLON
15   ;
16
17 function:
18   function_visibility
19   TK_IDENTIFIER                { begin_function($2); }
20   TK_LEFT_PARENTHESIS
21   function_parameter_list
```

```
22    TK_RIGHT_PARENTHESIS          { end_of_function_parameter
23                                        _declaration(); }
24    TK_COLON
25    type                          { push_function_return_type(); }
26    block                         { end_function(); }
27    ;
28
29  function_visibility:
30    TK_PLUS
31    | TK_MINUS
32    ;
33
34  function_parameter_list:
35    function_parameter
36    | function_parameter_list
37      TK_COMMA
38      function_parameter
39    | /* no parameter */
40    ;
41
42  function_parameter:
43    TK_IDENTIFIER
44    TK_COLON
45    type                          { push_function_parameter($1); }
46    ;
47
48  block:
49    TK_LEFT_BRACE
50    statement_list
51    TK_RIGHT_BRACE
52    | statement
53    ;
54
55  statement_list:
56    statement
57    | statement_list statement
58    ;
59
60  statement:
61    expression TK_SEMICOLON     { push_expression(); }
62    | for_loop
63    | if
64    | return TK_SEMICOLON
65    ;
66
67  type:
68    TK_INT                        { push_type(TYPE_INT); }
69    | TK_REAL                     { push_type(TYPE_REAL); }
70    ;
71
72  return:
73    TK_RETURN                     { push_return(false); }
74    | TK_RETURN expression        { push_return(true); }
75    ;
76
77  for_loop_range:
```

```
78    TK_LEFT_BRACKET
79    expression
80    TK_COMMA
81    expression
82    TK_RIGHT_BRACKET
83    ;
84
85  for_loop:
86    TK_FOR
87    TK_IDENTIFIER
88    TK_IN
89    for_loop_range              { begin_for_loop($2); }
90    block                       { end_if_statement(); }
91    ;
92
93  if:
94    TK_IF
95    TK_LEFT_PARENTHESIS
96    expression
97    TK_RIGHT_PARENTHESIS        { begin_if_statement(); }
98    block                       { end_if_statement(); }
99    ;
100
101 expression:
102   relational
103   | expression TK_ASSIGN     relational
104                               { push_binary_operation("="); }
105   | expression TK_ADD_ASSIGN relational
106                               { push_binary_operation("+="); }
107   | expression TK_SUB_ASSIGN relational
108                               { push_binary_operation("-="); }
109   | expression TK_MUL_ASSIGN relational
110                               { push_binary_operation("*="); }
111   | expression TK_DIV_ASSIGN relational
112                               { push_binary_operation("/="); }
113   ;
114
115
116 relational:
117   add
118   | relational TK_EQUAL add    { push_binary_operation("==");  }
119   | relational TK_LESS add     { push_binary_operation("<");  }
120   | relational TK_GREATER add  { push_binary_operation(">");  }
121   | relational TK_LEQ add      { push_binary_operation("<="); }
122   | relational TK_GEQ add      { push_binary_operation(">="); }
123   ;
124
125 add:
126   mul
127   | add TK_PLUS  mul           { push_binary_operation("+"); }
128   | add TK_MINUS mul           { push_binary_operation("-"); }
129   ;
130
131 mul:
132   unary
133   | mul TK_ASTERISK unary      { push_binary_operation("*"); }
```

