

Skript zur Vorlesung

# Programmiersprachen und Übersetzer

Sommersemester 2016

Dr. Hans H. Brüggemann  
Institut für Praktische Informatik



# Inhaltsverzeichnis

<b>1 Sprachen als Kommunikationsmittel</b>	<b>1</b>		
1.1 Syntax, Semantik und Pragmatik . .	1		
1.2 Abstraktion von Maschinensprachen	2		
1.3 Über Assemblersprachen zu höheren Programmiersprachen . . . . .	2		
1.4 Klassen höherer Programmiersprachen	3		
1.5 Kurze Programmiersprachen-Historie	5		
<b>2 Sprachprozessoren</b>	<b>14</b>		
2.1 Interpretierer . . . . .	14		
2.2 Übersetzer . . . . .	14		
2.3 Aufbau eines Übersetzers . . . . .	16		
2.3.1 Lexikale Analyse . . . . .	16		
2.3.2 Syntaktische Analyse . . . . .	17		
2.3.3 Semantische Analyse . . . . .	17		
2.3.4 Zwischencode-Erzeugung . .	17		
2.3.5 Codeoptimierung . . . . .	18		
2.3.6 Code-Erzeugung . . . . .	18		
2.3.7 Symboltabellen-Verwaltung .	18		
2.3.8 Fehlerbehandlung . . . . .	18		
2.4 Bootstrapping und Portierung . . . .	19		
2.4.1 Bootstrapping . . . . .	19		
2.4.2 Portierung . . . . .	21		
2.5 Reale und virtuelle Maschinen . . . .	22		
2.5.1 Sprachprozessoren kombiniert	22		
2.5.2 Aufbau virtueller Maschinen	23		
<b>3 Zwischencode für funktionale Abstraktion</b>	<b>24</b>		
3.1 Konzepte imperativer Sprachen . . .	24		
3.2 Speicherbelegung im Keller . . . . .	25		
3.3 Ausdrücke . . . . .	26		
3.3.1 Geschachtelter Ausdruck . . .	26		
3.3.2 Wertzuweisung . . . . .	28		
3.4 Anweisungen . . . . .	30		
3.4.1 Anweisungsfolge . . . . .	30		
3.4.2 Alternative . . . . .	30		
3.4.3 Wiederholung . . . . .	31		
3.5 Speicherplatzbedarf und Überlauf . .	31		
3.6 Funktionen . . . . .	32		
3.6.1 Inkarnation und Aufrufbaum	32		
3.6.2 Auswertungsstrategien . . . .	33		
3.6.3 Gültigkeit von Namen . . . .	35		
3.6.4 Laufzeitkeller . . . . .	36		
3.6.5 Adressierung lokaler Variablen	37		
3.6.6 Betreten und Verlassen . . .	39		
3.6.7 Funktionsaufruf . . . . .	40		
3.6.8 Funktionsdefinition . . . . .	42		
3.7 Übersetzung eines Programms . . . .	43		
<b>4 Zwischencode für die Datenabstraktion</b>	<b>44</b>		
4.1 Datenverbunde und Datenfelder . . .	44		
4.2 Namenlose Daten auf der Halde . . .	46		
4.3 Konzepte objektorientierter Sprachen	49		
4.3.1 Objekte . . . . .	49		
		4.3.2 Objektklassen . . . . .	50
		4.3.3 Vererbung und Teiltypen . .	50
		4.3.4 Generizität . . . . .	51
		4.3.5 Informationskapselung . . . .	52
		4.4 Speicherorganisation für Objekte . .	53
		4.5 Methodenaufruf . . . . .	55
		4.6 Definition von Methoden . . . . .	56
		4.7 Verwendung von Konstruktoren . . .	56
		4.8 Definition von Konstruktoren . . . .	57
		4.9 Definition von Klassen . . . . .	58
		4.10 Mehrfachbeerbung . . . . .	58
<b>5 Syntaktische Analyse</b>	<b>59</b>		
5.1 Theoretische Grundlagen . . . . .	59		
5.1.1 Kontextfreie Grammatik, erzeugte Sprache, Ableitungsbau, Linksreduktion . . . .	59		
5.1.2 Kellerautomat . . . . .	62		
5.1.3 Reguläre Sprache, regulärer Ausdruck, endlicher Automat	63		
5.2 Aufbau von lexikalischen Scannern . . .	66		
5.3 Deterministische linksableitende Syntaxanalyse . . . . .	67		
5.3.1 Gemeinsames Präfix und linksrekursive Produktion . .	67		
5.3.2 LL(1)-Grammatik . . . . .	68		
5.3.3 Top-down-Parser mit Tabelle	72		
5.3.4 oder rekursivem Abstieg	74		
5.3.5 Beispielparser für Mini-Sprache	78		
5.4 Deterministische linksreduzierende Syntaxanalyse . . . . .	81		
5.4.1 Reduktionsstelle, tabellengesteuerter Bottom-up-Parser .	81		
5.4.2 Deterministischer LR-Parser .	83		
5.4.3 Mehrdeutige Grammatik beim Bottom-up Parsing . . .	86		
5.4.4 Sackgassenvermeidung . . . .	88		
5.5 Stadiumautomat und Parser . . . . .	90		
5.5.1 Stadiumautomat . . . . .	90		
5.5.2 Links- und Rechtsparser . . .	92		
5.5.3 Charakteristischer Automat .	94		
5.5.4 Kanonischer LR(0)-Automat	97		
5.5.5 Kanonischer LR(1)-Parser . .	100		
5.5.6 SLR(1)- und LALR(1)-Parser	104		
5.5.7 Optimierter (LA)LR(1)-Parser	106		
<b>6 Semantische Analyse</b>	<b>110</b>		
6.1 Typprüfung . . . . .	110		
6.1.1 Typ-Ausdrücke . . . . .	111		
6.1.2 Typ-Äquivalenz . . . . .	112		
6.1.3 Typ-Umwandlung . . . . .	113		
6.1.4 Überlagerung von Funktionen	113		
6.1.5 Typvariablen . . . . .	114		
6.2 Unifikation . . . . .	115		
6.2.1 Substitution und Unifikator .	115		
6.2.2 Allgemeinsten Unifikator . . .	117		

6.2.3	Typinferenz . . . . .	119	<b>8</b>	<b>Logische Programmierung (Prolog)</b>	<b>136</b>
<b>7</b>	<b>Funktionale Programmierung (Haskell)</b>	<b>121</b>	8.1	Konzepte logischer Sprachen . . . . .	136
7.1	Konzepte funktionaler Sprachen . . .	121	8.1.1	Fakten, Regeln und Anfragen	136
7.1.1	Funktionen und Variablen . .	121	8.1.2	Resolution und Unifikation .	138
7.1.2	Ändern von Datenstrukturen	121	8.2	Computer-Arithmetik in Prolog . . .	141
7.1.3	Rekursive Datenstrukturen .	121	8.3	Listen in Prolog . . . . .	143
7.1.4	Speicherverwaltung . . . . .	122	8.4	Verkleinerung des Suchraums . . . .	144
7.1.5	Mustervergleich . . . . .	122	8.5	Zur Übersetzung von Prolog . . . . .	145
7.1.6	Funktionale . . . . .	122	8.6	Funktional-logische Sprachen . . . .	146
7.1.7	Bedarfsauswertung . . . . .	122		<b>Anhang</b>	
7.1.8	Typisierung . . . . .	123	<b>A</b>	<b>Funktionen zur Codeerzeugung</b>	<b>148</b>
7.2	Benutzung von Haskell . . . . .	123	A.1	Berechnung von Adressen: $\text{code}_A$ . .	148
7.3	Datentypen . . . . .	124	A.2	Berechnung von Werten: $\text{code}_W$ . . .	148
7.3.1	Tupel . . . . .	124	A.3	Anweisungen: $\text{code}$ . . . . .	149
7.3.2	Algebraische Datentypen . .	125	<b>B</b>	<b>Befehlssatz der C-Maschine</b>	<b>150</b>
7.3.3	Listen . . . . .	125	<b>C</b>	<b>Eine einfache Programmiersprache</b>	<b>151</b>
7.4	Fallunterscheidung . . . . .	126	C.1	Entfernung gemeinsamer Präfixe . .	151
7.5	Funktionen . . . . .	126	C.2	First- und Follow-Funktionswerte . .	151
7.5.1	Mustervergleich und Wächter	126	C.3	Syntaxanalyse-Tabelle . . . . .	152
7.5.2	Präfix- und Infixnotation . .	127	C.4	Beispiel für eine Syntaxanalyse . . .	152
7.6	Typvariablen und Typklassen . . . .	128	<b>D</b>	<b>Klassen formaler Sprachen</b>	<b>153</b>
7.7	Gemeinsame Teilausdrücke . . . . .	128	<b>E</b>	<b>Der Parser-Generator yacc</b>	<b>157</b>
7.8	Die Bildung von Listen . . . . .	129	E.1	Eine Beispieleingabe für yacc . . . .	158
7.8.1	durch Eigenschaften . . . . .	129	E.2	Syntax-gesteuerte Übersetzung . . .	159
7.8.2	und arithmetische Folgen . .	129	<b>F</b>	<b>Syntaktischer Zucker in Haskell</b>	<b>162</b>
7.9	Funktionale . . . . .	130	F.1	Funktionsdefinitionen . . . . .	162
7.9.1	Currying . . . . .	130	F.2	Ausdrücke . . . . .	162
7.9.2	Anonyme Funktionen . . . . .	131	F.3	Typ-Ausdrücke . . . . .	162
7.10	Monaden . . . . .	132		<b>Literatur</b>	<b>163</b>
7.11	Funktionale Programmierstile . . . .	132		<b>Index</b>	<b>165</b>
7.12	Bemerkungen zur Übersetzung funktionaler Programmiersprachen . . . .	132			
7.12.1	Höhere Funktionen . . . . .	132			
7.12.2	Striktheitsanalyse . . . . .	133			
7.12.3	Call-by-Need . . . . .	133			
7.12.4	Unendliche Listen . . . . .	135			
7.12.5	Übersetzung von Haskell . . .	135			

## Vorwort

Ich freue mich, das Skript von Prof. Parchmann weiterführen zu dürfen. Neben vielen kleinen Änderungen habe ich –um Raum für die Kapitel über Code-Erzeugung zu bekommen–, die Kapitel über die historische Entwicklung der Programmiersprachen und über Scanner gekürzt.

Die Graphen, die die Berechnung der First- und Follow-Mengen erleichtern, heißen jetzt nach ihrem Erfinder Parchmann-Graphen.

Hannover, März 2012

H. H. Brüggemann

Die Veranstaltung „Programmiersprachen und Übersetzer“ ist in erster Linie eine Pflichtveranstaltung für den Bachelor-Studiengang Informatik. In dieser Veranstaltung bauen die Studierenden ihre bisherige Programmiersprachenerfahrung aus und lernen unterschiedliche Programmierparadigmen kennen. Aufbauend auf den vorhandenen Kenntnissen von imperativen und objektorientierten Sprachen werden deren wesentliche Sprachkonstrukte in maschinennahe Befehle für eine virtuelle Maschine übersetzt.

Die Syntaxanalyse für deterministische kontextfreie Sprachen wird sowohl im top-down- als auch im bottom-up-Ansatz behandelt. Insbesondere werden hier Ergebnisse der theoretischen Informatik nutzbringend praktisch angewendet. Die semantische Analyse behandelt unter anderem die Typprüfung und die Typermittlung durch Inferenz. Die Unifikation kann aber nicht nur dafür eingesetzt werden, sondern auch für den Mustervergleich und zur Beschreibung der Arbeitsweise der logischen Programmiersprache *Prolog*. Als funktionale Programmiersprache wird ab diesem Semester *Haskell* vorgestellt, eine Sprache mit verzögerter Auswertung.

Ich danke Prof. Parchmann, dass er mir den Einstieg durch Überlassen von Skript und Folien sehr erleichtert hat.

Hannover, März 2014

H. H. Brüggemann

In der aktuellen Auflage wurden vor allem die Kapitel über Bottom-up-Syntaxanalyse überarbeitet und einiges über die Programmiersprache *Haskell* ergänzt. Außerdem wurde ein Anhang mit Informationen über verschiedene Sprachklassen angefügt.

Hannover, März 2015

H. H. Brüggemann

In der aktuellen Auflage wurde u.a. ein Abschnitt über Auswertungsstrategien, zur Realisierung unendlicher Listen und über funktional-logische Programmiersprachen eingefügt.

Hannover, März 2016

H. H. Brüggemann

# 1 Sprachen als Kommunikationsmittel für Menschen und Rechner

Sprachen sind das wichtigste Kommunikationsmittel zwischen Menschen. Natürliche Sprachen werden mündlich und schriftlich verwendet. **Programmiersprachen** sind künstliche Sprachen und werden fast nur schriftlich verwendet. Mit einem Programm einer Programmiersprache will ein Programmierer mit einem Rechner kommunizieren; Programmiersprachen müssen daher **Algorithmen** beschreiben können.



Abb. 1.1: Das Wort „Computer“ in natürlichen Sprachen.

## 1.1 Syntax, Semantik und Pragmatik von Sprachen

Jede Sprache –auch Programmiersprachen– hat die Aspekte Syntax, Semantik und Pragmatik.

Die **Syntax** einer Sprache beschreibt die Beziehung der Zeichen untereinander:

- Nach welchen Regeln des Satzbaus muss ich einen Text schreiben?
- Aus welchen Bestandteilen besteht ein gegebener Text und ist er grammatikalisch korrekt?

**Beispiel 1.1:** Werden Elemente einer Liste durch Komma oder Zwischenraum getrennt?

Die **Semantik** einer Sprache beschreibt die Beziehung der Zeichen zur realen Welt:

- Welche Bedeutung will ich mit einem Text vermitteln?
- Was sagt ein gegebener Text aus?

**Beispiel 1.2:** Wird die Abbruchbedingung einer Schleife am Anfang oder Ende geprüft?

Die **Pragmatik** einer Sprache beschreibt die Beziehung der Zeichen zu ihren Benutzern:

- Wie formuliere ich, damit mich die Zielgruppe gut versteht?
- Kann ich aus der Formulierung eines Textes die Absicht des Schreibers ableiten?

**Beispiel 1.3:** Ist ein Sprachkonstrukt (z.B. Sprung, Zeiger) fehleranfällig? Kann ich unproblematischere Konstrukte (z.B. Fallunterscheidung und Schleife, rekursiver Datentyp) einsetzen?

Auch die Pragmatik ist wichtig, denn ein Programm soll nicht nur vom Computer ausgeführt werden, sondern Menschen (insbesondere der Programmierer selbst – auch noch nach Monaten!) sollen es verstehen.

## 1.2 Programmiersprachen als Abstraktion der Maschinensprachen

Programme einer Programmiersprache sollen gut lesbar sein, d.h. mehrere Programmierer sollen mit Hilfe der Programmiersprache untereinander über Algorithmen kommunizieren können. Insbesondere soll ein Programmierer so auch mit sich selbst kommunizieren können: Die Programmiersprache soll helfen, Gedachtes festzuhalten, um später ein Verfeinern, Präzisieren oder Überprüfen eines Programms zu ermöglichen.

Programmiersprachen sind entstanden, um Algorithmen in einer Form zu beschreiben, die für Rechner *und* Menschen lesbar ist. Bzgl. der „Lesbarkeit“ gibt es aber ein breites Spektrum von Programmiersprachen. So sind Assemblersprachen für den Rechner viel einfacher „lesbar“ als eine höhere Programmiersprache wie **Java**, während für den Programmierer das Umgekehrte gilt.

Darüber hinaus haben die Strukturen der erlernten bzw. benutzten Programmiersprache einen großen und häufig unterschätzten Einfluss auf die Art, wie ein Programmierer von einem Problem zu einem Algorithmus gelangt (**Mutterspracheneffekt**). Man denke z.B. an das Problem des Durchlaufs durch einen binären Baum und an einen Programmierer, der mit einer funktionalen Sprache wie **Haskell** kompetent umgeht und im Gegensatz dazu an einen Programmierer, der mit einer Sprache ohne rekursive Aufrufe wie **BASIC** groß geworden ist. Beide werden diese Aufgabe mit sehr unterschiedlichen Algorithmen lösen!

Ein Rechner verarbeitet unmittelbar nur Anweisungen *einer* Programmiersprache, nämlich seiner **Maschinensprache**. Ein Maschinenprogramm ist eine Folge von Nullen und Einsen, also von Bits, die zu Bit-Blöcken –Byte oder Wort– zusammengefasst werden. Ein Mensch kann ein solches Programm nur äußerst fehleranfällig schreiben oder lesen; daher wird man nur ungern ein Programm direkt in einer Maschinensprache notieren.

Jede andere Programmiersprache benötigt einen **Sprachprozessor**: Entweder ein Übersetzungsprogramm, das ein in dieser Sprache vorgelegtes Programm in die Maschinensprache des Rechners übersetzt (**Compiler**, **Übersetzer**) oder ein Simulationsprogramm, das einen Rechner simuliert, der diese Programmiersprache als Maschinensprache hat (**Interpreter**). Mischformen sind möglich und werden vielfach benutzt.

Eine Programmiersprache soll dem Programmierer Mechanismen und sprachliche Werkzeuge zur Verfügung stellen, um Algorithmen eindeutig, knapp und für andere Menschen leicht nachvollziehbar zu formulieren. Im Idealfall wird die Ausdruckstärke durch wenige, intuitive Konzepte erreicht, die beliebig kombinierbar (orthogonal) sind; im schlimmsten Fall durch viele Konzepte mit einer großen Zahl von (fehlerträchtigen) Kombinationsverboten und Ausnahmen. Eine einfache Struktur der Programmiersprache ermöglicht einen einfachen, effizienten Übersetzer oder Interpreter. Natürlich soll eine Programmiersprache von der Maschinensprache des Rechners abstrahieren. Auch soll der Programmierer auf vorgefertigte Programmteile zugreifen und diese einfach in das eigene Programm integrieren können.

## 1.3 Von Maschinensprachen über Assemblersprachen zu höheren Programmiersprachen

Ein Programm einer **Maschinensprache** besteht aus Maschinenbefehlen. Jeder Maschinenbefehl ist eine Bitfolge. Der erste Teil der Bitfolge, der **Operationsteil**, bestimmt die Klasse des Maschinenbefehls, der Rest, der **Operandenteil**, beschreibt den/die Operand(en) dieses Maschinenbefehls. Dabei hängt vom Operationsteil ab, wie viele Operanden folgen und wie lang (in bit) der Maschinenbefehl ist. Lesbar ist das nur sehr mühsam.

**Assemblersprachen** verbessern daher vor allem die Lesbarkeit.

- Assembler bieten für den Operationsteil einen **mnemotechnischen Code** an, eine leicht zu merkende Abkürzung, z.B. `jmp` (*jump*) für Sprung- oder `add` für Additionsoperationen.
- Mehrere Operanden werden durch Komma getrennt und nicht als Bitfolgen fester Länge hintereinander geschrieben.
- Konstante Operanden dürfen dezimal (oder hexadezimal) angegeben werden statt dual.

Absolute Speicher- und Sprungadressen behindern die Wiederverwendbarkeit von Programmteilen. Deshalb beschreibt man

- einen Speicherplatz (für einen variablen Operanden) durch einen (Variablen-) **Namen**,
- die Adresse eines Sprungziels durch eine symbolische **Marke** (*label*).

Eine Assemblersprache muss noch genutzt werden, wenn Programmteile zeitkritisch sind (z.B. Gerätetreiber für Grafikkarten), mit wenig Speicherplatz auskommen müssen (z.B. eingebettete Systeme) oder noch keine entsprechende Hochsprachen-Bibliothek existiert (z.B. neue Technik).

**Höhere Programmiersprachen** abstrahieren stärker vom Maschinenmodell. Schon in imperativen Programmiersprachen können

- Ausdrücke beliebig geschachtelt werden, z.B.  $2 + \sin(\pi/3) * 3$ ,
- häufig benutzte Programmiermuster zur Steuerung des Programmablaufs verwendet werden, z.B. Alternativen (`if...then...else...fi`) und Schleifen (`while...do...od`),
- durch **Deklaration** von Namen Schreibfehler erkannt werden, die z.B. eine Fehlermeldung „*undeclared identifier*“ verursachen anstatt als verschiedene Variable behandelt zu werden,
- Ausdrücke bzw. Anweisungen zu einer Einheit (**Funktion**, **Prozedur**) zusammengefasst werden und zusammengehörige Variable ebenso (**Datenverbund**, **Datenfeld**).

Das Aufschreiben detaillierter Berechnungsschritte ist nicht die einzige Art des Programmierens. Man kann auch mit einer Spezifikation der gewünschten Eigenschaften des Ergebnisses beginnen und daraus anschließend die nötigen Berechnungsschritte (z.T. automatisch) ableiten.

## 1.4 Sprachklassen für höhere Programmiersprachen

Programme werden heute meist in einer problem-orientierten, höheren **Programmiersprache** (*programming language*) geschrieben. Diese Sprachen abstrahieren –unterschiedlich stark– von der Struktur und den Details der Rechner, auf denen die Programme ausgeführt werden sollen.

Die wichtigsten Klassen von **universell einsetzbaren Programmiersprachen** enthalten:

- **imperative (prozedurale) Programmiersprachen**,  
z.B. Algol 60, Algol 68, FORTRAN, COBOL, Pascal, Ada, Modula-2, C.

Diese Sprachen orientieren sich aus Effizienzgründen eng an der Architektur des **von-Neumann-Rechners** aus (aktiver) Zentraleinheit (*central processing unit*, CPU), (passivem) Speicher und einem Bus für den Austausch von Daten zwischen Zentraleinheit und Speicher. Diese Architektur liegt fast allen kommerziell erhältlichen Rechnern zugrunde.

Grundeinheit imperativer Sprachen ist die **Anweisung** (*statement*). Anweisungen verändern den **Zustand** (*state*) des Programms und steuern den Programmablauf.

Imperative Sprachen kennen neben Anweisungen auch Ausdrücke. Ausdrücke liefern Werte. Durch die Auswertung von Ausdrücken verändert sich manchmal sogar der Zustand (**Nebenwirkung**, *side effect*).

Imperative Sprachen bieten **funktionale Abstraktion** an (mit Nebenwirkungen!).

*You can  
never  
understand  
one language  
until you  
understand  
at least two.*

Geoffrey  
Willans

- **objektorientierte Programmiersprachen**,  
z.B. Simula, Smalltalk, Eiffel, C++, Modula-3, Java.

Objektorientierte Programmiersprachen enthalten **Klassen** zur **Datenabstraktion** und das Konzept der **Vererbung** ermöglicht eine „evolutionäre“ Art der Software-Entwicklung. Sie sind im Kern aber meist imperativ.

- **funktionale (applikative) Programmiersprachen**,  
z.B. (pure) Lisp, Scheme, Standard ML, Hope, Miranda, FP, OCaml, Haskell.

Hier soll jeder Funktionsaufruf für gleiche Parameter stets das gleiche Ergebnis liefern – unabhängig vom Programmzustand („frei von Nebenwirkungen“), wie bei einer mathematischen Funktion. Diese Eigenschaft erleichtert die Verifikation eines Programms erheblich.

Funktionale Sprachen unterscheiden nicht zwischen Anweisungen und Ausdrücken. Namen bezeichnen Ausdrücke und Funktionen, aber keine Speicherzellen.

Argumente und Ergebnisse von Funktionen können wieder Funktionen sein.

Grundeinheit funktionaler Sprachen ist der **Ausdruck** (*expression*).

Es gibt keinen durch Anweisungen *explizit* angegebenen Programmablauf.

Ausführungsprinzip ist die **Reduktion** (*reduction*): Ein Ausdruck wird schrittweise durch einen äquivalenten, einfacheren ersetzt, bis eine **Normalform** (*normal form*) erreicht wird.

- **logische (prädikative) Programmiersprachen**, z.B. Prolog (und seine Dialekte).

Diese Sprachen nutzen eine operationale Sicht der **Prädikatenlogik** (*predicate calculus*) und erlauben insbesondere freie Variablen im Aufruf.

Grundeinheit logischer Sprachen ist die **Horn-Klausel** (*horn clause*).

Ausführungsprinzip ist die **Resolution** (*resolution*), ein Verfahren um Implikationen in der Prädikatenlogik erster Stufe zu beweisen.

Außerdem gibt es **Sprachen für spezielle Anwendungen**, z.B.

- **Hardware-Beschreibungssprachen** (z.B. VHDL) spezifizieren (Komponenten von) Rechner(n), etwa ihr funktionales Verhalten, ihren hierarchischen Aufbau oder die räumliche Anordnung ihrer Komponenten.
- **Kommandosprachen von Betriebssystemen** (z.B. Unix shell) ermöglichen, **Prozesse** zu erzeugen, mehrere Prozesse koordiniert zusammenwirken zu lassen oder zu beenden. Dadurch können Benutzerprogramme ablaufen oder Systemfunktionen aktiviert werden.
- **Datenbanksprachen** (z.B. SQL) ermöglichen Anfragen an große Datenbestände und deren Änderung, oft ohne Ausführungsdetails vorzugeben.
- **Spezifikationssprachen für Druckseiten, Grafikobjekte oder Animationen** (z.B. Postscript) müssen neben den geometrischen Eigenschaften der darzustellenden Objekte auch zeitliche Abläufe und Reaktionen auf Ereignisse beschreiben können.
- **Sprachen für semi-strukturierte Dokumente** (z.B. SGML, XML) dienen der Beschreibung und dem Austausch von (meist) komplexen Daten(-familien).



## 1.5 Kurzer historischer Abriss zu Programmiersprachen

Um Programmiersprachen vorzustellen und zu vergleichen, verwendet man am besten denselben Algorithmus. Solche Vergleichsprogramme sind z.B.

- „Hello World!“, <http://www.roesler-ac.de/wolfram/hello.htm> oder
- „99 bottles of beer“, <http://www.99-bottles-of-beer.net> .

Wir benutzen den TPK-Algorithmus [11] von LUIS TRABB PARDO und DONALD E. KNUTH, der typische Sprachelemente von imperativen Programmiersprachen enthält:

- mathematische Standardfunktionen,
- Ein- und Ausgabe,
- bedingte Anweisungen,
- Schleifen,
- Unterprogramme sowie
- Felder und indizierte Variablen.

Der Algorithmus liest eine Folge von 11 Gleitpunktzahlen ein, kehrt die Reihenfolge um, berechnet für jede Zahl einen Funktionswert und gibt ihn aus, falls er nicht zu groß ist.

Falls in einer Programmiersprache einige Konzepte fehlen, werden entsprechende Vereinfachungen vorgenommen.

Zunächst eine „moderne“ Version dieses Algorithmus:

### TPK-Algorithmus in Pascal

```
program TPK (input, output);
var i: integer;
    y: real;
    a: array [0..10] of real;

function f (t: real) : real;
begin
    f := sqrt(abs(t)) + 5*t*t*t
end;

begin
for i:= 0 to 10 do read(a[i]);
for i:= 10 downto 0 do
    begin
        y := f(a[i]);
        if y > 400 then
            writeln(i, 'TOO LARGE')
        else
            write(i, y)
        end
    end
end.
```

Einen Überblick über viele Programmiersprachen sowie weitere Informationen zu einzelnen Sprachen bekommt man z.B. über die WWW-Seiten von

ÉRIC LÉVÉNEZ (<http://www.levenez.com/lang/>) und

BILL KINNERSLEY (<http://people.ku.edu/~nkinners/LangList/Extras/langlist.htm>).

**Plankalkül** wurde Mitte der 1940er Jahre vom Erfinder des ersten Computers, KONRAD ZUSE, entwickelt, eine (damals nicht implementierte) Programmiersprache, die schon viele Konzepte imperativer Sprachen vorwegnahm. Da die Sprache aber erst in den 1970er Jahren bekannter wurde, konnte sie die Entwicklung der Programmiersprachen nicht beeinflussen.

**Short Code:** WILLIAM F. SCHMITT schrieb von 1950–52 diesen algebraischen Interpretierer für den von JOHN P. ECKERT und JOHN W. MAUCHLY entwickelten UNIVAC (*universal automatic computer*). Der Interpretierer arbeitete die codierte Darstellung –von rechts nach links– ab; die Übersetzung in die codierte Darstellung erfolgte per Hand. **Short Code** ermöglicht Konstanten im Dezimalsystem, Gleitpunktzahlen, symbolische Marken für Programmadressen.

**TPK-Algorithmus in Short Code**  
(Die Farben dienen nur dem Verständnis.)

Short Code:	codierte Darstellung:
i = 10	00 00 00 00 W0 03 Z2
0: y = ( $\sqrt{\quad}$ abs t) + 5 cube t	01 T0 02 07 Z5 11 T0
y 400 if $\leq$ to 1	02 00 Y0 03 09 20 06
i print, 'TOO LARGE' print-return	03 00 00 00 Y0 Z3 41
0 0 if = to 2	04 00 00 Z4 59 W0 58
1: i print, y print-return	05 00 00 00 Z0 Z0 72
2: T0 U0 shift	06 00 00 Y0 59 W0 58
i = i-1	07 00 00 00 T0 U0 99
0 i if $\leq$ to 0	08 00 W0 03 W0 01 Z1
stop	09 00 00 00 Z0 W0 40
	10 00 00 00 00 ZZ 08

Variable: Speicherbelegung: i = W0, t = T0, y = Y0.  
Die Eingaben kommen nacheinander in die elf Speicherzellen U0, T9, ..., T0.

Konstanten: Z0 = 000000000000  
Z1 = 010000000051 (1.0, floating decimal form)  
Z2 = 010000000052 (10.0)  
Z3 = 040000000053 (400.0)  
Z4 = TOO LARGE  
Z5 = 050000000051 (5.0)

Marken: 0: = Zeile 01, 1: = Zeile 06, 2: = Zeile 07 (der Codierung)

Code-Operator-Zuordnung [13]:

01 -	06 abs value	1n (n+2)te Potenz	59 print und return
02 )	07 +	2n (n+2)te Wurzel	7n if = to n
03 =	08 pause	4n if $\leq$ to n	99 zykl. Shift des Speichers
04 /	09 (	58 print und tab	00 no operation

Die Multiplikation hat keinen Operator-Code; sie wird stets implizit vorgenommen.

**FORTRAN** (*formula translating system*) wurde 1954 unter Leitung von JOHN W. BACKUS (IBM) für die numerische Programmierung entwickelt. Ziel war ein effizienter Maschinencode – er sollte mit dem Code eines guten Assemblerprogrammierers vergleichbar sein. Der Compiler wurde 1957 ausgeliefert.

FORTRAN führte Kommentare, verzweigende IF-Anweisungen (<, =, > Null), Schleifen (DO) mit Endmarkierung und Laufvariable sowie Felder (DIMENSION) ein, auf deren Elemente über –zur Laufzeit berechnete– Indizes zugegriffen werden konnte. Ein FORMAT-Interpreter ermöglicht formatierte Ein- und Ausgabe.

FORTRAN-Programme waren Lochkarten-orientiert; die Bedeutung eines Zeichens variierte mit der Spalte (Kommentar-Kennzeichnung, Anweisungsnummer, Markierung als Fortsetzungszeile, Anweisung, ggf. Lochkartennummer). Leerzeichen wurden ignoriert, waren also kein Trennsymbol.

Variablen brauchten nicht deklariert zu werden; der Anfangsbuchstabe des Namens beeinflusst dann den Typ.

FORTRAN wurde oft weiterentwickelt (FORTRAN II, FORTRAN IV, FORTRAN 66, FORTRAN 77, Fortran90, Fortran95, Fortran2003, Fortran2008) und durch die Unterstützung von IBM sehr verbreitet.

Der sehr effiziente Maschinencode und die große Zahl von Unterprogrammpaketen für numerische Anwendungen in FORTRAN erhält die Sprache am Leben.

#### TPK-Algorithmus in FORTRAN I

```

C      THE TPK-ALGORITHM IN FORTRAN I
      FUNF(T) = SQRTF(ABSF(T))+5.0*T**3
      DIMENSION A(11)
1     FORMAT (6F12.4)
      READ 1,A
      DO 10 J = 1,11
        I = 11-J
        Y = FUNF(A(I+1))
        IF (400.0 - Y) 4,8,8
4     PRINT 5,I
5     FORMAT (I10, 10H TOO LARGE)
      GO TO 10
8     PRINT 9,I,Y
9     FORMAT (I10,F12.7)
10    CONTINUE
      STOP
  
```

Da Leerzeichen in FORTRAN einfach überlesen wurden und keine Namen oder Schlüsselwörter begrenzten, war manchmal eine weite Vorschau nötig: Erst nach dem Lesen des Kommas in der Anweisung `DO 10 J = 1,11` wird klar, dass DO ein Schlüsselwort ist und es nicht um eine Wertzuweisung an die (undeklarierte) Variable `DO10J` geht.

**COBOL** (*common business-oriented language*) wurde 1959 unter Leitung von GRACE M. HOPPER vom Verteidigungsministerium der USA zusammen mit Computerherstellern und Anwendern für kaufmännische Anwendungen entwickelt.

COBOL führte Record-Strukturen für Daten ein.

Die Sprache sieht umfangreiche Formatierungsmöglichkeiten der Daten vor, fällt aber auch durch starke Geschwätzigkeit und ein fehlendes Prozedurkonzept auf.

COBOL war zeitweise die am meisten benutzte Programmiersprache der Welt. Dies lag auch an der Kompatibilität von COBOL-Compilern durch frühe Standardisierung der Sprache.

1,11 ?  
Schreibt man doch mit Dezimalpunkt!  
Ich mach daraus mal 'ne 1.11  
Legende oder wahr:  
Mariner 1 erreichte die Venus nie! (1962)

### TPK-Algorithmus in COBOL

```

IDENTIFICATION DIVISION.
PROGRAM-ID. TPK-Algorithmus.
AUTHOR. mak.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
    CONSOLE IS CRT,
    DECIMAL-POINT IS COMMA.
INPUT-OUTPUT SECTION.

DATA DIVISION.
FILE SECTION.
WORKING-STORAGE SECTION.
77 i PIC S99.
01 y USAGE IS COMPUTATIONAL-1.
01 a-hilf.
    02 a USAGE IS COMPUTATIONAL-1 OCCURS 11 TIMES.
LINKAGE SECTION.

PROCEDURE DIVISION.
    PERFORM VARYING i FROM 1 BY 1 UNTIL i > 11
        ACCEPT a(i)
    END-PERFORM.
    PERFORM VARYING i FROM 11 BY -1 UNTIL i < 1
        PERFORM BERECHNUNG
            IF y > 400 THEN
                DISPLAY i
                DISPLAY "TOO LARGE"
            ELSE
                DISPLAY i
                DISPLAY y
            END-IF
        END-PERFORM.
    STOP RUN.
BERECHNUNG.
    IF a(i) IS NEGATIVE THEN
        COMPUTE y = (-a(i)) ** 0,5 + 5*a(i)*a(i)*a(i)
    ELSE
        COMPUTE y = a(i) ** 0,5 + 5*a(i)*a(i)*a(i)
    END-IF.

```

**Algol 60** (*algorithmic language*) entstand zur gleichen Zeit durch ein europäisch-amerikanisches Komitee (u.a. JOHN W. BACKUS, FRIEDRICH L. BAUER, JOHN MCCARTHY, PETER NAUR, ALAN J. PERLIS, HEINZ RUTISHAUSER, KLAUS SAMELSON). Die Syntax von Algol 60 wurde durch eine kontextfreie Grammatik (in **Backus-Naur-Form**) formal definiert.

Algol 60 führte Blockstruktur, rekursive Prozeduren, explizite Deklaration von Variablen, reservierte Wörter, dynamische Felder (mit variabler Feldgröße) und eine (Lochkarten-) formatfreie Eingabe ein. Fehlende Anweisungen für Ein- und Ausgabe erzeugten aber In-

kompatibilitäten zwischen den Implementierungen.

Algol 60 ist der Prototyp einer imperativen Programmiersprache, d.h. einer Sprache, die Variablen verwendet und Anweisungen –insbesondere die Wertzuweisung– bereitstellt, um Werte von Variablen zu verändern. Sie hatte –zusammen mit ISWIM– gewaltigen Einfluss auf spätere imperative Sprachen wie Pascal, C, Modula-2, Ada und Java. Algol 60 wurde Referenzsprache zur Veröffentlichung von Algorithmen; wirtschaftlichen Erfolg hatte sie nicht.

#### TPK-Algorithmus in Algol 60

```
begin
  integer i; real y; real array a[0:10];
  real procedure f(t); real t; value t;
    f := sqrt(abs(t))+5*t↑3;
  for i := 0 step 1 until 10 do read(a[i]);
  for i := 10 step -1 until 0 do
    begin
      y := f(a[i]);
      if y > 400 then write(i,'too large')
      else write(i,y)
    end
  end
end
```

**Lisp** (*list processor*) wurde ebenfalls in dieser Zeit (1958) entwickelt. JOHN MCCARTHY wollte FORTRAN um Listen erweitern. Daraus entwickelte sich eine Sprache [12], die im Bereich der künstlichen Intelligenz Verbreitung fand.

*lots of  
irritating  
superfluous  
parentheses*

Lisp basiert auf dem  $\lambda$ -Kalkül von ALONSO CHURCH und STEPHEN C. KLEENE. Daten und Programme werden einheitlich als Listen (in Klammerschreibweise) dargestellt (S-Ausdrücke); der Programmablauf wesentlich über Rekursion gesteuert. Speicherplatz für Listenelemente wird dynamisch zur Laufzeit bereitgestellt, nicht mehr benötigter Speicher wird durch **automatische Speicherbereinigung** (*garbage collection*) wieder eingezo-gen. Funktionen sind in Lisp „normale“ Daten.

Werden Berechnungen –nur– durch Anwendungen von Funktionen auf Daten ausgeführt und Wertzuweisungen überflüssig, so spricht man von –rein– funktionaler (applikativer) Programmierung. Moderne funktionale Sprachen sind Scheme, Common Lisp, CLOS, ML, OCaml, Miranda oder Haskell.

**BASIC** (*beginner's all-purpose symbolic instruction code*) wurde 1963/64 von THOMAS E. KURTZ und JOHN G. KEMENY entwickelt als einfach erlernbare Sprache, auch für Studenten anderer Fächer. Sie unterstützte den Wechsel vom (Lochkarten-)Stapelbetrieb zum Time-Sharing-Betrieb. Weil die Sprache zunächst nur interpretiert wurde und relativ klein und einfach war, eignete sie sich später gut für die ersten „Home Computer“.

**SNOBOL** (*string oriented symbolic language*) von DAVID J. FARBER, RALPH E. GRISWOLD und IVAN P. POLONSKY (AT&T Bell Labs), 1964 vorgestellt, war eine Sprache zur Verarbeitung von Zeichenketten und führte den Mustervergleich ein. Moderne Nachfolger sind SNOBOL 4 und ICON, aber auch Skriptsprachen wie Perl oder Tcl.

**PL/I** (*programming language one*) –1965 zunächst NPL (*new programming language*) genannt– wollte die wichtigsten Konzepte von FORTRAN, COBOL und Algol 60 vereinigen. Heraus-

gekommen ist eine Anhäufung von sich z.T. widersprechenden „Features“, z.B. gibt es für Variablen wahlweise eine statische Speicherzuordnung (wie in FORTRAN), eine Blockstruktur (wie in Algol) oder eine dynamische Speicherzuordnung über explizite Aufrufe (wie auch in Pascal). Ohne die Marktmacht von IBM wäre diese Sprache kaum erfolgreich geworden.

**Simula 67** wurde 1967 von OLE-JOHAN DAHL und KRISTEN NYGAARD als Algol-ähnliche Sprache vorgestellt mit Erweiterungen für die diskrete Simulation. Hier wurde das Konzept der **Klasse** als Erweiterung des Blockkonzeptes eingeführt. Eine Klasse besteht aus einer Menge von Datenvereinbarungen und Prozeduren. Ein Objekt einer Klasse wird dynamisch zur Laufzeit erzeugt und seine Lebensdauer ist nicht an die Blockstruktur gebunden. Merkmale von Klassen können vererbt werden. **Simula** legte damit die Grundlage zur objektorientierten Programmierung. Diese Konzepte findet man in relativ reiner Form in **Smalltalk**, oder –in Verbindung mit anderen Konzepten höherer Programmiersprachen– in **C++**, **C#**, **Delphi**, **Eiffel**, **Java**, **Objective-C** und **Python** wieder sowie in den objektorientierten Erweiterungen von **Lisp** (z.B. **Loops**, **Flavors**).

**Algol 68** entstand durch systematische Verallgemeinerung von **Algol 60**. Es sollten wenige, aber beliebig kombinierbare (orthogonale) Sprachkonzepte in der Sprache Verwendung finden. Diese Sprache fand außerhalb der Universitäten nur geringe Akzeptanz, da die formale Definition über zweistufige Grammatiken (**Wijngaarden-Grammatik**) festgelegt wurde und die Handbücher zur Sprache dadurch schwer lesbar waren.

**Algol 68** spezifizierte die Typen der Parameter im Funktionskopf (und nicht getrennt wie in **Algol 60**) und ermöglichte die Initialisierung von Variablen direkt bei der Deklaration (sogar für Felder). Die Blockklammerung durch Schlüsselwörter (z.B. **if-then-else-fi**, **for-do-od**) ermöglicht eine eindeutige Zuordnung des else-Teils.

#### TPK-Algorithmus in Algol 68

```
begin
  [0:10] real a;

  proc f = (real t) real :
    sqrt(abs(t))+5*t↑3;

  for i from 0 to 10 by 1 do read(a[i]) od;
    comment Laufvariable i ist implizit deklariert comment
  for i from 10 to 0 by -1 do
    real y := f(a[i]);
    if y > 400 then print((i, "too large")) else print((i, y)) fi
  od
end
```

**Pascal** (nach BLAISE PASCAL) war 1971 die Antwort von NIKLAUS WIRTH auf die Komplexität von **Algol 68**. Trotz vieler Sprach-Einschränkungen wurde **Pascal**, speziell als Ausbildungssprache, sehr beliebt. Abgelöst wurde **Pascal** Ende der 1980er Jahre durch **Modula-2** und **Oberon**. Für praktische Zwecke wurde die Sprache erst durch Erweiterungen wie in **Borlands Turbo-Pascal** brauchbar. Eine objektorientierte Erweiterung von **Pascal** ist **Delphi**.

**Prolog** (*programming in logic*) wurde 1972 von ALAIN COLMEAUER entwickelt. **Prolog** nutzt als nicht-prozedurale Sprache Teile der mathematischen Logik direkt als Programmiersprache. Fakten und Regeln bilden eine Datenbasis, an die man Anfragen stellen kann. Das System versucht dann, durch logisches Schließen die Anfrage aus der Datenbasis zu beantworten.

Prolog definiert einen speziellen Typ von Programmierung, die logische (deklarative) Programmierung, und ist die wichtigste Programmiersprache für „Expertensysteme“.

**Smalltalk** entstand Anfang der 1970er Jahre im Xerox Forschungslabor in Palo Alto im Rahmen des Dynabook Projekts. Das Dynabook basierte auf einer Idee von ALAN C. KAY für einen persönlichen Rechner, der eine für die damalige Zeit revolutionäre graphische Benutzeroberfläche anbot. Informationen wurden in überlappenden Fenstern angezeigt und die Steuerung des Rechners geschah mit Hilfe einer Maus über Befehle, die man sich nicht mehr merken musste, da sie in Menüs angezeigt wurden. Außerdem sollte das Dynabook Musik verarbeiten und über Datenleitungen Verbindungen zu anderen Geräten oder Datenbanken herstellen können.

*The best way to predict the future is to invent it.*  
Alan Kay

Das System wurde in einer neuen, rein objektorientierten Programmiersprache **Smalltalk** implementiert. Die erste Version war **Smalltalk-72** (1972), die letzte –heute noch aktuelle– Version ist **Smalltalk-80** (ADELE GOLDBERG, 1980). **Smalltalk**-Programme werden von einem Compiler in einen Bytecode übersetzt und dieser wird dann von einem Interpretierer, der „Smalltalk Virtual Machine“, abgearbeitet. **Smalltalk** erlaubt –wie **Simula**– eine einfache Vererbung von Merkmalen.

Die durch dieses Projekt gewonnenen Erkenntnisse sind sehr(!) langsam in den „normalen“ Computermarkt geflossen. Viele haben von diesem Projekt „gelernt“, so **Apple** mit dem **Macintosh** (1983), **Microsoft** mit **Windows**, **Unix** mit **X-Windows**.

## C und C++

C wurde 1972 von DENNIS RITCHIE als Programmiersprache geschaffen, um große Teile des UNIX-Betriebssystems in einer höheren Sprache schreiben zu können. Der Erfolg von C beruht in erster Linie auf dem Erfolg von UNIX. C erlaubt eine relativ maschinennahe Programmierung, verleitet jedoch manchmal zur „Trickprogrammierung“.

### TPK-Algorithmus in C++

```
#include <iostream.h>
#include <math.h>                // smart header file; "-lm" required

double f (double x) {
    return (sqrt(fabs(x)) + 5.0*pow(x,3.0));
}

int main (int argc, char** argv) {
    double A[11];
    for (int i=0; i<11; i++) {
        cin >> A[i];
    }
    /* In reverse order, apply "f" to each element of "A" and print */
    for (i=10; i>=0; i--) {      // i already declared in previous loop
        double y = f(A[i]);
        if (y > 400.0) {
            cout << i << "TOO LARGE\n";
        } else {
            cout << i << ' ' << y << '\n';
        }
    }
    return 0;
}
```

C wurde Anfang der 1980er Jahre durch BJARNE STROUSTRUP zu „C with classes“, der objekt-orientierten Programmiersprache C++ weiterentwickelt. C++ ermöglicht, von mehreren Klassen zu erben.

Ende 1989 wurde C vom ANSI, 1998 C++ von der ISO standardisiert. Das ANSI-C bietet die gleiche Typ-Sicherheit wie Pascal, ohne die Möglichkeiten der maschinennahen Programmierung zu beschneiden.

**Ada** (nach ADA KING, Countess of Lovelace) erschien 1980 nach langer Vorbereitung als eine vom amerikanischen Verteidigungsministerium geforderte und initiierte Programmiersprache. Wie bei PL/I wollte man eine Sprache für ein weites Anwendungsspektrum entwickeln. Ada enthält neue Sprachkonzepte zur Kapselung von Daten (*package*), zur Parallelverarbeitung (*task*) und zur Ausnahmefall-Bearbeitung (*exception*) und eignet sich damit gut zur Steuerung komplexer Systeme. Ada wird viel im militärisch-technischen Bereich benutzt.

**ML** (*meta language*) entstand ab 1973 in Edinburgh als Zwischensprache zu ROBIN MILNERS ambitioniertem Projekt LCF (*logic for computable functions*) eines Theorem-Beweisers und vereint die Flexibilität einer funktionalen Sprache mit einem sehr ausgereiften Typ-System, das eine starke Typ-Prüfung erlaubt. ML wurde 1986 standardisiert (Standard ML, SML).

Charakteristisch für ML ist, dass Typen nicht mehr deklariert werden müssen. Stattdessen schließt dieses **Hindley-Milner-Typsystem** aus dem Gebrauch des Namens auf mögliche Typen und prüft, ob alle Vorkommen konsistent sind (Typinferenz). Außerdem können Typen parametrisiert werden (polymorphe Typen).

Eine Variante ist Caml (*categorical abstract machine* + ML), die um 1985 von GÉRARD HUET entwickelt und 1990 von XAVIER LEROY um objekt-orientierte Konzepte zu OCaml (*objective Caml*) erweitert wurde.

#### TPK-Algorithmus in OCaml

```
let tpk l =
  let f x = sqrt x +. 5.0 *. (x ** 3.0) in
  let p x = x < 400.0 in
  List.filter p (List.map f (List.rev l))
```

**Miranda** wurde 1985 von DAVID TURNER als erste funktionale Sprache für den kommerziellen Gebrauch veröffentlicht. Ein Miranda-Programm ist eine Menge von Gleichungen, die mathematische Funktionen und abstrakte Datentypen definieren. Die Reihenfolge der Gleichungen ist irrelevant. Miranda nutzt den Call-By-Need. Blöcke werden nicht geklammert, sondern durch Einrückung gekennzeichnet. Turners Vorläufer von Miranda waren SASL (*St. Andrews Static Language*) und KRC (*Kent Recursive Calculator*).

**Haskell** (nach HASKELL B. CURRY) entstand ab 1987 aus einer breiten Initiative für eine rein funktionale Programmiersprache mit verzögerter Auswertung. Zu den Entwicklern gehörten PAUL R. HUDAK, PHILIP WADLER und SIMON PEYTON JONES. Die Sprache ist streng typgebunden, ohne dass die Typen von Namen deklariert werden müssen. Neu sind Typklassen. Fallunterscheidungen in Funktionsdefinitionen können durch Mustervergleich (*pattern matching*) beschrieben werden. Argumente werden erst ausgewertet, wenn sie gebraucht werden (verzögerte Auswertung, *lazy evaluation*). 1990 wurde der erste Haskell-Report veröffentlicht, 1999 wurde Haskell 98 veröffentlicht (revidiert 2003). Die aktuelle Version ist Haskell 2010. Haskell beeinflusste u.a. Python und Scala und ist die Basis für funktional-logische Sprachen wie Escher oder Curry.



**Java** entstand –unter dem Namen **Oak**– ab 1991 in einem Team um JAMES GOSLING bei Sun Microsystems. Man wollte eine plattformunabhängige, objektorientierte Software für Geräte der Unterhaltungselektronik entwickeln, die –z.B. über Fernsehkanäle verschickt– in Endgeräte geladen wird. Diese Entwicklung war kein großer Erfolg.

Das World Wide Web (WWW) eröffnete neue Anwendungsmöglichkeiten für das System, das dann den Namen **Java** bekam. 1996 wurde die Entwicklungsumgebung Java Development Kit (JDK) vorgestellt. **Java**-Programme werden vom Compiler in eine Zwischensprache, den Java Byte Code, übersetzt. Dieser wird dann auf dem Zielrechner mit einem Interpretierer, der Java Virtual Machine (JVM), ausgeführt. **Java** enthält viele Konzepte der objektorientierten Programmierung und basiert aus Gründen der besseren Akzeptanz auf C bzw. C++. Allerdings verzichtet **Java** auf einige komplexere Elemente von C++, z.B. auf die Mehrfachbeerbung von Klassen.

**Python** wurde Anfang der 1990er von GUIDO VAN ROSSUM als Lehrsprache entworfen. Ziel ist darum ein gut lesbarer Quellcode.

#### TPK-Algorithmus in Python

```
import math
def f(x):
    return math.sqrt(abs(x)) + 5 * x**3
vals = [float(raw_input()) for i in range(11)]
for i, x in enumerate(reversed(vals)):
    y = f(x)
    print('{0}: {1}'.format(i, y if y <= 400 else 'TOO LARGE'))
```

**Scala** (*scalable language*) entstand 2001–2003 in der Schweiz um MARTIN ODESKY. Die Sprache ist objekt-orientiert und funktional. Sie kann auch auf der Java Virtual Machine ausgeführt werden. Jeder Wert ist ein Objekt; es gibt keine primitiven Datentypen mehr. Statt einer Mehrfachbeerbung können Klassen explizit um (vorhandene) Implementierungen erweitert werden.

Aktuell entwickeln große Konzerne eigene Programmiersprachen: Google die Sprachen **Go** (2009) und **Dart** (2011), Microsoft die Sprache **TypeScript** (2012) und Apple die Sprache **Swift** (2014). Eine konzernunabhängige Entwicklung ist **Rust** (2012).

## 2 Sprachprozessoren für Programmiersprachen

Bevor Programme einer Programmiersprache  $L$  auf einem Rechner ausgeführt werden können, muss diese Programmiersprache auf diesem Rechnertyp verfügbar gemacht, also durch ein Programm, einen Sprachprozessor, **implementiert** werden. Zwei Typen von Sprachprozessoren lösen diese Aufgabe: Interpretierer und Übersetzer.

### 2.1 Interpretierer

Ein **Interpretierer** (*interpreter*)  $I_L$  bekommt als Eingabe ein Programm  $p_L$  einer Programmiersprache  $L$  sowie eine Eingabe  $e$  und berechnet daraus eine Ausgabe  $a$ .

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

Evtl. führt die Interpretation von  $p_L$  aber auch zu einem Fehler. Sind Ein- und Ausgabe Folgen aus einem Bereich  $D$ , so erhält man:

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

Ein Interpretierer bearbeitet das Programm  $p_L$  und die Eingabe  $e$  zur gleichen Zeit.

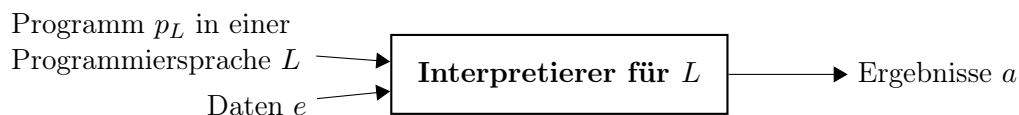


Abb. 2.1: Schema eines Interpretierers.

Auch bei wiederholter Ausführung nutzt er keine Information des Programms aus, die von den Eingabedaten unabhängig ist, wie z.B. die Anzahl der deklarierten Variablen.

Interpretierer gibt es u.a. für die Sprachen BASIC, Lisp, Unix-Kommandosprache (shell), SQL.

### 2.2 Übersetzer

Ein **Übersetzer** (*compiler*)  $\ddot{U}_{L,M}$  soll die Ineffizienzen der Interpretation vermeiden. Dazu nutzt man ein in der Informatik oft nützliches Prinzip, die **Vorberechnung**, **partielle Auswertung** (*partial evaluation*) bzw. **gemischte Berechnung** (*mixed computation*).

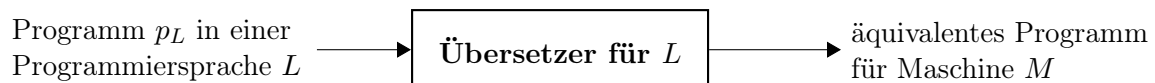


Abb. 2.2: Schema eines Übersetzers.

Während der Interpretierer  $I$  seine beiden Argumente, Programm und Eingabe, zur gleichen Zeit bekommt und verarbeitet, werden jetzt das Programm und die Eingabe zu zwei verschiedenen Zeiten verarbeitet:

- Zunächst, zur **Übersetzungszeit** (*compile time*), wird das Programm  $p_L$  vorverarbeitet, d.h. unabhängig von Eingabedaten analysiert und in eine Form überführt, die die effiziente Ausführung des Programms mit beliebigen Eingaben erlaubt. Dabei nimmt man an, dass sich der zusätzliche Aufwand für die Vorverarbeitung des Programms bei der (meist mehrfachen) Ausführung mit (verschiedenen) Eingaben lohnt.

Diese Vorverarbeitung besteht meist in der Übersetzung eines Programms  $p_L$  einer Sprache  $L$  in ein Programm  $p_M$  der Maschinen- oder Assemblersprache  $M$  eines konkreten oder

virtuellen Rechners. Die Sprache  $L$  heißt **Quellsprache** (*source language*), die Sprache  $M$  **Zielsprache** (*target language*);  $p_L$  **Quellprogramm** (*source program*), das Programm  $p_M$  **Zielprogramm** (*object program*) zu  $p_L$ .

Die Übersetzung von Quellprogrammen in Zielprogramme beseitigt die Ineffizienzen der Interpretation: Jedes Programm wird nur einmal zur Übersetzungszeit analysiert; die einzige noch vorzunehmende Analyse des Zielprogramms (in der Maschinsprache eines konkreten Rechners) ist die Dekodierung des Befehlscodes durch die Befehlsinheit des Rechners.

Der effiziente Zugriff auf Werte von Variablen (bzw. Konstanten) wird durch ein Speicherverwaltungsschema ermöglicht, das jeder Variable eine feste (Relativ-)Adresse zuordnet. (Nur) diese Adressen stehen im erzeugten Zielprogramm. Das erschwert die Zuordnung von Laufzeitfehlern zum Programmtext. Möchte man für Tests oder Fehlersuche den Wert von Variablen (oder Ausdrücken) an bestimmten Stellen im Programm wissen, ist ein **Debugger** nötig.

- Zu einer späteren Zeit, der **Ausführungszeit** oder **Laufzeit** (*run time*), wird das erzeugte Zielprogramm  $p_M$  mit der Eingabe  $e$  ausgeführt. Natürlich verlangen wir von der Übersetzung, dass das Zielprogramm  $p_M$  bei der Ausführung mit der Eingabe  $e$  ebenfalls das Ergebnis  $a$  produziert.

Dabei gibt es verschiedene Fehlersituationen:

- $p_L$  kann syntaktische Fehler oder Verletzungen von Kontextbedingungen enthalten in Programmteilen, die der Interpretierer bei der Ausführung mit Eingabe  $e$  gar nicht berührt. In diesem Fall kann die Interpretation erfolgreich ablaufen, während der Übersetzer –nach Analyse des ganzen Programms– die Übersetzung wegen entdeckter Fehler ablehnt.
- $I_L$  kann, obwohl  $p_L$  syntaktisch korrekt ist und alle Kontextbedingungen erfüllt, bei der Ausführung mit Eingabe  $e$  auf einen (Laufzeit-)Fehler stoßen (z.B. Division durch Null, Additionsüberlauf). Wenn wir  $I_L$  als Definition der Semantik von  $L$  betrachten, dann muss auch das erzeugte Zielprogramm  $p_M$  bei der Ausführung mit  $e$  auf einen Fehler stoßen.

Eine Übersetzung ist zwingend notwendig, wenn die Maschine, auf der das Zielprogramm laufen soll, nicht groß genug für einen Sprachprozessor ist oder nicht die dafür nötigen Ein-/Ausgabemöglichkeiten hat. Dies kommt oft vor bei einem **eingebetteten System** (*embedded system*), bei dem ein Mikroprozessor Teil einer komplexen Anlage (z.B. Auto, Waschmaschine) ist. Ein **Cross-Compiler** erzeugt dann auf einem größeren Prozessor das Zielprogramm für das eingebettete System.

Eigenschaften eines Programms, die nicht von den Eingabedaten abhängen, heißen **statisch**; solche, die von den Eingabedaten abhängen, heißen **dynamisch**. Statische Eigenschaften können zur Übersetzungszeit ermittelt werden, dynamische Eigenschaften erst zur Laufzeit.

Übersetzer gibt es nicht nur für Programmiersprachen! Z.B. werden L<sup>A</sup>T<sub>E</sub>X-Dokumente in dvi, pdf oder html übersetzt oder C-Programme in Ätzvorlagen für entsprechende Schaltkreise.

## 2.3 Prinzipieller Aufbau eines Übersetzers

Das Quellprogramm durchläuft mehrere Stufen, die sich in zwei Phasen gliedern: In der **Analysephase** wird es lexikal, syntaktisch und semantisch untersucht; in der **Synthesephase** wird Code erzeugt und optimiert. Informationen über Namen werden in einer Symboltabelle gespeichert und können dort abgerufen werden. Auch müssen auftretende Fehler behandelt werden.

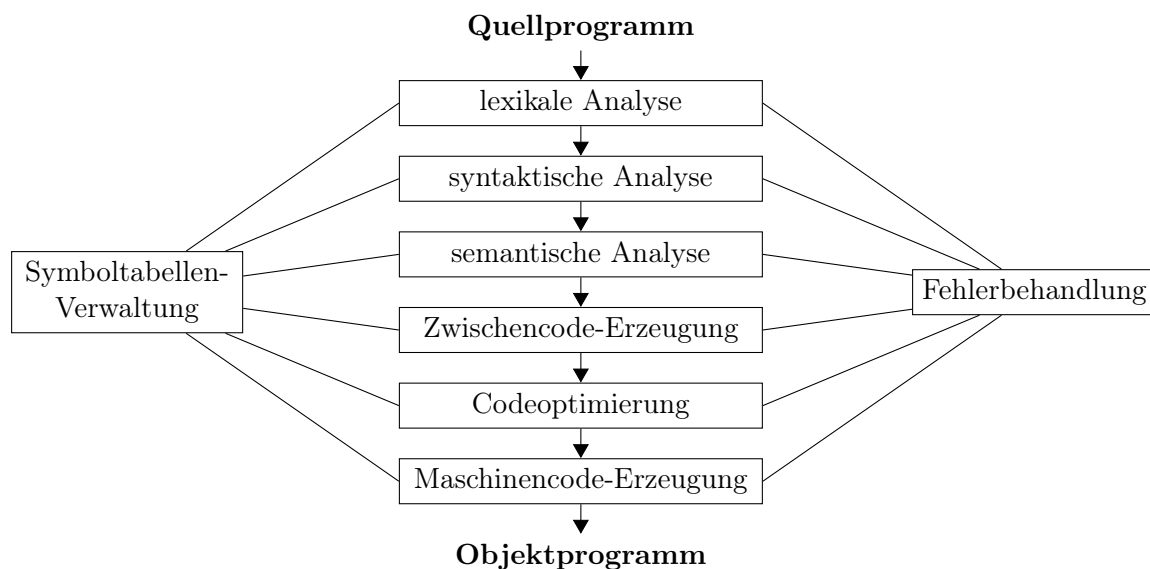


Abb. 2.3: Aufbau eines Übersetzers.

### 2.3.1 Lexikale Analyse

Diese Komponente des Übersetzers heißt (**Lexical**) **Scanner** oder **Tokenizer**. Bei der lexikalischen Analyse wird die Eingabe Zeichen für Zeichen eingelesen und zu größeren Einheiten (**Lexem**) zusammengefasst. Typische Einheiten sind Namen, Konstanten (z.B. Zahlen, Zeichenketten) oder Schlüsselwörter. Bedeutungslose Zeichen (z.B. Leerzeichen, Kommentare) werden überlesen.

Ein Lexem wird im Übersetzer intern durch ein **Token** dargestellt. Ein Token repräsentiert also eine Folge von Eingabezeichen, die als Einheit betrachtet wird; z.B. wird die Zeichenfolge 1234 als eine Zahl und nicht als Folge von vier Ziffern aufgefasst. Ein Token wird einer **Tokenklasse** (hier: *Zahl*) zugeordnet und hat oft einen **Tokenwert** (hier: 1234). Tokenklassen bei Programmiersprachen sind z.B. *Zahl* oder *Identifizier* (für Namen). Jedes Schlüsselwort erhält meist eine eigene Tokenklasse (z.B. *do*, *begin*, *end*). Was als Token gilt, hängt von der beabsichtigten Anwendung ab und wird vom Konstrukteur des Übersetzers definiert.

#### Beispiel 2.1:

Die Eingabezeile

```
position := initial + rate * 60
```

könnte die nebenstehende Tokenfolge liefern.

Dabei ist die Tokenklasse kursiv, der Tokenwert –falls vorhanden– normal gedruckt.

<i>Identifizier</i>	position
<i>Wertzuweisungszeichen</i>	
<i>Identifizier</i>	initial
<i>Pluszeichen</i>	
<i>Identifizier</i>	rate
<i>Multiplikationszeichen</i>	
<i>Zahl</i>	60

Der Scanner trägt Namen in eine **Symboltabelle** ein, die oft weitere Eigenschaften (Attribute) enthält. Der Tokenwert ist dann ein Zeiger auf den Eintrag in der Symboltabelle.

### 2.3.2 Syntaktische Analyse

Diese Komponente des Übersetzers heißt **Syntaxanalysator** oder **Parser**. Ihre Aufgabe ist:

1. Prüfung, ob die von der lexikalischen Analyse gelieferten Token in einer Reihenfolge auftreten, die der (syntaktischen) Beschreibung der Programmiersprache entspricht. (Z.B. gibt *identifizier* + \* *identifizier* in Pascal einen Fehler, in C jedoch nicht!)
2. Gruppieren der Token zu größeren syntaktischen Strukturen, etwa in Form eines Baumes (**Syntaxbaum**). Die internen Knoten des Syntaxbaumes stellen Operationen dar auf Werte, die durch die Teilbäume unterhalb des Knotens bestimmt sind.

**Beispiel 2.2:** Die Token aus Beispiel 2.1 können zu einem Syntaxbaum zusammengefasst werden (vgl. Abb. 2.4).

Die Priorität und die Assoziativität der Operatoren spiegelt sich im Syntaxbaum wider!

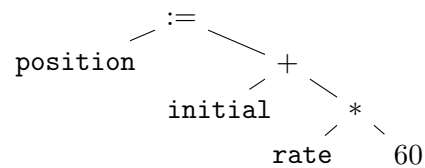


Abb. 2.4: Syntaxbaum.

### 2.3.3 Semantische Analyse

Wichtigste Aufgabe der semantischen Analyse ist die **Überprüfung der Typen** der Namen bzgl. der auf sie angewendeten Operationen und evtl. eine automatische Typanpassung. Ein Feld mit einem Index vom Typ **real** sollte einen Fehler liefern!

**Beispiel 2.3:** Ist die Variable **rate** vom Typ **real**, muss die **int**-Konstante 60 in die **real**-Konstante 60.0 (andere interne Zahldarstellung!) umgewandelt werden (vgl. Abb. 2.5).

Die semantische Analyse überprüft auch, ob alle verwendeten **Namen deklariert** wurden und ob die **Anzahl der Parameter** stimmt.

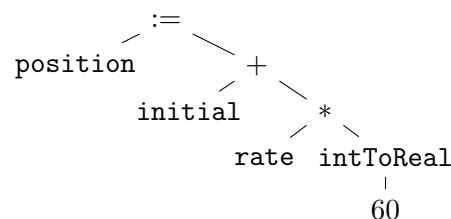


Abb. 2.5: Ergänzter Syntaxbaum.

### 2.3.4 Zwischencode-Erzeugung

Die Zwischencode-Erzeugung erledigt die Übersetzung in einen Zwischencode, d.h. in die Maschinensprache einer virtuellen Maschine mit möglichst einfacher Struktur. Dieser Zwischenschritt ist besonders bei optimierenden Compilern wichtig. Der Zwischencode (z.B. Syntax-Bäume, Byte-Code, 3-Adress-Befehle) sollte so aufgebaut sein, dass er leicht aus der höheren Programmiersprache erzeugbar, aber auch leicht in Maschinensprache übersetzbar ist.

In manchen Fällen, z.B. bei **Smalltalk**, **Java** oder **Python**, ist der Zwischencode bereits der Zielcode, der von einem Programm –wie der Java Virtual Machine– interpretiert wird.

**Beispiel 2.4:** Der Zwischencode wird hier durch 3-Adress-Befehle der Form **A := B op C** angegeben, wobei **A** das Ergebnis, **B** und **C** die Operanden (durch ihre Speicheradressen beschrieben) und **op** den Operator bezeichnen.

Das Beispiel 2.3 könnte folgende Übersetzung in 3-Adress-Befehle liefern.

Dabei sind **t1**, **t2** und **t3** neu erzeugte Namen.

```

t1 := intToReal 60
t2 := rate * t1
t3 := initial + t2
position := t3
  
```

### 2.3.5 Codeoptimierung

Die Codeoptimierung soll den Zwischencode verbessern bzgl. Speicherplatzbedarf und Laufzeit. Dazu benötigt man Algorithmen, die z.B. überflüssige Berechnungen entfernen, gemeinsame Teilausdrücke nur einmal berechnen, algebraische Vereinfachungen durchführen, nicht erreichbaren Code entfernen, Berechnungen zur Laufzeit durch Berechnungen zur Übersetzungszeit ersetzen sowie schleifeninvariante Berechnungen erkennen und diese Teile vor den Schleifenbeginn verschieben. Viele Optimierungen hängen bereits stark von den Eigenschaften der Zielmaschine ab.

**Beispiel 2.5:** Nach der Codeoptimierung ergibt sich folgender Zwischencode:

```
t2 := rate * 60.0
position := initial + t2
```

### 2.3.6 Code-Erzeugung

Die Code-Erzeugung soll verschiebbaren Maschinencode (d.h. ein Objektprogramm) bzw. Assemblercode erzeugen. Schwierig ist (besonders bei RISC-CPU's oder Parallelrechnern), die wenigen freien Register so zuzuordnen, dass möglichst wenig Speicherzugriffe nötig sind.

**Beispiel 2.6:**

Seien R1 und R2 freie Floating-Point-Register:

```
MOVf rate, R2
MULF #60.0, R2
MOVf initial, R1
ADDF R2, R1
MOVf R1, position
```

### 2.3.7 Symboltabellen-Verwaltung

Die **Symboltabelle** speichert die im Programm vorkommenden Namen. Zusätzlich enthält sie weitere Informationen, z.B. bei Variablenamen den Typ der Variable, die zugeordnete Relativadresse und ggf. die Schachtelungstiefe der Deklaration, um verschiedene Variablen gleichen Namens zu unterscheiden; bei Typnamen z.B. die Größe des Typs. Aus den Informationen der Symboltabelle kann z.B. die Adressumgebung (siehe Kapitel 3.2) ermittelt werden.

### 2.3.8 Fehlerbehandlung

Programmierer wären unzufrieden, wenn der Übersetzer die Übersetzung nur mit der Meldung „Ihr Programm ist kein Wort der Programmiersprache“ oder „Programm enthält noch Fehler“ beenden würde. Gewünscht sind hilfreiche Angaben über Position und Art der Fehler. Dazu muss der Übersetzer nicht nur merken, dass das Programm kein Wort der Programmiersprache ist, sondern aus der Art der Abweichung auf mögliche Fehlerquellen schließen.

## 2.4 Bootstrapping und Portierung von Übersetzern

Nachdem man eine neue Programmiersprache (im folgenden  $L$  genannt) entworfen hat, muss man für sie, um sie praktisch benutzen zu können, einen Compiler schreiben.

Einen Compiler symbolisiert man gern durch ein **T-Diagramm**:

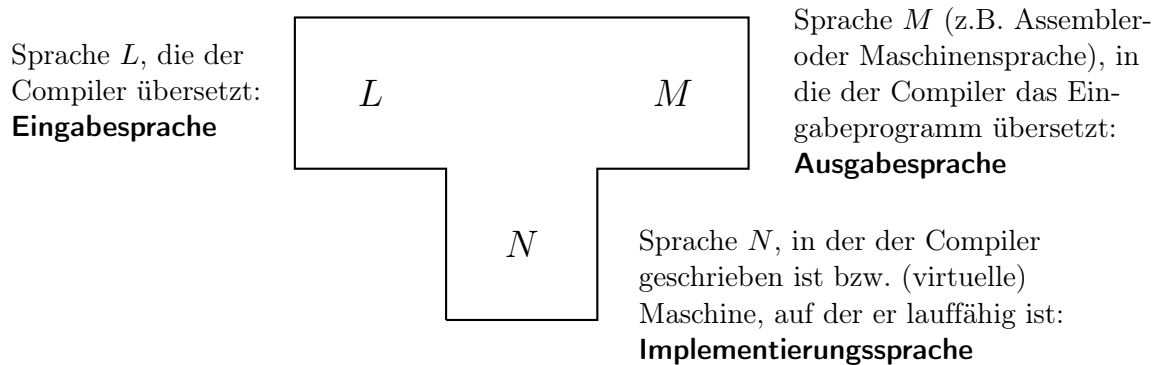


Abb. 2.6: T-Diagramm für Übersetzer.

Den Compiler schreibt man zweckmäßigerweise in der Sprache  $L$  selbst; denn da ein Compiler ein relativ komplexes Programm ist, erkennt man dabei die Leistungsfähigkeit (oder auch Mängel) der neuen Programmiersprache, und man erhält ein (umfangreiches) Testprogramm, mit dem man die korrekte Funktion des neuen Compilers überprüfen kann. Die Ausgabesprache dieses in  $L$  geschriebenen Compilers für  $L$  kann die Maschinen- oder Assemblersprache  $M$  des für die Entwicklung verwendeten Rechners sein.

### 2.4.1 Bootstrapping

- Der erste und aufwändigste Schritt des Bootstrapping besteht im manuellen Erstellen eines Compilers ①, geschrieben in der zu übersetzenden Sprache.
- Aus diesem Compiler ① entwickelt man einen deutlich einfacheren Compiler ②.

$L'$  ist eine Teilmenge der Sprache  $L$ , die nur unbedingt nötige Sprachkonstrukte enthält, und  $M'$  eine Teilmenge der Maschinensprache  $M$ , die auf effiziente Spezialbefehle verzichtet.

Dieser Compiler darf also langsam sein, einen hohen Speicherbedarf haben und nicht optimierte, also langsame, platzverschwendende  $M'$ -Programme erzeugen.

- Der letzte manuell, aber schon mechanischer ablaufende Schritt ist die Übertragung von Compiler ② in einen Compiler ③. Dies kann durch direktes Übersetzen von  $L'$  in  $M'$  geschehen oder durch Übersetzen von  $L'$  in eine Programmiersprache  $S$ , für die ein Compiler vorhanden ist (z.B. C oder FORTRAN), und Benutzen dieses Compilers.

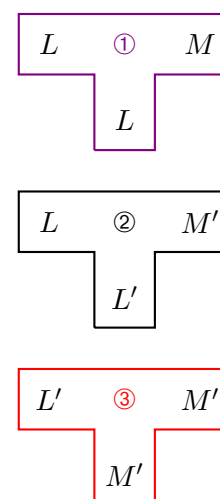


Abb. 2.7: Die manuellen Schritte des Bootstrapping.

- Nun übersetzt man den Compiler ② mit Compiler ③ und erhält den Compiler ④.

Compiler ④ bearbeitet den vollen Sprachumfang der Programmiersprache  $L$ , aber langsam und mit hohem Platzbedarf und erzeugt ebensolche Programme.

- Man übersetzt jetzt Compiler ① mit dem Compiler ④ und erhält einen Compiler ⑤ für die Sprache  $L$ , der schnelle, platzsparende Maschinenprogramme erzeugt, selbst aber noch träge arbeitet.

- Der letzte Schritt des Bootstrapping besteht in der Übersetzung von Compiler ① mit dem Compiler ⑤.

Dadurch erhält man den gewünschten Compiler ⑥ für  $L$ , der schnell und platzsparend arbeitet und ebensolche Programme erzeugt.

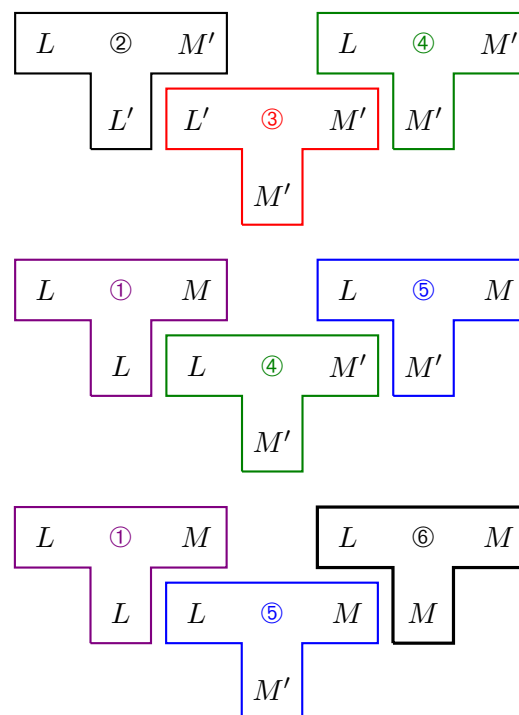


Abb. 2.8: Die automatischen Schritte des Bootstrapping.

T-Diagramme lassen sich gut kombinieren:

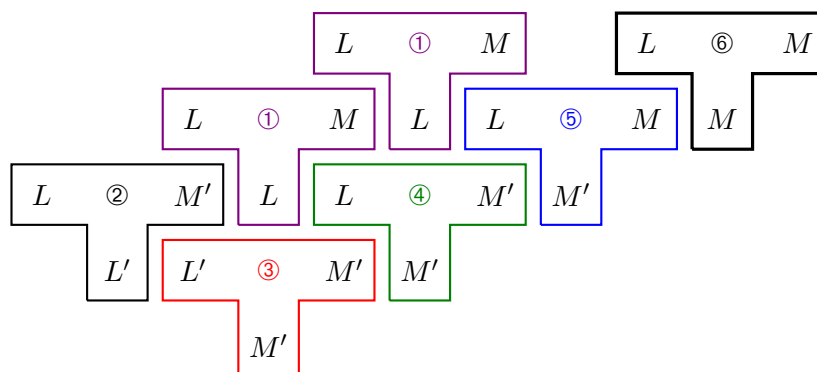
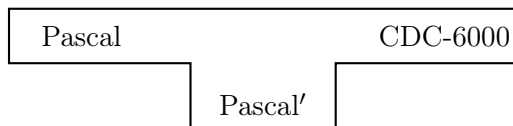
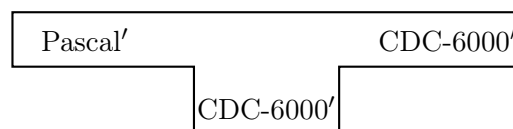


Abb. 2.9: T-Diagramme der letzten drei automatisch ablaufenden Schritte.

**Beispiel 2.7:** NIKLAUS WIRTH, „Erfinder“ von Pascal, versuchte 1968 einen Pascal-Compiler für die CDC-6000 in FORTRAN zu schreiben [33]. Aber FORTRAN erwies sich als zu unhandlich, um u.a. dynamische Datenstrukturen für die Symboltabellen zu realisieren. 1969 wurde der Compiler



Daraus wurde manuell der Compiler



geschrieben, ein ca. 4000 Zeilen langes Pascal-Programm.

gewonnen (ca. 3000 Zeilen). Pascal' ist dabei eine Teilsprache von Pascal, die auf alle Sprachkonstrukte verzichtet, die ein Compiler nicht unbedingt benötigt, z.B. auf Funktionen, auf Parameter von Prozeduren, auf Datentypen wie **real**, **set**, **packed record**, **packed array**. Bzgl. der

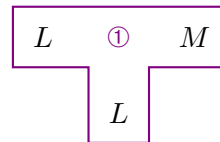


Zielmaschine verzichtete man z.B. auf Operationen zur Gleitpunktarithmetik und zur bitweisen Manipulation von Wörtern. Damit konnte dann das Bootstrapping durchgeführt werden.

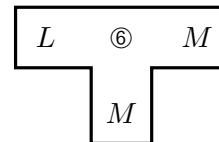
### 2.4.2 Portierung

Ein weiteres Problem ist es, einen Compiler für  $L$ , der in  $L$  geschrieben ist und auf der Maschine  $M$  läuft, auf eine andere Hardware  $N$  zu portieren. Dafür bietet sich z.B. folgende Lösung an:

Die Compiler

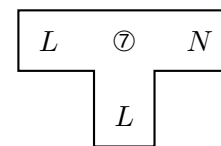


und



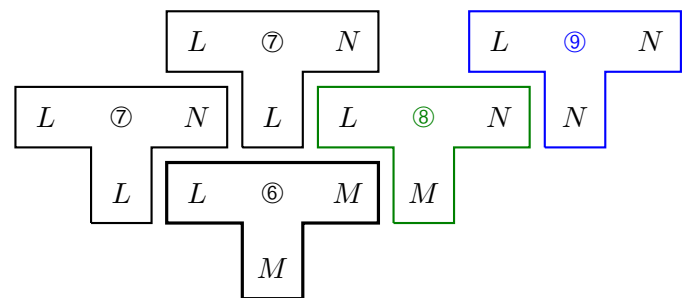
seien gegeben.

Man modifiziert Compiler ① zu Compiler ⑦. Dazu sind nur die „unteren“ Module des Compilers (s. Abb. Seite 16) wie Code-Optimierung und Maschinencode-Erzeugung wesentlich zu ändern.

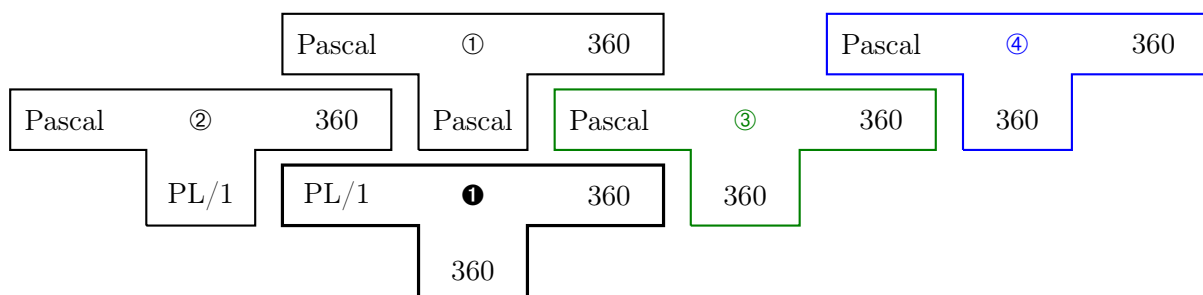


Dann verfährt man nach nebenstehendem Diagramm.

Man beachte dabei, dass der Compiler ⑧ auf der Maschine  $M$  läuft, aber Programme für die Maschine  $N$  erzeugt. Ein solcher Compiler heißt **Cross-Compiler**.



**Beispiel 2.8:** Zur Portierung eines Pascal-Compilers von der CDC-6000 auf die IBM /360 Rechnerfamilie wurde der Compiler ① geschrieben, manuell in den Compiler ② übersetzt und mit dem vorhandenen PL/1 Compiler der IBM /360 in Compiler ③ übersetzt. Damit konnte dann der Compiler ① zu dem angestrebten Compiler ④ verarbeitet werden [24]:



## 2.5 Reale und virtuelle Maschinen

Dem Programmierer steht meist eine **reale Maschine** zur Verfügung, d.h. ein Rechner als Hardware mit Prozessor und Speicherchips. Die Zielsprache der Übersetzung ist dann durch den Typ des Prozessors bestimmt: für diesen erzeugt man **nativen Code**.

Reale Rechner passen immer noch gut zu imperativen Programmiersprachen, d.h. viele Operationen und Strukturen realer Rechner (z.B. indirekte Adressierung, unbedingte Sprünge, linearer Speicher, indizierter Zugriff) finden sich in Konzepten imperativer Programmiersprachen wieder (z.B. Zeiger, Sprünge, Felder, Zugriff auf Feldkomponenten) und die anderen Konzepte imperativer Sprachen sind relativ leicht übersetzbar in Strukturen und Befehlsfolgen realer Rechner.

Will man für eine höhere Programmiersprache Code erzeugen, so würde man oft gerne Befehle verwenden, die von der realen Maschine so nicht bereitgestellt werden. Außerdem ändern sich die Befehlssätze moderner Rechner so schnell, dass man den Compiler nicht auf zufällig bereitgestellte Operationen festlegen will. Diese Festlegung kann nämlich bedeuten, dass man den Compiler für die nächste Rechnergeneration neu schreiben muss.

Bereits bei der Implementierung des ersten **Pascal**-Übersetzers entstand die Idee, zuerst Code für eine leicht idealisierte **virtuelle Maschine** zu erzeugen, deren Befehle dann nur noch auf den verschiedenen realen Zielrechnern zu implementieren waren. Auch die Übersetzung moderner Programmiersprachen wie **Prolog**, **Haskell** oder **Java** basiert auf diesem Prinzip. Dieses Vorgehen erleichtert die **Portierbarkeit** des Übersetzers. Aber es vereinfacht auch die Übersetzung selbst, da man den Befehlssatz passend zur zu übersetzenden Programmiersprache wählen kann.

Auch Internet-Anwendungen machen virtuelle Maschinen attraktiv. Durch die Portierbarkeit, die man durch eine Implementierung einer Programmiersprache auf einer virtuellen Maschine gewinnt, kann man Systeme unter verschiedenen Betriebssystemen –**plattformunabhängig**– lauffähig machen. Dann kann der Code auch über das Internet verbreitet, also **mobil** gemacht werden. Das ist eine der Ideen hinter der Programmiersprache **Java**.

Mit der Ausführung fremden Codes auf dem eigenen Rechner setzt man sich aber Angriffen böswilliger Angreifer aus. Auch hier bieten virtuelle Maschinen eine Lösung: da der Code nicht unmittelbar auf der eigenen Hardware ausgeführt wird, kann man das Verhalten des auszuführenden Codes beobachten und seine Zugriffsrechte auf Ressourcen des Rechners einschränken: **Sandkasten-Prinzip** (*sand boxing*).

### 2.5.1 Sprachprozessoren kombiniert: Interpretierte oder zweistufige Übersetzung

Die Kombination von Übersetzung und Interpretation ist möglich: Oft werden die in die Sprache einer virtuellen Maschine übersetzten Quellprogramme interpretiert.

Allerdings kann man die Sprache der virtuellen Maschine auch weiter (in die Sprache eines realen Rechners) übersetzen. Unterschiede ergeben sich, zu welcher Zeit und auf welcher Maschine dieser zweite Übersetzungsschritt erfolgt.

Die zweite Übersetzung kann auch zur Ausführungszeit erfolgen (**just-in-time Übersetzung, JIT**). Ziel der JIT-Übersetzung ist die Kombination von Effizienz und Portierbarkeit.

Meist werden die beiden Übersetzungsschritte auf verschiedenen Rechnern ausgeführt:

- Der erste Schritt passiert auf dem Rechner des Programmierers. Hier wird ein portabler Zwischencode in der Sprache einer virtuellen Maschine erzeugt. Dieser oft sehr aufwändige Übersetzungsschritt wird nur einmal gemacht.

- Der Zwischencode wird i.d.R. auf einen anderen Rechner heruntergeladen. Der zweite, einfachere Übersetzungsschritt wird auf diesem Rechner durchgeführt und erzeugt einen möglichst effizienten nativen Code für *diesen* Rechner.  
Es werden also Übersetzungszeit und Ausführungszeit kombiniert; der JIT-Übersetzer wird zur Ausführungszeit für ein Stück des Programms –z.B. eine Funktion– aufgerufen, meist zu dem Zeitpunkt, wenn diese Funktion gebraucht wird. Der native Code wird in einem lokalen Speicher (*cache*) gesammelt, so dass jede Funktion nur einmal übersetzt wird.

### 2.5.2 Architektur virtueller Maschinen

Eine virtuelle Maschine stellt **Befehle** (*instruction*) für eine virtuelle Hardware zur Verfügung. Diese virtuelle Hardware wird meist in Software **emuliert**. Ein **Laufzeitsystem** verwaltet dann Datenstrukturen für den Ausführungszustand, der durch die Befehle verändert wird.

Die Architektur von virtuellen Maschinen ist oft ähnlich. Wir beschreiben daher zunächst die Gemeinsamkeiten: die Speichersegmente, die Register und den Hauptausführungszyklus. Weitere Speichersegmente und Register werden bei Bedarf eingeführt.

Die virtuelle Maschinen enthalten zwei Datenspeicher und einen Programmspeicher (Abb. 2.10):

- einen pulsierenden **Keller** (*stack*) S für deklarierte Daten und Zwischenergebnisse.  
Der **Kellerzeiger**, ein Register **SP** (*stack pointer*), zeigt stets auf die oberste belegte Zelle des Kellers. Der **Rahmenzeiger** (*frame pointer*), ein Register **FP**, zeigt auf den Kellerrahmen, in dem die lokalen Variablen des aktuellen Funktionsaufrufs liegen.
- eine **Halde** (*heap*) H für dynamische Daten. Der **Haldenzeiger** (*heap pointer*), ein Register **HP**, zeigt stets auf die unterste belegte Zelle der Halde.  
Jede Zelle von S oder H sollte ein Datenelement (z.B. Zahl, Adresse) aufnehmen können.
- einen **Programmspeicher** C für das auszuführende Programm.  
Der **Befehlszähler**, ein Register **PC** (*program counter, instruction pointer*), zeigt auf den nächsten auszuführenden Befehl im Programmspeicher. Dieser wird in ein **Befehlsregister** **IR** (*instruction register*) geladen und anschließend ausgeführt. Jeder Befehl der virtuellen Maschine belegt genau eine Zelle im Programmspeicher, so dass der Inhalt des Registers PC nach jedem Befehl um 1 erhöht wird. Nur bei einem Sprung wird der Inhalt des Befehlszählers PC mit der Zieladresse überschrieben.  
Das Register PC hat zu Beginn den Wert 0.  
Die virtuelle Maschine hält an, wenn sie den Befehl **halt** ausführt.  
Der **Hauptausführungszyklus** (*main cycle*) der virtuellen Maschinen lautet daher:

```
while (true) {
    IR ← C[PC];
    PC++;
    execute(IR);
}
```

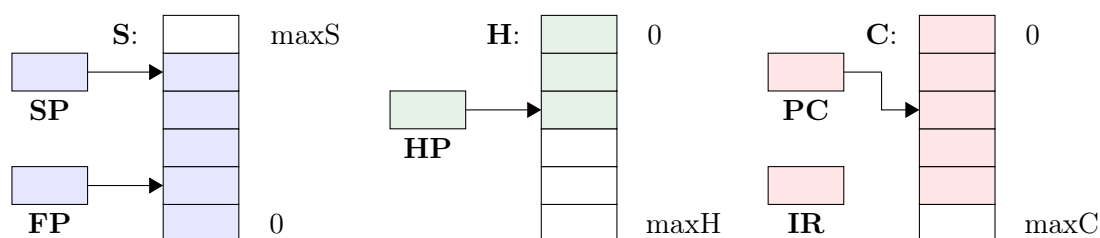


Abb. 2.10: Datenspeicher (Keller und Halde) und Programmspeicher der virtuellen Maschinen.