```
134   | mul TK_DIVIDE   unary       { push_binary_operation("/"); }
135   ;
136
137 unary:
138   TK_IDENTIFIER                 { push_variable($1); }
139   | TK_IDENTIFIER               { push_function_call($1); }
140     TK_LEFT_PARENTHESIS
141     function_argument_list
142     TK_RIGHT_PARENTHESIS
143   | TK_CONST_INT                { push_const_int($1); }
144   | TK_CONST_REAL               { push_const_real($1); }
145   | TK_LEFT_PARENTHESIS
146     expression
147     TK_RIGHT_PARENTHESIS
148   ;
149
150 function_argument_list:
151   function_argument
152   | function_argument_list
153     TK_COMMA
154     function_argument
155   | /* no argument */
156   ;
157
158 function_argument:
159   expression                    { push_function_argument(); }
160   ;
```

B.3 Aufrufkonventionen im Iom-Compiler

Die Abbildungen B.1 bis B.7 zeigen ein Beispiel für die **Aufrufkonvention** (engl. calling conventions).

Abb. B.1: Iom-Funktionsaufruf 1/7: Initialisierung: Framepointer (FP) und Stackpointer (SP) werden auf die Adresse $1000 gesetzt. Die `main` Funktion ($80) wird aufgerufen.

Abb. B.2: Iom-Funktionsaufruf 2/7: Wir allokieren Speicher für die Stackvariablen `x` *und* `y`*. Der Einfachheit wegen, werden wir hier den Größen der Datentypen keine Aufmerksamkeit schenken.*

Abb. B.3: Iom-Funktionsaufruf 3/7: Wir legen (push) beide Argumente auf den Stack und rufen letztendlich die `add` *Funktion auf.*

Abb. B.4: Iom-Funktionsaufruf 4/7: Es sei auf die Definition "complex instruction" für `call` *verwiesen.*

Abb. B.5: Iom-Funktionsaufruf 5/7: Wir reservieren wieder Speicher auf dem Stack und führen Berechnungen aus.

Abb. B.6: Iom-Funktionsaufruf 6/7: Es wird zur Aufrufadresse zurückgekehrt. Siehe "complex instruction" von `return`.

Abb. B.7: Iom-Funktionsaufruf 7/7: Zum Schluss werden die Argumente vom Stack gelöscht (pop).

Anhang C

DSL Beispiele

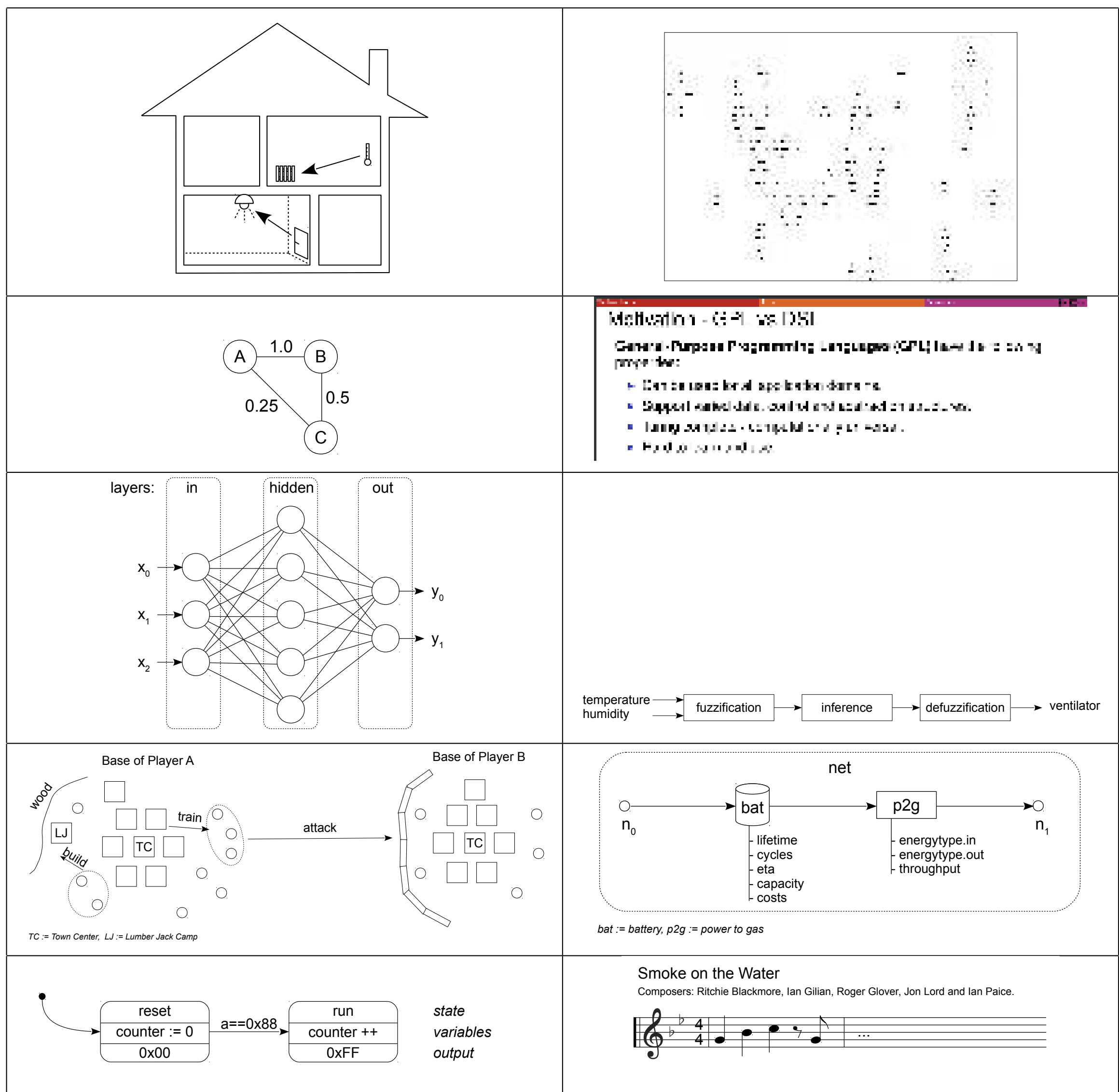
Die folgenden Codefragmente dienen der **Inspiration**. Nicht alle Beispiele sind optimal gewählt, sondern laden zur **Diskussion** ein. Tabelle C.1 zeigt die zugehörigen Abbildungen.

C.1 Hausautomatisierung

Das Beispiel Hausautomatisierung . . .

- beschreibt eine Gerätedatenbank, bestehend aus Sensoren und Aktoren.
- instanziiert ein Smart Home, welches konkrete Technologien, sowie auch Regeln definiert, um Aktoren basierend auf Sensordaten zu steuern.