Aus praktischen Gründen zählen wir Adressen beim Keller von unten nach oben, so dass der Keller –wie gewohnt– nach oben wächst, und sonst von oben nach unten.

### 3 Zwischencode-Erzeugung: Funktionale Abstraktion und imperative Programmiersprachen

Die beiden folgenden Kapitel sollen eine Vorstellung vermitteln, *was* ein Übersetzer tut, ohne schon zu erklären, *wie* er das tut. Wir definieren dazu präzise, aber intuitiv die Korrespondenz zwischen Programmen einer imperativen Quellsprache und den übersetzten Zielprogrammen.

Als Quellsprache wählen wir einen Ausschnitt der Programmiersprache C. Allerdings haben wir die konkrete Syntax geringfügig angepasst: Eine Wertzuweisung  $x = e$  schreiben wir  $x \leftarrow e$  und einen Vergleich  $e_1 == e_2$  schreiben wir  $e_1 = e_2$ . Eine Bedingung klammern wir mit Schlüsselwörtern statt mit runden Klammern, z.B. schreiben wir **if**  $e$  **then**  $s$  statt **if** ( $e$ )  $s$ .

Als Zielsprache der Übersetzung wählen wir die Maschinsprache einer virtuellen Maschine, die wir parallel zu unseren Überlegungen zur Übersetzung entwerfen und **C-Maschine** nennen. Die bei der Übersetzung zu erzeugenden Befehlsfolgen werden für jedes behandelte C-Konstrukt angegeben mit Hilfe von Übersetzungsfunktionen `code` und einer Speicherbelegungsfunktion  $\rho$ .

Wir kümmern uns noch nicht um das Problem, wie ein C-Programm syntaktisch analysiert wird. Außerdem sei die Typkorrektheit des Eingabeprogramms schon geprüft.

#### 3.1 Konzepte imperativer Programmiersprachen

Wir müssen u.a. folgende Konzepte imperativer Programmiersprachen in Maschinen abbilden:

- Der **Zustand** (*state*) eines Programms kann durch eine **Anweisung** (*statement*) verändert werden, z.B. durch eine Wertzuweisung (*assignment*).

Ein **Wert** (*value*) kann in einer (Programm-) **Variablen**<sup>1</sup> (*variable*) gespeichert werden, die als Behälter für Datenobjekte dient. Der Wert kann sich während der Programmausführung ändern und damit der Zustand des Programms.

Variable, Konstante und Funktionen werden im Programm durch **Namen** (*identifier*) bezeichnet. Falls nötig, sprechen wir genauer von Variablenbezeichnungen etc.

Einer Variablenbezeichnung wird die **Adresse** einer Speicherzelle der Maschine zugeordnet, diese Speicherzelle (und ggf. die folgenden) enthalten den aktuellen Wert der Variablen.

- Ein **Ausdruck** (*expression*) ist ein Term aus Konstanten, Namen und Operatoren, der bei der Programmausführung **ausgewertet** wird. Der Wert eines Ausdrucks hängt i.d.R. vom Zustand ab, da bei jeder Auswertung die aktuellen Werte der im Ausdruck enthaltenen Variablen (und Funktionsaufrufe) benutzt werden.
- **Anweisungen** steuern den **Programmablauf** explizit:
  - Der **Sprung** (`goto`) –sofern er noch zugelassen wird– kann in einen unbedingten Sprungbefehl der Zielmaschine übersetzt werden.

Eine **Alternative** (`if`) oder eine **Schleife** (`while`, `do-while`, `repeat`, `for`) wird mit Hilfe bedingter Sprünge übersetzt: Auf die Befehlsfolge für die Bedingung folgt ein bedingter Sprung.

Eine allgemeine **Fallunterscheidung** (`case`, `switch`) lässt sich oft effizienter durch indizierte Sprünge realisieren. Dazu wird die im Befehl angegebene Sprungadresse mit einem vorher berechneten Wert verändert.

---

<sup>1</sup> Der mathematische Begriff einer Variable ist ein anderer!

- Die Definition einer **Funktion** macht aus einer Folge von Anweisungen eine neue Anweisung (**funktionale Abstraktion**). Ein **Aufruf** (*call*) dieser Funktion erzeugt für die in der Definition angegebene Folge von Anweisungen eine **Inkarnation** dieser Funktion (*function activation*). Nach ihrer Abarbeitung wird mit dem Funktions-**Ergebnis** (*result*) an die Aufrufstelle zurückgekehrt. Ein Unterprogramm-Sprung merkt sich daher seine Herkunft.

Durch **Parameter** können *unterschiedliche* Inkarnationen der Funktion aufgerufen werden. Der Rumpf der Funktion wird bei jedem Aufruf mit **aktuellen Parametern** versorgt.

Der Rumpf einer (direkt) **rekursiven Funktion** enthält das aktuell zu definierende Funktionssymbol. Streng genommen ist diese Gleichung keine Definition, sondern eine einschränkende Bedingung. Diese Gleichung hat evtl. keine Lösung (wenn z.B. die Ausführung nicht terminiert) oder mehrere Lösungen (und eine wird ausgewählt: „kleinster Fixpunkt“).

Bei jedem Aufruf einer Funktion benötigt die Inkarnation Speicherplatz für ihre lokalen Variablen. Nach Verlassen der Funktion wird dieser Speicherplatz nicht mehr gebraucht. Der Speicher wird daher wie ein **Keller** organisiert.

## 3.2 Speicherbelegung im Keller

Jede Variable des Programms erhält zur *Übersetzungszeit* eine **Adresse** im Datenspeicher S, wo ihr Wert zur Laufzeit gespeichert wird. Der Übersetzer nutzt diese Adressen, um Code zu erzeugen, der die Werte der Variablen von dort lädt oder dort speichert (Abb. 3.2).

Diese Information wird den Übersetzungsfunktionen zur Verfügung gestellt als Funktion  $\rho$ , die für jede Variable  $x$  die (relative) Adresse von  $x$  liefert. Die Funktion  $\rho$  heißt **Speicherbelegungsfunktion** oder **Adressumgebung** (*address environment*).

Dazu müssen wir für jeden Datentyp  $t$  seine **Größe**  $|t|$  festlegen, nämlich die Anzahl der benötigten Speicherzellen für einen Wert dieses Typs.

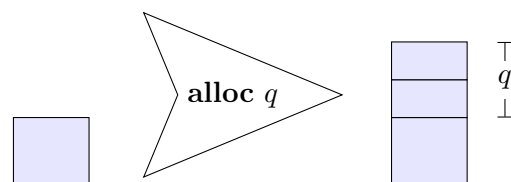
Die einfachen Typen **int**, **float** und **char** sowie Zeigertypen erhalten die Größe 1, d.h. jeder Variable dieser Typen ordnen wir *eine* Speicherzelle zu. Wir versuchen nicht –wie es in realen Übersetzern geschieht– mehrere „kleine Werte“ in *ein* Wort zu packen. Die Wortlänge ist nicht festgelegt; sie muss nur groß genug sein, um **int**-Werte und Adressen aufzunehmen.

Dann ergibt sich ein einfaches Schema zur Speicherbelegung: Den Variablen im Deklarationsteil  $t_1 x_1; \dots; t_k x_k$ ; des C-Programms ordnen wir in der Reihenfolge ihres Vorkommens lückenlos Adressen am Anfang des Kellerspeichers zu. Die erste zugeteilte Adresse ist 1.<sup>2</sup>

Die Speicherbelegungsfunktion  $\rho$  wird daher definiert durch:

$$\rho(x_1) = 1 \quad \text{und} \quad \rho(x_i) = \rho(x_{i-1}) + |t_{i-1}|, \text{ für } i > 1.$$

Um  $q$  Speicherplätze im Keller zur *Laufzeit* zu reservieren, führen wir die Befehle **alloc**  $q$  ein (Abb. 3.1, *allocate*). Der Befehl **alloc** 0 ist ohne Wirkung und kann vom Optimierer entfernt werden.



Für die Freigabe von Kellerspeicher nutzen wir die **alloc**-Befehle mit negativem Argument.

$$SP \leftarrow SP + q;$$

Abb. 3.1: Der Befehl **alloc**  $q$ .

Bevor die Auswertung von Ausdrücken beginnt, wird mit *einem* **alloc**-Befehl der Speicherplatz für *alle* deklarierten Variablen (des Hauptprogramms oder einer Funktionsinkarnation) reserviert.

<sup>2</sup> Die Adresse 0 reservieren wir für die Rücksprungradresse.

Die Adresse einer Variablen ist in Wirklichkeit eine **Relativadresse**, d.h. eine konstante Differenz zwischen zwei absoluten Adressen in S, nämlich der tatsächlichen Adresse der Zelle für diese Variable und der Anfangsadresse des Speicherbereichs der Funktion, in der die Variable deklariert ist. Die absolute Adresse 1 fassen wir also als Adresse 1 relativ zur Basis 0 auf.

Die so zugeteilten Relativadressen sind *statische* Größen, da sie sich aus dem Quellprogramm ergeben, genauer aus der Position der Variablen im Deklarationsteil.

Die Adressen liegen im Keller der C-Maschine. Wenn wir Funktionen behandeln, wird klar, dass es zwei ineinander geschachtelte Keller gibt:

- einen „großen“ Keller aus den Datenbereichen aller zu einem Zeitpunkt aktiven Funktionen, der wächst bzw. schrumpft, wenn eine Funktion betreten bzw. verlassen wird, und
- einen „kleinen“ Keller (Abb. 3.2) zu jeder aktiven Funktion für ihre Variablen und Zwischenergebnisse.

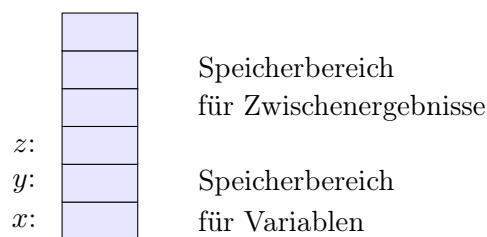


Abb. 3.2: Der „kleine“ Keller.

### 3.3 Ausdrücke

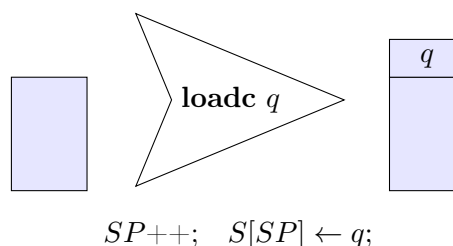
Wir übersetzen zunächst arithmetische und logische Ausdrücke. Der Übersetzer muss für jeden Ausdruck eine Befehlsfolge für die virtuelle Maschine erzeugen, die bei Ausführung den Wert des Ausdrucks liefert.

#### 3.3.1 Arithmetische und logische Ausdrücke

Übersetzen wir den Ausdruck  $(1 + 7) \cdot (5 - 2)$ . Wie sieht eine Befehlsfolge aus, die diesen Ausdruck auswertet und sein Ergebnis oben auf dem Keller hinterlässt?

Besteht der Ausdruck nur aus einer Konstanten, z.B. 7, benötigen wir nur einen Befehl, der diesen Wert oben auf den Keller schreibt.

Der Befehl **loadc** *q* (*load constant*, Abb. 3.3) erwartet keine Operanden auf dem Keller. Er legt die Konstante *q* oben auf dem Keller ab.



$SP++;$   $S[SP] \leftarrow q;$

Abb. 3.3: Der Befehl **loadc** *q*.

Sonst berechnen wir zuerst die Werte der Teilausdrücke,  $1 + 7$  und  $5 - 2$ , und hinterlassen ihre Werte in den oberen Kellerzellen, um dann den äußeren Operator „ $\cdot$ “ auf diese Werte anzuwenden. Die Anwendung des Operators löscht dann die Zwischenergebnisse 8 und 3 auf dem Keller und hinterlässt das Ergebnis des gesamten Ausdrucks 24 wieder oben auf dem Keller.

Wir treffen daher folgende Entwurfsentscheidung zur Berechnung von Ausdrücken:

Ein Befehl erwartet seine Operanden oben auf dem Keller<sup>3</sup> und ersetzt sie durch das Ergebnis.

Ein rekursives Vorgehen gemäß der Struktur eines Programmfragments (hier: von Ausdrücken) finden wir oft vor. Dieses Vorgehen passt gut zu unserer Entwurfsentscheidung.

<sup>3</sup> Der Begriff Keller meint die Art, wie der Speicher wächst oder schrumpft und wo die Operationen ihre Operanden finden. Zusätzlich benötigen wir Befehle, die die Operanden erst an diese Stelle bringen.

Der binäre Multiplikations-Befehl **mul** erwartet zwei Argumente oben auf dem Keller, multipliziert sie, löscht die Operanden und schreibt das Ergebnis oben auf den Keller:  $8 \cdot 3$  (Abb. 3.4).

Die übrigen binären Befehle arbeiten analog: die arithmetischen Befehle **add**, **sub**, **div**, **mod** die Vergleiche **eq**, **neq**, **le**, **leq**, **gr** und **geq** sowie die logischen Befehle **and** und **or**.

Das Ergebnis einer Vergleichsoperation ist ein logischer Wert: **true** oder **false**. In Abb. 3.5 vergleicht der **leq**-Befehl zwei Zahlen:  $3 \leq 7$ .

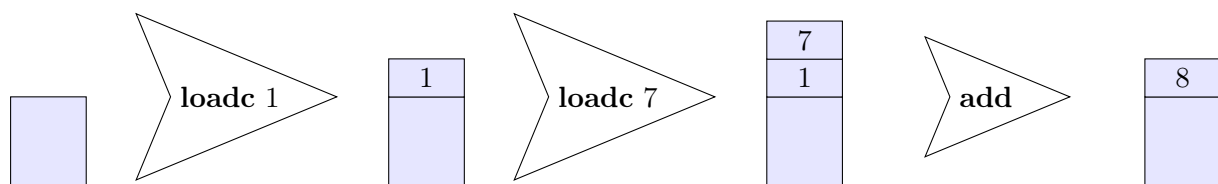
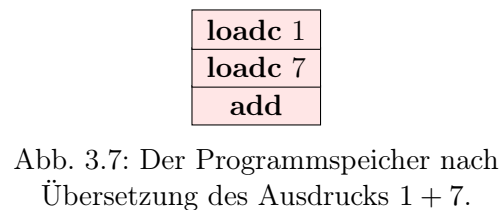
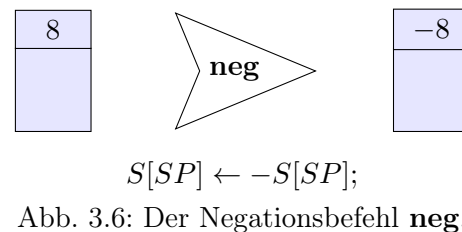
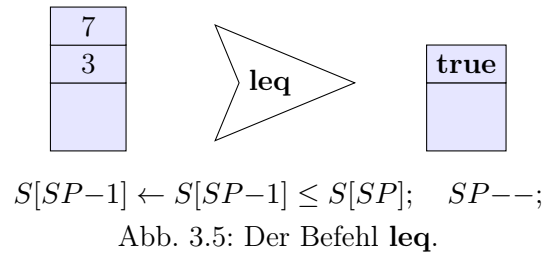
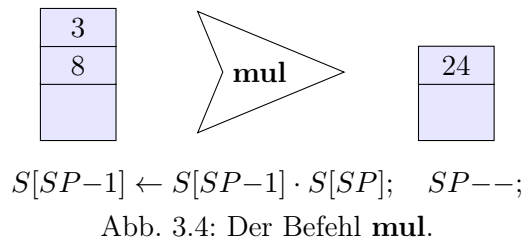
In der Programmiersprache C werden Wahrheitswerte durch ganze Zahlen dargestellt: 0 steht für **false**, alle anderen Werte für **true**.

Unäre Befehle wie **neg** und **not** konsumieren nur einen Operanden. Da sie ebenfalls einen Wert als Ergebnis liefern, ersetzen sie den Wert in der obersten Kellerzelle. Der **neg**-Befehl dreht das Vorzeichen einer Zahl um (Abb. 3.6).

Der Ausdruck  $1 + 7$  wird in die Befehle

**loadc 1**, **loadc 7**, **add**

der virtuellen Maschine übersetzt (Abb. 3.7), die zur Laufzeit abgearbeitet werden (Abb. 3.8).



Welche Befehlsfolgen für eine Anweisung oder einen Ausdruck erzeugt werden, wird durch **Übersetzungsfunktionen** festgelegt. Diese Funktionen bekommen als Argumente ein Programmfragment und eine Adressumgebung  $\rho$ . Sie zerlegen das Programmfragment rekursiv und setzen die für die Komponenten erzeugten Befehlsfolgen zu *einer* Befehlsfolge zusammen.

$\text{code}_W^\rho(e_1 + e_2) = \text{code}_W^\rho e_1, \text{code}_W^\rho e_2, \text{add}$	// andere binäre Operatoren analog
$\text{code}_W^\rho(-e) = \text{code}_W^\rho e, \text{neg}$	// andere unäre Operatoren analog
$\text{code}_W^\rho q = \text{loadc } q$	// Konstante

### 3.3.2 Wertzuweisung

In imperativen Sprachen werden Variablen auf zwei Arten verwendet. Betrachten wir z.B. die Wertzuweisung  $x \leftarrow y + 1$ . Von der Variablen  $y$  benötigen wir offenbar ihren **Wert**, den **Inhalt** ihrer Speicherzelle, um den Wert des Ausdrucks  $y + 1$  zu ermitteln. Von der Variablen  $x$  benötigen wir aber die **Adresse** ihrer Speicherzelle, um den neu berechneten Wert dort ablegen zu können.

Eine Variable auf der *linken* Seite einer Wertzuweisung muss daher anders übersetzt werden als eine auf der *rechten* Seite. Wir indizieren die Übersetzungsfunktionen deshalb mit A bzw. W:

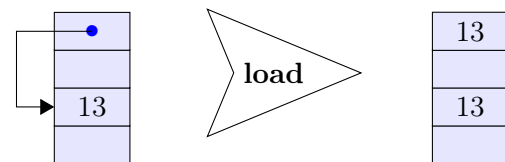
- $\text{code}_A$  erzeugt Befehle zur Berechnung der Adresse einer Variablen oder eines Ausdrucks;
- $\text{code}_W$  erzeugt Befehle zur Berechnung des Wertes eines Ausdrucks;
- $\text{code}$  (ohne Index) übersetzt Anweisungen.

Variablen einer Wertzuweisung werden daher folgendermaßen übersetzt:

$\text{code}_A^\rho x = \text{loadc } \rho(x)$	// linke Seite
$\text{code}_W^\rho x = \text{code}_A^\rho x, \text{load}$	// rechte Seite

Um den Wert einer Variablen  $x$  zu ermitteln, benötigen wir einen Befehl **load** (Abb. 3.9), der den Inhalt einer Speicherzelle, deren Adresse oben auf dem Keller liegt, auf den Keller lädt.

Jeder Ausdruck hat einen Wert, aber nicht immer eine Adresse. Der Wert des Ausdrucks  $y + 1$  ist z.B. nur vorübergehend auf dem Keller vorhanden und deshalb nicht adressierbar.

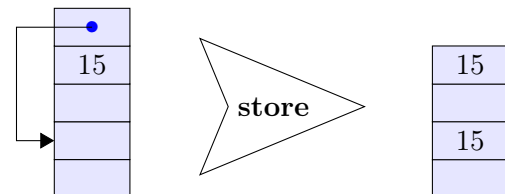


$S[SP] \leftarrow S[S[SP]];$

Abb. 3.9: Der Befehl **load**.

In der Programmiersprache C ist eine **Wertzuweisung** ein **Ausdruck**! Der Wert dieses Ausdrucks ist der Wert der rechten Seite der Zuweisung. Der Wert der Variablen auf der linken Seite der Zuweisung ändert sich als **Nebenwirkung** der Auswertung des Ausdrucks.

Zur Implementierung der Wertzuweisung benötigen wir einen Befehl **store** (Abb. 3.10). Der Befehl **store** erwartet zwei Argumente auf dem Keller: einen Wert  $w$  und darüber eine Adresse  $a$ . Er schreibt den Wert  $w$  an der Adresse  $a$  in den Speicher und lässt ihn als Ergebnis oben auf dem Keller zurück.



$S[S[SP]] \leftarrow S[SP-1]; \quad SP--;$

Abb. 3.10: Der Befehl **store**.

Eine Wertzuweisung  $x \leftarrow e$  übersetzen wir:

$\text{code}_W^\rho (x \leftarrow e) = \text{code}_W^\rho e, \text{code}_A^\rho x, \text{store}$

In einer Adressumgebung  $\rho = \{x \mapsto 5, y \mapsto 7\}$  berechnet die folgende Befehlsfolge den Wert der Wertzuweisung  $x \leftarrow y + 1$ :

**loadc 7, load, loadc 1, add, loadc 5, store**

Die Befehlsfolge ermittelt zuerst den Wert der rechten Seite, dann die Adresse der linken Seite. Die Wertzuweisung wird schließlich vom Befehl **store** ausgeführt (Abb. 3.11).

<b>S:</b>		9	<b>C:</b>	<b>loadc 7</b>	0
		8		<b>load</b>	1
$y:$	23	7		<b>loadc 1</b>	2
		6		<b>add</b>	3
$x:$		5		<b>loadc 5</b>	4
		4		<b>store</b>	5

Abb. 3.11: Übersetzung der Wertzuweisung  $x \leftarrow y + 1$ .



**Beispiel 3.1:** Eine Wertzuweisung  $a \leftarrow (b + (b \cdot c))$  mit Adressumgebung  $\rho = \{a \mapsto 5, b \mapsto 6, c \mapsto 7\}$  wird übersetzt zu:

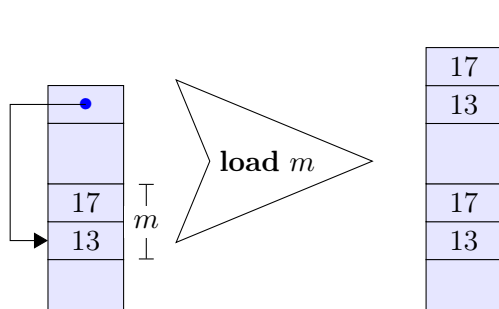
$$\begin{aligned} & \text{code}_W^\rho (a \leftarrow (b + (b \cdot c))) \\ &= \text{code}_W^\rho (b + (b \cdot c)), \text{code}_A^\rho a, \text{store} \\ &= \text{code}_W^\rho b, \text{code}_W^\rho (b \cdot c), \text{add}, \text{code}_A^\rho a, \text{store} \\ &= \text{loadc } 6, \text{load}, \text{code}_W^\rho (b \cdot c), \text{add}, \text{code}_A^\rho a, \text{store} \\ &= \text{loadc } 6, \text{load}, \text{code}_W^\rho b, \text{code}_W^\rho c, \text{mul}, \text{add}, \text{code}_A^\rho a, \text{store} \\ &= \text{loadc } 6, \text{load}, \text{loadc } 6, \text{load}, \text{code}_W^\rho c, \text{mul}, \text{add}, \text{code}_A^\rho a, \text{store} \\ &= \underbrace{\text{loadc } 6, \text{load}}_{\text{loada } 6}, \underbrace{\text{loadc } 6, \text{load}}_{\text{loada } 6}, \underbrace{\text{loadc } 7, \text{load}}_{\text{loada } 7}, \text{mul}, \text{add}, \underbrace{\text{loadc } 5, \text{store}}_{\text{storea } 5} \end{aligned}$$

Routineaufgaben wiederholen sich und führen zu ähnlichen Befehlsfolgen. Oft wird ein Wert von einer konstanten –zur Übersetzungszeit bekannten– Adresse geladen bzw. an diese Adresse geschrieben. Als Optimierung führen wir dafür **Spezialbefehle** ein:

**loada**  $q \equiv \text{loadc } q, \text{load}$  (load from address)  
**storea**  $q \equiv \text{loadc } q, \text{store}$  (store at address)

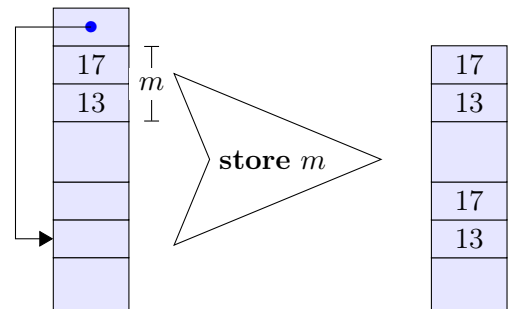
Spezialbefehle erhöhen die Effizienz: Der erzeugte Code wird kürzer und die Implementierung oft effizienter, z.B. kann **storea**  $q$  statt durch  $\underbrace{SP++; S[SP] \leftarrow q; S[S[SP]] \leftarrow S[SP-1]; SP--}_{\text{loadc } q}$ ;  $\underbrace{S[q] \leftarrow S[SP]}_{\text{store}}$  implementiert werden.

Um Daten bewegen zu können, die größer sind als eine Speicherzelle, verallgemeinern wir die Befehle **load** und **store** zu Befehlen **load**  $m$  (Abb. 3.12) bzw. **store**  $m$  (Abb. 3.13) für nicht-negative Werte  $m$ . Der Befehl **load**  $m$  legt den Inhalt von  $m$  aufeinanderfolgenden Zellen auf den Keller – ab der Adresse, die oben auf dem Keller liegt. Der Befehl **load** 1 entspricht dem Befehl **load**. Die *absteigende* Folge für  $i$  ist wichtig, damit  $S[SP]$  nicht zu früh überschrieben wird!



**for**  $i \leftarrow m-1; i \geq 0; i--$  **do**  
 $S[SP+i] \leftarrow S[SP-m+i];$   
 $SP \leftarrow SP-m;$

Abb. 3.12: Der Befehl **load**  $m$ .



**for**  $i \leftarrow 0; i < m; i++$  **do**  
 $S[SP+i] \leftarrow S[SP-m+i];$   
 $SP--;$

Abb. 3.13: Der Befehl **store**  $m$ .

Natürlich können wir wieder Spezialbefehle einführen und abkürzen:

**loada**  $q \ m \equiv \text{loadc } q, \text{load } m$   
**storea**  $q \ m \equiv \text{loadc } q, \text{store } m$

### 3.4 Anweisungen

#### 3.4.1 Anweisung und Anweisungsfolgen

In der Programmiersprache C gilt: Ist  $e$  ein Ausdruck, dann ist  $e$ ; eine **Anweisung** (*statement*). Eine Anweisung liefert keinen Wert zurück. Daher muss der Kellerzeiger SP vor und nach einer Anweisung den gleichen Wert haben. Deshalb entfernen wir das Ergebnis des Ausdrucks  $e$ . Ist  $|e|$  die Größe (des Typs) von  $e$ , so erzeugen wir:

$$\text{code}^\rho(e;) = \text{code}_W^\rho e, \text{alloc} - |e|$$

Für eine **Folge von Anweisungen** konkatenieren wir die Codesequenzen für die Anweisungen:

$$\begin{aligned} \text{code}^\rho(s\ ss) &= \text{code}^\rho s, \text{code}^\rho ss && //\ s\ \text{Anweisung, } ss\ \text{Folge von Anweisungen} \\ \text{code}^\rho(\varepsilon) &= \varepsilon && //\ \varepsilon\ \text{leere Folge von Anweisungen} \end{aligned}$$

#### 3.4.2 Alternative (binäre Fallunterscheidung)

Als nächstes übersetzen wir die **Alternative**, die **binäre Fallunterscheidung**.

Wir geben ein Übersetzungsschema an für die **if**-Anweisung:  $\text{if } e \text{ then } s_1 \text{ else } s_2$  wobei  $e$  ein Ausdruck und jedes  $s_i$  eine Anweisung oder eine zu einem Block zusammengefasste Anweisungsfolge ist.

Um von der linearen Ausführungsreihenfolge abzuweichen, benötigen wir Sprungbefehle.

Ein **unbedingter Sprung** setzt die Ausführung des Programms an einer anderen Stelle fort (Abb. 3.14). Ein **bedingter Sprung** weicht nur dann von der sequentiellen Ausführung ab, wenn eine bestimmte Bedingung zutrifft. Dazu testen wir, ob das oberste Element im Keller 0 (**false**) ist (*jump on zero*, Abb. 3.15).

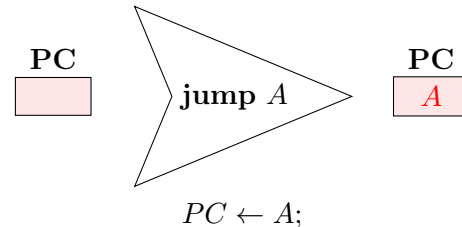
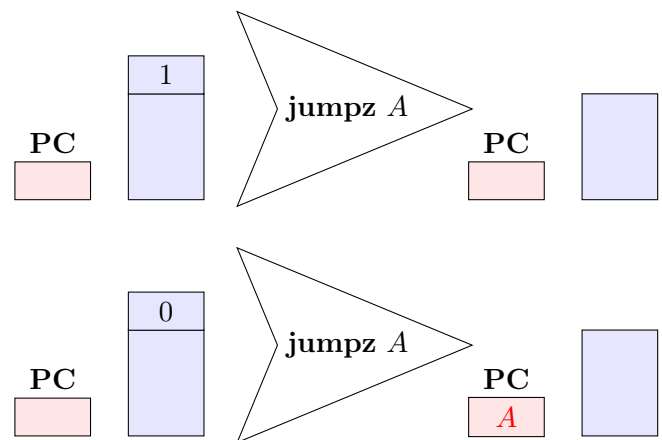


Abb. 3.14: Der Sprungbefehl **jump**.

Für die Übersetzung von bedingten Anweisungen und Schleifen benötigen wir ein neues Sprachmittel. Wir markieren Befehle oder das Ende des Schemas durch symbolische **Marken** (*label*), die wir in Sprungbefehlen als Ziele verwenden. Eine Marke steht für die –zu diesem Zeitpunkt ggf. noch unbekannte– Adresse des Befehls, der diese Marke trägt. Im nächsten Durchlauf können die Marken durch absolute Codeadressen ersetzt werden.

Statt absoluter Codeadressen könnte man auch **relative Adressen** als Sprungziele nutzen, d.h. die Sprungziele in den Sprungbefehlen relativ zum aktuellen PC angeben. Dann würden meist kürzere Adressen ausreichen und der Code wäre leichter **relokierbar**, d.h. er könnte an beliebiger Stelle im Speicher stehen.



$$\text{if } S[SP] = 0 \text{ then } PC \leftarrow A; \quad SP--;$$

Abb. 3.15: Der Sprungbefehl **jumpz**.

Um die Alternative zu übersetzen, legen wir Codesequenzen für  $e$ ,  $s_1$  und  $s_2$  in den Programmspeicher. Dazwischen fügen wir Sprünge ein (Abb. 3.16): Hinter die Bedingung schreiben wir einen bedingten Sprung zum Anfang des **else**-Teils  $s_2$ : Wird die Bedingung  $e$  zur Laufzeit zu **false** (0) ausgewertet, springen wir dorthin. Sonst wird der Code für  $s_1$  ausgeführt. Damit danach nicht auch noch der Code für  $s_2$  ausgeführt wird, fügen wir hinter den Code für  $s_1$  einen unbedingten Sprung hinter die Fallunterscheidung ein.

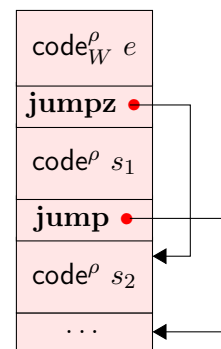


Abb. 3.16: Code für die Alternative.

$\text{code}^\rho(\text{if } e \text{ then } s_1 \text{ else } s_2) = \text{code}_W^\rho e, \text{jumpz } A, \text{code}^\rho s_1, \text{jump } B, A: \text{code}^\rho s_2, B: \dots$

**Beispiel 3.2:** Für die Adressumgebung  $\rho = \{x \mapsto 4, y \mapsto 7\}$  ist

$\text{code}^\rho(\text{if } x > y \text{ then } x \leftarrow x - y; \text{ else } y \leftarrow y - x;) =$

**loada 4, loada 7, gr, jumpz A, loada 4, loada 7, sub, storea 4, alloc -1, jump B,**  
**A: loada 7, loada 4, sub, storea 7, alloc -1, B: ...**

Ein Codeschema kann zur Übersetzung eines Programms mehrfach oder rekursiv angewendet werden; daher werden bei jeder Anwendung eines Codeschemas *neue* Marken verwendet (indem z.B. die Marke um eine laufende Nummer ergänzt wird).

### 3.4.3 Wiederholung (Schleife)

Abb. 3.17 zeigt die Befehlsfolge für eine **while-Schleife**: **while**  $e$  **do**  $s$ . Hinter den Code für die Bedingung  $e$  fügen wir einen bedingten Sprung ein, der die Schleife verlässt. Am Ende des Codes für den Rumpf wird ein unbedingter Sprung zurück an den Anfang der Schleife – vor den Code der Bedingung – eingefügt.

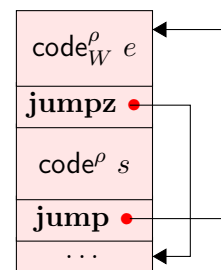


Abb. 3.17: Code für die while-Schleife.

$\text{code}^\rho(\text{while } e \text{ do } s) = A: \text{code}_W^\rho e, \text{jumpz } B, \text{code}^\rho s, \text{jump } A, B: \dots$

**Beispiel 3.3:** Gegeben sei die Anweisung  $s \equiv \text{while } a > 0 \text{ do } \{c \leftarrow c + 1; a \leftarrow a - b;\}$

und die Adressumgebung  $\rho = \{a \mapsto 7, b \mapsto 8, c \mapsto 9\}$ . Dann liefert  $\text{code}^\rho s$  die Folge:

**A: loada 7, loadc 0, gr, jumpz B, loada 9, loadc 1, add, storea 9, alloc -1, loada 7,**  
**loada 8, sub, storea 7, alloc -1, jump A, B: ...**

Im Rumpf einer Schleife sind auch **break**- oder **continue**-Anweisungen erlaubt: Durch ein **break** wird die Schleife durch einen unbedingten Sprung verlassen. Ein **continue** beendet nur den aktuellen Schleifendurchlauf und springt an das Ende des Schleifenrumpfs. Die Übersetzungsfunktion  $\text{code}$  muss auch die Sprungziele für **break** bzw. **continue** erzeugen und verwalten.

## 3.5 Speicherplatzbedarf für Zwischenergebnisse und Kellerüberlauf

Bisher haben wir den Wert des Registers SP unbefangen erhöht, um neue Speicherzellen des Datenspeichers zu nutzen. Um einen **Kellerüberlauf** (*stack overflow*) zu vermeiden, hätten wir

vor jeder Erhöhung –durch Vergleich mit  $\text{maxS}$ – prüfen müssen, ob die maximale Größe des Datenspeichers schon erreicht ist.

Bereits zur Übersetzungszeit lässt sich für eine gegebene Anweisungsfolge die Anzahl der maximal nötigen Speicherplätze für Zwischenergebnisse berechnen.

Wir ermitteln daraus die Adresse der obersten Kellerzelle, auf die SP bei der Auswertung von Ausdrücken im Anweisungsteil des aktuellen Funktionsaufrufs zeigen kann und sparen damit die vielen Vergleiche bei der Vergrößerung des Kellers ein.

Für das Hauptprogramm und jeden Funktionsaufruf prüfen wir dann nur, ob diese Adresse kleiner gleich der oberen Grenze des Kellers ist.

Für spätere Zwecke (Kapitel 4.2) speichern wir diese Adresse im **Schrankenzeiger**, einem Register **EP** (*extreme pointer*) der C-Maschine (Abb. 3.18).

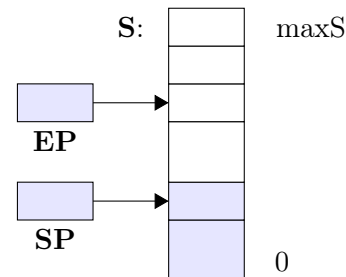


Abb. 3.18: Der Keller der C-Maschine.

### 3.6 Funktionen

Bevor wir Funktionen übersetzen, bereiten wir die zugehörigen Konzepte, Begriffe und Probleme kurz auf. So besteht die **Deklaration** einer Funktion aus:

- einem **Namen**, unter dem die Funktion aufrufbar ist,
- einer **Ein-/Ausgabeschnittstelle**: der Spezifikation (Namen und Typen) der formalen Parameter und der Rückgabewerte,
- einem **Rumpf** (*body*), der aus (lokalen) Deklarationen und Anweisungen besteht.

Viele Sprachen erlauben nur einen Rückgabewert; dann ist kein Name dafür nötig. Liefert die Funktion keinen Rückgabewert, ist also eine **Prozedur**, geben wir als Rückgabebetyp **void** an.

Funktionen werden **aufgerufen**, d.h. aktiviert, wenn ein Vorkommen ihres Namens im Anweisungsteil einer Funktion abgearbeitet wird. Eine aufgerufene Funktion kann weitere Funktionen aufrufen (andere oder sich selbst). Hat eine aufgerufene Funktion ihren Anweisungsteil abgearbeitet, so wird sie **verlassen** und ihr Aufrufer, d.h. die Funktion, die sie aktiviert hat, fährt in der Ausführung hinter dem Aufruf fort.

#### 3.6.1 Inkarnation und Aufrufbaum

Die Funktionsaufrufe, die während der Ausführung eines Programms entstehen, bilden einen geordneten Baum, den **Aufrufbaum** des Programmlaufs. Die Wurzel des Aufrufbaums ist markiert mit dem Namen der Funktion *main*, mit deren Aufruf die Programmausführung startet. Jeder Knoten im Aufrufbaum ist markiert mit einem Funktionsnamen  $f$ , sein direkter Vorgänger mit dem Namen der Funktion, die diesen Aufruf von  $f$  ausgeführt hat; seine direkten Nachfolger bilden die Liste der Funktionen, die  $f$  aufgerufen hat – geordnet in der Reihenfolge ihrer Aufrufe. Eine Markierung  $f$  kann mehrfach im Aufrufbaum auftreten; sie tritt so oft auf, wie  $f$  im Laufe der Programmabarbeitung aufgerufen wird. Jedes Vorkommen von  $f$  im Aufrufbaum heißt **Inkarnation** von  $f$ ; der Weg von der Wurzel des Aufrufbaums bis zu diesem Knoten (**Inkarnationsweg**) charakterisiert diese Inkarnation.

Betrachten wir den Zustand der Programmausführung, wenn eine bestimmte Inkarnation von

$f$  aktiv ist. Alle Vorfahren dieser Inkarnation, also alle Knoten auf dem Inkarnationsweg, sind bereits aufgerufen, aber noch nicht verlassen worden; sie sind zu diesem Zeitpunkt **lebendig**.

**Beispiel 3.4:** Betrachten wir das folgende C-Programm und seinen Aufrufbaum. Der zweite rekursive Aufruf der Funktion  $fac$  ist rot dargestellt. Zu diesem Zeitpunkt sind die Funktion  $main$  und zwei Inkarnationen von  $fac$  lebendig.

```
int fac (int n) {
    if n ≤ 0 then return 1;
    else return n · fac(n-1);
}

int main() {
    int n;
    n ← 2;
    return fac(n) + fac(n-1);
}
```

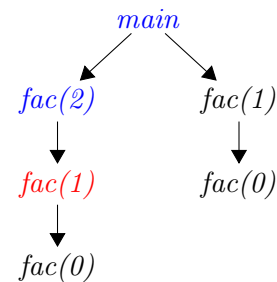


Abb. 3.19: Der Aufrufbaum.

Abhängig vom Wert der globalen Variablen und der aktuellen Parameter gibt es zu einem Programm mehr als einen Aufrufbaum oder sogar unendlich große Aufrufbäume.

### 3.6.2 Auswertungsstrategien und Parameterübergabe

Die Auswertung eines Ausdrucks  $e$  geschieht durch **Termersetzung**, indem folgende zwei Schritte wiederholt werden, bis kein Ersetzungsschritt mehr möglich ist:

1. Man sucht einen Teilausdruck (**Redex**, *reducible expression*) in  $e$ , der mit der linken Seite einer Funktionsdefinition übereinstimmt, wobei Variablen der linken Seite durch geeignete Ausdrücke ersetzt werden.
2. Der Redex wird durch die rechte Seite der Funktionsdefinition ersetzt, wobei die Variablen auf der rechten Seite durch die gleichen Ausdrücke ersetzt werden.

Eine **Auswertungsstrategie** ist ein Algorithmus zur Auswahl des nächsten Redex.

Wir betrachten Auswertungsreihenfolgen am Beispiel einer Funktion  $\text{square}(x) = x \cdot x$  und dem Ausdruck  $\text{square}(15/3)$ .

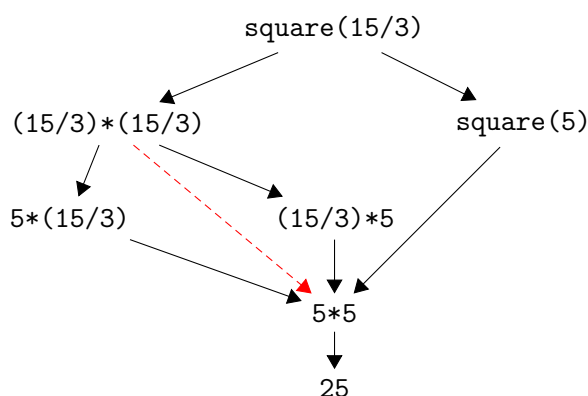


Abb. 3.20: Auswertungsstrategien.

Die **strikte Auswertungsstrategie** (*eager evaluation*) wählt einen der im Ausdruck am weitesten *innen* stehenden Redexe. Gibt es mehrere, wird von diesen der am weitesten links stehende Redex ausgewählt (*leftmost innermost evaluation*). Diese Strategie beschreibt der am weitesten rechts verlaufende Weg in Abb. 3.20.

Die **nicht-strikte Auswertungsstrategie** (*lazy evaluation*, **Bedarfsauswertung**, **verzögerte Auswertung**) wählt einen der im Ausdruck am weitesten *außen* stehenden Redexe. Gibt es mehrere, wird von diesen der am weitesten links stehende Redex gewählt (*leftmost outermost evaluation*). Die Argumente der Funktion sind dann i.Allg. noch nicht ausgewertet. In Abb. 3.20 ist dies der links verlaufende Weg.

Beide Strategien haben Vor- und Nachteile.

- Bei nicht-strikter Auswertung werden nur Teilausdrücke ausgewertet, deren Wert zum End-

ergebnis beiträgt; bei strikter Auswertung werden evtl. weitere Teilausdrücke ausgewertet. Das ist kritisch, wenn die Auswertung des Teilausdrucks teuer ist, zu einem Laufzeitfehler führt oder nicht terminiert. Für die (nicht-terminierende) Funktion  $f(x) = f(x+1)$  und die (konstante) Funktion  $g(y) = 5$  terminiert die nicht-strikte Auswertung des Ausdrucks  $g(f(0))$ , die strikte Auswertung aber nicht.

- Andererseits berechnet die nicht-strikte Auswertung manchmal denselben Wert mehrfach (in Abb. 3.20: 15/3), obwohl dies bei strikter Auswertung nicht nötig ist. Diesen Nachteil vermeidet die **Bedarfsauswertung mit gemeinsamer Nutzung** (*sharing*), die als gleich erkannte Teilausdrücke nur einmal auswertet (rot gestrichelt in Abb. 3.20). Dazu werden Zeiger auf den gemeinsamen Teilausdruck gesetzt, der beim ersten Bedarf ausgewertet und durch seinen Wert ersetzt wird.

Insgesamt gilt: Alle Auswertungsstrategien, die terminieren, liefern das gleiche Ergebnis. Terminiert eine Strategie, so terminiert auch die nicht-strikte Auswertung (aber evtl. nicht die strikte).

Der Programmierer erkennt die Auswertungsstrategie oft nur indirekt über die Verfahren zur **Parameterübergabe**. Die regelt, in welcher Form aktuelle Parameter übergeben und zu welchem Zeitpunkt diese ausgewertet werden. Wir unterscheiden vier Möglichkeiten:

Strikte Auswertungsstrategien:

- **Adressaufruf / Call-By-Reference** (z.B. Pascal, C++):  
Es wird stets eine Adresse übergeben; dort steht der Wert des Parameters.  
Vorteile: Einfache Übergabe von Datenverbunden und Datenfeldern. Über die Adresse kann –als Nebenwirkung– ein (weiteres) Ergebnis an die aufrufende Funktion geliefert werden.  
Nachteil: Konstante oder Ausdrücke können für diesen Parameter nicht benutzt werden.
- **Wertaufruf / Call-By-Value** (*applicative order evaluation*) (z.B. Algol 60, C, Pascal):  
Die Parameter werden zuerst ausgewertet und ihre Werte an die Funktion übergeben.  
Vorteil: Die Parameter werden nur einmal ausgewertet; außer der Auswertung entsteht kein zusätzlicher Aufwand.  
Nachteil: Jeder Parameter wird ausgewertet, auch wenn sein Wert für die Auswertung des Funktionsrumpfs gar nicht benötigt wird. Das kann eine Terminierung verhindern.

Nicht-strikte Auswertungsstrategien:

- **Formelaufruf / Call-By-Name** (*normal order evaluation*) (z.B. Algol 60):  
Man wertet den Rumpf der Funktion aus und jedesmal, wenn der Wert eines Parameters benötigt wird, wird der Ausdruck des zugehörigen aktuellen Parameters berechnet.<sup>4</sup>  
Vorteil: Ein Parameter wird nur ausgewertet, wenn sein Wert tatsächlich benötigt wird.  
Nachteil: Wird ein Parameterwert mehrmals benötigt, wird er mehrfach berechnet.
- **Formelaufruf mit Ergebnisteilhabe / Call-By-Need** (z.B. Haskell, Scala):  
Ein Parameter wird nur ausgewertet, wenn sein Wert benötigt wird, und dann nur einmal. Der erste Zugriff bewirkt die Auswertung des Parameters; alle weiteren Zugriffe nutzen den bereits berechneten Wert.  
Vorteil: Call-By-Need vereint das gute Terminierungsverhalten von Call-By-Name mit der Effizienz von Call-By-Value.  
Nachteil: Verwaltungsaufwand: Wie erkennt man, ob Parameter bereits ausgewertet wurden? Benötigte nicht-lokale Werte müssen bei Bedarfsauswertung gespeichert werden.

<sup>4</sup> Eine trickreiche Nutzung des Call-By-Name ist Jensen's Device zur Berechnung der Summe von Feldelementen:

```
function sum(real add, int lv): real;
  real temp ← 0.0;
  for lv ← 1 to 10 do temp ← temp+add;
  return temp;
```

Für eine Variable  $V$  ergibt  $\text{sum}(V, L)$  gerade  $10 \cdot V$ .  
Für ein Feld  $F[1:10]$  liefert aber  $\text{sum}(F[L], L)$  die Summe der Feldelemente.

### 3.6.3 Gültigkeitsbereich von Namen

In einem Programm kann ein **Name** mehrmals vorkommen und sogar mehrfach deklariert werden. Daher müssen wir festlegen, wie wir zu einem Namen die richtige Deklaration finden. Ein **definierendes Vorkommen** (*binding occurrence*) eines Namens ist ein Vorkommen, bei denen der Name in einer Deklaration definiert oder in einer formalen Parameterliste spezifiziert wird; alle anderen Vorkommen heißen **angewandte Vorkommen** (*applied occurrence*).

Betrachten wir die in Funktionen auftretenden Namen. Die als formale Parameter oder durch lokale Deklarationen eingeführten Namen nennen wir **lokal** oder **gebunden**. Beim Aufruf einer Funktion wird für jeden lokalen Namen eine neue **Inkarnation** erzeugt. Dazu wird –entsprechend der Größe ihrer Typen– Platz für Variablen bereitgestellt. Die formalen Parameter werden mit den Werten der aktuellen Parameter initialisiert. Die **Lebensdauer** der erzeugten Inkarnationen entspricht der Lebensdauer der Funktionsinkarnation. Der durch sie belegte Platz kann daher bei Verlassen der Funktion freigegeben werden.<sup>5</sup> Eine kellerartige Speicherverwaltung ist dafür gut geeignet, denn derselbe Speicherbereich, der bei Betreten der Funktion für die formalen Parameter, die lokalen Variablen und für anfallende Zwischenergebnisse (Abb. 3.22) bereitgestellt wurde, wird bei Verlassen der Funktion wieder freigegeben.

Nicht ganz so einfach ist die Behandlung von nicht lokalen Namen. Wir bezeichnen diese Namen als **global** oder **frei** – relativ zur betrachteten Funktion. Die **Sichtbarkeits-** bzw. **Gültigkeitsregeln** einer Programmiersprache legen fest, wie das zu einem angewandten Vorkommen eines Namens gehörende definierende Vorkommen gefunden wird. Die umgekehrte, aber äquivalente Sicht geht von einem definierenden Vorkommen eines Namens aus und legt fest, in welchem Programmstück alle angewandten Vorkommen des Namens sich auf dieses definierende Vorkommen beziehen.

Algol-ähnliche Sprachen haben folgende Sichtbarkeitsregel: Ein definierendes Vorkommen eines Namens ist sichtbar in der Programmeinheit (Funktion oder **Block**), in der der Name (durch Deklaration bzw. Spezifikation) definiert wurde, ohne die von dieser Programmeinheit echt umfassten („inneren“) Programmeinheiten, die diesen Namen neu definieren.

Suchen wir also zu einem angewandten Vorkommen das zugehörige definierende Vorkommen, so beginnen wir mit der Suche bei den Definitionen der Programmeinheit, in der das angewandte Vorkommen steht. Ggf. setzen wir die Suche in der direkt umfassenden Programmeinheit fort. Findet sich auch in allen umfassenden („äußeren“) Programmeinheiten (einschließlich des gesamten Programms) kein definierendes Vorkommen, so liegt ein Programmierfehler vor.

Die andere Sicht – ausgehend von einem definierenden Vorkommen – überstreicht die enthaltende Programmeinheit und ordnet allen angewandten Vorkommen des Namens dieses definierende Vorkommen zu. Nur an Programmeinheiten, die eine Redefinition des gleichen Namens enthalten, wird der überstreichende Strahl abgeblockt.

Durch diese Sichtbarkeitsregel erhält man die **statische Bindung** (*static scoping*), d.h. globale Namen werden definierenden Vorkommen in textlich umgebenden Programmeinheiten zugeordnet. Diese Zuordnung ist statisch, da sie nur auf dem Programmtext und nicht auf der (dynamischen) Ausführung des Programms beruht. Jede Benutzung eines globalen Namens zur Ausführungszeit betrifft eine Inkarnation des statisch zugeordneten definierenden Vorkommens.

Im Gegensatz dazu betrifft die **dynamische Bindung** (*dynamic scoping*) stets die *zuletzt angelegte* Inkarnation dieses Namens – egal, in welcher Funktion er definierend auftrat.

Statische Bindung wird von allen Algol-ähnlichen Sprachen und modernen funktionalen Sprachen wie Haskell vorgeschrieben, während ältere Lisp-Dialekte die dynamische Bindung benutzen.

<sup>5</sup> Wir ignorieren lokale Variable, die durch eine Zusatzspezifikation (**own** in Algol 60, **static** in PL/I oder C) ihre Funktionsinkarnation überleben.

**Beispiel 3.5:** Im folgenden Programm bezieht sich bei statischer Bindung das angewandte Vorkommen der Variablen  $x$  in der Funktion  $q$  auf die globale Variable  $x$  und liefert den Wert 1.

```

int  $x \leftarrow 1$ ;
void  $q()$  {
     $\text{printf}(\text{"\%d", } x);$ 
}

int  $\text{main}()$  {
    int  $x \leftarrow 2$ ;
     $q();$ 
}

```

Das zuletzt angelegte Vorkommen einer Variablen  $x$  vor dem Aufruf der Funktion  $q$  in der Funktion  $\text{main}$  ist aber deren lokale Variable  $x$ . Bei dynamischer Bindung liefert das angewandte Vorkommen von  $x$  in  $q$  den Wert 2.

Die Programmiersprache ANSI-C verbietet –im Gegensatz zu Pascal– geschachtelte Funktionsdefinitionen. Diese Entwurfsentscheidung vereinfacht die Verwaltung der Sichtbarkeitsbereiche erheblich; der **statische Vorgänger** einer Funktion ist hier immer das Hauptprogramm.

In ANSI-C unterscheiden wir daher nur zwei Arten von Variablen:

- **globale Variable**, die außerhalb der Funktionsdefinitionen deklariert sind, und
- **lokale Variable**, die lokal zu einzelnen Funktionen definiert sind.

### 3.6.4 Speicherorganisation für Funktionen: Der Laufzeitkeller

Eine einfache Implementierung ersetzt jeden Funktionsaufruf *inline* durch den Code für den Funktionsrumpf (*open subroutine*). Rekursive Funktionen lassen sich so aber nicht übersetzen.

Ein raffinierteres Vorgehen erzeugt nur *einmal* Code für eine Funktion (*closed subroutine*). Dann benutzen alle Inkarnationen einer Funktion denselben Code; jede Inkarnation hat aber i.d.R. unterschiedliche lokale Daten (auch wenn Anzahl und Typen der Daten für jede Inkarnation gleich sind) und benötigt dafür einen eigenen Datenbereich, den **Kellerrahmen** (*stack frame*).

Die Speicherorganisation, der **Laufzeitkeller**, enthält einen Speicherbereich für jede lebende Inkarnation, und zwar in der Reihenfolge, in der diese Inkarnationen auf dem Inkarnationsweg auftreten (in Abb. 3.21 für Beispiel 3.4).

Um eine Funktion effizient verlassen zu können, sind die Inkarnationen so verkettet, dass jede auf die Inkarnation zeigt, von der sie durch einen Aufruf angelegt wurde: auf ihren **dynamischen Vorgänger**.

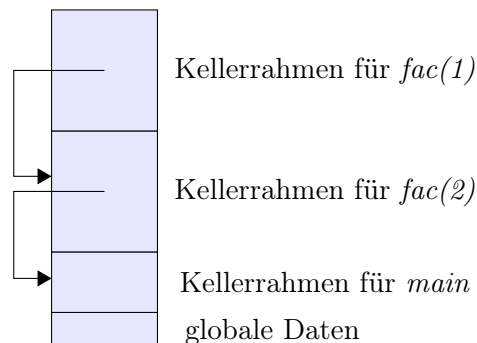


Abb. 3.21: Der Laufzeitkeller der C-Maschine.

Ein Register, der **Rahmenzeiger FP** (*frame pointer*), zeigt stets auf eine bestimmte Stelle des aktuellen Kellerrahmens und adressiert Parameter und lokale Variablen (Abb. 3.22).

Der Kellerrahmen enthält auch Platz für die Register, deren Inhalt bei Betreten einer Funktion gerettet wird, weil er bei Verlassen der Funktion wiederhergestellt werden muss. Diese Zellen heißen **organisatorische Zellen**, weil sie eine korrekte und effiziente Organisation von Funktionsanfang und -ende ermöglichen.

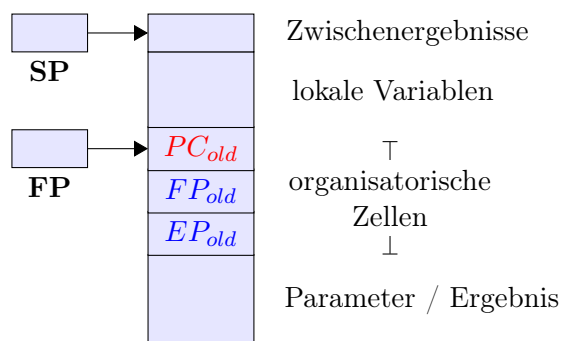


Abb. 3.22: Ein Kellerrahmen.

In der C-Maschine sind dies die drei Register PC, FP und EP:



- Der gerettete Inhalt des PC ist die **Rücksprungadresse** im Programmspeicher, an der die Berechnung nach Beendigung des Funktionsaufrufs fortfahren soll.
- Der gerettete Inhalt des FP verweist auf die Daten der aufrufenden Funktion, genauer auf den Kellerrahmen des dynamischen Vorgängers des aktuellen Funktionsaufrufs.
- Vorsorglich retten wir auch den Inhalt des EP.

Oberhalb der organisatorischen Zellen beginnt der Datenbereich der aktuellen Funktionsinkarnation. Hier legen wir die lokalen Variablen der aktuellen Funktion ab. Dadurch können wir auf diese Variablen mit **festen Relativadressen** relativ zum Rahmenzeiger FP zugreifen. Zeigt der Rahmenzeiger auf die oberste organisatorische Zelle ( $PC_{old}$ ), können für die lokalen Variablen Relativadressen ab 1 vergeben werden.

Oberhalb des Datenbereichs legen wir den **lokalen Keller**, das ist der Keller, der uns bei der Auswertung von Ausdrücken begegnet ist. Seine maximale Länge ist statisch bestimmbar.

Zwei Dinge müssen noch im Kellerrahmen untergebracht werden: die aktuellen Parameter und der Rückgabewert der Funktion. Eine *konstante* Anzahl von **aktuellen Parametern** könnten wir zwischen organisatorischen Zellen und lokalen Variablen platzieren. Die Programmiersprache C erlaubt aber Funktionsdefinitionen mit **variablen Parameterlisten**, z.B. ist bei der Funktion *printf* nur der erste Parameter **obligatorisch**, alle weiteren aktuellen Parameter sind **optional**, ihre Anzahl kann also von Funktionsaufruf zu Funktionsaufruf variieren.

Um trotzdem für die lokalen Variablen feste Relativadressen vergeben zu können, nutzen wir einen Trick: die aktuellen Parameter werden *unterhalb* der organisatorischen Zellen und *in umgekehrter Reihenfolge* auf den Keller gelegt (Abb. 3.23)! Der erste Parameter liegt also oberhalb des zweiten usw. Als Relativadressen stehen die negativen Zahlen ab  $-3$  zur Verfügung. Hat der erste Parameter z.B. die Größe  $|t|$ , bekommt er die Relativadresse  $-(|t|+2)$ .

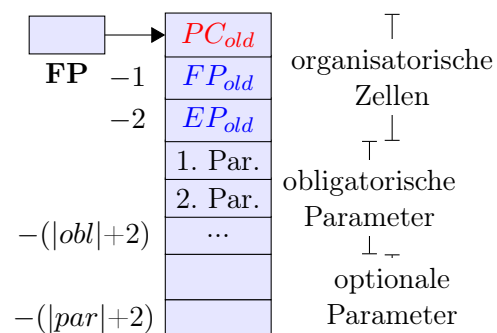


Abb. 3.23: Parameter im Kellerrahmen.

Liefert die Funktion einen **Rückgabewert** zurück, sollten wir dafür einen Standardplatz vorsehen, auf den sich mit einer festen Relativadresse relativ zum FP zugreifen lässt. Wir könnten dazu einen Extrabereich unterhalb der organisatorischen Zellen wählen. Da der Platz für die aktuellen Parameter bei Rückkehr aus einem Funktionsaufruf nicht mehr benötigt wird, werden wir diesen Bereich zum Ablegen des Rückgabewerts wiederverwenden.

### 3.6.5 Adressierung lokaler Variablen und des Rückgabewerts

Wir haben deklarierten *globalen* Namen bereits Speicherzellen bzw. Adressen zugeordnet. Dabei griffen wir über den Kellerzeiger SP auf Variablen zu (**absolute Adressierung**).

Bei Parametern und lokalen Variablen wollen wir aber auf Instanzen im aktuellen Funktionsaufruf zugreifen. Die Adressierung erfolgt daher relativ zum Rahmenzeiger FP. Um diesen Unterschied bei der Codeerzeugung berücksichtigen zu können, erweitern wir die Adressumgebung  $\rho$  so, dass sie außer der Relativadresse auch die Information verwaltet, ob der Name global oder lokal ist. Die Adressumgebung  $\rho$  erhält deshalb die Funktionalität:

$\rho : Names \rightarrow \{G, L\} \times \mathbb{Z}$ , die Etiketten  $G$  und  $L$  bezeichnen globale bzw. lokale Gültigkeit.

Für Zugriffe auf lokale Variablen oder Parameter verallgemeinern wir die Übersetzungsfunktion  $code_A$  für Namen:

$$\text{code}_A^\rho x = \begin{cases} \text{loadrc } j & , \text{ falls } \rho(x) = (G, j) \\ \text{loadrc } j & , \text{ falls } \rho(x) = (L, j) \end{cases}$$

Dabei laden die neuen Befehle **loadrc**  $j$  den Wert  $FP+j$  oben auf den Keller (Abb. 3.24).

Natürlich können wir wieder Spezialbefehle für häufig vorkommende Befehlsfolgen einführen:

**loadr**  $j \equiv \text{loadrc } j$ , **load**  
**storer**  $j \equiv \text{loadrc } j$ , **store**

Falls  $m$  Speicherzellen bewegt werden sollen, nutzen wir die Befehle **loadr**  $j \ m$  und **storer**  $j \ m$ .

Mit dieser Änderung können wir die Codeerzeugung, wie wir sie bisher Schritt für Schritt entwickelten, auch auf die Rümpfe von Funktionen anwenden.

Ein Name kann auch eine Funktion bezeichnen. Auch Funktionsnamen wollen wir eine Adresse zuordnen, nämlich die Anfangsadresse ihres Codes im *Programmspeicher* C. Diese Anfangsadresse des Codes einer Funktion  $f$  beschreiben wir durch die Marke  $\_f$ .

Wir benötigen noch eine systematische Methode zur **Berechnung der Adressumgebung**  $\rho$ , die dafür sorgt, dass die Adressumgebung an jeder Anwendungsstelle genau die dort sichtbaren Namen mit der jeweils aktuellen Information zeigt.

Für eine Variablendeklaration  $d \equiv t \ x$ ; definieren wir Funktionen, die für ein Paar  $(\rho, n)$  aus Adressumgebung  $\rho$  und erster freier Relativadresse  $n$  eine erweiterte Adressumgebung und die nächste freie Relativadresse liefern. Für eine Deklaration

- einer globalen Variablen:  $\text{elab\_global}(\rho, n) (t \ x) = (\rho \oplus \{x \mapsto (G, n)\}, n + |t|)$
- einer lokalen Variablen:  $\text{elab\_local}(\rho, n) (t \ x) = (\rho \oplus \{x \mapsto (L, n)\}, n + |t|)$
- eines formalen Parameters:  $\text{elab\_formal}(\rho, z) (t \ x) = (\rho \oplus \{x \mapsto (L, z - |t|)\}, z - |t|)$

Der Ausdruck  $\rho \oplus \{x \mapsto a\}$  bezeichnet die (partielle) Funktion, die für das Argument  $x$  den Eintrag  $a$  zu  $\rho$  hinzufügt bzw. einen alten Eintrag für  $x$  mit dem neuen Wert  $a$  überschreibt.

Die Funktionen **elab\_global** und **elab\_local** unterscheiden sich nur durch das Etikett. Bei der Funktion **elab\_formal** erhält jeder weitere Parameter eine *kleinere* Adresse. Hier übergeben wir die unterste bisher belegte Adresse  $z$  im Kellerrahmen anstelle der ersten freien Relativadresse.

Listen von Deklarationen arbeiten wir durch wiederholte Anwendung dieser Funktionen ab:

$\text{elab\_globals}(\rho, n) () = (\rho, n)$   
 $\text{elab\_globals}(\rho, n) (t \ x; ll) = \text{elab\_globals}(\text{elab\_global}(\rho, n) (t \ x)) (ll)$   
 $\text{elab\_locals}(\rho, n) () = (\rho, n)$   
 $\text{elab\_locals}(\rho, n) (t \ x; ll) = \text{elab\_locals}(\text{elab\_local}(\rho, n) (t \ x)) (ll)$   
 $\text{elab\_formals}(\rho, z) () = (\rho, z)$   
 $\text{elab\_formals}(\rho, z) (t \ x, sp) = \text{elab\_formals}(\text{elab\_formal}(\rho, z) (t \ x)) (sp)$

Sei  $f$  eine Funktion ohne Rückgabewert mit einer Spezifikation  $sp$  formaler Parameter und einer Deklaration  $ll$  lokaler Variablen und  $\_f$  die Anfangsadresse des Codes von  $f$ . Aus einer Adressumgebung  $\rho$  für globale Namen entsteht dann die Adressumgebung  $\rho_f$  für die Funktion  $f$ :

$\rho_f = \text{let } \rho = \rho \oplus \{f \mapsto (G, \_f)\}$   
 $\quad \text{in let } (\rho, \_) = \text{elab\_formals}(\rho, -2) \ sp$   
 $\quad \quad \text{in let } (\rho, \_) = \text{elab\_locals}(\rho, 1) \ ll$   
 $\quad \quad \text{in } \rho$

Hat die Funktion  $f$  einen **Rückgabewert**, verwalten wir dessen Relativadresse unter dem lokalen Namen  $ret$  ebenfalls in der Adressumgebung  $\rho_f$  für  $f$ .

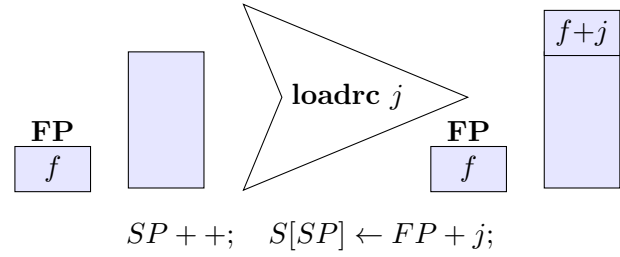


Abb. 3.24: Der Befehl **loadrc**  $j$ .

Sei  $t$  der Typ des Rückgabewerts und  $|obl|$  der Platzbedarf der obligatorischen Parameter.

- Ist  $|t| \leq |obl|$ , können wir den Rückgabewert *am unteren Rand* des Blocks für die obligatorischen Parameter ablegen, d.h. ab Relativadresse  $-(|obl|+2)$  (Abb. 3.25).
- Ist  $|t| > |obl|$ , benötigen wir für die Rückgabe evtl. einen größeren Block als für die Parameter. Dann legen wir den Rückgabewert ab Adresse  $-(|t|+2)$  ab (Abb. 3.26).

Wir erweitern daher unsere Definition von  $\rho_f$  um die Bindung  $ret \mapsto (L, -(max(|obl|, |t|)+2))$ .

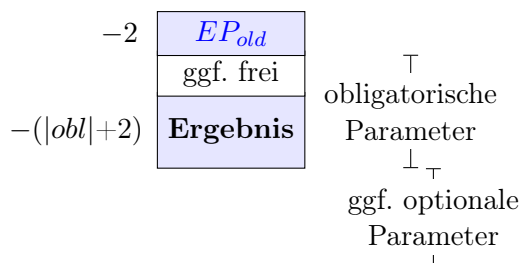


Abb. 3.25: Rückgabewert mit  $|t| \leq |obl|$ .

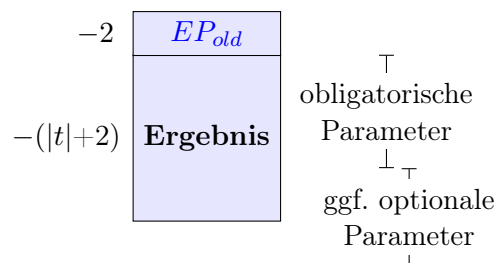


Abb. 3.26: Rückgabewert mit  $|t| > |obl|$ .

**Beispiel 3.6:** Betrachten wir erneut das C-Programm aus Beispiel 3.4.

- Als globale Namen haben wir keine globalen Variablen, aber  $fac$  und  $main$  für Funktionen. Dann ist  $\rho_0 = \emptyset$  die Adressumgebung vor der Deklaration der Funktion  $fac$ .
- Aus  $\rho_0$  erhalten wir die Adressumgebung  $\rho_{fac}$  innerhalb von  $fac$ , indem wir zuerst die Bindung  $fac \mapsto (G, \_fac)$  zu  $\rho_0$  hinzufügen und dann die Bindung  $n \mapsto (L, -3)$  für den formalen Parameter  $n$ . Schließlich ermitteln wir die Relativadresse des Rückgabewerts zu  $-3$ . Wir erhalten  $\rho_{fac} = \{fac \mapsto (G, \_fac), n \mapsto (L, -3), ret \mapsto (L, -3)\}$
- Mit dem Ende der Funktionsdefinition  $fac$  werden alle lokalen Bindungen entfernt. Vor der Definition der Funktion  $main$  ist bereits der Name der Funktion  $fac$  bekannt. Deshalb ist  $\rho_1 = \{fac \mapsto (G, \_fac)\}$
- Für die Funktion  $main$  tragen wir zunächst eine Bindung für den globalen Funktionsnamen ein. Die Funktion hat keine Parameter, aber einen Rückgabewert der Größe 1. Deshalb erhält dieser die Relativadresse  $-3$ .  $\rho_{main}$  enthält eine Bindung für die lokale Variable  $n$ . Dann ergibt sich die Adressumgebung  $\rho_{main}$  innerhalb der Funktion  $main$  als:  
 $\rho_{main} = \{fac \mapsto (G, \_fac), main \mapsto (G, \_main), n \mapsto (L, 1), ret \mapsto (L, -3)\}$

### 3.6.6 Betreten und Verlassen von Funktionen

Betrachten wir zunächst den **Aufruf** einer Funktion. Sei  $f$  die gegenwärtig aktive Funktion. Ihr Kellerrahmen ist daher der oberste auf dem Keller. Jetzt rufe die Funktion  $f$  eine Funktion  $g$  auf. Wir wollen für den Aufruf von  $g$  eine Befehlsfolge erzeugen, die den Aufruf auswertet und den Rückgabewert oben auf dem Keller hinterlässt.

Bei einem Aufruf der Funktion  $g$  müssen folgende Aktionen ausgeführt werden, bevor die Anweisungen von  $g$  abgearbeitet werden:

- B1 Platz für Funktionsergebnis reservieren, falls die obligatorischen Parameter zu klein sind
  - B2 Werte der aktuellen Parameter ermitteln und auf den Keller schreiben
  - B3 Alte Werte der Register EP und FP auf den Keller retten
  - B4 Anfangsadresse der Funktion  $g$  berechnen
  - B5 Rücksprungsadresse ermitteln und auf den Keller retten
  - B6 Register FP auf den aktuellen Kellerrahmen setzen
  - B7 zur Anfangsadresse von  $g$  springen
- 
- B8 Platz für lokale Variablen von  $g$  reservieren
  - B9 Register EP auf den Wert für die neue Inkarnation setzen

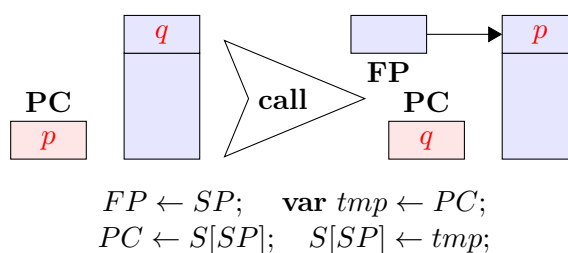
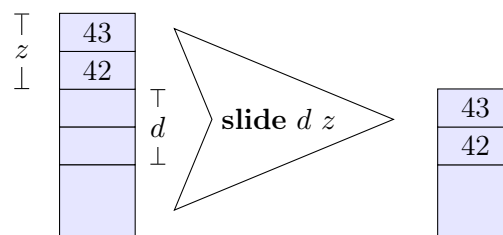


die Rückkehradresse! Dadurch kann der Befehl **call** die Aufgabe B5 miterledigen, indem er die Zieladresse oben auf dem Keller durch den aktuellen PC ersetzt.

Schließlich ordnen wir diesem Befehl auch noch die Aufgabe B6 zu, das Register FP auf den aktuellen Wert zu setzen, also auf die oberste belegte Kellerzelle.

Liegt der Rückgabewert nicht am unteren Rand des Kellerrahmens, weil optionale Parameter vorhanden sind, so verschieben wir ihn –nach dem Rücksprung, Aufgabe V5– entsprechend nach unten. Dazu dienen die Befehle **slide**  $d\ z$  (Abb. 3.29).

Hat  $g$  keine optionalen Parameter ( $|obl|=|par|$ ), so ist das Argument  $d$  des **slide**-Befehls 0.

Abb. 3.28: Der Befehl **call**.

```

if d > 0 then
  if z = 0 then SP ← SP-d;
  else {
    SP ← SP-d-z;
    for i ← 0; i < z; i++ do {
      SP++; S[SP] ← S[SP+d]; }
  }

```

Abb. 3.29: Der Befehl **slide**  $d\ z$ .

Sei  $t$  der Typ des Ergebnisses der Funktion  $g$ ,  $|obl|$  der Platzbedarf für die obligatorischen Parameter und  $|par| \geq |obl|$  der Platzbedarf für alle Parameter.

Dann lautet das Übersetzungsschema für einen **Funktionsaufruf**  $e \equiv g(e_1, \dots, e_n)$ :

$\text{code}_W^\rho g(e_1, \dots, e_n) = \text{alloc } q, \text{ code}_W^\rho e_n, \dots, \text{code}_W^\rho e_1, \text{ mark, code}_W^\rho g, \text{ call, slide } d\ |t|$   
 mit  $q = \max(|t| - |obl|, 0)$ ,  $d = \text{if } |t| \leq |obl| \text{ then } |par| - |obl| \text{ else } \max(|par| - |t|, 0)$

Das Schema erzeugt für jeden aktuellen Parameter  $e_i$  Code, der den Wert von  $e_i$  (in der Adressumgebung  $\rho$  von  $f$ ) berechnet. Dies entspricht einer Parameterübergabe **by value**.

Pascal bzw. C++ unterstützen auch eine Übergabe der Parameter **by reference**.

Dann soll statt des Wertes die Adresse des aktuellen Parameters am Platz des zugehörigen formalen Parameters  $x$  abgelegt werden. Jeder Zugriff auf den formalen Parameter  $x$  im Rumpf der Funktion erfordert dann eine **Indirektion** über seine Relativadresse. Für einen Referenzparameter  $x$ , dem wir das Etikett  $R$  zuordnen, berechnen wir seine Adresse deshalb als:

$\text{code}_A^\rho x = \text{loadr } j \quad , \text{ falls } \rho(x) = (R, j).$

In C wird für den Ausdruck  $g$ , der die Anfangsadresse der aufzurufenden Funktion liefern soll, ebenfalls Code zur Berechnung des Wertes erzeugt. Dadurch kann man eine Funktion auch über Funktionszeiger aufrufen. Einfache Funktionsnamen aber werden in C –wie Felder– als **Referenzen** aufgefasst, deren Wert gleich der Adresse ist:  $\text{code}_W^\rho f = \text{code}_A^\rho f = \text{loadc } \_f$

Hier bietet sich ein Spezialbefehl an: **calld**  $a \equiv \text{mark, loadc } a, \text{ call}$

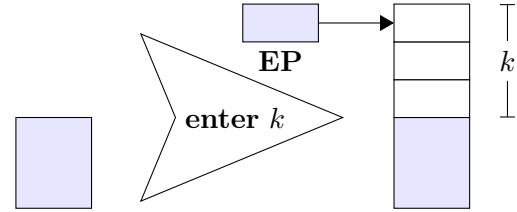
Für den rekursiven Funktionsaufruf  $\text{fac}(n-1)$  in dem Programm aus Beispiel 3.4 erhalten wir dann die Befehlsfolge:

**alloc** 0, **loadr** -3, **loadc** 1, **sub**, **calld**  $\_fac$ , **slide** 0 1

### 3.6.8 Übersetzung einer Funktionsdefinition

Zu Beginn der Übersetzung einer Funktionsdefinition erzeugen wir Code für die Schritte B8 und B9: Mit einem **alloc**-Befehl reservieren wir Platz für die lokalen Variablen (B8).

Den neuen EP (B9) setzen wir relativ zum aktuellen SP mit dem Befehl **enter**  $k$ , wobei  $k$  der maximale Platzbedarf für die Zwischenergebnisse der Funktion  $g$  ist (Abb. 3.30). Der Befehl überprüft auch, ob auf dem Keller genug Platz zur Abarbeitung des aktuellen Aufrufs bleibt. Sonst wird die Ausführung des Programms mit einer Fehlermeldung abgebrochen.



$EP \leftarrow SP + k; \text{testoverflow}$

Abb. 3.30: Der Befehl **enter**  $k$ .

**testoverflow** ist eine Abkürzung für:

**if**  $EP > \text{maxS}$  **then** **error**(„Stack Overflow“);

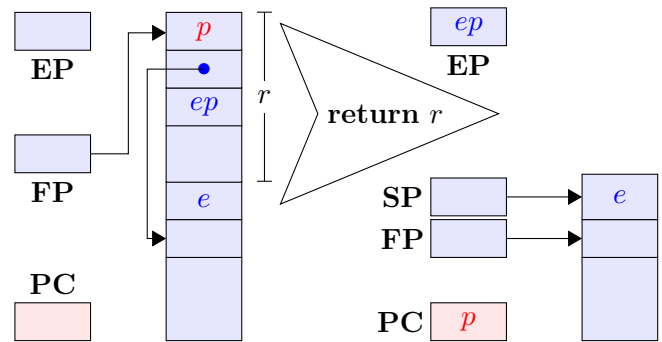
Die Aufgabe V1 –die Speicherung des Rückgabewerts– behandeln wir wie eine Wertzuweisung, da wir die Anfangsadresse eines Rückgabewerts in der Adressumgebung verwalten.

Im Befehl **return**  $r$  (Abb. 3.31) fassen wir die Aufgaben V2–V4 zusammen. Insbesondere sollen  $r$  Zellen oberhalb des Rückgabewerts freigegeben werden. **testoverflow** prüft, ob auf dem Keller noch genügend Platz zur Abarbeitung der aufrufenden Funktion ist.

Sei  $|obl|$  der Platzbedarf für die obligatorischen Parameter,  $t$  der Rückgabebetyp der Funktion  $g$ ,  $r = 3 + \max\{|obl| - |t|, 0\}$ .

Wir übersetzen die C-Anweisungen **return**  $e$ ; bzw. **return**; für einen Ausdruck  $e$  vom Typ  $t$ :

$\text{code}^\rho(\text{return};) = \text{return } (|obl|+3)$   
 $\text{code}^\rho(\text{return } e;) =$   
 $\text{code}_W^\rho e, \text{storer } \rho(\text{ret}) |t|, \text{return } r$



$PC \leftarrow S[FP]; \quad EP \leftarrow S[FP-2];$   
**testoverflow**  
 $SP \leftarrow FP - r; \quad FP \leftarrow S[FP-1];$

Abb. 3.31: Der Befehl **return**  $r$ .

Sei  $l$  die Anzahl Speicherzellen für die lokalen Variablen,  
 $k$  der maximale Platzbedarf für die Zwischenergebnisse der Funktion  $g$ ,  
 $\rho_g$  die Adressumgebung für  $g$ , die man aus  $\rho$ ,  $params$ ,  $locals$  sowie  $|t|$  erhält.

Damit übersetzen wir die **Definition einer Funktion**  $g$  mit Rückgabebetyp  $t$ , formalen Parametern  $params$ , lokalen Variablen  $locals$  und Anweisungsfolge  $ss$ :

$\text{code}^\rho(t \ g \ (params) \ \{locals \ ss\}) = \_g: \text{alloc } l, \text{enter } k, \text{code}^{\rho_g} ss, \text{return } r$

Das **return**  $r$  am Ende der übersetzten Folge wird gebraucht, falls die Funktion durch Abarbeitung des textuellen Endes der Funktion beendet wird.

**Beispiel 3.7:** Dann können wir den Rumpf der Funktion  $fac$  aus Beispiel 3.4 so übersetzen (Der Platzbedarf  $k$  für Zwischenergebnisse beträgt 3):

~~$\_fac: \text{alloc } 0, \text{enter } 3, \text{loadr } -3, \text{loadc } 0, \text{leq}, \text{jumpz } A, \text{loadc } 1, \text{storer } -3, \text{return } 3, \text{jump } B,$~~   
 ~~$A: \text{loadr } -3, \text{alloc } 0, \text{loadr } -3, \text{loadc } 1, \text{sub}, \text{calld } \_fac, \text{slide } 0, \text{mul}, \text{storer } -3, \text{return } 3,$~~   
 ~~$B: \text{return } 3$~~

Ein Optimierer könnte Befehle ohne Wirkung und nicht erreichbare Befehle entfernen.

### 3.7 Übersetzung eines ganzen Programms

Vor der Programmausführung setzen wir alle Register auf den Wert 0. Die Ausführung des Programms der C-Maschine beginnt mit dem Befehl in  $C[0]$ .

Das Programm  $p$  ist eine Folge von Deklarationen von globalen Variablen und Funktionen, von denen eine die Funktion **int**  $main()$  definiert. Zur Vereinfachung erlauben wir keine Kommandozeilenparameter bei der Funktion  $main$ . Der Code für das Programm  $p$  enthält damit:

1. Code zum Anlegen der globalen Variablen;
2. Code für die Funktionsdefinitionen;
3. Code für den Aufruf der Funktion  $main$ ;
4. den Befehl **halt**, um die Programmausführung zu beenden.

Sei  $dd$  eine Folge von Variablendeklarationen und seien  $df_1, \dots, df_n$  die Deklarationen der Funktionen  $f_1, \dots, f_n \equiv main$ . Zur Vereinfachung nehmen wir an, dass alle Variablendeklarationen im Programm  $p$  vor den Funktionsdeklarationen stehen:  $p \equiv dd \ df_1 \ \dots \ df_n$

$(\rho_0, k) = \mathbf{elab\_globals}(\emptyset, 1) \ dd$

ist dann das Paar aus der Adressumgebung für die globalen Variablen von  $p$  und der ersten freien Relativadresse; dabei steht  $\emptyset$  für die leere Adressumgebung.

Die Adressumgebung vor Abarbeitung der  $i$ -ten Funktionsdefinition ist dann:

$\rho_i = \rho_{i-1} \oplus \{f_i \mapsto (G, \_f_i)\} \quad (i = 1, \dots, n)$ , dabei ist  $\_f_i$  die Anfangsadresse der Funktion  $f_i$ .

Dann erhalten wir als Übersetzungsschema für das Programm  $p$ :

$\text{code } p = \mathbf{alloc } k, \mathbf{enter } 3, \mathbf{call} \ \_f_n, \mathbf{slide } (k-1) \ 1, \mathbf{halt}, \quad \text{code}^{\rho_1} \ df_1, \dots, \text{code}^{\rho_n} \ df_n$

Vor dem Aufruf der Funktion  $main$  reservieren wir oberhalb der Speicherzelle mit Adresse 0 insgesamt  $k-1$  Speicherzellen für globale Variablen sowie eine Zelle für den Rückgabewert von  $f_n$ . Da der SP vor der Ausführung den Wert 0 hat, müssen wir ihn um  $k$  erhöhen. Bis zur Ausführung des ersten Befehls **call** werden drei weitere Zellen auf den Keller gelegt. Deshalb erhält der **enter**-Befehl das Argument 3.

Nach Abarbeitung des ersten Aufrufs liegt der Rückgabewert in der Speicherzelle mit Adresse  $k$ . Der Befehl **halt**, der ihn an das Betriebssystem zurückliefern soll, erwartet ihn aber in der Speicherzelle 1. Deshalb müssen wir ihn nach dem Aufruf um  $k-1$  Zellen nach unten verschieben. Das leistet der Befehl **slide**  $(k-1) \ 1$ .

**Warnung:** Wir haben –zur besseren Verständlichkeit– stark vereinfacht. Einige Realitäten wurden ignoriert. Insbesondere muss die **Wortgröße** festgelegt werden.

Z.B. haben wir eine Konstante immer direkt als Operanden im Befehl untergebracht, in einigen Befehlen sogar zwei. Die zur C-Maschine verwandte P-Maschine für **Pascal** unterscheidet, ob ein konstanter Operand in den Befehl hineinpasst oder nicht. Zu große Konstanten werden in einer **Konstantentabelle** gespeichert, die am oberen Speicherende oberhalb der Halde angelegt wird. Die Befehle enthalten dann nur Verweise in diese Tabelle. Es gibt dann *verschiedene Befehle für eine Operation*, je nachdem, ob die Operanden im Befehl oder in der Konstantentabelle stehen.



## 4 Zwischencode-Erzeugung: Datenabstraktion und objekt-orientierte Programmiersprachen

Bisher haben wir Ausdrücke und Anweisungen übersetzt und die funktionale Abstraktion betrachtet, also die Bereitstellung neuer Funktionalität durch Kombination von bereits bekannten Sprach-Bausteinen. Dies ist der Kernbereich imperativer Sprachen.

Nun betrachten wir die **Datenabstraktion**, also die Bereitstellung neuer strukturierter Datentypen. Die gemeinsame Kapselung von Funktion und Daten führt dann schnell zu objekt-orientierten Sprachkonzepten. Aber bereits in imperativen Sprachen gibt es Formen der Datenabstraktion, z.B. Datenverbunde und Datenfelder. Rekursive und irreguläre Datentypen lassen sich mit Hilfe von Zeigern implementieren. Diese werden auch zur Implementierung von Objekten genutzt, um das Problem der zur Übersetzungszeit unbekannten Größe von Objekten zu lösen.

Um rekursive Definitionen von Funktionen möglich zu machen, musste man die Implementierung als *open subroutine* ersetzen, weil das textuelle Einkopieren des Codes für den Funktionsrumpf dann nicht terminiert, durch eine *closed subroutine* mit dem zusätzlichen Aufwand eines Laufzeitkellers. Ebenso sind rekursiv definierte Daten in einer strikten Sprache –also bei Verwendung des Call-By-Value– unmöglich, weil dann alle Argumente vor dem Funktionsaufruf ausgewertet werden und dieser Prozess im rekursiven Fall nicht terminiert. **Rekursiv definierte Datenstrukturen** sind daher in vielen Programmiersprachen nicht enthalten. Um sie zu ermöglichen, benötigt man nicht-strikte (*lazy*) Sprachelemente, die Argumente nur bei Bedarf auswerten. Darauf werden wir im Kapitel über Funktionale Programmiersprachen näher eingehen.

### 4.1 Datenverbunde und Datenfelder

Werden mehrere Daten als eine Einheit aufgefasst, sprechen wir von **strukturierten Daten** oder einem **Datenverbund** (*record*, *struct*).

Sind alle Daten vom gleichen Typ, so können sie auch als **Datenfeld** (*array*) realisiert werden.

Betrachten wir die Speicherbelegung und Adressierung bei **Verbunden** oder **Strukturen**.

Sei eine Verbundvariable  $x$  deklariert durch: **struct**  $t$  {**int**  $a$ ; **int**  $b$ ; }  $x$ ;

Der Verbundvariablen  $x$  ordnen wir wieder die Adresse der ersten freien Speicherzelle zu. Die **Komponenten** von  $x$  erhalten Adressen relativ zum Anfang der Struktur:  $a \mapsto 0$ ,  $b \mapsto 1$ . Diese Relativadressen hängen nur vom Typ  $t$  ab. Wir sammeln diese in einer Funktion **offsets**, die Paaren  $(t, c)$  aus Verbundtyp und Komponente die zugehörige Relativadresse zuordnet.