```
devices {
  sensors {
    tempSensor {
      payload <float> data;
      protocol temp;
    }
  }
  actors {
    heating {
      regulator <int> r;
      ...
    }
    light {
      switch s;
    }
  }
}
home myhome {
  room bedroom {
    light l1, l2;
    tempSensor t with ipv6 = fd5c:8a8f:196e:9912::/64:XXXX:X;
    heating h;
    rules {
      if t.payload < 15 then r = 3;
    }
```



Tab. C.1: Grafische Darstellungen zu den Beispiel-DSLs. Von oben links nach unten rechts: Hausautomatisierung, Conway's Game of Life, Graphen, Textsatz, Neuronale Netze, Fuzzy Logik, Echtzeitstrategiespiele, Energienetzmodellierung, Zustandsautomaten, Musiknotation.

```
  }
}
```

C.2 Conway's Game of Life

Das Beispiel Game of Life demonstriert eine kurze Notation, um den bekannten zellulären Automaten von Conway zu implementieren. Die Syntax ist nicht auf das konkrete Beispiel limitiert.

```
for k=0:10, x=0:25, y=0:25
  n =+ u=-1:1, v=-1:1, u != 0, v != 0 : cells[k,x+u,y+v]
  cells[k=0, x=0, y=0] = 0
  cells[k=0, x>0, y>0] = random(1/4)
  cells[k>0, x,   y  ] = ( cells[k-1,x,y] and (n < 2 or n > 3) )
      or ( cells[k-1,x,y] and (n==3) )
```

C.3 Graphenbeschreibung

Das folgende Beispiel schlägt eine Notation vor, um Graphen zu beschreiben. Weiterhin werden Funktionen für Graphalgorithmen bereitgestellt.