Hat ein Verbundtyp  $t$  die Komponenten  $c_1 \dots c_k$  mit Typen  $t_1 \dots t_k$ , so definieren wir:

$$\text{offsets}(t, c_1) = 0 \quad \text{und} \quad \text{offsets}(t, c_i) = \text{offsets}(t, c_{i-1}) + |t_{i-1}| \quad , \text{ für } i > 1$$

Die Komponenten können auch zusammengesetzt sein. Deshalb benötigen wir zur Konstruktion der Speicherbelegung eine Hilfsfunktion (in C: **sizeof**), die uns für jeden Typ  $t$  seine **Größe**  $|t|$  berechnet, nämlich die Anzahl der benötigten Speicherzellen.

Die **Größe eines Verbundtyps**  $t$  ergibt sich als

Summe der Größen seiner Komponenten  $t_i$ :

$$|t| = \sum_{i=1}^k |t_i|$$

Deklarierte Variablen werden wir weiter aufeinanderfolgend im Keller ablegen.

Die **Adresse einer Verbundkomponente** wird dann in zwei Schritten ermittelt:

- Laden der Anfangsadresse des Verbundes;
- Erhöhung der Adresse um die Relativadresse der Komponente.

Sei  $e$  ein Ausdruck von einem Verbundtyp  $t$  mit Komponente  $c$ . Dann wird zur Berechnung der



Adresse von  $e.c$  der folgende Code erzeugt:

$$\text{code}_A^\rho(e.c) = \text{code}_A^\rho e, \text{loadc } m, \text{add } , \text{ wobei } m = \text{offsets}(t, c).$$

Wie ermittelt man den **Wert einer Verbundkomponente**?

Für Komponenten, deren Typ zusammengesetzt ist, reicht das Laden des Inhalts der adressierten Zelle nicht aus, sondern es muss ein ganzer Block oben auf den Keller geladen werden. Das Übersetzungsschema, um den Wert eines Ausdrucks  $e$  der Größe  $|t|$  zu berechnen, lautet daher:

$$\text{code}_W^\rho e = \text{code}_A^\rho e, \text{load } |t|$$

Die Zuweisung eines Werts vom Verbundtyp  $t$  wird dann übersetzt mit:

$$\text{code}_W^\rho(e_1 \leftarrow e_2) = \text{code}_W^\rho e_2, \text{code}_A^\rho e_1, \text{store } |t|$$

Als nächstes wollen wir ein **Datenfeld** (*array*) übersetzen. In einem Feld haben alle Komponenten den gleichen Typ. Die Komponenten haben ganze Zahlen als Standardnamen (**Index**), die auch als Wert eines Ausdrucks errechnet werden können.

Die Programmiersprache C stellt nur **statische Felder** zur Verfügung, d.h. die Größe und Anzahl der Feldkomponenten ist zur Übersetzungszeit bekannt. Die Indexuntergrenze in C ist stets 0.

Betrachten wir die folgende Deklaration eines Feldes  $a$ : `int a [8];`

Wieviele Speicherzellen belegt das Feld  $a$  zur Laufzeit? Nun,  $a$  besteht aus den 8 Komponenten

$a[0], a[1], a[2], a[3], a[4], a[5], a[6], a[7],$

von denen jede –da vom Typ `int`– eine Zelle belegt. Damit benötigt das Feld elf Speicherzellen für seine Komponenten. Die Elemente legen wir aufeinanderfolgend in den Keller. In der Speicherbelegungsfunktion setzen wir als Anfangsadresse für  $a$  die Anfangsadresse der Komponente  $a[0]$ .

Die Größe  $|t'|$  eines Feldtyps  $t' \equiv t[k]$  ergibt sich als Produkt aus Anzahl der Komponenten  $k$  und Größe  $|t|$  des Komponententyps:  $|t'| = k \cdot |t|$

Die Speicherbelegungsfunktion kann tatsächlich *zur Übersetzungszeit* –aus dem Deklarationsteil des C-Programms– berechnet werden! So könnten wir zum Abspeichern, z.B. des Wertes 42, in die Komponente  $a[0]$  des obigen Feldes  $a$  die folgende Befehlsfolge benutzen:

`loadc 42, loadc  $\rho(a)$ , store, alloc -1` , wobei die Adresse  $\rho(a)$  statisch bekannt ist.

Die Übersetzung der Wertzuweisung  $a[i] \leftarrow 42$ ; wird interessant, denn die `int`-Variable  $i$  erhält ihren Wert erst *zur Ausführungszeit* des Programms. Daher müssen Befehle erzeugt werden, die zuerst den aktuellen Wert von  $i$  ermitteln und dann diesen Wert zur Anfangsadresse  $\rho(a)$  addieren, um die richtige Komponente des Felds zu auswählen:

`loadc 42, loadc  $\rho(a)$ , loada  $\rho(i)$ , add, store, alloc -1`

Sei  $a$  ein Ausdruck für ein Feld von Komponenten des Typs  $t$ . Für die Anfangsadresse der Komponente  $a[e]$  muss erst die Anfangsadresse des Felds  $a$  ermittelt werden. Dann wird der Wert von  $e$  berechnet und damit die Nummer der selektierten Komponente bestimmt. Dieser Index muss mit dem Platzbedarf  $|t|$  für eine Komponente multipliziert werden, bevor er zur Anfangsadresse des Feldes addiert wird, um die Anfangsadresse des Ausdrucks  $a[e]$  zu erhalten: (Adresse von  $a[e]$ ) = (Adresse von  $a$ ) +  $|t| \cdot$  (Wert von  $e$ )

Wir erweitern deshalb die Übersetzungsfunktion  $\text{code}_A$  auf indizierte Feldausdrücke durch:

$$\text{code}_A^\rho a[e] = \text{code}_A^\rho a, \text{code}_W^\rho e, \text{loadc } |t|, \text{mul}, \text{add}$$

Ist die Adresse eines indizierten Feldausdrucks  $a[e]$  bekannt, so kann dessen Wert bestimmt werden, indem der Inhalt der adressierten Speicherzelle (mit **load** bzw. **load m**) geladen wird.

Wir könnten das Übersetzungsschema für Verbunde auch auf Felder anwenden. In der Programmiersprache C ist der Wert eines Feldes  $a$  jedoch nicht die Folge der Werte der Komponenten von  $a$ , sondern die **Anfangsadresse** von  $a$ . Denn nach der C-Philosophie wird der Wert des Felds  $a$  aufgefasst als **Zeiger** auf den Speicherbereich, in dem die Komponenten von  $a$  liegen. Der Wert des Felds  $a$  ist deshalb vom Typ  $t^*$ , falls die Komponenten von  $a$  vom Typ  $t$  sind. Repräsentiert also der Ausdruck  $e$  ein Feld, gilt:

$$\text{code}_W^{\rho} e = \text{code}_A^{\rho} e$$

## 4.2 Namenlose Daten auf der Halde

**Zeiger** und dynamische Speicherbelegung für **namenlose Daten** sind eng verwandte Konzepte in imperativen Programmiersprachen. Bisher haben wir nur **deklarierte Daten** betrachtet: In der Deklaration wird ein Name für ein Datum angegeben und diesem Namen wird statisch eine (Relativ-)Adresse zugeordnet. Mit Zeigern kann man auf namenlose Daten zugreifen und dynamisch wachsende und schrumpfende verkettete Strukturen realisieren, wobei die Daten nicht durch eine Deklaration, sondern durch die Ausführung eines Befehls erzeugt werden.

Die Semantik von C spezifiziert die **Lebensdauer** dynamisch erzeugter Daten nicht sehr präzise. Natürlich kann die Implementierung einer Programmiersprache den belegten Speicher schon vor dem Ende der Lebensdauer wieder freigeben, wenn das laufende Programm auf diese Daten nicht mehr zugreifen kann. Der Prozess der Freigabe von Speicher, der von unerreichbaren Daten belegt ist, heißt **automatische Speicherbereinigung** (*garbage collection*).

Die kellerartige Speicherbelegung passt nicht zur Lebensdauer dynamisch erzeugter Daten. Deshalb werden dynamisch erzeugte Daten nicht im Keller, sondern in einem anderen Datenspeicher, der **Halde** (*heap*), untergebracht.

Um den physischen Speicher gut auszunutzen, bekommen Keller und Halde keine festen Speicherbereiche, sondern wir legen den Keller ans untere Ende und die Halde ans obere Ende eines gemeinsamen Datenspeichers (Abb. 4.1). Natürlich dürfen sich die Speicherbereiche nicht überlappen.

Die Halde wächst bei Anlage eines Datums in Richtung Keller. Der **Haldenzeiger HP** (*heap pointer*) zeigt immer auf die unterste belegte Zelle der Halde (und wird daher vor der Programmausführung mit  $\text{maxS}+1$  initialisiert).

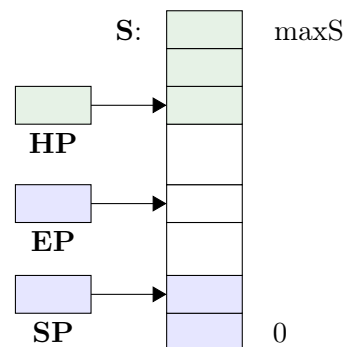


Abb. 4.1: Der Datenspeicher der C-Maschine mit Halde (grün) und Keller (blau).

Um einen **Kellerüberlauf** (*stack overflow*) festzustellen, vergleichen wir  $EP$  jetzt nicht mehr mit  $\text{maxS}$ , sondern mit  $HP$ . Die Abkürzung **testoverflow** wird daher endgültig festgelegt als:

**testoverflow**  $\equiv$  **if**  $EP \geq HP$  **then error**(„Stack Overflow“);

Der Test ist nur nötig, wenn sich  $EP$  oder  $HP$  ändern. Ein möglicher Kellerüberlauf wird bereits bei Betreten (Befehl **enter**) oder beim Verlassen (Befehl **return**) einer Funktion festgestellt.

Was kann man mit Zeiger(-Werten) tun?

- Zeiger **erzeugen**: Zeiger auf Speicherzellen setzen; sowie
- Zeiger **dereferenzieren**: durch Zeiger auf Werte von Speicherzellen zugreifen.

Technisch ist ein Zeiger nur eine Speicheradresse. In C gibt es zwei Arten, Zeiger zu erzeugen:

- durch einen Aufruf der Bibliotheksfunktion **malloc** (*memory allocation*) oder
- durch die Anwendung des **Adressoperators** **&**.

Ein Aufruf **malloc**( $e$ ) berechnet den Wert  $m$  des Ausdrucks  $e$  und liefert einen Verweis auf die unterste Zelle eines neuen Speicherabschnitts der Größe  $m$  zurück.

Ein Aufruf der Funktion **malloc** schlägt nie fehl. Selbst wenn nicht genügend Platz für das neue Datum zur Verfügung steht, liefert der Aufruf eine Adresse zurück: im Fehlerfall die Adresse 0 (besser wäre eine Fehlermeldung **Haldenüberlauf**). Ein Programmierer muss daher den Rückgabewert immer auf 0 testen, um diese Fehlersituation zu erkennen und angemessen zu behandeln.

Wir übersetzen:

$$\text{code}_W^\rho(\text{malloc}(e)) = \text{code}_W^\rho e, \text{ new}$$

Ein neues Datum auf der Halde wird mit dem Befehl **new** erzeugt (Abb. 4.2).

Der Befehl **new** erwartet oben auf dem Keller die Größe des neuen Datums und liefert die Anfangsadresse des neuen Speicherbereichs zurück. Vorher muss überprüft werden, ob genügend Speicherplatz für das neue Datum vorhanden ist. Falls nicht, liefert der Befehl **new** den Wert 0 zurück.

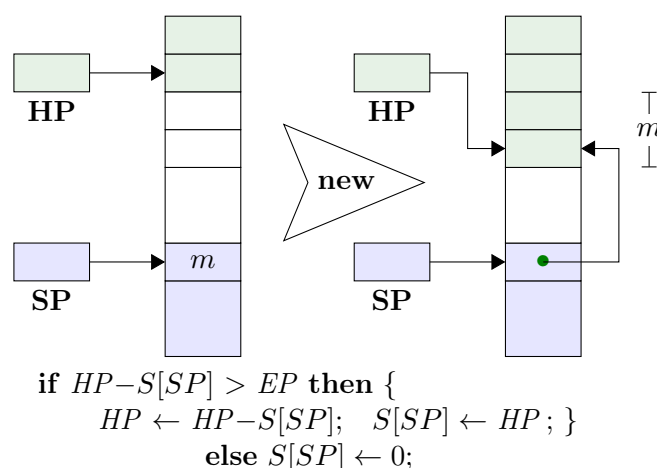


Abb. 4.2: Der Befehl **new**.

Die Anwendung **&** $e$  des **Adressoperators** auf einen Ausdruck  $e$  liefert einen Zeiger auf das Speicherobjekt von  $e$ , d.h. auf dasjenige, das an der Anfangsadresse von  $e$  beginnt und den Typ von  $e$  besitzt. Folglich ist der Wert des Ausdrucks **&** $e$  gerade die Adresse des Ausdrucks  $e$ :

$$\text{code}_W^\rho(\&e) = \text{code}_A^\rho e$$

Sei  $e$  ein Ausdruck, der zu dem Wert eines Zeigers  $p$  ausgewertet wird. Dieser Zeigerwert ist dann die Adresse des Datums, auf das der Zeiger zeigt. Dieses Datum erhalten wir, wenn wir den Zeiger  $p$  **dereferenzieren**, d.h. den Präfix-Operator **\*** anwenden. D.h., der Wert von  $e$  ist die Adresse von  $*e$ :

$$\text{code}_A^\rho(*e) = \text{code}_W^\rho e$$

**Beispiel 4.1:** Wir verfolgen das Zusammenwirken der verschiedenen Übersetzungsschemata.

Für eine Deklaration:

```
struct t { int a[7]; struct t *b; };
int i, j;
struct t *pt;
```

wollen wir folgenden Ausdruck übersetzen:

$$e' \equiv ((pt \rightarrow b) \rightarrow a)[i+1]$$

Der Operator  $\rightarrow$  bedeutet Dereferenzierung, gefolgt von einer Selektion:  $e \rightarrow c \equiv (*e).c$

Hat  $e$  den Typ  $t^*$  für eine Struktur  $t$  mit einer Komponente  $c$  und  $m = \text{offsets}(t, c)$ , so gilt:

$$\text{code}_A^\rho(e \rightarrow c) = \text{code}_A^\rho((*)e).c = \text{code}_A^\rho(*e), \text{ loadc } m, \text{ add} = \text{code}_W^\rho e, \text{ loadc } m, \text{ add}$$

Im Beispiel gilt  $\text{offsets}(t, a) = 0$ , und  $\text{offsets}(t, b) = 7$ . Die Speicherbelegungsfunktion sei  $\rho = \{(G, i) \mapsto 1, (G, j) \mapsto 2, (G, pt) \mapsto 3\}$ . Dann ergibt sich für den Ausdruck  $e'$ :

$$\text{code}_A^\rho((pt \rightarrow b) \rightarrow a)[i+1] = \text{code}_A^\rho((pt \rightarrow b) \rightarrow a), \text{code}_W^\rho(i+1), \text{loadc } 1, \text{mul}, \text{add} = \text{code}_A^\rho((pt \rightarrow b) \rightarrow a), \text{loada } 1, \text{loadc } 1, \text{add}, \text{loadc } 1, \text{mul}, \text{add}$$

Dabei ist  $\text{code}_A^p((pt \rightarrow b) \rightarrow a) = \text{code}_W^p(pt \rightarrow b), \text{loadc } 0, \text{add}$   
 $= \text{loada } 3, \text{loadc } 7, \text{add}, \text{load}, \text{loadc } 0, \text{add}$

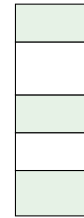
Insgesamt erhalten wir die Befehlsfolge:

**loada 3, loadc 7, add, load, ~~loadc 0, add~~, loada 1, loadc 1, add, ~~loadc 1, mul~~, add**

Ohne die systematische Ableitung wären wir kaum so leicht auf diese Befehlsfolge gekommen. Andererseits bietet sie noch Möglichkeiten zur **Optimierung**. So können wir die Addition von 0 einsparen und die Multiplikation mit 1. Das hätten wir schon bei der Codeerzeugung berücksichtigen können durch Fallunterscheidungen in den Übersetzungsschemata. Der Übersichtlichkeit halber haben wir darauf verzichtet und hoffen, dass ein Postpass-Optimierer solche lokalen Codeverbesserungen getrennt durchführt.

Damit bleibt bei der Behandlung von Zeigern nur noch die **explizite Freigabe** von Speicherbereichen zu regeln. Die Freigabe eines Speicherblocks, auf den ein Zeiger zeigt, ist problematisch, weil es weitere Zeiger auf diesen Speicherbereich geben kann. Diese würden durch die Freigabe zu **hängenden Zeigern** (*dangling references*).

Trotzdem kann die Halde nach einigen (korrekten!) Freigaben **fragmentiert** sein wie in Abb. 4.3. Die freien Speicherabschnitte können sehr unregelmäßig über die Halde verstreut sein.



Zur Speicherbereinigung benötigt das Laufzeitsystem weitere Datenstrukturen, deren Verwaltung die Aufrufe der Funktionen **malloc** oder **free** verteuern.

Abb. 4.3: Die Halde nach Freigabe einiger Blöcke (grün = belegt, weiß = frei).

Die Algorithmen zur **automatischen Speicherbereinigung** wenden verschiedene Strategien an:

- Bei Verfahren mit **Verweiszählung** (*reference counting*) berechnet für jede Zelle ein eigener Zähler die Anzahl der Verweise auf diese Zelle. Fällt dieser Zähler auf 0, so wird die Zelle wiederverwendet. Diese Verfahren haben den Nachteil, dass zyklische Strukturen zu Komplikationen und unterschiedlich große Zellen zu Effizienzeinbußen führen.

Die nachfolgenden Algorithmen unterbrechen die eigentliche Programmausführung, wenn kein Speicherplatz mehr zugewiesen werden kann (Vorsicht bei Echtzeitanwendungen!):

- **Markieren und Löschen** (*mark and sweep*) erfolgt in zwei Phasen: In der ersten Phase werden alle lebenden (noch verwendeten) Zellen markiert. In der Löschphase wird die gesamte Halde durchlaufen, um die unmarkierten Zellen zu finden. Danach wird oft noch eine Kompaktierung durchgeführt, so dass alle lebenden Zellen, aber auch der freie Speicher je einen zusammenhängenden Bereich bilden. Die Markierungs- und die Kompaktierungsphase erfordert zwar nur je ein Durchlaufen aller lebenden Zellen; die Laufzeit des Algorithmus ist wegen der Löschphase aber proportional zur Größe der Halde.
- **Lebende Objekte kopieren** (*copy collector*): Dazu braucht man zwei gleich große Speicherbereiche. Ist der benutzte Speicherbereich voll, so werden alle lebenden Zellen in den anderen Speicherbereich kopiert. Die Laufzeit ist daher proportional zur Größe des Speicherplatzes für die lebenden Zellen. Nach dem Kopieren ist der freie Haldenbereich zusammenhängend; daher können Speicherzellen effizient allokiert werden, indem der Zeiger auf das nächste freie Speicherwort einfach erhöht wird.

In unserem minimalen Compiler tun wir bei einer Speicherfreigabe – nichts!

Diese Implementierung ist korrekt, aber natürlich nicht speicheroptimal. Der Ausdruck  $e$  muss trotzdem ausgewertet werden, da er Nebenwirkungen (z.B. Wertzuweisungen) enthalten kann.

$$\text{code}^p(\text{free}(e);) = \text{code}_W^p e, \text{alloc } -|e|$$

### 4.3 Konzepte objektorientierter Programmiersprachen

Softwaresysteme werden immer größer und komplexer. Damit wächst die Notwendigkeit, die Entwicklung dieser Systeme effizienter und transparenter zu machen. Eine Hoffnung besteht darin, auch Softwaresysteme –wie Hardwaresysteme (und viele Produkte des täglichen Lebens, z.B. Autos, Waschmaschinen)– aus vorgefertigten Standardbausteinen zusammenzusetzen.

Dieser Hoffnung versucht man u.a. durch folgende Ideen näherzukommen:

- Modularisierung,
- Wiederverwendbarkeit von Modulen,
- Erweiterbarkeit von Modulen und
- Abstraktion.

Objektorientierte Sprachen bieten hier neue Möglichkeiten. Objektorientierung wird deshalb heute als ein wesentliches Paradigma angesehen, um die Komplexität von Softwaresystemen zu beherrschen. Die wichtigsten Konzepte objektorientierter Sprachen sind:

- Objektorientierte Sprachen bieten als weitere Modularisierungseinheit **Objektklassen**. Objektklassen können Daten und Funktionen auf diesen Daten kapseln.
- Das **Vererbungskonzept** vereinfacht die Erweiterung oder Variation von Objektklassen.
- Das Typsystem objektorientierter Sprachen nutzt das Vererbungskonzept: Erbende Klassen werden **Teiltypen** der Basisklassen; ihre Objekte können an (fast) allen Stellen benutzt werden, an denen Objekte der Basisklasse zulässig sind.
- **Vererbungshierarchien** führen unterschiedliche Abstraktionsebenen in Programme ein. Dies erlaubt, an verschiedenen Stellen innerhalb eines Programms oder Systems auf unterschiedlichen Abstraktionsstufen zu arbeiten.
- **Abstrakte Klassen** können in Spezifikationen verwendet und durch schrittweise Vererbung verfeinert und dann implementiert werden. Der Übergang zwischen Spezifikation, Entwurf und Implementierung wird dadurch fließend.
- **Informationskapselung** trennt die abstrakte Sicht auf die Klassenbedeutung von der konkreten Sicht auf ihre Implementierung.
- **Generizität** erlaubt, Klassendefinitionen zu parametrisieren. Dadurch können Datenstrukturen wie Listen, Keller, Warteschlangen und Mengen und ihre Algorithmen unabhängig vom Datentyp ihrer Elemente implementiert werden.

Objektorientierte Sprachen sind mit imperativen Sprachen viel enger verwandt als mit funktionalen oder gar logischen Programmiersprachen. Sie benutzen dasselbe Ausführungsmodell: ein von-Neumann-Rechner, bei dem unter expliziter Programmkontrolle ein komplexer Zustand verändert wird. Objektorientierte Sprachen sind daher bis auf weiteres imperative Sprachen mit zusätzlichen Konzepten für Abstraktion und Modularisierung.

#### 4.3.1 Objekte

Imperative Programmiersprachen bieten eine **funktionale Abstraktion** an: eine komplexe Berechnung kann in eine Funktion oder Prozedur „verpackt“ und durch Aufruf aktiviert werden. Die typische Modularisierungseinheit imperativer Sprachen ist daher die Funktion (Prozedur). Solange die Komplexität der Daten gegenüber der Komplexität der Programmabläufe vernachlässigbar ist, ist dies angemessen. Aber für Aufgaben, die komplexe Datenstrukturen erfordern, sollte auch eine **Datenabstraktion** möglich sein, also die Kapselung der Datenstrukturen und der auf diesen Strukturen arbeitenden Funktionen zu einer Einheit.

Grundkonzept objektorientierter Sprachen ist das **Objekt**. **Merkmale** eines Objekts sind seine **Attribute** (lokale Daten) und **Objekt-Methoden** (darauf anwendbare Funktionen). Die Werte der Attribute eines Objekts bestimmen seinen Objektzustand. Ein Objekt kapselt in seinem Zustand Daten und in seinen Methoden die auf diesen Daten durchführbaren Operationen: Die Werte der

Attribute eines Objekts werden nur(!) durch den Aufruf von Methoden dieses Objekts geändert. Die wichtigste Basisoperation objektorientierter Sprachen ist die **Aktivierung einer Methode**  $f$  für ein Objekt  $o$  (geschrieben  $o.f$ ). Innerhalb von  $f$  kann man mit dem Schlüsselwort **this** auf dieses Objekt  $o$  Bezug nehmen.

Im Vordergrund steht dabei das Objekt; die Methode ist nur Bestandteil des Objekts. Diese Objektorientierung hat der Sprachklasse ihren Namen gegeben.

### 4.3.2 Objektklassen

Für eine sichere und effiziente Programmentwicklung müssen Inkonsistenzen und Fehler in Programmen frühzeitig und zuverlässig erkannt werden. Übersetzer nutzen (statische) Typinformation zur Überprüfung.

Statische –also dem Übersetzer bekannte oder daraus ableitbare– Typinformation besteht aus Angaben zu den im Programm verwendeten Namen. Sie legen fest, dass die an diese Namen gebundenen Laufzeitobjekte den angegebenen Typ besitzen. Ein **Typ** steht für eine Menge zulässiger Werte und bestimmt, welche Operationen auf diese Werte angewendet werden dürfen.

Übersetzer können Typinformation z.B. nutzen, um

- Inkonsistenzen zu erkennen,
- Typkonversionen einzufügen und
- Operatoren eindeutig zu bestimmen,
- effizienteren Code zu erzeugen.

Einige objektorientierte Sprachen (z.B. Smalltalk, Python, PHP, Ruby) knüpfen den Typ eines Objekts daran, dass bestimmte Methoden oder Attribute vorhanden sind („*duck typing*“). Diese dynamische Typprüfung ist flexibel, reduziert aber die Möglichkeit, Fehler schon zur Übersetzungszeit zu finden.

Deshalb erweitern Sprachen wie C++, Java oder C# die von imperativen Sprachen wie Pascal oder C bekannten statischen Typkonzepte. Ihre Typen heißen **Objektklassen** oder kurz: **Klassen**.

Eine Objektklasse legt Attribute und Methoden fest, die ein Objekt mindestens haben muss, um zu dieser Klasse zu gehören. Für Attribute wird ihr Typ und für Methoden ihr Prototyp (Typen für Parameter und Rückgabewert) vorgegeben. In einigen objektorientierten Sprachen (z.B. Eiffel) kann die Semantik einer Methode noch genauer festgelegt werden, z.B. durch Beschreibung von **Voraussetzung** (*precondition*) und **Effekt** (*postcondition*). Oft definiert die Klasse auch die Methoden; diese Definitionen können jedoch evtl. überschrieben werden. Außerdem können u.U. Objekte der Klasse angehören, die neben den geforderten Merkmalen noch zusätzliche besitzen.

Die Objektklasse ist das Konzept zur Datenabstraktion in objektorientierten Sprachen. Sie erzeugt –durch gleichnamige Konstruktoren– ihre Objekte, die **Instanzen** dieser Klasse.

### 4.3.3 Vererbung und Teiltypen (Inklusionspolymorphie)

**Vererbung** (*inheritance*) bedeutet die Übernahme aller Merkmale einer Klasse  $K$  in eine neue Klasse  $U$  (ohne den Code zu kopieren!). Die Klasse  $U$  kann zusätzliche Merkmale definieren und geerbte Methoden evtl. überschreiben.

Erbt  $U$  von  $K$ , dann heißt  $U$  eine von  $K$  **abgeleitete Klasse** oder **Unterklasse** von  $K$ ;  $K$  heißt **Basisklasse** oder **Oberklasse** von  $U$ .

Die Vererbung vereinfacht Erweiterungen und Variationen. Durch Bildung von **Vererbungshierarchien** können Klassenbibliotheken strukturiert und Abstraktionsstufen gebildet werden.

Vererbung erleichtert, Teile einer bestehenden Implementierung zu nutzen, zu erweitern und sie bei Bedarf lokal –durch Überschreiben von Methoden– an spezielle Anforderungen anzupassen.

When I see  
a bird that  
walks like  
a duck and  
swims like  
a duck and  
quacks like  
a duck, I  
call that  
bird a duck.  
J.W. Riley

Zusätzlich können wir **abstrakte Klassen** definieren. Abstrakte Klassen enthalten undefinierte Methoden und besitzen keine eigenen Instanzen, d.h. alle ihre Objekte stammen aus echten Unterklassen. Programmiersprachen erhalten so ein Äquivalent für abstrakte Begriffe und unterschiedliche Abstraktionsstufen in natürlichen Sprachen. Was auf einer höheren Abstraktionsstufe formuliert werden kann, hat einen größeren Anwendungsbereich und ist öfter wiederverwendbar.

Typisierte objektorientierte Sprachen besitzen eine Vererbungshierarchie in ihrem Typsystem. Erbt eine Klasse  $U$  öffentlich von der Klasse  $K$ , dann ist der  $U$  zugeordnete Typ ein **Teiltyp** des Typs zu  $K$ . Jedes Objekt eines Teiltyps ist automatisch auch Element der Oberklasse; eine erbende Klasse  $U$  wird **Unterklasse** der beerbten Klasse  $K$ . Daher dürfen an einer Eingabeposition (Funktionsparameter, rechte Seiten von Zuweisungen) oder als Funktionsrückgabewert Objekte eines beliebigen Teiltyps des angegebenen Typs auftreten (**Teiltypregel**).

Dieses nützliche Prinzip hat aber Konsequenzen: Objekte der Unterklassen von  $K$  können über zusätzliche Merkmale verfügen und brauchen dann mehr Speicherplatz als  $K$ -Objekte!

- Nutzt die Programmiersprache *Verweise* auf Objekte statt der Objekte (z.B. Eiffel, Java), so ist die Teiltypregel problemlos anwendbar: Verweise haben stets die gleiche Speichergröße – unabhängig von der Klasse des Objekts, auf das sie zeigen.
- Sonst muss die Programmiersprache genau zwischen Objekten und Verweisen auf Objekte unterscheiden: C++ ruft deshalb für Parameter, dessen Typ eine Unterklasse  $U$  ist, einen (modifizierten) *copy*-Konstruktor  $U(\text{const } U\& x)$  der Klasse  $U$  auf, der die zusätzlichen Merkmale weglässt und so für eine **Typanpassung** sorgt.

Wir nennen die Objekte von  $K$ , die nicht auch Objekte einer echten Unterklasse sind, **unmittelbare Objekte** von  $K$ . Entsprechend nennen wir  $K$  den **unmittelbaren Typ** der unmittelbaren  $K$ -Objekte. Damit besitzt jedes Objekt einen eindeutig bestimmten unmittelbaren Typ: den kleinsten (in der Klassenhierarchie tiefsten) Typ, zu dem das Objekt gehört. Es ist darüber hinaus Element jeder Oberklasse seines unmittelbaren Typs.

Wegen der Teiltypregel akzeptieren Methoden und Funktionen in objektorientierten Sprachen auf einer Parameterposition (Verweise auf) Objekte von verschiedenen unmittelbaren Typen und damit von unterschiedlicher Struktur. Dies ist eine Form von **Polymorphie**.

Die Teiltypregel und die Möglichkeit, dass Klassen eine geerbte Methode überschreiben dürfen, hat die für Programmierer interessante und für Übersetzer wichtige Konsequenz, dass sich die bei einem Methodenaufwurf  $e.f(\dots)$  aufgerufene Methode  $f$  nach dem unmittelbaren Typ des Objekts richten muss, zu dem sich der Ausdruck  $e$  zur Laufzeit auswertet. Damit muss der Übersetzer Code für die Aktivierung einer Methode erzeugen, die er zum Zeitpunkt der Übersetzung u.U. noch nicht kennt.<sup>6</sup> Diese Auswahl der Methoden zur Laufzeit (*dynamic lookup*) ist offenbar eine Form von **dynamischer Bindung**.

Diese **dynamische Auswahl der Methoden** bewirkt, dass das Objekt den auszuführenden Code (für eine Methode) bestimmt und damit, wie es auf einen Methodenaufwurf („Nachricht“) reagiert. Die gleiche Nachricht an unterschiedliche Objekte kann daher zu unterschiedlichen Ergebnissen („Antworten“) führen.

#### 4.3.4 Generizität (parametrische Polymorphie)

Streng typisierte Sprachen zwingen häufig zu einer Reimplementierung der gleichen Funktion für verschiedene Typen, etwa von den Parametern. Diese mehrfachen Funktionsinstanzen erschweren die Implementierung, machen Programme unübersichtlich und ihre Wartung aufwändig.

<sup>6</sup> Bei einer **Multimethode** (*multiple dispatch*) wird die zugehörige Implementierung einer Methode zur Laufzeit nicht anhand des Typs eines Objekts, sondern der Typen mehrerer Objekte (z.B. aller Parameter) ermittelt.

Das auf Vererbung beruhende Typkonzept objektorientierter Sprachen erspart uns oft eine Vervielfachung von Funktionsimplementierungen. Für eine wichtige Problemklasse führt Vererbung aber nicht zu eleganten Lösungen. Die Implementierung von Datenstrukturen für Behälter wie Listen, Keller oder Warteschlangen haben einen natürlichen Parameter: den Typ ihres Inhalts. Durch **Generizität** (*genericity*) können wir für solche Datenstrukturen und ihre Methoden eine Mehrfachimplementierung vermeiden. Sie erlaubt, **Typdefinitionen** (und Funktionsdefinitionen) zu **parametrisieren**; als Parameter sind selbst wieder (evtl. eingeschränkte) Typen zugelassen.

Z.B. kann eine einzige Definition für die parametrisierte Klasse **list**<*t*> Listen mit beliebigem Elementtyp *t* beschreiben. Listen mit spezifischem Typ werden durch **Instantiierung** der generischen Klasse erzeugt: **list**<**int**> bezeichnet eine Liste, deren Elemente vom Typ **int** sind.

Generizität wird von einigen objektorientierten Sprachen (z.B. C++, Java) unterstützt; sie ist aber keine Erfindung objektorientierter Sprachen, sondern ein wesentliches Merkmal moderner funktionaler Programmiersprachen wie OCaml oder Haskell. Sogar die imperative Sprache Ada hatte bereits ein sehr ausgefeiltes Konzept von Generizität.

### 4.3.5 Informationskapselung und Programmstruktur

Die meisten objektorientierten Sprachen stellen Konstrukte zur Verfügung, mit denen die Merkmale einer Klasse als **privat** oder **öffentlich** klassifiziert werden können.

Private Merkmale sind in bestimmten Kontexten unsichtbar oder zumindest nicht zugreifbar. Manche objektorientierte Sprachen unterscheiden verschiedene **Sichtbarkeitskontexte**, z.B. innerhalb der Klasse, in abgeleiteten Klassen, in fremden Klassen, in bestimmten Klassen. Sprachkonstrukte oder Regeln können festlegen, in welchen Kontexten welche Merkmale sichtbar bzw. lesbar/schreibbar oder aufrufbar sind.

Diese Informationskapselung ermöglicht eine **Abstraktion** (*abstraction*) vom internen Aufbau der Objekte. Meist existiert für Objekte eine **Schnittstelle** (*interface*) aus öffentlichen Funktionen, über die die versteckten internen Daten des Objekts gelesen und verändert werden können.

Der Übersetzer realisiert solche Konstrukte einfach, indem er unsichtbare Namen nicht in die Adressumgebung aufnimmt. Wir gehen daher nicht weiter auf die Informationskapselung ein, obwohl sie wichtig ist für eine klare Trennung zwischen der abstrakten Sicht der Klassenbedeutung und der konkreten Sicht ihrer Implementierung.

Die Informationskapselung hat auch Auswirkungen auf die **Programmstruktur**:

**Beispiel 4.2:** Man möchte für alle Angehörigen der Universität Informationen anzeigen oder das Gehalt auszahlen. Prozedural würde man zwei Funktionen schreiben:

```

info(x) = case type(x) of
    Professor:  ["zeige Info über Professor"];
    Mitarbeiter: ["zeige Info über Mitarbeiter"];
    Hilfskraft: ["zeige Info über Hilfskraft"];
end;

und      bezahle(x) = case type(x) of
    Professor:  ["zahle Professor"];
    Mitarbeiter: ["zahle Mitarbeiter weniger"];
    Hilfskraft: ["zahle Hilfskraft viel weniger"];
end;
```

In einem objektorientierten Programm gibt es Klassen für Professoren, Mitarbeiter und Hilfskräfte und jede Klasse enthält die Methoden **info** und **bezahle**:



```

class Professor = method info = ["zeige Info über Professor"];
                  method bezahle = ["zahle Professor"];

class Mitarbeiter = method info = ["zeige Info über Mitarbeiter"];
                    method bezahle = ["zahle Mitarbeiter weniger"];

class Hilfskraft = method info = ["zeige Info über Hilfskraft"];
                   method bezahle = ["zahle Hilfskraft viel weniger"];

```

Beide Vorgehensweisen lassen sich durch folgende Tabelle gut vergleichen:

Operation	Professor	Mitarbeiter	Hilfskraft
info	info_Professor	info_Mitarbeiter	info_Hilfskraft
bezahle	bezahle_Professor	bezahle_Mitarbeiter	bezahle_Hilfskraft

In konventionellen Programmiersprachen wird der Code *zeilenweise* in einer Funktion gruppiert, die auf allen auftretenden Arten von Daten arbeitet. In objektorientierten Programmiersprachen wird der Code *spaltenweise* gebündelt, in dem die einzelnen Funktionsteile mit den Daten, auf denen sie arbeiten sollen, gruppiert werden.

In der funktionsorientierten Organisation konventioneller Programmiersprachen kann man leicht neue Operationen hinzufügen (z.B. `zahle_Weihnachtsgeld`), aber für einen neuen Datentyp (z.B. Junior-Professor) muss jede Funktion geändert werden. In der datenorientierten Organisation objektorientierter Sprachen kann man einen neuen Datentyp leicht als neue Klasse hinzufügen, dagegen muss für eine neue Operation oft jede vorhandene Klasse geändert werden.

In objektorientierten Sprachen könnte man eine abstrakte Oberklasse Uni-Angehörige einführen und am Zahhtag eine einfache Schleife (Iterator) mit der Methode `bezahle` ausführen über alle Elemente dieser Klasse (ohne Fallunterscheidung!), da die Auswahl der Methode ja dynamisch nach dem unmittelbaren Typ des Objekts geschieht.

## 4.4 Speicherorganisation für Objekte

Wir erweitern jetzt unsere Implementierung von C auf Klassen, Objekte und ihre Merkmale. Als Beispiel für eine objektorientierte Sprache wählen wir eine einfache Teilmenge von C++.

**Klassen** werden aufgefasst als Erweiterungen von Verbundtypen; sie enthalten **Attribute**, also Datenfelder, und **Methoden**, die auch als **virtual** deklariert und dann –in Unterklassen– überschrieben werden können, sowie **Konstruktoren**, die neu angelegte Objekte initialisieren.

**Beispiel 4.3:** Die Klassendefinition für `list` enthält zwei Attribute `info` und `next` der Typen `int` bzw. `list *`.

Ein Konstruktor initialisiert neue Listenobjekte und eine Methode `last` liefert den Inhalt des Attributs `info` des letzten über `next`-Verweise erreichbaren `list`-Objekts zurück.

Diese –als **virtual** gekennzeichnete– Methode darf in Unterklassen neu definiert werden.

```

class list {
    int info;
    list * next;
    list (int x) {info ← x; next ← null;}
    virtual int last() {
        if next = null
            then return info;
        else return next→last();
    }
};

```

Zur Vereinfachung verzichten wir auf Angaben zur Sichtbarkeit. Für ein lauffähiges C++-Programm könnten wir alle Attribute oder Methoden der Klasse z.B. als **public** deklarieren. Auch vor der Oberklasse einer Klassendefinition müsste man eine Qualifizierung, z.B. **public**, einfügen, um Merkmale der Oberklasse außerhalb der erbenden Klasse verwenden zu können.

In **Java** liegen alle Objekte auf der Halde.

In **C++** können Objekte –wie Verbunde– auch direkt auf dem Keller abgelegt werden.

Bei der Implementierung von **Objekten** wollen wir nur diejenigen Dinge im Objekt selbst anlegen, die sich von Objekt zu Objekt unterscheiden. Nicht-überschreibbare Methoden ergeben sich aus dem Typ des Objekts und brauchen nicht beim Objekt abgespeichert werden.

Adressen von Attributen und überschreibbaren Methoden können aber nicht immer zur Übersetzungszeit ermittelt werden.

- Um Attribute einer Klasse in allen Unterklassen gleich adressieren zu können, erhalten neue Attribute einer Unterklasse *größere* Relativadressen als die Attribute der Oberklasse.
- Die zutreffende Implementierung von überschreibbaren Methoden ergibt sich erst zur Laufzeit aus dem unmittelbaren Typ des Objekts. Statt die Anfangsadressen dieser Methoden in jedem Objekt abzuspeichern, wird für jede Klasse  $K$  vor der Programmausführung eine Tabelle  $\text{üM}_K$  angelegt, die die **Anfangsadressen aller überschreibbaren Methoden** für *unmittelbare*  $K$ -Objekte enthält. Diese Objekte erhalten dann (an der Relativadresse 0) einen Verweis auf diese Tabelle  $\text{üM}_K$ . Überschreibbare Methoden einer Unterklasse erhalten ebenfalls eine *größere* Relativadresse als die überschreibbaren Methoden der Oberklasse.

Ein Objekt der Klasse **list** enthält einen Verweis auf die Tabelle der überschreibbaren Methoden der Klasse **list**, gefolgt von den Zellen für die Attribute *info* und *next* (Abb. 4.4).

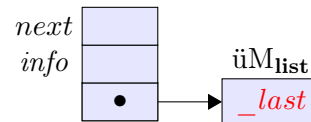


Abb. 4.4: Ein Objekt der Klasse **list**.

**Beispiel 4.4:** Nun sei eine Klasse **mylist** als Unterklasse von **list** definiert:

```
class mylist : list {
    int moreInfo;
    virtual int length() {
        if next = null then return 1;
        else return 1+next→length();
    }
};
```

Objekte der Klasse **mylist** enthalten einen Verweis auf die Tabelle der überschreibbaren Methoden der Klasse **mylist**. Diese unterscheidet sich von der Tabelle der Oberklasse durch die zusätzliche Methode *length*. Außerdem wird hinter den beiden Feldern für die Attribute der Oberklasse **list** ein weiteres Feld für das neue Attribut *moreInfo* bereitgestellt (Abb. 4.5).

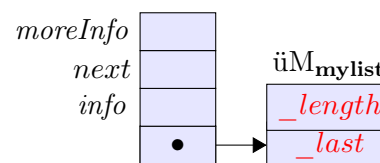


Abb. 4.5: Ein Objekt der Unterklasse **mylist**.

Eine Methode fassen wir als eine Funktion mit einem zusätzlichen *impliziten* ersten Parameter auf: das aktuelle Objekt. Der erste Parameter steht im Keller direkt unter den organisatorischen Zellen. Allerdings kennen wir zur Übersetzungszeit die Größe des aktuellen Objektes nicht. Über das Schlüsselwort **this** soll innerhalb des Methodenrumpfs auf das aktuelle Objekt zugegriffen werden können. Deshalb übersetzen wir das aktuelle Objekt *by reference* und finden dann dessen Adresse stets an der Relativadresse  $-3$  (bzgl. des Rahmenzeigers FP). Dann können wir das Schlüsselwort **this** einheitlich übersetzen und es bezeichnet den *Verweis* auf das aktuelle Objekt:

$\text{code}_W^\rho(\text{this}) = \text{loadr } -3$

Statt **this** im aktuellen Kellerrahmen abzuspeichern, hätten wir ein Register **COP** (*current*

*object pointer*) einführen können. Dieses Register müsste dann vor jedem Aufruf einer nicht-überschreibbaren Methode gerettet und nach Rückkehr aus solchem Aufruf restauriert werden. Wir bleiben lieber nah an der Architektur der C-Maschine.

Jede Klasse  $K$  erhält eine eigene Adressumgebung  $\rho_K$ , die jeden in der Klasse  $K$  sichtbaren Namen  $x$  auf eine –um ein Etikett erweiterte– Relativadresse abbildet. Wir unterscheiden folgende Arten von Namen durch Etiketten:

globale Variable	$(G, a)$
lokale Variable, formale Parameter	$(L, a)$
Attribut	$(A, a)$
nicht-überschreibbare Methode	$(N, a)$
überschreibbare Methode	$(V, a)$

Für eine nicht-überschreibbare Methode  $x$  enthält  $\rho_K$  die Anfangsadresse des Codes für  $x$ . Für eine überschreibbare Methode  $x$  liefert  $\rho_K$  die Relativadresse innerhalb der Tabelle  $\text{üM}_K$ , wo die Anfangsadresse von  $x$  abgelegt ist.

Für die anderen Arten von Variablen wird folgender Code zur Berechnung der Adresse erzeugt:

$$\text{code}_A^\rho x = \begin{cases} \text{loadc } a & , \text{ falls } \rho(x) = (G, a) \\ \text{loadr } a & , \text{ falls } \rho(x) = (L, a) \\ \text{code}_W^\rho(\text{this}), \text{ loadc } a, \text{ add} & , \text{ falls } \rho(x) = (A, a) \end{cases}$$

Attribute des aktuellen Objekts werden also über den Verweis **this** adressiert.

Zur Optimierung der Adressierung von Objektattributen können wir Spezialbefehle einführen:

**loadmc**  $q \equiv \text{loadr } -3, \text{ loadc } q, \text{ add}$

**loadm**  $q \ m \equiv \text{loadmc } q, \text{ load } m$

**storem**  $q \ m \equiv \text{loadmc } q, \text{ store } m$  , wobei wir ein Argument  $m = 1$  weglassen.

## 4.5 Methodenaufruf

Ein Methodenaufruf  $e_1.f(e_2, \dots, e_n)$  wird wie ein Funktionsaufruf behandelt, dem als zusätzlicher erster Parameter ein Verweis auf das Objekt  $o$  übergeben wird;  $o$  ist der Wert des Ausdrucks  $e_1$ . Ein Methodenaufruf  $f(e_2, \dots, e_n)$  (ohne Objekt) steht für den Aufruf:

$$\text{this} \rightarrow f(e_2, \dots, e_n) \quad \text{bzw.} \quad (*\text{this}).f(e_2, \dots, e_n)$$

Zur Vereinfachung betrachten wir hier keine optionalen Parameter und nehmen an, dass der Platz der aktuellen Parameter (incl. des übergebenen Objektzeigers) für den Rückgabewert ausreicht.

Sei  $K$  die (statisch bekannte) Klasse des Ausdrucks  $e_1$ . Den Methodennamen  $f$  werten wir bzgl. der Adressumgebung  $\rho_K$  der Klasse  $K$  aus. Das Objekt  $o$ , zu dem sich  $e_1$  auswertet, wird **by reference** übergeben. Daher wird für den Parameter  $e_1$  Code zur Berechnung der Adresse erzeugt und nicht zur Berechnung des Wertes wie für die übrigen Parameter:

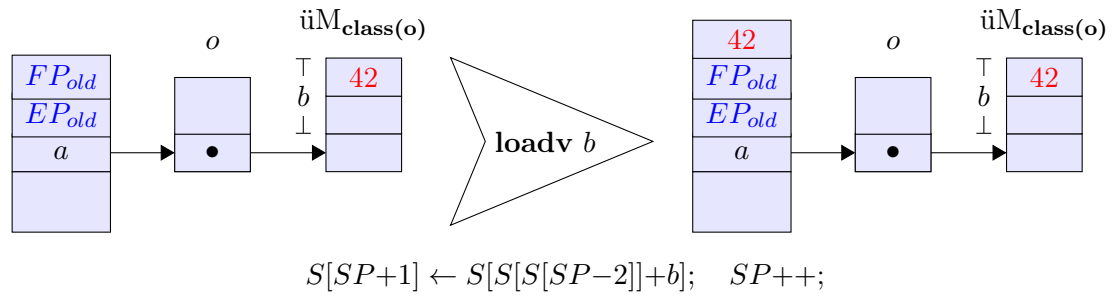
$$\text{code}_W^\rho e_1.f(e_2, \dots, e_n) = \text{code}_W^\rho e_n, \dots, \text{code}_W^\rho e_2, \text{code}_A^\rho e_1, \text{mark}, \text{code}_A^{\rho_K} f, \text{call}$$

$\text{code}_A^{\rho_K} f$  soll die Anfangsadresse der zugehörigen Implementierung der Methode  $f$  liefern:

$$\text{code}_A^{\rho_K} f = \begin{cases} \text{loadc } \_f & , \text{ für eine nicht-überschreibbare Methode } f \text{ mit } \rho_K(f) = (N, \_f) \\ \text{loadv } b & , \text{ für eine überschreibbare Methode } f \text{ mit } \rho_K(f) = (V, b) \end{cases}$$

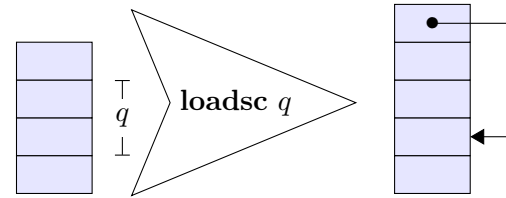
Für eine überschreibbare Methode kann die anzuspringende Codeadresse erst zur Laufzeit ermittelt werden. Aber bei allen Unterklassen von  $K$  steht die Adresse für  $f$  an der gleichen Stelle in deren Tabelle  $\text{üM}$  der überschreibbaren Methoden!  $f$  wird also indirekt über Objekt  $o$  aufgerufen.

Der Befehl **loadv**  $b$  berechnet aus der Relativadresse  $b$  von  $f$  innerhalb des Objekts  $o$ , zu dem sich  $e_1$  auswertet, die Anfangsadresse der gewünschten Implementierung von  $f$  (Abb. 4.6).

Abb. 4.6: Der Befehl **loadv b**.

Die Anfangsadresse  $a$  des Objekts  $o$  steht –als erster Parameter– im Keller direkt unter den organisatorischen Zellen (nach dem Befehl **mark**, der die ersten beiden organisatorischen Zellen geladen hat) an der Adresse  $SP-2$ . Mit ihr wird die Anfangsadresse der Tabelle  $\ddot{u}M$  der virtuellen Methoden des Objekts  $o$  ermittelt. In der Tabelle steht die Anfangsadresse der Methode an der Adresse  $b$ . Der Befehl **loadv b** adressiert das Objekt  $o$  *relativ zum SP*, da der FP für den neuen Rahmen erst im anschließenden Befehl **call** gesetzt wird.

Zunächst lädt der Befehl **loadsc q** die *Adresse*  $-q$  relativ zum Kellerzeiger SP (Abb. 4.7). Damit wird der Befehl **loads q** implementiert, der den *Inhalt* der Speicherzelle  $-q$  bzgl. SP lädt:  
**loads q**  $\equiv$  **loadsc q, load**



$$S[SP+1] \leftarrow SP - q; \quad SP++;$$

Abb. 4.7: Der Befehl **loadsc q**.

Der Befehl **loadv b** wird dann durch die Folge einfacherer Befehle realisiert:

**loadv b**  $\equiv$  **loads 2, load, loadc b, add, load**

**Beispiel 4.5:** Den rekursiven Aufruf  $next \rightarrow last()$  im Rumpf der überschreibbaren Methode **last** der Klasse **list** übersetzen wir zu:

mit der Adressumgebung  $\rho_{list} = \{info \mapsto (A, 1), next \mapsto (A, 2), list \mapsto (N, \_list), last \mapsto (V, 0)\}$

Der Befehl **loadm 2** ermittelt die Adresse des Ausdrucks  $(*next)$ , d.h. den Wert von  $next$ .

Hier bietet sich ein neuer Spezialbefehl an:

**callv b**  $\equiv$  **mark, loadv b, call**.

## 4.6 Definition von Methoden

Die Definition einer Methode  $f$  einer Klasse  $K$  ist von der Form:  $t \ f(t_2 \ x_2, \dots, t_n \ x_n) \ \{ \text{ss} \}$

Außer, dass wir Methoden als Funktionen mit einem zusätzlichen ersten Parameter auffassen, der den Verweis auf das aktuelle Objekt  $o$  enthält, und wir innerhalb des Rumpfs  $\text{ss}$  der Methode mit dem Schlüsselwort **this** auf diesen Verweis zugreifen können, ändert sich nichts am Übersetzungsschema für Methoden in C++ gegenüber Funktionen in C.

**Beispiel 4.6:** Die Implementierung der Methode **last** der Klasse **list** liefert:

**\_last:** **alloc 0, enter 6, loadm 2, loadc 0, eq, jumpz A, loadm 1, storer -3, return 3,**  
**A:** **loadm 2, callv 0, storer -3, return 3**

Der Unterschied gegenüber C betrifft nur die Befehle **loadm** / **storem**, um auf Attribute zuzugreifen, und den Befehl **callv**, der die Anfangsadresse der überschreibbaren Methode nutzt.

## 4.7 Verwendung von Konstruktoren

In objektorientierten Sprachen definiert man Konstruktoren für eine Klasse  $K$ , um neue Objekte zu initialisieren. Der Konstruktor trägt denselben Namen wie die Klasse.



Klasse **list** liefert mit  $\rho_{list} = \{info \mapsto (A, 1), next \mapsto (A, 2), list \mapsto (N, \_list), last \mapsto (V, 0)\}$ :  
`_list: alloc 0, enter 3, loadc _üMlist, storem 0, alloc -1, loadr -4, storem 1, alloc -1,`  
`loadc 0, storem 2, alloc -1, return 4`

Ein Konstruktor kann **Konstruktoren der Oberklasse**  $O$  aufrufen. In C++ steht der Aufruf im Kopf der Konstruktor-Deklaration:

$$K(t_2\ x_2, \dots, t_m\ x_m) : O(e_2, \dots, e_n)\{ss\}$$

Vor der Auswertung des Rumpfs  $ss$  muss der Konstruktor der Oberklasse aufgerufen werden. Da dieser für dasselbe Objekt aufgerufen wird wie der aufrufende Konstruktor, wird auch ihm der Verweis auf das aktuelle Objekt übergeben. Seine aktuellen Parameter werden ausgewertet relativ zur Adressumgebung der Klasse  $K$ , erweitert um die formalen Parameter des Konstruktors der Oberklasse. Da der Konstruktor der Oberklasse evtl. seine eigenen Versionen der überschreibbaren Methoden aufruft, warten wir mit der Initialisierung des Tabellenverweises, bis der Konstruktor der Oberklasse ausgeführt wurde:

$$\text{code}^{\rho} K(t_2\ x_2, \dots, t_m\ x_m) : O(e_2, \dots, e_n)\{ss\} = \_K: \text{alloc } l, \text{enter } k, \text{code}_W^{\rho_1} e_n, \dots, \text{code}_W^{\rho_1} e_2, \\ \text{loadr } -3, \text{calld } \_O, \text{initVirtual } K, \text{code}^{\rho'} ss, \text{return } r$$

wobei  $\rho_1$  die Adressumgebung zur Auswertung der aktuellen Parameter des Konstruktors der Oberklasse und  $\rho'$  die Adressumgebung innerhalb des Konstruktors  $K$  sind.

Jeder Konstruktor verwendet dann die Versionen virtueller Methoden, die in seiner Klasse gelten. Daher wird im neuen Objekt der Verweis auf die Tabelle der überschreibbaren Funktionen erst *nach* dem Aufruf der Konstruktoren der Oberklassen angelegt.

Java macht es anders: Dort kann der Konstruktor der Oberklasse schon auf die Methoden der Unterklasse zugreifen; die Tabelle der virtuellen Methoden wird dort *vor* dem Konstruktor-Aufruf für die Oberklasse angelegt.

## 4.9 Definition von Klassen

Bei der Übersetzung einer Klassendefinition **class cname** muss die Tabelle  $\_üM_{\text{cname}}$  für die Anfangsadressen der überschreibbaren Methoden angelegt werden: Hat die Klasse eine Oberklasse, so wird zunächst eine Kopie der entsprechenden Tabelle der Oberklasse angelegt, sonst eine leere Tabelle. Für jede neue überschreibbare Methode der Klasse wird diese Tabelle erweitert, für jede redefinierte Methode wird die Adresse überschrieben.

Die Anfangsadresse  $\_üM_{\text{cname}}$  der Tabelle  $\_üM_{\text{cname}}$  wird als globale Adresse in die Adressumgebung aufgenommen.

## 4.10 Ein kurzer Blick: Mehrfachbeerung

Einige Programmiersprachen (z.B. C++, CLOS, Eiffel, Perl, Python, Scala) erlauben einer Klasse, von mehreren Oberklassen gleichzeitig zu erben (*multiple inheritance*). Dafür hat sich der Begriff **mehrfache Vererbung** eingebürgert. Sprachlich ist **mehrfache Beerung** korrekt.

Die Herausforderung besteht in der Erfindung raffinierter Implementierungstechniken, aber auch im Sprachentwurf selbst. Ein Problem bereitet die Auflösung von möglichen Mehrdeutigkeiten:

- Es werden verschiedene Merkmale geerbt, die –zufällig– den gleichen Namen haben oder
- eine Oberklasse vererbt dasselbe Merkmal auf mehreren Wegen.

Auch die Implementierungs-idee, dass Attribute und überschreibbare Methoden in allen Unterklassen die gleiche Adresse haben wie in der Oberklasse, funktioniert nur, solange eine Klasse von *einer* Oberklasse erbt. Stattdessen bekommt jede Klasse  $K$  eine Hash-Tabelle  $h_K$ , die jedem Methodenamen  $f$  die in  $K$  gültige Codeadresse der zugehörigen Implementierung zuordnet.

## 5 Syntaktische Analyse

### 5.1 Theoretische Grundlagen

#### 5.1.1 Kontextfreie Grammatik, erzeugte Sprache, Ableitungsbaum, Linksreduktion

**Frage:** Wie definiert man eine unendliche Menge von (syntaktisch korrekten) Programmen?

**Prinzip:** Wie bei natürlichen Sprachen – durch eine Grammatik für die Programmiersprache!

Formale Sprachen zur Beschreibung der Syntax von natürlichen Sprachen wurden Mitte der 1950er Jahre vom Linguisten NOAM CHOMSKY eingeführt. Wichtige Anwendungen haben formale Grammatiken heute

- bei der Beschreibung von Programmiersprachen und im Compilerbau,
- in Systemen zur Erkennung und Verarbeitung natürlicher Sprachen,
- in der syntaktischen Mustererkennung und
- bei der Beschreibung biologischer Wachstumsprozesse (Lindenmayer-Systeme).

Formale Grammatiken dienen also zur Beschreibung strukturierter Zeichenketten. Es wird die Syntax (die Form der Zeichenketten) und nicht die Semantik (deren Bedeutung) beschrieben.

**Definition:** Ein **Alphabet**  $X$  ist eine endliche, nichtleere Menge von Zeichen oder Symbolen.

Ein **Wort**  $w$  über dem Alphabet  $X$  ist eine endliche Folge von Symbolen  $w = a_1 \dots a_n$ ,  $a_i \in X$ ,  $n \geq 0$ ,  $1 \leq i \leq n$ . Die **Länge**  $|w|$  eines Wortes  $w$  ist die Anzahl seiner Zeichen. Das **leere Wort**  $\varepsilon$  hat die Länge  $|\varepsilon| = 0$ . Die Menge aller Wörter über dem Alphabet  $X$  wird mit  $X^*$  bezeichnet. Jede Teilmenge  $L$  von  $X^*$  ist eine **formale Sprache**.

**Beispiel 5.1:** Sei  $X = \{a, b, c\}$ . Dann ist  $w = aacbb = a^2cb^2$  ein Wort über  $X$  der Länge 5. Die Menge  $L = \{a^n cb^n \mid n \geq 0\} = \{c, acb, aacbb, \dots\}$  ist eine formale Sprache über  $X$ .

**Beispiel 5.2:** Die Menge aller gültigen Java-Programme ist eine formale Sprache über der Menge  $X$  aller ASCII-Zeichen.

**Definition:** Eine **kontextfreie Grammatik**  $G$  ist ein Tupel  $G = (N, T, P, S)$  mit

$N$  ist ein Alphabet (**nichtterminale Symbole**, Variablen, Metazeichen),

$T$  ist ein Alphabet mit  $T \cap N = \emptyset$  (**terminale Symbole**),

$P$  ist eine endliche Menge von **Produktionen** (Regeln) der Form

$$A \rightarrow \alpha \text{ mit } A \in N \text{ und } \alpha \in (N \cup T)^*,^7$$

$S \in N$  ist das **Startsymbol**.

Da Grammatiken zur Definition von Programmiersprachen sehr umfangreich sind<sup>8</sup>, hat man für deren Produktionen kompaktere Schreibweisen entwickelt:

Eine Produktion  $A \rightarrow \alpha$  aus  $P$  heißt **A-Produktion**. Für alle A-Produktionen einer Grammatik  $A \rightarrow \alpha_1$ ,  $A \rightarrow \alpha_2$ ,  $\dots$ ,  $A \rightarrow \alpha_n$  schreibt man kurz  $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$ .

Das Startsymbol der Grammatik und die Alphabete für die terminalen und die nichtterminalen Symbole werden oft nur *implizit* durch die Produktionen festgelegt:

- Das Symbol auf der linken Seite der *ersten* Produktion ist das Startsymbol.
- Jedes Symbol auf der linken Seite einer Produktion ist ein nichtterminales Symbol.
- Jedes *andere* Symbol auf der rechten Seite einer Produktion ist ein terminales Symbol.

<sup>7</sup> Manchmal erlaubt man Produktionen mit dem leeren Wort  $\varepsilon$  auf der rechten Seite (*nullable rule*) nur dann, wenn das Startsymbol auf der linken Seite steht. Wir brauchen diese Einschränkung nicht.

<sup>8</sup> Die Original-Grammatik für Pascal (1976) hatte ca. 220 Produktionen und ca. 120 nichtterminale Symbole.

*Formal methods will never have any impact until they can be used by people that don't understand them.*  
T. Melham

**Frage:** Wie erzeugt man aus einer kontextfreien Grammatik Wörter einer Sprache?

**Definition:** Seien  $\gamma_1, \gamma_2 \in (N \cup T)^*$ , also Wörter aus terminalen und nichtterminalen Symbolen.  $\gamma_2$  lässt sich aus  $\gamma_1$  bzgl.  $G$  **in einem Schritt ableiten** (Schreibweise  $\gamma_1 \Rightarrow \gamma_2$ ), falls gilt:

1.  $\gamma_1 = \beta_1 A \beta_2$  mit  $A \in N$ ,  $\beta_1, \beta_2 \in (N \cup T)^*$  ( $\gamma_1$  enthält ein nichtterminales Symbol  $A$ );
2.  $A \rightarrow \alpha$  ist eine Produktion aus  $P$  (es gibt eine  $A$ -Produktion in  $P$ );
3.  $\gamma_2 = \beta_1 \alpha \beta_2$  ( $\gamma_2$  entsteht aus  $\gamma_1$  durch Ersetzung von  $A$  durch  $\alpha$ ).

Eine Produktion  $A \rightarrow \alpha$  wird auf ein Wort  $w$  angewendet, indem *ein* Vorkommen des nichtterminalen Symbols  $A$  in  $w$  durch die rechte Seite  $\alpha$  der Produktion ersetzt wird.

Seien  $\gamma_1, \dots, \gamma_m \in (N \cup T)^*$ ,  $m \geq 1$  und gelte  $\gamma_i \Rightarrow \gamma_{i+1}$ ,  $1 \leq i \leq m-1$ . Dann lässt sich  $\gamma_m$  aus  $\gamma_1$  (in  $m-1$  Schritten) ableiten. Kann man aus einem Wort  $\beta \in (N \cup T)^*$  ein Wort  $\gamma$  in Null oder mehr Schritten ableiten, so schreibt man  $\beta \Rightarrow^* \gamma$ .

Jedes aus dem Startsymbol mit Regeln aus  $P$  ableitbare Wort heißt **Satzform** von  $G$ .

**Definition:** Die von einer Grammatik  $G$  **erzeugte Sprache**  $L(G)$  ist die Menge aller terminalen Wörter, die man vom Startsymbol  $S$  ableiten kann, d.h.  $L(G) = \{w \in T^* \mid S \Rightarrow^* w\}$ . Eine kontextfreie Grammatik  $G$  erzeugt eine **kontextfreie Sprache**  $L(G)$ .

**Beispiel 5.3:** Sei  $G_1 = (N_1, T, P_1, S)$  gegeben mit  $N_1 = \{S\}$ ,  $T = \{a, *, +, (, )\}$  und  $P_1$  mit den folgenden Produktionen:  $S \rightarrow S + S$ ,  $S \rightarrow S * S$ ,  $S \rightarrow (S)$ ,  $S \rightarrow a$ . Für Grammatik  $G_1$  gilt:  $S \Rightarrow (S) \Rightarrow (S + S) \Rightarrow (S + S * S) \Rightarrow \dots \Rightarrow (a + (a + a) * a)$ , d.h.  $(a + (a + a) * a) \in L(G_1)$ .

**Beispiel 5.4:** Sei  $G_2 = (N_2, T, P_2, S)$  gegeben mit  $N_2 = \{S, R\}$  und  $P_2$  mit den folgenden Produktionen:  $S \rightarrow aR$ ,  $S \rightarrow (S)R$ ,  $R \rightarrow +S$ ,  $R \rightarrow *S$ ,  $R \rightarrow \varepsilon$ . Aus Grammatik  $G_2$  kann man ableiten:  $S \Rightarrow (S)R \Rightarrow (aR)R \Rightarrow (a + S)R \Rightarrow (a + (S)R)R \Rightarrow (a + (S) * S)R \Rightarrow (a + (S) * aR)R \Rightarrow \dots \Rightarrow (a + (a + a) * a)$ , d.h.  $(a + (a + a) * a) \in L(G_2)$ .

Um ein Wort abzuleiten, können Produktionen oft in verschiedener Reihenfolge angewendet werden. Manchmal will man diese Reihenfolge genauer festlegen:

**Definition:** Eine Ableitungsfolge  $S \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \dots \Rightarrow \gamma_k = w \in T^*$ , in der stets das linkeste nichtterminale Zeichen ersetzt wird, heißt **Linksableitung** (*leftmost derivation*) des Wortes  $w$ . Dann ist in der Definition des Ableitungsschritts  $\beta_1 \in T^*$ . Ersetzt man stets das rechteste nichtterminale Zeichen ( $\beta_2 \in T^*$ ), entsteht eine **Rechtsableitung** (*rightmost derivation*).

**Satz:** Gilt  $S \Rightarrow^* w$  mit  $w \in T^*$ , dann gibt es auch eine Linksableitung (Rechtsableitung) für  $w$ .

**Beispiel 5.5:** Das Wort  $(a + (a + a) * a)$  hat bzgl. der Grammatik  $G_1$  (Beispiel 5.3) folgende

- Linksableitung:  $S \Rightarrow (S) \Rightarrow (S + S) \Rightarrow (a + S) \Rightarrow (a + S * S) \Rightarrow (a + (S) * S) \Rightarrow (a + (S + S) * S) \Rightarrow (a + (a + S) * S) \Rightarrow (a + (a + a) * S) \Rightarrow (a + (a + a) * a)$
- Rechtsableitung:  $S \Rightarrow (S) \Rightarrow (S + S) \Rightarrow (S + S * S) \Rightarrow (S + S * a) \Rightarrow (S + (S) * a) \Rightarrow (S + (S + S) * a) \Rightarrow (S + (S + a) * a) \Rightarrow (S + (a + a) * a) \Rightarrow (a + (a + a) * a)$ .

Mit einem **Ableitungsbaum** (*parse tree*) kann man beschreiben, welche Produktionen einer kontextfreien Grammatik ein Wort ableiten, ohne die *Reihenfolge* der Ableitungsschritte anzugeben.

Jeder interne Knoten eines Ableitungsbaums ist mit einem nichtterminalen Symbol  $A$  markiert; die direkten Nachfolger dieses Knotens tragen von links nach rechts die Symbole der rechten Seite einer  $A$ -Produktion (vgl. Abb. 5.1).

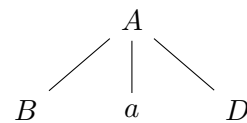


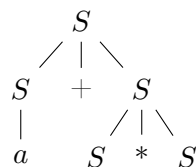
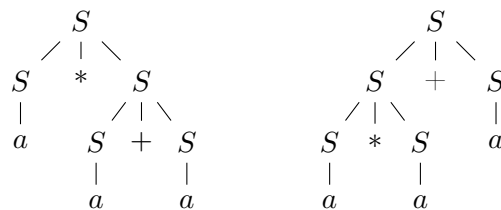
Abb. 5.1: Die Produktion  $A \rightarrow BaD$  im Ableitungsbaum.



Die Blätter eines Ableitungsbaums sind mit terminalen oder nichtterminalen Symbolen oder dem leeren Wort  $\varepsilon$  markiert und bilden von links nach rechts gelesen ein Wort. Dieses Wort lässt sich aus dem Symbol an der Wurzel des Ableitungsbaums ableiten.

Ist die Wurzel mit dem Startsymbol der Grammatik  $G$  und sind alle Blätter mit terminalen Symbolen oder dem leeren Wort  $\varepsilon$  markiert, so liegt das abgelesene Wort in der von der Grammatik erzeugten Sprache  $L(G)$ .

**Beispiel 5.6:** Die Grammatik  $G_1$  aus Beispiel 5.3 hat für das Wort  $a + S * S$  einen Ableitungsbaum (Abb. 5.2) und für das Wort  $a * a + a$  zwei verschiedene Ableitungsbäume (Abb. 5.3).

Abb. 5.2: Ableitungsbaum für  $a + S * S$ .Abb. 5.3: Ableitungsbäume für  $a * a + a$ .

Hat ein Wort  $w \in L(G)$  mehrere Ableitungsbäume, so heißt die Grammatik  $G$  **mehrdeutig** (*ambiguous*). Hat jedes Wort  $w \in L(G)$  genau einen Ableitungsbaum, so heißt die Grammatik  $G$  **eindeutig** (*unambiguous*).

**Beispiel 5.7:** Die kontextfreie Grammatik mit den Produktionen  $S \rightarrow a \mid aSa$  ist eindeutig.

Der Ableitungsbaum beschreibt die syntaktische Struktur eines Wortes bzgl. einer gegebenen kontextfreien Grammatik. Das Bestimmen der syntaktischen Struktur eines Wortes heißt **Syntaxanalyse** (*parsing*). Die Syntaxanalyse ist eine Teilaufgabe von Übersetzern, da ein Programm überprüft werden muss, ob es ein Element der (Programmier-)Sprache („syntaktisch richtig“) ist.

Im Übersetzerbau betrachtet man eine Ableitungsfolge oft *rückwärts* vom Wort zum Startsymbol. Man spricht dann statt von Ableitungsschritten von **Reduktionsschritten**.

Eine Ableitung beginnt stets mit dem Startsymbol, aber bei einer Reduktion muss man aufpassen, dass man mit dem Startsymbol endet (und nicht in einer Sackgasse!) (vgl. Beispiel 5.35).

Eine Reduktion heißt **Linksreduktion**, wenn für jeden Reduktionsschritt  $\alpha \leftarrow \beta$  gilt:

In  $\alpha = w_1 \dots w_n$  ( $w_s \in T \cup N$ ,  $1 \leq s \leq n$ ) wird ein Wort  $w_i \dots w_j$  ersetzt und keine Ersetzung, die links davon beginnt ( $w_k \dots w_l$  mit  $k < i$ ), kann man durch Reduktionsschritte bis zum Startsymbol fortsetzen. Eine Linksreduktion entspricht einer Rechtsableitung, allerdings werden die Produktionen in umgekehrter Reihenfolge angewendet.

**Beispiel 5.8:** Für Grammatik  $G_1$  und  $w = (a + a) * a$  erhält man folgende Linksreduktion:  
 $(a + a) * a \leftarrow (S + a) * a \leftarrow (S + S) * a \leftarrow (S) * a \leftarrow S * a \leftarrow S * S \leftarrow S$

Kontextfreie Grammatiken von Programmiersprachen werden oft in einem „Dialekt“ beschrieben, der **Backus-Naur-Form (BNF)**:

- Oft benutzt man die Zeichenfolge  $::=$  für das Relationssymbol  $\rightarrow$ .
- Nichtterminale Symbole haben die Form  $\langle \text{name} \rangle$ .
- Terminale Symbole sind einfache Zeichen (z.B.  $+$ ,  $-$ ) oder Tokenklassen (z.B. *tokenname*).

Die **erweiterte BNF** kann Wiederholungen oder Auslassungen in *einer* „Produktion“ beschreiben:

- **Wiederholung:** Eine Produktion der Form  $A \rightarrow \alpha \{\beta\} \gamma$  beschreibt, dass man aus  $A$  Zeichenketten ableiten kann, in denen  $\beta$  beliebig oft auftritt, also  $A \xRightarrow{*} \alpha \beta^i \gamma$ ,  $i \geq 0$ . Sie ist eine Abkürzung für  $A \rightarrow \alpha B \gamma$ ,  $B \rightarrow \varepsilon \mid \beta B$  mit neuem Nonterminal  $B$ .
- **Auslassung:** Eine Produktion der Form  $A \rightarrow \alpha [\beta] \gamma$  beschreibt, dass  $\beta$  höchstens einmal auftritt. Sie kürzt also  $A \rightarrow \alpha \gamma \mid \alpha \beta \gamma$  ab.

**Beispiel 5.9:** Eine Grammatik für einfache Wertzuweisungen kann dann so oder so aussehen:

$\langle \text{assign} \rangle ::= \text{variable} := \langle \text{expr} \rangle;$	$\langle \text{assign} \rangle ::= \text{variable} := \langle \text{expr} \rangle;$
$\langle \text{expr} \rangle ::= \langle \text{term} \rangle \mid \langle \text{term} \rangle + \langle \text{expr} \rangle$	$\langle \text{expr} \rangle ::= \langle \text{term} \rangle \{ + \langle \text{term} \rangle \}$
$\langle \text{term} \rangle ::= \langle \text{factor} \rangle \mid \langle \text{factor} \rangle * \langle \text{term} \rangle$	$\langle \text{term} \rangle ::= \langle \text{factor} \rangle \{ * \langle \text{factor} \rangle \}$
$\langle \text{factor} \rangle ::= \text{number} \mid \text{variable} \mid (\langle \text{expr} \rangle)$	$\langle \text{factor} \rangle ::= \text{number} \mid \text{variable} \mid (\langle \text{expr} \rangle)$

Das Startsymbol ist also  $\langle \text{assign} \rangle$ ,

das Alphabet der nichtterminalen Symbole ist  $\{\langle \text{assign} \rangle, \langle \text{expr} \rangle, \langle \text{term} \rangle, \langle \text{factor} \rangle\}$  und

das Alphabet der terminalen Symbole ist  $\{\text{variable}, \text{number}, :=, ;, +, *, (, )\}$ .

Wir wollen überflüssige Teile einer kontextfreien Grammatik entfernen. Dazu definieren wir:

**Definition:** Sei  $G = (N, T, P, S)$  eine kontextfreie Grammatik.

$a \in T$  heißt **nützlich**, falls  $a$  in einer Produktion aus  $P$  vorkommt.

$A \in N$  heißt (vom Startsymbol) **erreichbar**, falls es Wörter  $\alpha, \beta$  gibt mit  $S \xRightarrow{*} \alpha A \beta$ .

$A \in N$  heißt **produktiv**, falls es ein Terminalwort  $u \in T^*$  gibt mit  $A \xRightarrow{*} u$ .

$G'$  heißt **reduzierte<sup>9</sup> kontextfreie Grammatik** zu  $G$ , falls  $L(G') = L(G)$  und jedes terminale Symbol nützlich ist und jedes nichtterminale Symbol erreichbar und produktiv ist.

Um eine reduzierte Grammatik zu erhalten, entfernt man alle unnützen terminalen Symbole und alle unerreichbaren oder unproduktiven nichtterminalen Symbole sowie alle Produktionen, in denen solche Symbole vorkommen. Aber die richtige Reihenfolge macht's!

**Beispiel 5.10:** In der Grammatik  $G = (\{S, X, Y, Z\}, \{a, b\}, P, S)$  mit

$$P = \{S \rightarrow aX, X \rightarrow bS \mid aYbY, Y \rightarrow ba \mid aZ, Z \rightarrow aZX\}$$

ist  $Y$  offenbar produktiv. Dann sind auch  $X$  und  $S$  produktiv.  $Z$  ist nicht produktiv.

Da alle (verbliebenen) nichtterminalen Symbole erreichbar und alle terminalen Symbole nützlich sind, gilt:  $G' = (\{S, X, Y\}, \{a, b\}, P', S)$  mit  $P' = \{S \rightarrow aX, X \rightarrow bS \mid aYbY, Y \rightarrow ba\}$ .

**Beispiel 5.11:** In der Grammatik  $G = (\{S, U, V, X, Z\}, \{a, b, c, d\}, P, S)$  mit

$$P = \{S \rightarrow SZ \mid Sa \mid b, U \rightarrow V, X \rightarrow c, V \rightarrow Vd \mid d, Z \rightarrow ZX\}$$

ist  $Z$  erreichbar, ebenso  $X$ .  $U$  und  $V$  sind nicht erreichbar.  $Z$  ist nicht produktiv.

Nach Entfernung von  $Z$  und seiner Produktionen ist  $X$  nicht mehr erreichbar!

Die reduzierte Grammatik:  $G' = (\{S\}, \{a, b\}, P', S)$  mit  $P' = \{S \rightarrow Sa \mid b\}$ .

Um eine reduzierte Grammatik zu erhalten, entfernt man daher zuerst die nicht produktiven und dann die nicht erreichbaren nichtterminalen Symbole und schließlich die nicht nützlichen terminalen Symbole. Im folgenden betrachten wir nur reduzierte kontextfreie Grammatiken.

**Bemerkungen:**

- Für eine kontextfreie Sprache  $L$  gibt es viele kontextfreie Grammatiken  $G$  mit  $L(G) = L$ . Z.B. gilt  $L(G_1) = L(G_2)$  für Beispiel 5.3/5.4.
- Es gibt formale Sprachen, für die es keine kontextfreie Grammatik gibt:  $\{a^n b^n c^n \mid n \geq 0\}$ .
- Es ist nicht entscheidbar, ob zwei kontextfreie Grammatiken dieselbe Sprache erzeugen, d.h. es gibt keinen Algorithmus, der zwei beliebige kontextfreie Grammatiken als Eingabe hat und die Frage, ob beide die gleiche Sprache erzeugen, korrekt beantworten kann.

### 5.1.2 Kellerautomat

Kellerautomaten sind das Automatenmodell für kontextfreie Grammatiken.

**Definition:** Ein **Kellerautomat** (*push down automaton*) ist ein Tupel  $K = (Q, T, \Delta, q_0, F)$ , wobei

- $Q$  die endliche Menge der **Zustände**,

<sup>9</sup> Eine Reduzierung hat nichts mit Reduktionsschritten zu tun!

- $T$  das **Eingabealphabet** und
- $q_0 \in Q$  der **Anfangszustand** und
- $F \subseteq Q$  die Menge der **Endzustände** sind.
- Die **Übergangsrelation**  $\Delta$  ist eine endliche Relation zwischen  $Q^+ \times (T \cup \{\varepsilon\})$  und  $Q^*$ .  
 $\Delta$  kann man auch als endliche partielle Funktion von  $Q^+ \times (T \cup \{\varepsilon\})$  in die endlichen Teilmengen von  $Q^*$  betrachten.

An dieser Definition ist ungewöhnlich, dass die Menge der Zustände mit der Menge der Kellersymbole identifiziert wurde. Der Inhalt des Kellers ist also immer eine Folge von Zuständen. Das oberste Symbol im Keller wird als **aktueller Zustand** angesehen. Der Automat soll außerdem mehrere Symbole am oberen Kellerrand gleichzeitig berücksichtigen können.

Die Übergangsrelation beschreibt die möglichen Berechnungen des Kellerautomaten. Sie beschreibt *endlich* viele Übergänge. Die Anwendung eines Übergangs  $(\gamma, x, \bar{\gamma})$  ersetzt den oberen Abschnitt  $\gamma \in Q^+$  des Kellers durch eine neue Folge  $\bar{\gamma} \in Q^*$  von Zuständen und liest dabei  $x \in T \cup \{\varepsilon\}$  in der Eingabe. Der ersetzte Kellerabschnitt  $\gamma$  ist nicht leer. Ein Übergang, bei dem kein Eingabezeichen verarbeitet wird, heißt  $\varepsilon$ -Übergang.

Eine Konfiguration umfasst alle Komponenten eines Automatentyps, die für zukünftige Schritte des Automaten relevant sind. Hier ist das der Kellerinhalt  $\gamma$  und die restliche Eingabe  $w$ .

**Definition:** Eine **Konfiguration** des Kellerautomaten  $K$  ist ein Paar  $(\gamma, w) \in Q^+ \times T^*$ .

Die Relation  $\vdash_K$  beschreibt die **Konfigurationsübergänge**:

$(\alpha\beta, aw) \vdash_K (\alpha\beta', w) \Leftrightarrow (\beta, a, \beta') \in \Delta$  für  $\alpha, \beta' \in Q^*, \beta \in Q^+, a \in T \cup \{\varepsilon\}$ .

$(q_0, w)$  für beliebiges  $w \in T^*$  heißt eine **Anfangskonfiguration**,

$(q, \varepsilon)$  für  $q \in F$  eine **Endkonfiguration**.

Wort  $w \in T^*$  wird vom Kellerautomaten  $K$  **akzeptiert**, wenn  $(q_0, w) \vdash_K^* (q, \varepsilon)$  für ein  $q \in F$ .

Die **Sprache**  $L(K)$  ist die Menge der vom Kellerautomaten  $K$  akzeptierten Wörter:

$L(K) = \{w \in T^* \mid \exists q \in F : (q_0, w) \vdash_K^* (q, \varepsilon)\}$ .

Eine **akzeptierende Berechnung** für ein Wort  $w$  führt von der Anfangskonfiguration  $(q_0, w)$  zu einer Endkonfiguration. Es kann mehrere akzeptierende Berechnungen für ein Wort geben, aber auch Berechnungen, die nur einen Präfix des Wortes lesen können oder die das ganze Wort lesen können, aber keine Endkonfiguration erreichen. Da man akzeptierende Berechnungen nicht durch Probieren finden möchte, bevorzugt man **deterministische Kellerautomaten**, bei denen es zu jeder Konfiguration höchstens einen Übergang in eine Nachfolgekonfiguration gibt.

Bei Kellerautomaten gibt es mehrere Quellen für nichtdeterministisches Verhalten:

- Die Übergangsrelation  $\Delta$  kann zu  $(\gamma, a) \in Q^+ \times T$  mehrere Fortsetzungen anbieten.
- Ist  $\bar{\gamma} \in Q^+$  ein nichtleerer Suffix von  $\gamma$ , so kann die Übergangsrelation auch für  $(\bar{\gamma}, a)$
- oder sogar für  $(\bar{\gamma}, \varepsilon)$  weitere Fortsetzungsmöglichkeiten liefern.

**Definition:** Ein Kellerautomat  $K$  heißt **deterministisch**, wenn die Übergangsrelation  $\Delta$  folgende Eigenschaft erfüllt: Sind  $(\gamma_1, a, \gamma_2)$  und  $(\bar{\gamma}_1, a', \bar{\gamma}_2)$  zwei verschiedene Übergänge aus  $\Delta$  und ist  $\bar{\gamma}_1$  ein Suffix von  $\gamma_1$ , dann sind  $a', a \in T$  und verschieden, also  $\varepsilon \neq a \neq a' \neq \varepsilon$ .

### 5.1.3 Reguläre Sprache, regulärer Ausdruck und endlicher Automat

Eine wichtige Unterklasse der kontextfreien Grammatiken hat besonders einfache Produktionen.

**Definition:** Eine Grammatik  $G = (N, T, P, S)$  heißt **regulär (rechtslinear)**, wenn alle Produktionen die Form  $A \rightarrow \alpha B$  oder  $A \rightarrow \alpha$  mit  $A, B \in N, \alpha \in T^*$  haben.

Die von einer regulären Grammatik  $G$  erzeugte Sprache  $L(G)$  heißt **reguläre Sprache**.

**Bemerkungen:**

- Jede endliche Sprache ist regulär!
- Jede reguläre Sprache ist kontextfrei!
- Man kann reguläre Grammatiken auch mit *linkslinaren* Produktionen ( $A \rightarrow B\alpha$  oder  $A \rightarrow \alpha$ ) definieren. Beide Formen sind äquivalent, dürfen aber nicht gemeinsam auftreten.
- Es gibt kontextfreie Sprachen, für die es keine reguläre Grammatik gibt.  
Z.B. wird die Sprache  $\{a^n b^n \mid n \geq 0\}$  durch die kontextfreie Grammatik  $G = (N, T, P, S)$  mit  $N = \{S, A\}$ ,  $T = \{a, b\}$  und  $P = \{S \rightarrow \varepsilon, S \rightarrow aA, A \rightarrow Sb\}$  erzeugt.  $G$  ist nicht regulär!

**Beispiel 5.12:** Sei  $G_3 = (N, T, P, S)$  eine reguläre Grammatik mit  $N = \{S, A, B\}$ ,  $T = \{0, 1\}$  und den folgenden Produktionen:

$$\begin{array}{lll} A \rightarrow 0 & B \rightarrow 1 \\ S \rightarrow 0 & A \rightarrow 0A & B \rightarrow 0B \\ S \rightarrow 1B & A \rightarrow 1B & B \rightarrow 1A \end{array}$$

Die Sprache  $L(G_3)$  enthält die Dualdarstellung aller natürlichen Zahlen mit einer geraden Zahl von Einsen ohne führende Nullen.

Eine Ableitung für 101011 ist z.B.

$$\begin{aligned} S &\Rightarrow 1B \Rightarrow 10B \Rightarrow 101A \\ &\Rightarrow 1010A \Rightarrow 10101B \Rightarrow 101011 \end{aligned}$$

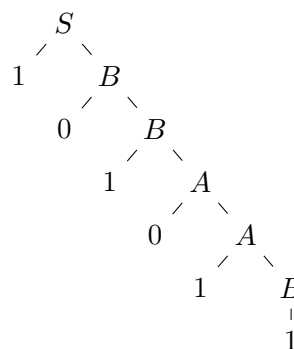


Abb. 5.4: Ableitungsbaum für das Wort 101011 bzgl.  $G_3$ .

Bei regulären Grammatiken gibt es nur Linksableitungen bzw. Rechtsableitungen. Die zugehörigen Ableitungsäume sind daher besonders einfach (vgl. Abb 5.4).

**Reguläre Ausdrücke** beschreiben ebenfalls reguläre Sprachen.

Zunächst definieren wir wichtige Operationen auf Mengen von Wörtern:

Seien  $M, M_1$  und  $M_2$  Mengen von Wörtern über einem Alphabet  $T$ . Dann ist

$$\begin{aligned} M_1 \cup M_2 &:= \{w \mid w \in M_1 \vee w \in M_2\}, \\ M_1 M_2 &:= \{uv \mid u \in M_1 \wedge v \in M_2\}, \\ M^0 &:= \{\varepsilon\}, \quad M^1 := M, \quad M^i := M M^{i-1} \quad \text{für } i > 0, \\ M^* &:= \bigcup_{i=0}^{\infty} M^i, \quad M^+ := \bigcup_{i=1}^{\infty} M^i, \quad \text{d.h. } M^+ = M^* \setminus \{\varepsilon\}, \text{ falls } \varepsilon \notin M \text{ ist.} \end{aligned}$$

**Beispiel 5.13:** Sei  $M_1 = \{a, bc\}$ ,  $M_2 = \{a, de, \varepsilon\}$  und  $M = \{0, 1\}$ . Dann gilt:

$$\begin{aligned} M_1 \cup M_2 &= \{a, bc, de, \varepsilon\} \\ M_1 M_2 &= \{a^2, bca, ade, bcde, a, bc\} \\ M^2 &= \{00, 01, 10, 11\} \\ M^3 &= \{000, 001, \dots, 111\} \\ M^* &= \{\varepsilon\} \cup \{0, 1\} \cup \{00, 01, 10, 11\} \cup \dots \\ M^+ &= \{0, 1\} \cup \{00, 01, 10, 11\} \cup \{000, \dots\} \cup \dots \end{aligned}$$

**Reguläre Ausdrücke** werden nun rekursiv definiert. Ein regulärer Ausdruck  $\alpha$  über einem Alphabet  $T$  bezeichnet eine Menge von Wörtern  $L(\alpha) \subseteq T^*$ , also eine formale Sprache über  $T$ . Man definiert zunächst die einfachsten regulären Ausdrücke und beschreibt deren Wortmengen. Dann gibt man Operationen an, wie man aus einfachen Ausdrücken kompliziertere zusammensetzen kann und wie die zugehörigen Wortmengen zu bilden sind.