```
vertices V(x,y) = {
  A=(0,0), B=(10,0), C=(10,10)
};

edges E(w) = {
  (A,B,1.0), (B,C,0.5), (C,A,0.25)
};

graph G1 = (V, E);

int n = ORDER(G1);            # ORDER := number of nodes
int m = SIZE(G1);             # SIZE := number of edges
path p = SPP(G1.A, G1.C);    # SPP := shortest path
```

C.4 Schriftsatzsystem

Das Beispiel Schriftsatz demonstriert, wie die Folien dieser Lehrveranstaltung erstellt werden.

```
SLIDE Motivation - GPL vs DSL
  . **General-Purpose Programming Languages (GPL)** have the foll...
  | **Universalsprachen (GPL)** haben die folgenden Eigenschaften:
  ITEMS
    . Can be used for all application domains.
    | Koennen fuer alle Anwendungsgebiete verwendet werden.
    . Support varied data, control and abstraction structures.
    | Unterstuetzen eine Vielzahl an Daten-, Kontroll- und Ab...
```

C.5 Künstliche Neuronale Netze

Das Beispiel KNN/ANN erstellt und traininert ein künstliches neuronales Netz.

```
inputlayer  = matrix 3 x 1 of neurons
  with net=ident act=ident out=ident
hiddenlayer = matrix 5 x 1 of neurons
  with net=dist_from_center act=gauss out=ident
outputlayer = matrix 2 x 1 of neurons
  with net=wsum act=ident out=ident
layers = fully feedforward matrix of inputlayer
  -> hiddenlayer -> outputlayer

rbf = create ann of layers

inputdata = readdata("input.csv")
outputdata = readdata("output.csv")
train(rbf, inputdata, outputdata)
```

C.6 Fuzzy-Logik

Das Beispiel Fuzzy-Logik beschreibt eine Fuzzy-Steuerung für einen Ventilator

```
VAR temperature
  TERM cold = LEFT_HAND_SADDLE(left=-20, right=10) END
  TERM ok   = TRIANGULAR(-15, 0, 15) END
  TERM hot  = LEFT_HAND_SADDLE(30, 10) END
END

VAR humidity
  ...
END

VAR ventilator
  ...
END

IF
  temperature=hot OR humidity=wet
THEN
  ventilator=fast
END
```

C.7 KI für Echtzeitstrategiespiele

Das Beispiel RTS zeigt die Modellierung einer einfachen künstlichen Intelligenz für Echtzeitstrategiespiele.

```
ruleif age = ancient and wood>cost(lumberjack_camp) and
  num(idle(builders)) > 0
  => build lumberjack_camp near (towncenter, large(wood))

```

```
ruleif age = middle age and food > 50*cost(samurai)
  => train (0.4*food / food(samurai)) samurai
  => attack(largest(opponent))

define attack(opponent)
  select 90 % of idle(military) near homebase
  pathto towncenter(opponent)
```

C.8 Energieverbundsysteme

Das Beispiel Energieverbundsysteme beschreibt (partiell) ein Netzwerk, welches aus Energiespeichern und -konvertern besteht. Ein Szenario definiert das Ziel minimaler Investitionskosten.

```
storage battery:
  energytype = power
  lifetime = 5 year
  cycles.max = 100k
  eta.in = 0.8
  eta.out = 0.8
  capacity = 1 .. 4 kWh
  costs.investement = 10k * exp(0.05 * capacity) EUR

converter power_to_gas:
  ...

network net:
  node n0 at pos(50,50)
  storage battery as bat at pos(100,50)
  converter power_to_gas as p2g at pos(200,50)
  node n1 at pos(100,50)
  n0 -> bat.in
  bat.out -> p2g.in
  p2g.out -> n1

scenario s1:
  use network net as N s.t. behavior is ideal
  objective = minimize n.bat.costs.investment

run:
  simulate s1
```

C.9 Zustandsautomaten

Das Beispiel Zustandsautomaten zeigt ein Fragment, um Transduktoren gemäß Mealy zu modellieren.