**Definition:** Gegeben sei ein Alphabet  $T$ . Dann gilt:

0. Das Symbol  $\emptyset$  ist ein regulärer Ausdruck, der die leere Menge  $L(\emptyset) = \emptyset$  bezeichnet.
1. Das leere Wort  $\varepsilon$  ist ein regulärer Ausdruck, der die Menge  $L(\varepsilon) = \{\varepsilon\}$  bezeichnet.<sup>10</sup>
2. Für jedes  $a \in T$  ist  $a$  ein regulärer Ausdruck.  $a$  bezeichnet die Menge  $L(a) = \{a\}$ .

Sind  $\alpha$  und  $\beta$  reguläre Ausdrücke mit  $L(\alpha) = M$  und  $L(\beta) = M'$ , so ist

3.  $\alpha \mid \beta$  ein regulärer Ausdruck, der  $L(\alpha \mid \beta) = M \cup M'$  bezeichnet.
4.  $\alpha \beta$  ein regulärer Ausdruck, der  $L(\alpha \beta) = M M'$  bezeichnet.
5.  $\alpha^*$  bzw.  $\alpha^+$  ein regulärer Ausdruck für die Menge  $L(\alpha^*) = M^*$  bzw.  $L(\alpha^+) = M^+$ .

Reguläre Ausdrücke können Klammern enthalten, um die Reihenfolge der Operationen festzulegen. Um die Zahl der Klammern gering zu halten, vereinbart man Prioritäten:  $*$  und  $+$  haben die höchste Priorität, dann kommt das Produkt und schließlich die Vereinigung. Alle Operationen sind linksassoziativ.

**Beispiel 5.14:** Der reguläre Ausdruck  $\alpha = a \mid bc^*d$  ist zu klammern als  $a \mid ((b(c)^*)d)$  und bezeichnet die formale Sprache  $L(\alpha) = \{a\} \cup \{bc^i d \mid i \geq 0\}$ .

Reguläre Ausdrücke können Namen erhalten, die in anderen Ausdrücken verwendet werden:

**Beispiel 5.15:** Die Menge der Identifier in Pascal wird durch folgende reguläre Ausdrücke beschrieben. Hier sind `<letter>`, `<digit>` und `<ident>` Namen für reguläre Ausdrücke.

```
<letter> = A | B | ... | Z | a | b | ... | z
<digit>  = 0 | 1 | ... | 9
<ident>  = <letter> ( <letter> | <digit> )*
```

**Satz:** Die durch reguläre Ausdrücke bezeichneten Mengen sind reguläre Sprachen, und jede reguläre Sprache lässt sich durch einen regulären Ausdruck bezeichnen.

Häufig ist der reguläre Ausdruck viel kürzer als eine äquivalente reguläre Grammatik.

**Beispiel 5.16:** Eine reguläre Grammatik in BNF für die Menge der Identifier in Pascal:

```
<ident> → A <idR> | B <idR> | ... | z <idR>
<idR>   → A <idR> | B <idR> | ... | z <idR> | 0 <idR> | ... | 9 <idR> | ε
```

**Beispiel 5.17:** Ein regulärer Ausdruck für alle Wörter, die Dualdarstellungen aller natürlichen Zahlen mit einer geraden Zahl von Einsen darstellen (ohne führende Nullen):  $0 \mid (10^*10^*)^*$

Für einige Teilausdrücke haben sich Kurzschreibweisen eingebürgert:

- **Auslassung:** Ein Ausdruck  $r \mid \varepsilon$  wird zu  $r?$  abgekürzt.
- Für einen Ausdruck  $c_1 \mid c_2 \mid \dots \mid c_k$  von *Terminalzeichen* schreibt man  $[c_1 c_2 \dots c_k]$ , für im Alphabet *aufeinanderfolgende* Terminalzeichen  $[c_1 - c_k]$ .

**Beispiel 5.18:** `<letter> = [A-Z] | [a-z]`

In Beispiel 5.15 können wir daher abkürzen: `<digit> = [0-9]`

Das Automatenmodell für reguläre Sprachen ist ein **endlicher Automat** (*finite state machine*), hier als **Akzeptor**.

**Definition:** Ein **deterministischer endlicher Automat** ist ein Tupel  $DEA = (Q, T, \delta, q_0, F)$  mit

- $Q$  die endliche Menge der **Zustände**,
- $T$  das **Eingabealphabet**,
- $\delta : Q \times T \rightarrow Q$  die (partielle) **Übergangsfunktion**,
- $q_0 \in Q$  der **Anfangszustand** und
- $F \subseteq Q$  die Menge der **Endzustände**.

<sup>10</sup> Ja, man kann  $\varepsilon$  durch den regulären Ausdruck  $\emptyset^*$  darstellen.

Die Übergangsfunktion lässt sich **erweitern** für Eingabewörter zu  $\hat{\delta} : Q \times T^* \rightarrow Q$  mit

$$\hat{\delta}(q, \varepsilon) = q \quad \text{und} \quad \hat{\delta}(q, ax) = \hat{\delta}(\delta(q, a), x) \quad \text{für alle } q \in Q, a \in T \text{ und } x \in T^*.$$

Die von *DEA* **akzeptierte Sprache** ist  $L(DEA) = \{x \in T^* \mid \hat{\delta}(q_0, x) \in F\}$ .

Ein **nichtdeterministischer endlicher Automat** *NEA* hat stattdessen die Übergangsfunktion  $\delta : Q \times T \rightarrow \mathcal{P}(Q)$  mit der Erweiterung  $\hat{\delta} : \mathcal{P}(Q) \times T^* \rightarrow \mathcal{P}(Q)$ ,

$$\hat{\delta}(Y, \varepsilon) = Y \quad \text{und} \quad \hat{\delta}(Y, ax) = \bigcup_{q \in Y} \hat{\delta}(\delta(q, a), x) \quad \text{für alle } Y \subseteq Q, a \in T \text{ und } x \in T^*$$

und der akzeptierten Sprache  $L(NEA) = \{x \in T^* \mid \hat{\delta}(\{q_0\}, x) \cap F \neq \emptyset\}$ .

Man kann die Übergangsfunktion auch als Relation  $\Delta \subseteq Q \times (T \cup \{\varepsilon\}) \times Q$  darstellen.

## 5.2 Aufbau von lexikalischen Scannern

Für gängige Programmiersprachen nutzt man Scanner, um Zeichen zu Token zusammenzufassen, also zur Erkennung von

- reservierten Wörtern (**begin**, **for**, **if**, ...),
  - Identifikatoren (Namen für Variable, Konstante, Typen etc.),
  - Konstanten (ganze Zahlen, Gleitpunktzahlen, Zeichenketten, ...),
  - (zusammengesetzten) Sonderzeichen (**:=**, **<=**, ...), und
  - Zwischenraum (*white space*) (als Folge von Leerzeichen, Tabulatoren, Zeilenwechsel).
- Manchmal (z.B. in *ISWIM*, *Occam*, *Miranda*, *Haskell*, *Python*) hat eine **Einrückung** eine Bedeutung und es gilt die **Abseitsregel** (*off-side rule*). Ein Scanner erzeugt dann bei verstärkter Einrückung ein *INDENT*-Token und bei verringerter Einrückung ein *DEDENT*-Token.

Damit der Scanner von einfacher Struktur bleibt, werden Tokenklassen so definiert, dass die Worte, die zu einer Tokenklasse gehören, eine reguläre Sprache bilden<sup>11</sup>. Also definiert man jede Tokenklasse durch einen regulären Ausdruck.

**Beispiel 5.19:** Festlegung der Tokenklassen für eine Sprache mit arithmetischen Ausdrücken, Wertzuweisungen und **if**-Anweisungen. Mit den Abkürzungen aus Beispiel 5.18 werden folgende Tokenklassen definiert:

<i>ident</i> (Identifier)	<code>&lt;letter&gt; ( &lt;letter&gt;   &lt;digit&gt; )*</code>
<i>number</i> (vorzeichenlose Zahlen: int, real)	<code>&lt;digit&gt;+ ( .&lt;digit&gt;+ )?</code>
<i>relop</i> (Vergleichsoperatoren)	<code>=   &lt;=   &lt;&gt;   &lt;   &gt;   &gt;=</code>
<i>arithop</i> (arithmetische Operatoren)	<code>+   -   *   /</code>

Die Tokenklassen **if**, **then** und **else** enthalten nur das gleichnamige reservierte Wort.

Offenbar können einige Wörter in mehreren Tokenklassen liegen. Z.B. liegt das Token **if** in den Tokenklassen **if** und **ident**! Dann ist eine zusätzliche Regelung nötig, um das Wort eindeutig einer Tokenklasse zuzuordnen.

Scanner-Generatoren, z.B. *LEX*, *FLEX* oder *JFLEX*, erzeugen Scanner automatisch; typisch ist folgendes Vorgehen:

- Erzeugung eines (nicht-deterministischen) Zustandübergangsgraphen eines endlichen Automaten für jeden regulären Ausdruck einer Tokenklasse.
- Entfernung der  $\varepsilon$ -Übergänge im nicht-deterministischen Zustandübergangsgraphen.
- Umwandlung des nicht-deterministischen Zustandübergangsgraphen ohne  $\varepsilon$ -Übergänge in einen deterministischen Zustandübergangsgraphen.
- Ableitung eines Scanners aus dem deterministischen Zustandübergangsgraphen.

Oft sind **direkt erzeugte Scanner** effizienter. Oder der Parser erledigt diese Aufgabe gleich mit.

<sup>11</sup> Schachtelbare Kommentare übersteigen daher die Fähigkeiten eines Scanners.

### 5.3 Deterministische linksableitende Syntaxanalyse

Nun befassen wir uns mit Methoden zur syntaktischen Analyse. Der Scanner des Übersetzers hat die Eingabe Zeichen für Zeichen gelesen und eine Tokenfolge  $w$  erzeugt. In der syntaktischen Analyse wird jetzt geprüft, ob die Token in einer Reihenfolge auftreten, die durch die Grammatik dieser Programmiersprache festgelegt ist, d.h. ob  $w$  ein Wort dieser Sprache ist.

Dieses **Wortproblem** ist für kontextfreie Sprachen mit dem Algorithmus von COCKE-YOUNGER-KASAMI in  $O(|w|^3)$  lösbar. Wir suchen aber ein effizienteres, lineares Verfahren.

Das Verfahren zur **linksableitenden Syntaxanalyse** (*top-down parsing*) soll

1. das gegebene Wort  $w$  nur einmal von links nach rechts lesen.
2. eine **Linksableitung** von  $w$  erzeugen. Das bedeutet, dass der Ableitungsbaum von oben nach unten und von links nach rechts konstruiert wird.

Außerdem soll das Verfahren **deterministisch** arbeiten:

3. Muss in der Linksableitung als nächstes das nichtterminale Symbol  $A$  verarbeitet werden, so ist die anzuwendende  $A$ -Produktion durch das nächste zu lesende Symbol von  $w$  (**Vorschau-Symbol, lookahead**) *eindeutig* bestimmt.

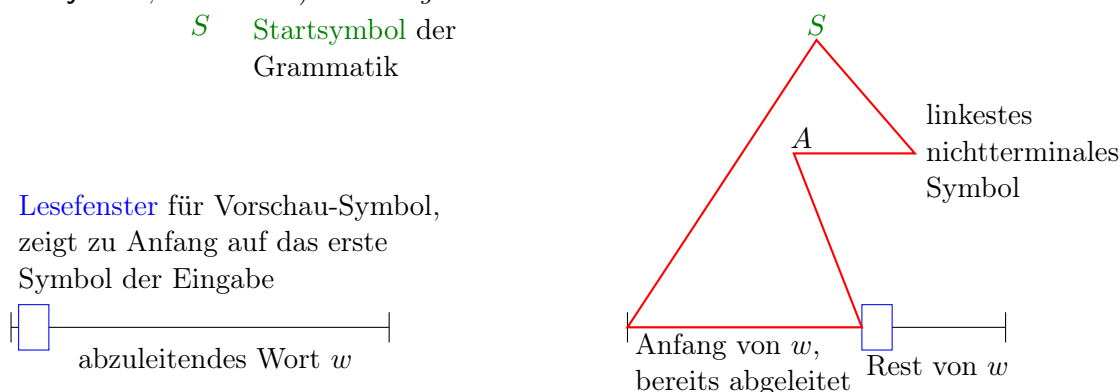


Abb. 5.5: Linksableitende Syntaxanalyse.

Nicht alle Grammatiken erlauben ein Parsen in dieser Art. Es gibt sogar kontextfreie Sprachen, z.B.  $\{a^n 0 b^n \mid n \geq 0\} \cup \{a^n 1 b^{2n} \mid n \geq 0\}$ , für die keine Grammatik existiert, die eine deterministische linksableitende Syntaxanalyse ermöglicht.

Aber eine deterministische linksableitende Syntaxanalyse ist für eine **s-Grammatik** stets möglich: Für jedes nichtterminale Zeichen  $A$  einer *s(imple deterministic)*-Grammatik gilt:

Alle rechten Seiten von  $A$ -Produktionen beginnen mit *verschiedenen terminalen* Zeichen.

(Das leere Wort wird nur direkt aus dem Startsymbol abgeleitet, das dann nicht auf der rechten Seite einer Produktion vorkommen darf.)

#### 5.3.1 Hindernisse: Gemeinsame Präfixe und linksrekursive Produktionen

Eine Grammatik kann die oben gestellten Forderungen z.B. nicht erfüllen, wenn es verschiedene Produktionen mit der gleichen linken Seite gibt, deren rechte Seiten ein gleiches Anfangsstück (**Präfix**) haben. In diesem Fall lässt sich die Grammatik aber umformen.

**Entfernen gemeinsamer Präfixe (Linksfaktorisierung):** Sei  $\alpha \neq \varepsilon$  ein gemeinsames Präfix, genauer: seien  $A \rightarrow \alpha\beta_1 \mid \dots \mid \alpha\beta_r \mid \gamma_1 \mid \dots \mid \gamma_s$  alle  $A$ -Produktionen der Grammatik, wobei kein  $\gamma_i$  das Präfix  $\alpha$  hat. Sei  $A'$  ein neues nichtterminales Symbol. Ersetze diese Produktionen durch

$$A \rightarrow \alpha A' \mid \gamma_1 \mid \dots \mid \gamma_s \quad \text{und} \\ A' \rightarrow \beta_1 \mid \dots \mid \beta_r$$

Offenbar erzeugt die so umgeformte Grammatik die gleiche Sprache.

**Beispiel 5.20:** Sei  $G = (\{B\}, \{\text{and}, \text{or}, \text{not}, (, ), \text{a}\}, P, B)$  gegeben mit den Produktionen  $B \rightarrow B \text{ and } B \mid B \text{ or } B \mid \text{not } B \mid (B) \mid \text{a}$

Offensichtlich haben die beiden ersten Produktionen ein gemeinsames Präfix  $B$ .

Also führt man ein neues nichtterminales Symbol  $B'$  ein und ersetzt die Produktionen durch:

$$B \rightarrow BB' \mid \text{not } B \mid (B) \mid \text{a} \quad \text{und} \\ B' \rightarrow \text{and } B \mid \text{or } B$$

Deterministisches Top-Down-Parsen ist auch nicht möglich, falls eine **linksrekursive Produktion** auftritt, also eine Produktion der Form  $A \rightarrow A\alpha$ ,  $\alpha \in (N \cup T)^+$ , da nicht erkennbar ist, *wie oft* diese Produktion angewandt werden soll. Linksrekursive Produktionen lassen sich ebenfalls durch Umformen der Grammatik entfernen.

**Entfernen linksrekursiver Produktionen:**

Seien  $A \rightarrow A\alpha_1 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \dots \mid \beta_n$  alle  $A$ -Produktionen, wobei kein  $\beta_i$  mit  $A$  beginnt und alle  $\alpha_i \neq \varepsilon$  sind. Sei  $A'$  ein neues nichtterminales Symbol. Ersetze diese Produktionen durch:

$$A \rightarrow \beta_1 A' \mid \dots \mid \beta_n A' \quad \text{und} \\ A' \rightarrow \alpha_1 A' \mid \dots \mid \alpha_m A' \mid \varepsilon$$

An der Korrespondenz der Ableitungsbäume für die ursprüngliche Grammatik (links) und transformierte Grammatik (rechts) erkennt man, dass die erzeugte Sprache unverändert bleibt:

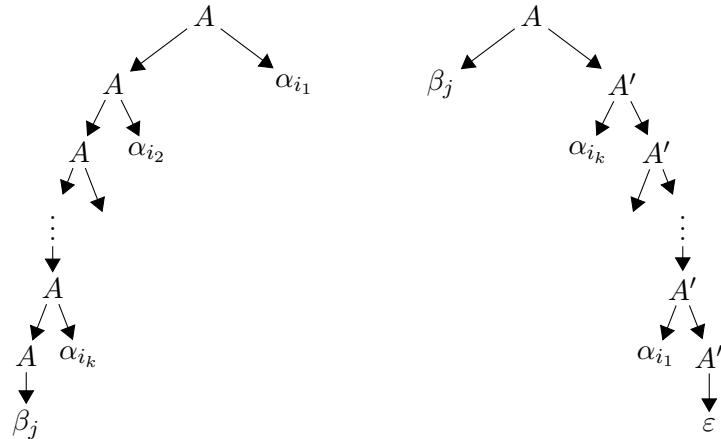


Abb. 5.6: Entfernen linksrekursiver Produktionen.

**Beispiel 5.21:** Sei  $G$  die Grammatik aus Beispiel 5.20. Die ersten beiden Produktionen sind linksrekursiv. Mit dem neuen nichtterminalen Symbol  $B'$  ersetzt man die Produktionen durch:

$$B \rightarrow \text{not } BB' \mid (B)B' \mid \text{a } B' \quad \text{und} \\ B' \rightarrow \text{and } BB' \mid \text{or } BB' \mid \varepsilon$$

Diese Umformungen führen aber nicht in jedem Fall zu einer Grammatik, die eine deterministische Syntaxanalyse erlaubt.

### 5.3.2 LL(1)-Grammatik

Wann erlaubt eine kontextfreie Grammatik eine deterministische linksableitende Syntaxanalyse?

Dazu definieren wir zunächst die Funktionen **First** und **Follow**.

**Definition:** Sei  $\alpha \in (N \cup T)^*$ . Dann ist

$$\mathbf{First}(\alpha) = \{a \in T \mid \alpha \xRightarrow{*} a\beta, \beta \in (N \cup T)^*\} \cup \{\varepsilon \mid \alpha \xRightarrow{*} \varepsilon\}.$$

$\mathbf{First}(\alpha)$  ist also die Menge derjenigen terminalen Zeichen, die bei Ableitungen von  $\alpha$  als erste auftreten. Kann man aus  $\alpha$  das leere Wort  $\varepsilon$  herleiten, so ist  $\varepsilon$  ebenfalls in  $\mathbf{First}(\alpha)$ .

**Beispiel 5.22:** In der Grammatik  $G_2$  aus Beispiel 5.4 gilt:  $a \in \mathbf{First}(S)$  wegen  $S \Rightarrow aR$ , und „ $($ “ liegt in  $\mathbf{First}(S)$  wegen  $S \Rightarrow (S)R$ . Aber es gilt auch  $a \in \mathbf{First}(RS)$ , da  $RS \Rightarrow S \Rightarrow aR$ .



**Definition:** Sei  $A \in N$  und  $\$$  ein neues Symbol, das das Ende der Eingabe markiert. Dann ist

$$\mathbf{Follow}(A) = \{a \in T \mid S \xRightarrow{*} \alpha A a \beta, \alpha, \beta \in (N \cup T)^*\} \cup \{\$ \mid S \xRightarrow{*} \alpha A, \alpha \in (N \cup T)^*\}$$

$\mathbf{Follow}(A)$  ist also die Menge der terminalen Zeichen, die bei einer Ableitung vom *Startsymbol*  $S$  direkt auf  $A$  folgen können. Ist  $A$  das letzte Zeichen, so ist die **Endmarke**  $\$$  in  $\mathbf{Follow}(A)$ .

**Beispiel 5.23:** Bezüglich der Grammatik  $G_2$  aus Beispiel 5.4 gilt: Es ist „ $\epsilon$ “ aus  $\mathbf{Follow}(R)$  wegen  $S \Rightarrow (S)R \Rightarrow (aR)R$ , und es ist  $\$ \in \mathbf{Follow}(R)$  wegen  $S \Rightarrow (S)R$ .

Generell kann man die Endmarke einführen, indem man die Grammatik durch Hinzufügen einer Produktion  $Z \rightarrow S\$$  erweitert, wobei  $S$  das Startsymbol und  $Z$  ein neues Nonterminal ist, das neues Startsymbol wird und wie die Endmarke  $\$$  nur dort vorkommt. Die erzeugte Sprache wird dadurch **präfixfrei**, d.h. kein Wort der Sprache ist Präfix eines anderen Wortes der Sprache.

Betrachten wir eine Linksableitung eines Wortes der Sprache und sei  $A$  das linkeste nichtterminale Zeichen, es ist also als nächstes eine  $A$ -Produktion anzuwenden. Die Auswahl dieser  $A$ -Produktion soll anhand des Vorschau-Symbols erfolgen. Wir analysieren daher die Grammatik und berechnen für jede  $A$ -Produktion die Menge aller möglichen Vorschau-Symbole. Diese Mengen müssen dann paarweise disjunkt sein. Gilt das für alle möglichen Nonterminals  $A$ , so ist ein deterministisches Top-down-Parsen möglich.

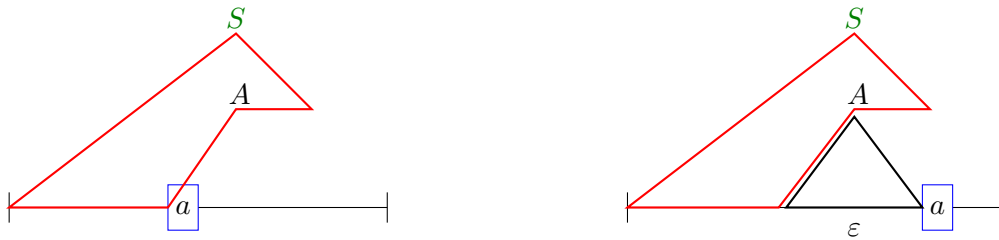


Abb. 5.7: Bestimmung der erwarteten Vorschau-Symbole.

Wir bestimmen zunächst aus der Grammatik die Menge der **erwarteten Symbole**, die in einer Linksableitung bei Anwendung einer Produktion  $A \rightarrow \alpha$  als Vorschau-Symbol vorkommen können:

$$\mathbf{Erwartet}(A \rightarrow \alpha) = \mathbf{First}(\alpha \mathbf{Follow}(A)) = \begin{cases} \mathbf{First}(\alpha) & , \text{ falls } \epsilon \notin \mathbf{First}(\alpha) \\ (\mathbf{First}(\alpha) \setminus \{\epsilon\}) \cup \mathbf{Follow}(A) & , \text{ falls } \epsilon \in \mathbf{First}(\alpha) \end{cases}$$

Nun können wir eine Bedingung anzugeben, die deterministisches Top-Down-Parsen garantiert: Das Vorschau-Symbol soll die anzuwendende  $A$ -Produktion *eindeutig* bestimmen.

- Also dürfen die rechten Seiten  $\alpha_i$  und  $\alpha_j$  zweier  $A$ -Produktionen nicht das gleiche terminale Anfangszeichen erzeugen, d.h.  $\mathbf{First}(\alpha_i) \cap \mathbf{First}(\alpha_j) = \emptyset$  für  $i \neq j$ .
- Kann man aber aus  $A$  –wie in Abb. 5.7 rechts– das leere Wort  $\epsilon$  herleiten, z.B. mit Hilfe der Produktion  $A \rightarrow \alpha_j$ , so muss *eindeutig* festliegen, ob aus  $A$  das leere Wort hergestellt werden soll oder nicht. In dem Fall ist  $\epsilon \in \mathbf{First}(\alpha_j)$ .

Dann darf das erste terminale Symbol, das  $A$  folgen kann, nicht mit einem Symbol übereinstimmen, das man als erstes aus der rechten Seite einer *anderen*  $A$ -Produktion herleiten kann, d.h.  $\mathbf{Follow}(A) \cap \mathbf{First}(\alpha_i)$  muss für  $i \neq j$  leer sein.

Diese Überlegungen führen zu folgender

**Definition:** Eine kontextfreie Grammatik  $G = (N, T, P, S)$  heißt **LL(1)-Grammatik**

(liest Eingabe von **L**inks nach **r**echts, erzeugt **L**inksableitung, nur **1** Zeichen als Vorschau, „**L**inksab**L**eitung mit Vorschautiefe **1**“), falls für alle  $A \in N$  gilt:

Seien  $A \rightarrow \alpha_1 \mid \dots \mid \alpha_n$  alle  $A$ -Produktionen in  $P$ .

1.  $\mathbf{First}(\alpha_1), \dots, \mathbf{First}(\alpha_n)$  sind paarweise disjunkt:  $\mathbf{First}(\alpha_i) \cap \mathbf{First}(\alpha_j) = \emptyset$ , falls  $i \neq j$ .
2. Ist  $\epsilon \in \mathbf{First}(\alpha_j)$ , dann ist  $\mathbf{Follow}(A) \cap \mathbf{First}(\alpha_i) = \emptyset$  für alle  $1 \leq i \leq n, i \neq j$ .

**Beispiel 5.24:** Zur Erläuterung der zweiten Bedingung der LL(1)-Eigenschaft betrachte man die Produktionen:

$$S \rightarrow aAaa \mid bAba \qquad A \rightarrow b \mid \varepsilon$$

Offensichtlich gilt  $\mathbf{First}(aAaa) = \{a\}$ ,  $\mathbf{First}(bAba) = \{b\}$ ,  $\mathbf{First}(b) = \{b\}$  und  $\mathbf{First}(\varepsilon) = \{\varepsilon\}$ . Damit ist die erste Bedingung der LL(1)-Eigenschaft erfüllt.

Für die zweite Bedingung bestimmt man  $\mathbf{Follow}(A) = \{a, b\}$  und stellt fest, dass  $\mathbf{Follow}(A) \cap \mathbf{First}(b) \neq \emptyset$  ist. Die Grammatik ist also nicht LL(1). Um das Wort  $bba$  abzuleiten, muss man nach der Ableitung  $S \Rightarrow bAba$  die Produktion  $A \rightarrow \varepsilon$  anwenden. Um das Wort  $bbba$  abzuleiten, muss man in der gleichen Situation, insbesondere dem gleichen Vorschau-Symbol  $b$ , die Produktion  $A \rightarrow b$  anwenden. Dieser Konflikt ist mit der gegebenen Information nicht lösbar.

Oft kann man durch „scharfes Hinsehen“ die **First**- und **Follow**-Mengen für eine gegebene Grammatik bestimmen. Der Vollständigkeit halber sollen jetzt zwei Algorithmen zur Berechnung der Funktionen **First** und **Follow** angegeben werden.

#### Algorithmen zur Berechnung der Funktionen First und Follow:

Gegeben sei eine kontextfreie Grammatik  $G = (N, T, P, S)$ .

Man berechne zunächst für alle  $X \in N \cup T$  die Menge  $\mathbf{First}(X)$  wie folgt:

1. Ist  $X \in T$ , dann setze  $\mathbf{First}(X) = \{X\}$ .
2. Ist  $X \rightarrow \varepsilon$  in  $P$ , dann füge das leere Wort  $\varepsilon$  zu  $\mathbf{First}(X)$  hinzu.
3. Durchlaufe die Menge  $P$  aller Produktionen und führe Schritt 4) solange aus, bis keine Änderungen an den **First**-Mengen mehr auftreten.
4. Für jede Produktion  $X \rightarrow X_1 X_2 \dots X_n$  mit  $n \geq 1$  füge  $a \in T$  zu  $\mathbf{First}(X)$  hinzu, falls  $\varepsilon \in \mathbf{First}(X_j)$  für alle  $j$ ,  $1 \leq j \leq i-1$ , (d.h.  $X_1 \dots X_{i-1} \xRightarrow{*} \varepsilon$ ) und  $a \in \mathbf{First}(X_i)$  für ein  $i$ ,  $1 \leq i \leq n$ .  
Ist  $\varepsilon \in \mathbf{First}(X_j)$  für alle  $j$ ,  $1 \leq j \leq n$ , so füge  $\varepsilon$  zu  $\mathbf{First}(X)$  hinzu.

Sei nun  $X_1 \dots X_n$  mit  $n \geq 1$ ,  $X_i \in N \cup T$ ,  $1 \leq i \leq n$  gegeben. Zur Berechnung der Menge  $\mathbf{First}(X_1 \dots X_n)$  wende man einmal den Schritt 4) des obigen Algorithmus auf eine „künstliche“ Produktion  $Y \rightarrow X_1 \dots X_n$  mit einem neuen nichtterminalen Symbol  $Y$  an. Es gilt offensichtlich  $\mathbf{First}(X_1 \dots X_n) = \mathbf{First}(Y)$ . Weiterhin gilt auf Grund der Definition  $\mathbf{First}(\varepsilon) = \{\varepsilon\}$ .

Die **Follow**-Funktion berechnet man folgendermaßen:

1. Sei  $\$$  ein neues Symbol, das das Ende eines Eingabewortes markiert.  
Füge  $\$$  zu  $\mathbf{Follow}(S)$  hinzu.
2. Durchlaufe die Menge  $P$  aller Produktionen und führe den folgenden Schritt solange aus, bis keine Änderungen an den **Follow**-Mengen mehr auftreten.
3. Für jede Produktion  $A \rightarrow \alpha B \beta$  mit  $\alpha, \beta \in (N \cup T)^*$  und  $B \in N$   
füge  $\mathbf{First}(\beta) \cap T$  (d.h. die Menge  $\mathbf{First}(\beta) \setminus \{\varepsilon\}$ ) zu  $\mathbf{Follow}(B)$  hinzu.  
Ist  $\varepsilon \in \mathbf{First}(\beta)$  (d.h.  $\beta \xRightarrow{*} \varepsilon$ ), dann füge  $\mathbf{Follow}(A)$  ebenfalls zu  $\mathbf{Follow}(B)$  hinzu.

**Beispiel 5.25:** Man betrachte wieder die Grammatik  $G_2$  aus Beispiel 5.4 mit den Produktionen  $S \rightarrow aR \mid (S)R$  und  $R \rightarrow +S \mid *S \mid \varepsilon$ . Zunächst berechne man

$$\begin{array}{ll} \mathbf{First}(S) = \{a, (\} & \mathbf{Follow}(S) = \{\$, )\} \\ \mathbf{First}(R) = \{\varepsilon, +, *\} & \text{und dann} \quad \mathbf{Follow}(R) = \{\$, )\}. \end{array}$$

Also:  $\mathbf{First}(aR) = \{a\}$ ,  $\mathbf{First}((S)R) = \{(\}$  und  $\mathbf{First}(aR) \cap \mathbf{First}((S)R) = \emptyset$   
 $\mathbf{First}(+S) = \{+\}$ ,  $\mathbf{First}(*S) = \{*\}$  und  $\mathbf{First}(\varepsilon) = \{\varepsilon\}$  sind paarweise disjunkt  
 und  $\mathbf{Follow}(R) = \{\$, )\}$ ,  $\mathbf{First}(+S) = \{+\}$  und  $\mathbf{First}(*S) = \{*\}$  ebenfalls.

Also ist  $G_2$  eine LL(1)-Grammatik mit der deterministisches Parsen durchgeführt werden kann.

Eine zweite, anschaulichere Methode zur Bestimmung der **First**- und **Follow**-Funktion benutzt die **Parchmann-Graphen**:

Gegeben sei eine kontextfreie Grammatik  $G = (N, T, P, S)$ .

Wir bestimmen, aus welchen Nichtterminalsymbolen das leere Wort  $\varepsilon$  abgeleitet werden kann:

**Algorithmus zur Berechnung von  $N_\varepsilon = \{A \in N \mid A \xRightarrow{*} \varepsilon\}$ :**

1.  $N_0 = \{A \mid A \rightarrow \varepsilon \text{ ist in } P\}$
2.  $N_{i+1} = N_i \cup \{A \mid A \rightarrow \alpha \text{ ist in } P \text{ und } \alpha \in N_i^*\}$
3. Ist  $N_{i+1} = N_i$ , dann setze  $N_\varepsilon = N_i$ .

Nun berechnen wir die First-Funktion für alle nichtterminalen Symbole der Grammatik:

**Algorithmus zur Berechnung der First-Funktion:**

Zur Berechnung von **First**( $A$ ) für alle  $A \in N$  konstruieren wir den Parchmann-Graphen  $\Gamma_{\text{first}}(G)$ :

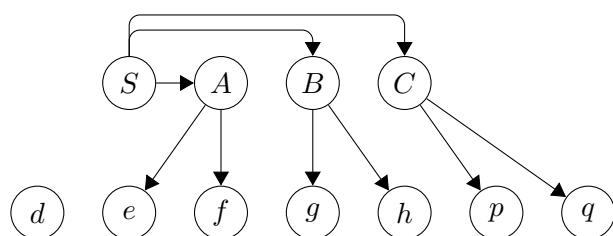
1. Jedes Symbol aus  $N \cup T$  wird durch einen Knoten in  $\Gamma_{\text{first}}(G)$  dargestellt.
2. Für jede Produktion  $A \rightarrow X_1 \dots X_n$  mit  $n \geq 1$ , fügt man eine Kante von  $A$  nach  $X_i$  hinzu, falls alle davorstehenden Symbole  $X_1, \dots, X_{i-1} \in N_\varepsilon$ .
3. Setze **First**( $A$ ) =  $\begin{cases} \{a \in T \mid \text{ex. Weg in } \Gamma_{\text{first}}(G) \text{ von } A \text{ nach } a\} & \text{falls } A \notin N_\varepsilon \\ \{\varepsilon\} \cup \{a \in T \mid \text{ex. Weg in } \Gamma_{\text{first}}(G) \text{ von } A \text{ nach } a\} & \text{falls } A \in N_\varepsilon \end{cases}$

Damit ist die **First**-Funktion auch für alle Wörter  $\alpha \in (N \cup T)^*$  bestimmt. Es gilt:

- Ist  $\alpha = \varepsilon$ , dann ist **First**( $\alpha$ ) =  $\{\varepsilon\}$ .
- Ist  $\alpha = a\beta$  mit  $a \in T$ , dann ist **First**( $\alpha$ ) =  $\{a\}$ .
- Ist  $\alpha = A\beta$  mit  $A \in N$ , dann ist **First**( $\alpha$ ) =  $\begin{cases} \text{First}(A) & \text{falls } A \notin N_\varepsilon \\ (\text{First}(A) - \{\varepsilon\}) \cup \text{First}(\beta) & \text{falls } A \in N_\varepsilon \end{cases}$

**Beispiel 5.26:** Betrachte Grammatik  $G = (N, T, P, S)$  mit  $N = \{S, A, B, C\}$ ,  $T = \{d, e, f, g, h, p, q\}$  und den Produktionen  $S \rightarrow ABCd$ ,  $A \rightarrow e \mid f \mid \varepsilon$ ,  $B \rightarrow g \mid h \mid \varepsilon$ ,  $C \rightarrow p \mid q$ .

Es ist  $N_\varepsilon = \{A, B\}$  und der zur Bestimmung der **First**-Funktion benötigte Graph  $\Gamma_{\text{first}}(G)$  ist



Also gilt:

**First**( $S$ ) =  $\{e, f, g, h, p, q\}$ ,

**First**( $A$ ) =  $\{\varepsilon, e, f\}$ ,

**First**( $B$ ) =  $\{\varepsilon, g, h\}$  und

**First**( $C$ ) =  $\{p, q\}$ .

und z.B.

**First**( $BAC$ ) =  $\{e, f, g, h, p, q\}$

Nun berechnen wir die **Follow**-Funktion.

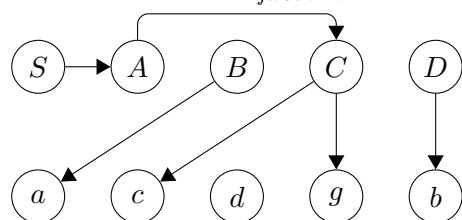
**Algorithmus zur Berechnung der Follow-Funktion:**

Der Parchmann-Graph  $\Gamma_{\text{follow}}(G)$  hat einen Knoten für jedes Symbol in  $N \cup T \cup \{\$\}$ .

1. Füge eine Kante vom Startsymbol  $S$  nach  $\$$  hinzu.
2. Für jedes Vorkommen eines Nonterminals  $B$  auf der rechten Seite einer Produktion  $A \rightarrow \alpha B \beta$  mit  $A, B \in N$ ,  $\alpha, \beta \in (N \cup T)^*$ , füge je eine Kante von  $B$  nach jedem  $a \in \text{First}(\beta) \cap T$  hinzu. Ist  $\varepsilon \in \text{First}(\beta)$  und  $A \neq B$ , dann füge eine Kante von  $B$  nach  $A$  hinzu.
3. Setze **Follow**( $A$ ) =  $\{a \in T \cup \{\$\} \mid \text{es gibt einen Weg von } A \text{ nach } a \text{ in } \Gamma_{\text{follow}}(G)\}$ .

**Beispiel 5.27:** Betrachte Grammatik  $G$  mit  $N = \{S, A, B, C, D\}$ ,  $T = \{a, b, c, d, g\}$  und Startsymbol  $S$ . Produktionen sind  $S \rightarrow AB$ ,  $B \rightarrow aAB \mid \varepsilon$ ,  $A \rightarrow CD$ ,  $D \rightarrow bCD \mid \varepsilon$ ,  $C \rightarrow cSd \mid g$ .

Der Parchmann-Graph  $\Gamma_{\text{first}}(G)$  zur Bestimmung der First-Funktion ist



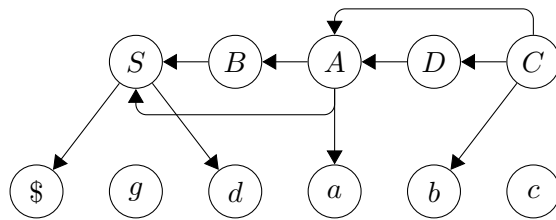
Es gilt also  $N_\varepsilon = \{B, D\}$  und

**First**( $S$ ) = **First**( $A$ ) = **First**( $C$ ) =  $\{c, g\}$ ,

**First**( $B$ ) =  $\{\varepsilon, a\}$  und

**First**( $D$ ) =  $\{\varepsilon, b\}$ .

Der Parchmann-Graph  $\Gamma_{follow}(G)$  zur Bestimmung der Follow-Funktion ist

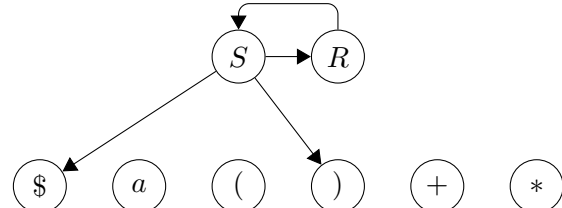
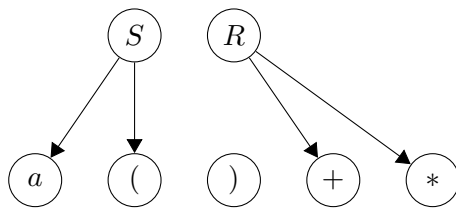


und damit gilt

**Follow**( $S$ ) = **Follow**( $B$ ) =  $\{d, \$\}$ ,  
**Follow**( $A$ ) = **Follow**( $D$ ) =  $\{a, d, \$\}$  und  
**Follow**( $C$ ) =  $\{a, b, d, \$\}$ .

**Beispiel 5.28:** Man betrachte wieder die Grammatik  $G_2$  aus Beispiel 5.4 mit den Produktionen  $S \rightarrow aR \mid (S)R$  und  $R \rightarrow +S \mid *S \mid \varepsilon$ .

Wir bestimmen zunächst die Parchmann-Graphen  $\Gamma_{first}(G_2)$  und  $\Gamma_{follow}(G_2)$



und dann die **First**- und **Follow**-Funktion. Wir erhalten (wie in Beispiel 5.25):

**First**( $S$ ) =  $\{a, (\}$   
**First**( $R$ ) =  $\{\varepsilon, +, *\}$  und

**Follow**( $S$ ) =  $\{\$, )\}$   
**Follow**( $R$ ) =  $\{\$, )\}$ .

### 5.3.3 Tabellengesteuerter Top-down-Parser

**Frage:** Wie erzeugt man aus einer LL(1)-Grammatik  $G$  einen Parser für die Sprache  $L(G)$ ?

Der Ableitungsbaum enthält noch viele überflüssige Informationen. Nur die erkannten Produktionen der Linksableitung müssen ausgegeben bzw. an die weiteren Teile des Übersetzers weitergegeben werden. Für den Parsingprozess muss man natürlich die noch zu bearbeitenden Teile des Ableitungsbaums speichern. Dazu verwendet man am besten einen Keller (siehe Abb. 5.8).

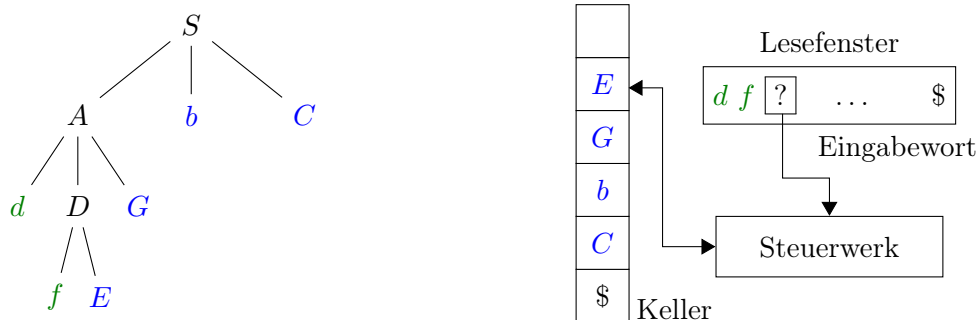


Abb. 5.8: Ableitungsbaum und Ableitung durch Top-down-Parser.

Von dem zu analysierenden Wort  $w$  sind bereits die ersten beiden Zeichen  $d$  und  $f$  abgeleitet worden, d.h. das Lesefenster steht auf dem nachfolgenden Zeichen. Noch abzuleiten sind, in dieser Reihenfolge, die nichtterminalen Symbole  $E$ ,  $G$  und  $C$ . Außerdem muss das terminale Symbol  $b$  vor der Ableitung von  $C$  in der Eingabe auftreten.

Benutzt man zur Speicherung dieser Symbole einen Keller, dann sieht man, dass der nächste Ableitungsschritt nur vom obersten Kellersymbol und vom Symbol im Lesefenster abhängt. Die **Endmarke**  $\$$  ist dabei ein neues Symbol, das das Ende der Eingabe bzw. des Kellers markiert.

Also wird der tabellengesteuerte Parser etwa folgendes Aussehen haben:

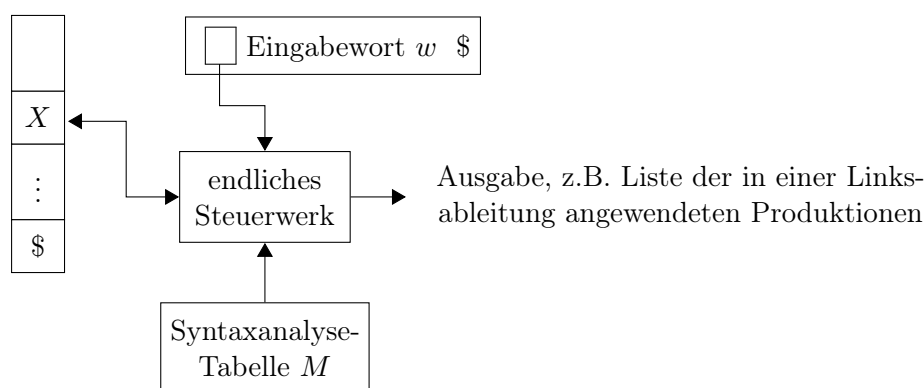


Abb. 5.9: Schema eines tabellengesteuerten Top-down-Parsers.