```
inputs:
  a : vec8
  b : bool
outputs:
  c : vec16
```

```
6  variables:
7    counter : int := 0
8
9  machine:
10   init state reset:
11     counter = 0
12     c = 0x00
13     transitions:
14       run if a == 0x88
15   state run:
16     counter ++
17     c = 0xFF
18     transitions:
19       reset if b
```

C.10 Musiknotation

Das Beispiel Musiknotation beschreibt die ersten Noten von “Smoke on the Water”.

```
1  Title: Smoke on the Water.
2
3  Composers: Ritchie Blackmore, Ian Gilian, Roger Glover,
4      Jon Lord and Ian Paice.
5
6  Line: 4/4 Key Bb ||: 1/4 g b C 1/8 . g | . 1/4 b 1/8 CIS 1/2 C | ...
7  ...
```

Abbildungsverzeichnis

1.1 Diagramme zu den Beispielgraphen G_1, G_2 und G_3. 12

2.1 Generator und Akzeptor 17
2.2 Chomsky Hierarchie 18
2.3 Deterministischer endlicher Automat (DEA) 1 19
2.4 Deterministischer endlicher Automat (DEA) 2 20
2.5 Nichtdeterministischer endlicher Automat (NEA) 21
2.6 Potenzmengenkonstruktion 22
2.7 Automat für die Dyck-Sprache 22
2.8 Prinzip eines Stackspeichers. 23
2.9 Erstes Beispiel zu Kellerautomaten. 24
2.10 Zweites Beispiel zu Kellerautomaten. 25
2.11 Erweiterte Chomsky Hierarchie 26

3.1 Phasen eines Compilers 31
3.2 Compiler Frontend 32
3.3 Compiler Backend 33
3.4 Beispiele für T-Diagramme 33
3.5 Textuelle und graphische Darstellung eines abstrakten Syntaxbaums . 34
3.6 Transduktor 35
3.7 LL-Parser 41
3.8 Hierarchische Auflistung von Bottom-Up Parsertypen 43
3.9 AST zum Beispielwort 43
3.10 Beispiel zur Alignierung. 55
3.11 Beispielübersetzung in die Iom-ISA 57
3.12 Codegenerierung mathematischer Ausdrücke 58
3.13 Codegenerierung bedingter Ausdrücke 58
3.14 Veranschaulichung des Peephole Ansatzes zur Codeoptimierung . . . 61
3.15 Clang und LLVM 68

4.1 Expertenwissen 70
4.2 Integrationsmöglichkeiten von DSLs 72
4.3 Domänenhierarchie 73
4.4 Einbeziehung der DomänennutzerInnen und -expertenInnen 74
4.5 Zusammenschluss von DSL-Fragmenten 75
4.6 Verschiedene Größen von Sprachen 76
4.7 Abdeckung, Modularität 77

5.1 Abstrakter Syntaxbaum mit zugehörigem s-Ausdruck 85

6.1 Zustandsautomat für eine Aufzugsteuerung als Beispieldomäne 89
6.2 Semantisches Modell für eine Transition eines Zustandsautomaten . . 89
6.3 Eingebettete Übersetzung und Baumkonstruktion 92

A.1 Flex und GNU Bison 98

B.1 Iom-Funktionsaufruf 1/7 114
B.2 Iom-Funktionsaufruf 2/7 115
B.3 Iom-Funktionsaufruf 3/7 115
B.4 Iom-Funktionsaufruf 4/7 116
B.5 Iom-Funktionsaufruf 5/7 116
B.6 Iom-Funktionsaufruf 6/7 117
B.7 Iom-Funktionsaufruf 7/7 117

Tabellenverzeichnis

1 Zuordnung: Folienkapitel, Skriptkapitel 9

1.1 Beispiele für domänenspezifische Sprachen. 14

2.1 Tabellarische Übergangsfunktion eines DEA 20
2.2 Tabellarische Übergangsrelation eines NEA 21
2.3 Syntaktische Elemente der Backus Naur Form (BNF). 27
2.4 Syntaktische Elemente der Erweiterten Backus Naur Form (EBNF). . 27
2.5 Beispiel für eine attributierte Grammatik mit Typregeln 28

3.1 Übersetzung von EBNF Konstrukten in Produktionsregeln 39
3.2 Beispiel für eine Shift-Reduce Tabelle 44
3.3 Unvollständige Liste von Parsergeneratoren 45
3.4 Action-Goto Tabelle 47
3.5 Beispiel für eine attributierte Grammatik mit Typregeln 48
3.6 Beispielcode und Kontext-IDs 51
3.7 Symboltabellen für das Beispielprogramm aus Tabelle 3.6 51
3.8 Beispiel einer attributierten Grammatik mit semantischer Auswertung 52
3.9 Beispiel einer Grammatik mit Aktionscode 52
3.10 Mapping der Konstanten auf Register 53
3.11 Maschinencode 53
3.12 Iom-Befehlssatzarchitektur 56
3.13 Erweiterung der Iom-Befehlssatzarchitektur 60

A.1 Vordefinierte Terminalsymbole in Xtext 105
A.2 Xtext Grammatikregeln 106

C.1 Grafische Darstellungen zu den Beispiel-DSLs 120

Index

Abdeckung, 77
Abstrakter Syntaxbaum, 34
Aktionstabelle, 46
Allzwecksprache, 14
Alphabet, 16
Angelegenheit/Concern, 78
angewandte DSL, 70
ANTLR, 99
attributierte Grammatik, 28, 48
Ausdrucksstärke, 76
Ausgaberegeln, 52

Backus Naur Form, 27
Baumkonstruktion, 91
Befehlssatzarchitektur, 55
BNF, 27
Bottom-Up Parsing, 42

Chomsky-Hierarchie, 18
Codeoptimierung, 59
Compiler, 31
Compilerbackend, 32
Compilerfrontend, 31
Compilermiddleend, 32

DAG, 13
Datenflusssprache, 79
Deklaration, 49
deklarative Sprache, 79
Deterministisch kontextfreie Sprache, 25
Deterministischer Endlicher Automat, 20
Dokumentation, 93
Domänenanalyse, 73
DomänenexpertInnen, 74
Domänenmodell, 89
Domänenmodellierung, 73
DomänennutzerInnen, 74
DSL Beispiele, 119

EBNF, 27
Eclipse Xtext, 99
eingebettete Übersetzung, 91
eingebettete DSL, 71, 81
eingebettete Interpretation, 92
Entwurfsdimension, 76
Entwurfsmuster, 82
ereignisgesteuerte Sprache, 79
Erweiterte Backus Naur Form, 27
externe DSL, 71, 87

First-Menge, 36
Follow-Menge, 36
Formale Sprache, 16
Fragment, 71
funktionale Sprache, 79
Funktionenfolge, 83

Geltungsbereich, 50
geschachtelte Funktionen, 83
GNU Bison, 98
Goto-Tabelle, 46
Grammatik, 17
Graph, 12
Graphfärbung, 13, 66

Homoikonizität, 86
Hostsprache, 82

imperative Sprache, 79
Instruktion, 53
Interpretation, 52
Iom Compiler, 109
Iom-Compiler, 67

JetBrains MPS, 108

Kellerautomat, 23
Konfiguration, 24
Konstantenfaltung, 60
Kontext, 50
Kontextfreie Sprache, 23

Konzept, 75

Lebendigkeit, 65
Lebensbereich, 65
Lexer, 34
Linksrekursion, 36, 39
Lisp, 84
LL(k)-Parser, 41
LL-Parser, 40
Look-ahead, 36
LR(k) Parser, 44

Maschinencode, 53
Metasprache, 26
Methodenverkettung, 82
modellbewusste Generierung, 90
Modellierung, 88
modellignorante Generierung, 90

Nichtdeterministischer Endlicher Automat, 21
Nominalphrase, 15

O-Notation, 11
Offlineoptimierung, 61
Onlineoptimierung, 60

Palindrom, 25
Parsergenerator, 45
Parsing, 36
Peephole-Optimierung, 61
Potenzierung, 16
Programm, 75

Reduce, 43
Reduce-Reduce Konflikt, 47
Register, 52
Registereinsatz, 65
Registermanagement, 53
Reguläre Sprache, 19
rekursiver Abstieg, 37

Scanner, 34
Scanning, 34
Schleifenoptimierung, 64
Semantik, 14, 77
semantische Analyse, 48
semantisches Modell, 88
Shift, 43
Shift-Reduce Konflikt, 47
Softwaretests, 93
Sprachgröße, 75
Sprachklassen, 18
Sprachkonzept, 75
Sprachmodularität, 78
Sprachumfang, 75
Sprachwerkbank, 93
Strukturierung, 79
Symboltabelle, 34, 49
Syntax, 13
syntaxgesteuerte Übersetzung, 91

technische DSL, 70
Templated Generierung, 90
Token, 15
Tokenizer, 34
Top-Down Parsing, 37
Transduktor, 29
Transformer Generierung, 90
trennzeichengesteuerte Übersetzung, 91
Typregel, 48

Universalsprache, 14

Verbalphrase, 15
Verhaltensparadigma, 79
verzögerte Codegenerierung, 60
Vollständigkeit, 78

Wörterbuch, 84
Wort, 16

Zielarchitektur, 54

Literaturverzeichnis

[1] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. Compilers, principles, techniques and tools. *Addison Wesley*, 7(8):9, 1986, 2006. "Dragon Book".

[2] Frances E. Allen and John Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137, 1976.

[3] Lorenzo Bettini. *Implementing domain-specific languages with Xtext and Xtend.* Packt Publishing Ltd, 2016.

[4] Gregory J Chaitin, Marc A Auslander, Ashok K Chandra, John Cocke, Martin E Hopkins, and Peter W Markstein. Register allocation via coloring. *Computer languages*, 6(1):47–57, 1981.

[5] Noam Chomsky. Three models for the description of language. *IRE Transactions on information theory*, 2(3):113–124, 1956.

[6] Noam Chomsky. *Syntactic structures.* 1957.

[7] Martin Fowler. *Domain-specific languages.* Pearson Education, 2010.

[8] Richard A Frost, Rahmatullah Hafiz, and Paul Callaghan. Parser combinators for ambiguous left-recursive grammars. In *International Symposium on Practical Aspects of Declarative Languages*, pages 167–181. Springer, 2008.

[9] Paul Graham. *ANSI common lisp*, volume 2003. Prentice Hall Englewood Cliffs, NJ, 1996.

[10] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. Introduction to automata theory, languages, and computation. *Acm Sigact News*, 32(1):60–65, 2001.

[11] Stephen C Johnson et al. *Yacc: Yet another compiler-compiler*, volume 32. Bell Laboratories Murray Hill, NJ, 1975.

[12] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. Design guidelines for domain specific languages. *arXiv preprint arXiv:1409.2378*, 2014.

[13] Gary A Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206. ACM, 1973.

[14] D.E. Knuth. *The Art of Computer Programming*, volume 1-4A. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1968-2011.

[15] Donald E Knuth. On the translation of languages from left to right. *Information and control*, 8(6):607–639, 1965.

[16] Donald E Knuth. Semantics of context-free languages. *Mathematical systems theory*, 2(2):127–145, 1968.

[17] Misc. Gnu bison documentation.
`https://www.gnu.org/software/bison/manual/bison.pdf`.

[18] Misc. Xtend documentation.
`https://www.eclipse.org/xtend/documentation/`.

[19] Misc. Xtext documentation.
`https://www.eclipse.org/Xtext/documentation/`.

[20] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.

[21] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart CL Kats, Eelco Visser, and Guido Wachsmuth. *DSL engineering: Designing, implementing and using domain-specific languages*. dsl-book.org, 2013.

[22] Edmund Weitz. *Common Lisp recipes: a problem-solution approach*. Apress, 2016.

[23] Niklaus Wirth. *Compiler construction*, volume 1. Citeseer, 1996.
`http://www.ethoberon.ethz.ch/WirthPubl/CBEAll.pdf`,
Oldenbourg Verlag.