Dabei hat die **Syntaxanalyse-Tabelle**  $M$  (*parse table*) folgende Gestalt:

Die Zeilen der Tabelle sind mit den nichtterminalen Symbolen der Grammatik und die Spalten mit den Vorschauzeichen (also einem terminalen Symbol oder der Endmarke  $\$$ ) markiert. Jeder Tabelleneintrag enthält eine Produktion der Grammatik oder bleibt leer.

#### Konstruktion der Syntaxanalyse-Tabelle $M$ :

Für jede Produktion  $A \rightarrow \alpha$  in  $P$  und für alle  $f \in \mathbf{Erwartet}(A \rightarrow \alpha)$  setze:  $M(A, f) := A \rightarrow \alpha$

Die LL(1)-Bedingung erzwingt, dass jeder Tabellenplatz *höchstens* einmal besetzt wird!

Jedes leere Feld beschreibt eine Fehlersituation.

**Beispiel 5.29:** Für die Grammatik  $G_2$  aus Beispiel 5.4 erhält man folgende Tabelle  $M$ :

$M$	$a$	$($	$)$	$+$	$*$	$\$$
$S$	$S \rightarrow aR$	$S \rightarrow (S)R$				
$R$			$R \rightarrow \varepsilon$	$R \rightarrow +S$	$R \rightarrow *S$	$R \rightarrow \varepsilon$

In der Praxis steht in jedem leeren Feld eine **Fehlermeldung** (bzw. deren Nummer).

Die Arbeitsweise dieses Parsers kann man nun wie folgt beschreiben:

**Initiale Situation:** Auf dem Eingabeband steht das zu analysierende Wort  $w$  gefolgt von der Endmarke  $\$$ . Das Lesefenster steht auf dem ersten Zeichen von  $w$ .

Der Keller enthält als unterstes Symbol  $\$$  und darüber das Startsymbol  $S$  der Grammatik.

**Arbeitsweise:** Sei  $X$  das oberste Kellersymbol und  $a$  das Zeichen im Lesefenster der Eingabe. Dann führe man die folgenden Schritte durch, bis eine Endsituation erreicht ist.

1. Ist  $X \in T \cup \{\$\}$ , dann vergleiche  $X$  und  $a$ :
  - a. Ist  $X \neq a$ , dann beende das Parsen. Es gilt  $w \notin L(G)$ .
  - b. Sonst ist  $X = a$ .
    - $\alpha$ . Ist  $a = \$$ , dann beende das Parsen. Das vorgelegte Wort  $w$  ist aus der von der Grammatik erzeugten Sprache  $L(G)$ :  $w \in L(G)$ .
    - $\beta$ . Ist  $a \in T$ , dann lösche  $X$  vom Keller und setze das Lesefenster ein Zeichen weiter.
2. Ist  $X \in N$ , dann betrachte den Tabelleneintrag  $M(X, a)$ :
  - a. Ist  $M(X, a)$  leer, dann beende das Parsen. Es gilt  $w \notin L(G)$ .
  - b. Enthält  $M(X, a)$  die Produktion  $X \rightarrow A_1 \dots A_r$ , dann lösche  $X$ , schreibe  $A_r, \dots, A_1$  (in dieser Reihenfolge!) auf den Keller und gib die Produktion aus.

**Beispiel 5.30:** Als Grammatik betrachten wir wieder  $G_2$  aus Beispiel 5.4. Das Eingabewort sei  $a * (a + a)$ . Dann sind die ersten Schritte des Parsers:

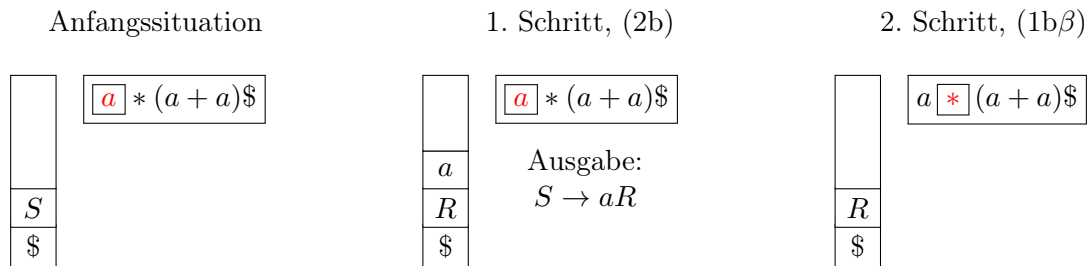


Abb. 5.10: Arbeitsweise des Parsers.

Um die Arbeitsweise eines tabellengesteuerten Top-Down-Parsers kompakt darzustellen, notiert man den momentanen Kellerinhalt  $X \dots \$$  (um 90° gegen den Uhrzeiger gedreht) und die restliche Eingabe ab dem Vorschau-Symbol  $a$  als ein Paar von Wörtern  $[X \dots \$, a \dots \$]$ . Dieses Paar heißt eine **Konfiguration** des Parsers. Jeder Schritt des Parsers erzeugt eine neue Konfiguration.

Ein Schritt der Form (1b $\beta$ ) soll durch  $[aY \dots \$, ab \dots \$] \vdash [Y \dots \$, b \dots \$]$  und ein Schritt der Form (2b) durch  $[XY \dots \$, a \dots \$] \vdash_{(i)} [A_1 \dots A_r Y \dots \$, a \dots \$]$  bezeichnet werden, wobei  $i$  die Nummer der angewendeten und ausgegebenen Produktion angibt.

**Beispiel 5.31:** Die Schritte des Parsers (u.a. die in Abb. 5.10 aufgeführten) werden nachfolgend durch Konfigurationen dargestellt. Dazu wurden die Produktionen der Grammatik  $G_2$  im Beispiel 5.4 von links nach rechts durchnummeriert:

$1 \ S \rightarrow (S)R,$	$2 \ S \rightarrow aR,$	$3 \ R \rightarrow +S,$	$4 \ R \rightarrow *S,$	$5 \ R \rightarrow \varepsilon.$
$\begin{array}{l} \vdash [ S \$ \quad \quad , a \star ( a + a ) \$ ] \\ \hline (2) \vdash [ a R \$ \quad \quad , a \star ( a + a ) \$ ] \\ \hline \vdash [ R \$ \quad \quad \quad , \star ( a + a ) \$ ] \\ \hline (4) \vdash [ \star S \$ \quad \quad , \star ( a + a ) \$ ] \\ \hline \vdash [ S \$ \quad \quad \quad , ( a + a ) \$ ] \\ \hline (1) \vdash [ ( S ) R \$ \quad , ( a + a ) \$ ] \\ \hline \vdash [ S ) R \$ \quad \quad , a + a ) \$ ] \\ \hline (2) \vdash [ a R ) R \$ \quad , a + a ) \$ ] \\ \hline \vdash [ R ) R \$ \quad \quad , + a ) \$ ] \end{array}$	$\begin{array}{l} \vdash [ + S ) R \$ \quad , + a ) \$ ] \\ \hline (3) \vdash [ S ) R \$ \quad \quad , a ) \$ ] \\ \hline \vdash [ a R ) R \$ \quad , a ) \$ ] \\ \hline (2) \vdash [ R ) R \$ \quad \quad , ) \$ ] \\ \hline \vdash [ ) R \$ \quad \quad \quad , ) \$ ] \\ \hline (5) \vdash [ R \$ \quad \quad \quad , \$ ] \\ \hline (5) \vdash [ \$ \quad \quad \quad , \$ ] \\ \hline \text{accept} \end{array}$			

### 5.3.4 Implementation eines Top-Down-Parsers nach der Methode des rekursiven Abstiegs (*recursive descent parser*)

Diese wohl einfachste Implementationsmethode lässt sich elegant über **Syntaxgraphen** einführen. Syntaxgraphen sind graphische Darstellungen von kontextfreien Grammatiken und werden häufig zur Darstellung der Syntax von Programmiersprachen benutzt<sup>12</sup>. Viele Übersetzer benutzen diese Methode zum Parsen, u.a. der Java-Compiler von SUN.

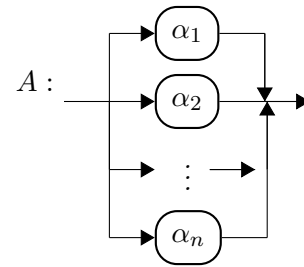
**Konstruktion von Syntaxgraphen aus einer kontextfreien Grammatik  $G = (N, T, P, S)$ :**

Für jedes nichtterminale Zeichen  $A \in N$  wird ein Syntaxgraph mit Namen  $A$  erzeugt. Er hat genau *eine* Eingangs- und *eine* Ausgangskante.

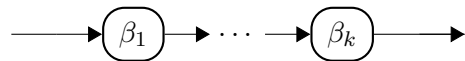
<sup>12</sup> Die Original-Grammatik für Pascal (1976) bestand aus 17 Syntaxgraphen.

1. Sind  $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$  alle  $A$ -Produktionen, so sieht der Graph  $A$  wie rechts angegeben aus,

wobei die Teilgraphen  $\alpha_i$  wie folgt bestimmt werden:



2. Ist  $\alpha_i = \beta_1 \dots \beta_k$  mit  $\beta_i \in N \cup T$ , dann hat der Teilgraph das Aussehen



$$\text{mit } \beta_i = \begin{cases} \boxed{B} & , \text{ falls } \beta_i = B \in N \\ \circ x & , \text{ falls } \beta_i = x \in T. \end{cases}$$

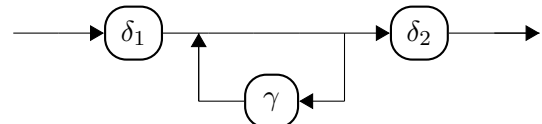
Ist  $\alpha_i = \varepsilon$ ,

so hat der Teilgraph die Form



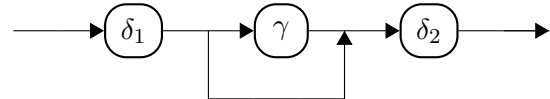
Ist  $\alpha_i = \delta_1 \{\gamma\} \delta_2$ ,

so hat der Teilgraph die Form



Ist  $\alpha_i = \delta_1 [\gamma] \delta_2$ ,

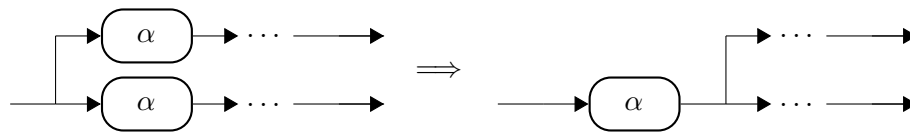
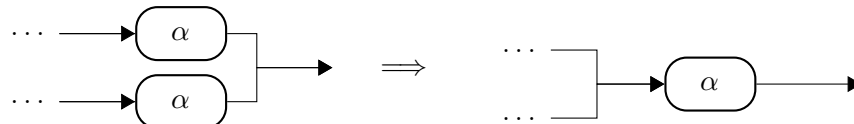
so hat der Teilgraph die Form



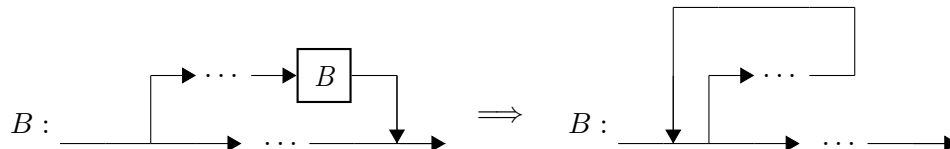
wobei die Teilgraphen  $\delta_1$ ,  $\delta_2$  und  $\gamma$  wieder wie in 2 bestimmt werden.

Diese Syntaxgraphen kann man durch folgende Regeln vereinfachen:

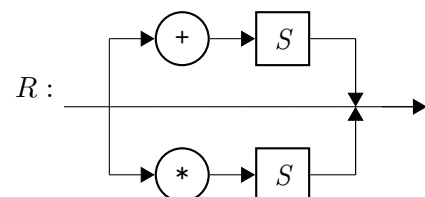
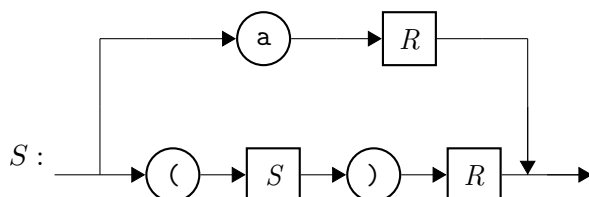
1. Ein Teilgraph  $\boxed{B}$  mit  $B \in N$  kann durch den Syntaxgraphen für  $B$  ersetzt werden (*Substitutionsregel*).
2. Identische Teilgraphen mit gemeinsamem Ausgang bzw. Eingang werden zusammengefasst:



3. Tritt am Ausgang eines Syntaxgraphen für  $B$  der Knoten  $\boxed{B}$  auf, so ersetzt man diesen durch eine Kante auf den Eingang des Syntaxgraphens (*Iteration statt Endrekursion*):



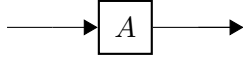
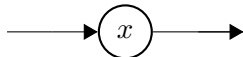
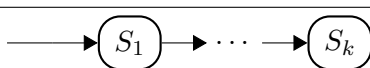
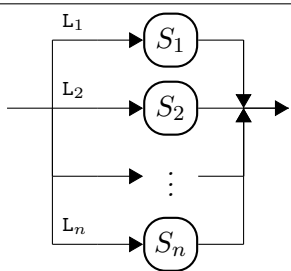
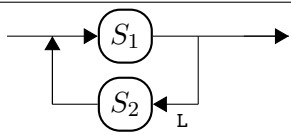
**Beispiel 5.32:** Die Syntaxgraphen für die Grammatik  $G_2$  aus Beispiel 5.4 sind:







Jedem Teilgraphen T wird ein Programmstück P(T) durch folgende Tabelle zugeordnet:

Teilgraph T	zugeordnetes Programmstück P(T)
Nonterminal 	A(); -- Aufruf des Unterprogramms A
Terminal 	if (token == x) getToken(); else error();
Sequenz 	{ P(S <sub>1</sub> ); ... ; P(S <sub>k</sub> ); }
Verzweigung (L <sub>i</sub> sind die Wegweiser) 	switch (token) { case in L <sub>1</sub> : P(S <sub>1</sub> ); break; : case in L <sub>n</sub> : P(S <sub>n</sub> ); break; default: error(); }
Schleife 	while (true) { P(S <sub>1</sub> ); if (token != L) break; P(S <sub>2</sub> ); } }

Die so erhaltenen Unterprogramme werden vereinfacht (z.B. überflüssige Abfragen entfernt).

**Beispiel 5.34:** Ein Parser für Beispiel 5.32 nach Vereinfachung der Unterprogramme:

```
char token;                                /* globale Variable fuer das Vorschau-Symbol */

void getToken(); { /* einfacher Scanner; überliest white space, liefert bei EOF
                  das Token $, jedes andere Zeichen wird als Token übergeben */
char c;
    while (isspace(c = getchar())) ;
    if (c == EOF) token = '$';
    else token = c;
}

void error() {
    printf ("Fehler in der Eingabe\n");
    exit();
}

void S() {
    switch (token) {
        case 'a':  getToken();
                   break;
        case '(':  getToken();
                   S();
                   if (token == ')') getToken();
                   else error();
                   break;
        default:   error();
    }
}
```

```

    if (token == '+' || token == '*') {
        getToken();
        S();
    }
    else
        if (token != '$' && token != ')') error();
    return;
}

void main () {
    getToken();
    S();
    if (token == '$')
        printf ("Wort gehoert zur Sprache\n");
    else error();
}

```

### 5.3.5 Beispielparser für eine Mini-Sprache mit arithmetischen Ausdrücken

Es wird ein Parser für eine einfache „Programmiersprache“ zur Abarbeitung arithmetischer Ausdrücke konstruiert.

Die Grammatik der Sprache wird durch folgende Produktionen beschrieben:

<code>&lt;program&gt;</code>	$\rightarrow$ <code>begin &lt;statementlist&gt; end</code>
<code>&lt;statementlist&gt;</code>	$\rightarrow$ <code>&lt;statement&gt;   &lt;statement&gt;; &lt;statementlist&gt;</code>
<code>&lt;statement&gt;</code>	$\rightarrow$ <code>ident <math>\leftarrow</math> &lt;expression&gt;   print &lt;expression&gt;</code>
<code>&lt;expression&gt;</code>	$\rightarrow$ <code>&lt;sign&gt;&lt;termlist&gt;</code>
<code>&lt;sign&gt;</code>	$\rightarrow$ <code> +   -   <math>\varepsilon</math></code>
<code>&lt;termlist&gt;</code>	$\rightarrow$ <code>&lt;term&gt;   &lt;term&gt; + &lt;termlist&gt;   &lt;term&gt; - &lt;termlist&gt;</code>
<code>&lt;term&gt;</code>	$\rightarrow$ <code>&lt;factorlist&gt;</code>
<code>&lt;factorlist&gt;</code>	$\rightarrow$ <code>&lt;factor&gt;   &lt;factor&gt; * &lt;factorlist&gt;  </code> <code>                  &lt;factor&gt; / &lt;factorlist&gt;</code>
<code>&lt;factor&gt;</code>	$\rightarrow$ <code>ident   number   ( &lt;expression&gt; )</code>

Als Variablennamen sind nur a, b, ..., z erlaubt; als Zahlen treten nur ganze Zahlen auf.

Der Scanner erkennt die Schlüsselwörter `begin`, `end` und `print` und liefert dann das jeweilige Token `begin`, `end` bzw. `print` zurück. Am Ende der Eingabe (durch das Zeichen “\$” markiert) gibt der Scanner das Token `eof` zurück. Bei einer Variable liefert er das Paar (`ident`, index), wobei der Index des *i*-ten Buchstabens im Alphabet *i* ist; bei Zahlen liefert er das Paar (`number`, Zahlenwert). Alle anderen Zeichen werden direkt übergeben. Die Rückgabewerte des Scanners stehen in den Variablen `token` bzw. `tokenValue`.

Bei Erkennen eines Syntaxfehlers wird die Prozedur `error("...")` aufgerufen.

Man kann die Parserprozeduren leicht erweitern, so dass die erkannten arithmetischen Ausdrücke gleich während des Parsens ausgewertet werden, und erhält einen **Interpretierer** (Programmtext des ursprünglichen Parsers ist blau dargestellt, die Erweiterungen zum Interpretierer kursiv).

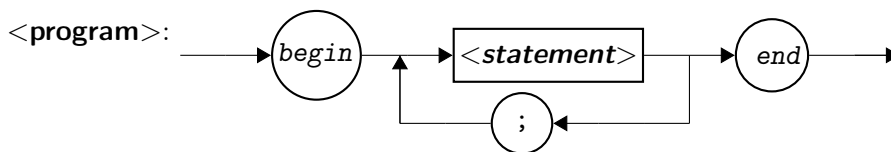
Die Prozeduren `expression`, `term` und `factor` liefern im Interpretierer als Funktionen Teilergebnisse zurück. Das Hauptprogramm bekommt ein Feld zur Speicherung der Variablenwerte.

```

void main() {
    int var[26];                                /* Feld für die Werte von Variablen */
    getToken();
    program();                                  /* program ist das Startsymbol der Grammatik */
    if (token == eof)
        printf("Wort gehoert zur Sprache\n");
    else error("main: eof expected");
}

```

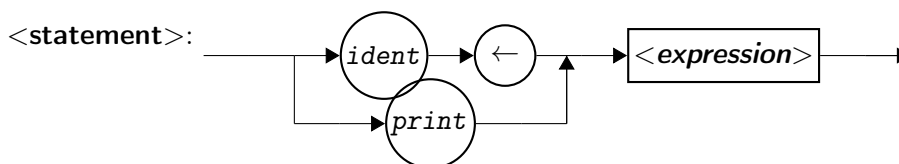
Die Syntaxgraphen und die zugeordneten Prozeduren sind dann:



```

void program() {
    if (token != begin) error("program: begin expected");
    do {
        getToken();
        statement();
    } while (token == ';' );
    if (token != end) error("program: end expected");
    getToken();
}

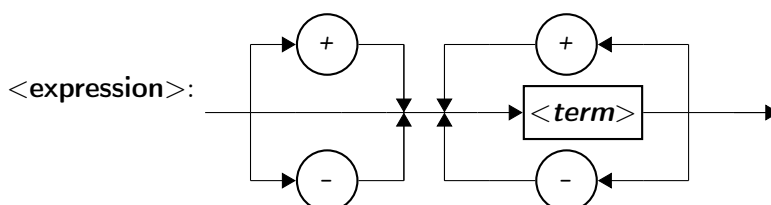
```



```

void statement() {
    int ewert, index;
    Boolean assign;
    if (token == ident) {
        index = tokenValue;
        assign = true;
        getToken();
        if (token != '←') error("statement: ← expected");
    }
    else
        if (token != print) error("statement: print expected");
        else assign = false;
    getToken();
    ewert = expression(); /* Interpreter: expression liefert Wert zurück */
    if (assign) var[index] = ewert;
    else printf("%d\n", ewert);
}

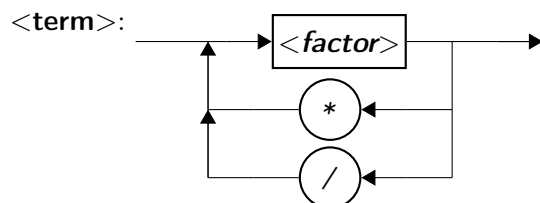
```



```

int expression() {                                     /* void beim Parser */
    int twert, ewert = 0;
    Boolean plus = true;
    if (token == '-') plus = false;
    if ((token == '-') || (token == '+')) getToken();
    do {
        twert = term();                                /* Interpreter: term liefert Wert zurück */
        if (plus) ewert = ewert + twert; else ewert = ewert - twert;
        if ((token != '-') && (token != '+')) break;
        plus = token == '+';
        getToken();
    } while (true);
    return (ewert);
}

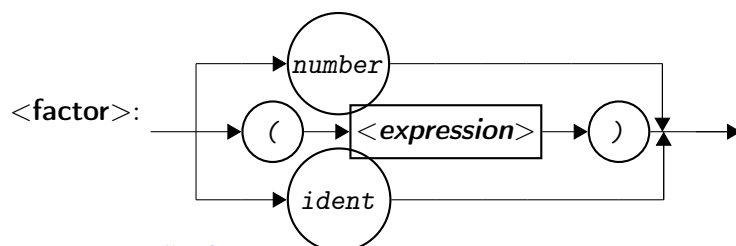
```



```

int term() {                                           /* void beim Parser */
    int fwert, twert = 1;
    Boolean mulop = true;
    do {
        fwert = factor();                              /* Interpreter: factor liefert Wert zurück */
        if (mulop) twert = twert * fwert; else twert = twert / fwert;
        if ((token != '*') && (token != '/')) break;
        mulop = token == '*';
        getToken();
    } while (true);
    return (twert);
}

```



```

int factor() {                                         /* void beim Parser */
    int fwert;
    switch (token) {
        case ident:  fwert = var[tokenValue];
                      getToken();
                      break;
        case number: fwert = tokenValue;
                      getToken();
                      break;
    }
}

```

```

    case '(':    getToken();
                fwert = expression();
                if (token != ')') error("factor: ) expected");
                getToken();
                break;
    default:    error("factor: identifier, number or ( expected");
}
return (fwert);
}

```

Um einen *tabellengesteuerten* Parser zu erzeugen, geht man anders vor: Die kontextfreie Grammatik wird durch Entfernen gemeinsamer Präfixe in eine geeignete Form gebracht (Anhang C.1). Als nächstes werden die **First**- und **Follow**-Funktionen für die nichtterminalen Symbole der Grammatik berechnet. Hier wird ein zu großer Aufwand betrieben, weil die Funktionswerte für *jedes* Symbol bestimmt werden (Anhang C.2). Danach kann die Syntaxanalyse-Tabelle gefüllt werden (Anhang C.3). Diese Art der Konstruktion ist daher eher für eine automatische Erstellung durch einen **Parser-Generator** (für LL(k)-Grammatiken z.B. ANTLR oder JavaCC) geeignet.

## 5.4 Deterministische linksreduzierende Syntaxanalyse

Bei der **linksreduzierenden Syntaxanalyse** (*bottom-up parsing*) wird das Eingabewort ebenfalls von links nach rechts gelesen. Es wird jedoch eine **Linksreduktion** erzeugt, d.h. der Ableitungsbaum wird von unten nach oben erzeugt. Dabei erhält man die Produktionen einer Rechtsableitung, allerdings in umgedrehter Reihenfolge. Das Verfahren soll deterministisch arbeiten<sup>13</sup>.

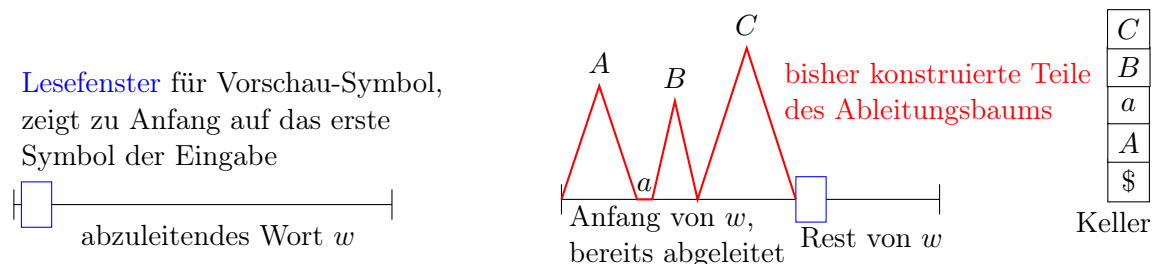


Abb. 5.11: Linksreduzierende Syntaxanalyse.

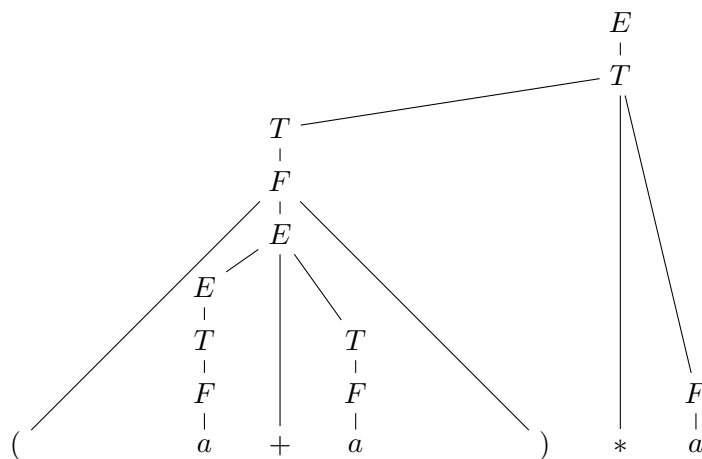
### 5.4.1 Reduktionsstelle, tabellengesteuerter Bottom-up-Parser

Wir müssen im abzuleitenden Wort bzw. im bereits konstruierten Teil des Ableitungsbaumes die rechte Seite einer Produktion identifizieren (im Beispiel 5.35 blau markiert) und durch die linke Seite dieser Produktion ersetzen.

Für eine Rechtsableitung  $S \xRightarrow{*} \gamma Ax \Rightarrow \gamma \alpha x$  und eine Produktion  $A \rightarrow \alpha$  heißt (die Position von)  $\alpha$  die **Reduktionsstelle** oder der **Griff** (*handle*) von  $\gamma \alpha x$ .

**Beispiel 5.35:** Betrachte das Wort  $(a+a)*a$  und die Grammatik  $G_4 = (N, T', P, E)$  für arithmetische Ausdrücke. Sei  $N = \{E, T, F\}$ ,  $T' = \{a, +, *, (, )\}$  und  $P$  enthalte folgende Produktionen: (1)  $E \rightarrow E + T$  (2)  $E \rightarrow T$  (3)  $T \rightarrow T * F$  (4)  $T \rightarrow F$  (5)  $F \rightarrow (E)$  (6)  $F \rightarrow a$

<sup>13</sup> Allgemeiner könnte man fordern, dass die anzuwendende Produktion zwar nicht eindeutig sein muss, aber dass es dann egal ist, welche der möglichen Produktionen angewendet wird, da alle zugehörigen Ableitungsfolgen in jedem Fall wieder zusammenfließen (**Church-Rosser-Sprachen**).

Abb. 5.12: Syntaxbaum für  $(a + a) * a$ .

Linksreduktion:  $(a + a) * a \Leftarrow (F + a) * a \Leftarrow (T + a) * a \Leftarrow (E + a) * a \Leftarrow (E + F) * a \Leftarrow (E + T) * a \Leftarrow (E) * a \Leftarrow F * a \Leftarrow T * a \Leftarrow T * F \Leftarrow T \Leftarrow E$

Rechtsableitung:  $E \xrightarrow{(2)} T \xrightarrow{(3)} T * F \xrightarrow{(6)} T * a \xrightarrow{(4)} F * a \xrightarrow{(5)} (E) * a \xrightarrow{(1)} (E + T) * a \xrightarrow{(4)} (E + F) * a \xrightarrow{(6)} (E + a) * a \xrightarrow{(2)} (T + a) * a \xrightarrow{(4)} (F + a) * a \xrightarrow{(6)} (a + a) * a$

Linksrekursive Produktionen stören beim Bottom-up-Parsen offenbar nicht!

Ein bottom-up-Parser führt bei jedem Schritt eine der folgenden Operationen aus:

- Eine **shift**-Operation schreibt das Vorschau-Symbol auf den Keller und schiebt das Lese-fenster um ein Zeichen weiter.
- Eine **reduce**-Operation ist nur anwendbar, wenn die obersten Symbole auf dem Keller die rechte Seite einer Produktion bilden. Diese Symbole werden gelöscht und durch das Symbol auf der linken Seite der Produktion ersetzt.
- Die **accept**-Operation, wenn der Parser das Ende der Eingabe erkennt und auf dem Keller nur noch das Startsymbol der Grammatik gefolgt von der Endmarke \$ steht.
- Die **error**-Operation, wenn der Parser einen Syntaxfehler in der Eingabe erkennt.

Auch ein Bottom-up Parser speichert die noch zu bearbeitenden Teile des Ableitungsbaumes im Keller. Allerdings benötigen wir jetzt Zugriff auf *mehrere* Symbole am oberen Ende des Kellers. Wir steuern den Parser wieder durch eine Syntaxanalyse-Tabelle.

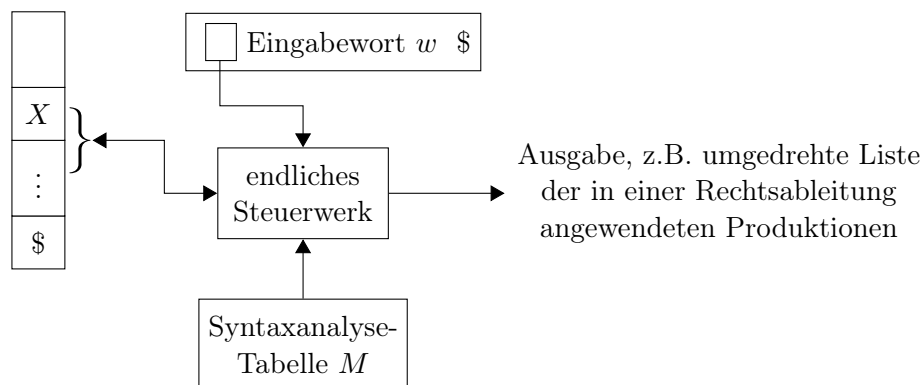


Abb. 5.13: Schema eines tabellengesteuerten Bottom-up-Parsers.

Die Arbeitsweise eines Bottom-up-Parsers wird wieder durch **Konfigurationen** beschrieben. Der Kellerinhalt wird hier um 90° im Uhrzeigersinn gedreht; das oberste Kellerelement steht rechts.

Wir vergleichen die Arbeitsweise von kontextfreier Grammatik und Kellerautomat:

kfr. Grammatik	Kellerautomat	
↓ Linksreduktion	Konfiguration	Operation
$(a + a) * a$	[\$ , $(a + a) * a$ ]	shift (
	[\$( , $a + a) * a$ ]	shift $a$
$\xleftarrow{(6)}$	[\$( $a$ , $+a) * a$ ]	reduce $F \rightarrow a$
$(F + a) * a \xleftarrow{(4)}$	[\$( $F$ , $+a) * a$ ]	reduce $T \rightarrow F$
$(T + a) * a \xleftarrow{(2)}$	[\$( $T$ , $+a) * a$ ]	reduce $E \rightarrow T$
$(E + a) * a$	[\$( $E$ , $+a) * a$ ]	shift +
	[\$( $E+$ , $a) * a$ ]	shift $a$
$\xleftarrow{(6)}$	[\$( $E + a$ , ) * $a$ ]	reduce $F \rightarrow a$
$(E + F) * a \xleftarrow{(4)}$	[\$( $E + F$ , ) * $a$ ]	reduce $T \rightarrow F$
$(E + T) * a \xleftarrow{(1)}$	[\$( $E + T$ , ) * $a$ ]	reduce $E \rightarrow E + T$
$(E) * a$	[\$( $E$ , ) * $a$ ]	shift )
$\xleftarrow{(5)}$	[\$( $E$ ) , $*a$ ]	reduce $F \rightarrow (E)$
$F * a \xleftarrow{(4)}$	[\$ $F$ , $*a$ ]	reduce $T \rightarrow F$
$T * a$	[\$ $T$ , $*a$ ]	shift *
	[\$ $T*$ , $a$ ]	shift $a$
$\xleftarrow{(6)}$	[\$ $T * a$ , \$]	reduce $F \rightarrow a$
$T * F \xleftarrow{(3)}$	[\$ $T * F$ , \$]	reduce $T \rightarrow T * F$
$T \xleftarrow{(2)}$	[\$ $T$ , \$]	reduce $E \rightarrow T$
$E$	[\$ $E$ , \$]	accept
↑ Rechtsableitung	[Keller , Resteingabe]	Operation

Die rechte Seite der angewendeten Produktion bzw. der Teil des Kellers, der im nächsten Schritt reduziert wird, ist blau markiert. Die Linksreduktion ergibt rückwärts (von unten nach oben) gelesen eine Rechtsableitung. Der Kellerautomat zeigt –wie die Grammatik– jeden Reduktionsschritt, darüberhinaus aber auch jede shift-Operation.

Im 9. Reduktionsschritt (rot) wird nicht mit der Produktion  $E \rightarrow T$  reduziert, da dieser Schritt in eine **Sackgasse**  $T * a \leftarrow E * a \leftarrow E * F \leftarrow E * T$  führen würde, also nicht bis zum Startsymbol der Grammatik verlängert werden kann! Zu einem Wort einer eindeutigen Grammatik gibt es genau einen Ableitungsbaum (eine Rechtsableitung, eine Linksreduktion), d.h. es gibt in jedem Reduktionsschritt nur eine „richtige“ zu ersetzende rechte Seite. Diese Reduktionsstelle zu finden, ist nicht so einfach! Der linkeste reduzierbare Teilstring ist offenbar nicht immer der Griff!

Natürlich soll der Bottom-up-Parser **deterministisch** arbeiten, also auch Sackgassen vermeiden.

#### 5.4.2 Aufbau eines deterministischen LR-Parsers

Kontextfreie Grammatiken, deren Bottom-up Parser deterministisch ist, heißen **LR(k)-Grammatiken** (Lesen der Eingabe von **L**inks nach rechts, Erzeugen einer umgekehrten **R**echtsableitung und **k** Zeichen als Vorschau, „**L**inks**R**eduktion mit Vorschautiefe **k**“).

Die theoretischen Grundlagen zum Bottom-up Parsen sind komplizierter als beim Top-down-Parsen. Viele **Parser-Generatoren** arbeiten jedoch mit dieser Methode, da viel mehr Sprachen

durch LR(1)-Grammatiken als durch LL(1)-Grammatiken beschreibbar sind. In der Praxis werden Syntaxanalyse-Tabellen zu LR(1)-Grammatiken jedoch sehr groß; Parser-Generatoren beschränken sich daher oft –wie yacc und Bison– auf die Teilklasse der **LALR(1)-Grammatiken**.

Man kann zeigen, dass die Menge der Kellerbelegungen aller erfolgreichen Reduktionen eine reguläre Sprache bilden. Für diese Sprache konstruiert ein Parser-Generator einen deterministischen endlichen Automaten, z.B. für Grammatik  $G_4$ :

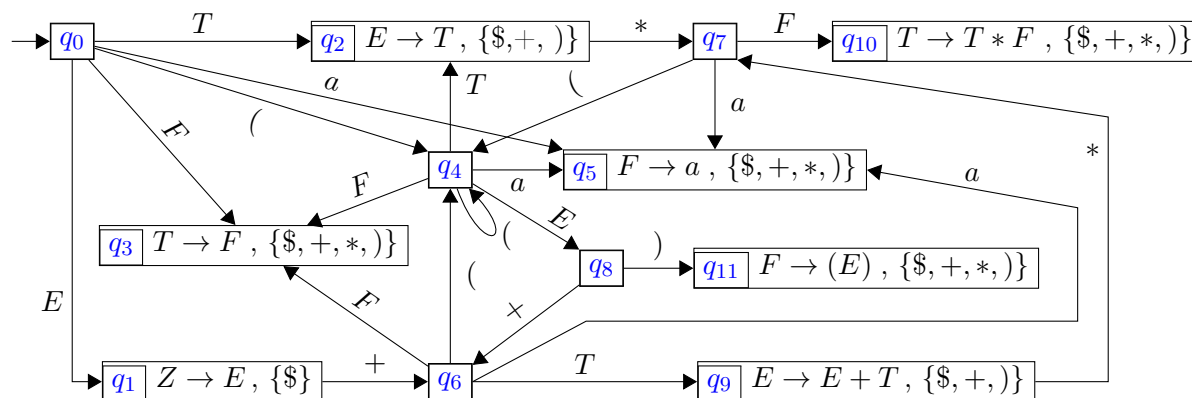


Abb. 5.14: deterministischer endlicher LALR(1)-Automat für die (erweiterte) Grammatik  $G_4$ .

Kommt man in einen Zustand mit einer Produktion und liegt das Vorschau-symbol in der angegebenen Menge, so reduziert man mit dieser Produktion. Sonst führt man eine shift-Operation aus und das eingelesene/bearbeitete Symbol bestimmt den Übergang in den Nachfolgezustand.

Damit der Automat nach einer reduce-Operation nicht wieder über den bisherigen Kellerinhalt laufen muss, merkt man sich nach jeder Operation den erreichten Zustand – auf dem Keller. Der Keller enthält also statt der Grammatiksymbole jetzt Zustände des endlichen Automaten.

Aus diesem endlichen Automaten leitet ein Parser-Generator die Syntaxanalyse-Tabelle ab, die jetzt aus zwei Teilen besteht, der Aktionstabelle und der Sprungtabelle.

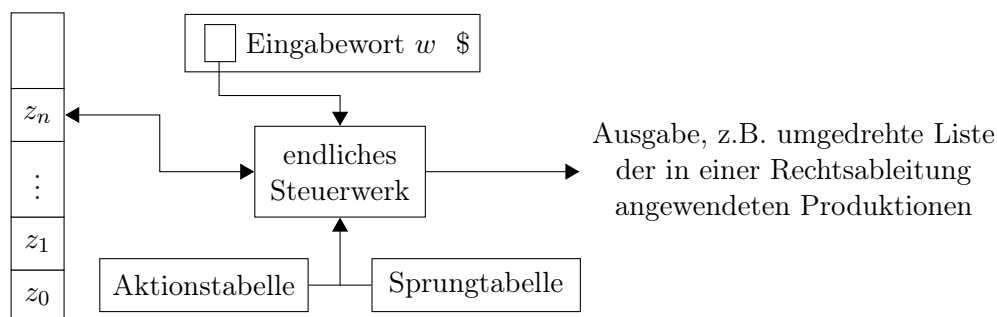


Abb. 5.15: Aufbau eines LR-Parsers.

Die  $z_i$  sind Zustände des endlichen Automaten.

Die Spalten der Aktionstabelle sind mit den Vorschau-symbolen (den terminalen Symbolen der Grammatik bzw. der Endmarke  $\$$ ) und die Spalten der Sprungtabelle mit den nichtterminalen Symbolen der Grammatik markiert. Die Zeilen beider Tabellen sind mit Zuständen markiert. Üblicherweise werden die Produktionen der Grammatik sowie die Zustände des endlichen Automaten durchnummeriert. Der Startzustand ist 0.



Ein Eintrag der **Aktionstabelle** (*action table*) hat eine der Formen:

*si* (**shift** <neuer Zustand>) oder  
*ri* (**reduce** <Produktion>) oder  
*acc* (**accept**).

Ein Eintrag in der **Sprungtabelle** (*goto table*) hat die Form:

*i* (<neuer Zustand>)

Freie Felder in den Tabellen bezeichnen Fehlerzustände.

**Beispiel 5.36:** Parser-Generatoren –wie **yacc**, **bison** oder **happy**– liefern für die Grammatik  $G_4$  die Syntaxanalyse-Tabelle in Abb. 5.16.

Zustand	aktion						sprung		
	<i>a</i>	+	*	(	)	\$	<i>E</i>	<i>T</i>	<i>F</i>
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4			9	3	
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Abb. 5.16: Syntaxanalyse-Tabelle für  $G_4$ .

Der tabellengesteuerte LR-Parser arbeitet wie folgt:

**Initiale Situation:** Das Lesefenster befindet sich auf dem ersten Zeichen des Eingabeworts, das mit der Endmarke \$ abgeschlossen ist. Im Keller befindet sich nur der Startzustand 0.

**Arbeitsschritt**<sup>14</sup>: Sei  $z$  der Zustand oben auf dem Keller und sei  $a$  das Vorschau-Symbol.

- Ist  $\text{aktion}[z, a] = \text{shift } z'$ , dann schreibe den neuen Zustand  $z'$  auf den Keller. Das Lesefenster wandert ein Symbol nach rechts.
- Ist  $\text{aktion}[z, a] = \text{reduce } A \rightarrow \alpha$ , dann lösche die obersten  $|\alpha|$  Einträge vom Keller. Sei danach  $z'$  der Zustand oben auf dem Keller. Schreibe den Zustand  $\text{sprung}[z', A]$  auf den Keller. Gib die Produktion  $A \rightarrow \alpha$  aus.
- Ist  $\text{aktion}[z, a] = \text{accept}$ , dann beende das Parsing. Das Eingabewort gehört zur Sprache.
- Ist  $\text{aktion}[z, a] = \text{error}$  (leerer Eintrag in der Aktionstabelle), dann beende das Parsing mit einer Fehlermeldung. Das Eingabewort gehört nicht zur Sprache.

**Beispiel 5.37:** Die Parserschritte aus Beispiel 5.35 notieren wir mit folgenden Konfigurationen:

[ 0 , ( a + a ) * a \$ ]		[ 0 4 8 , ) * a \$ ]
└─ [ 0 4 , a + a ) * a \$ ]		(1) └─ [ 0 4 8 11 , * a \$ ]
└─ [ 0 4 5 , + a ) * a \$ ]		└─ [ 0 3 , * a \$ ]
(6) └─ [ 0 4 3 , + a ) * a \$ ]		(5) └─ [ 0 2 , * a \$ ]
(4) └─ [ 0 4 2 , + a ) * a \$ ]		(4) └─ [ 0 2 7 , a \$ ]
(2) └─ [ 0 4 8 , + a ) * a \$ ]		└─ [ 0 2 7 5 , \$ ]
└─ [ 0 4 8 6 , a ) * a \$ ]		└─ [ 0 2 7 10 , \$ ]
└─ [ 0 4 8 6 5 , ) * a \$ ]		(6) └─ [ 0 2 , \$ ]
(6) └─ [ 0 4 8 6 3 , ) * a \$ ]		(3) └─ [ 0 1 , \$ ]
(4) └─ [ 0 4 8 6 9 , ) * a \$ ]		(2) └─ accept

<sup>14</sup> Ja, man kann zusätzlich das gelesene Symbol (bei einer shift-Operation) bzw. die linke Seite der angewendeten Produktion (bei einer reduce-Operation) auf den Keller schreiben. Nötig für die Funktionsweise ist das nicht. (Natürlich muss man dann bei einer reduce-Operation doppelt so viele Einträge löschen.)

### 5.4.3 Mehrdeutige Grammatik beim Bottom-up Parsing

Ein Parser-Generator liefert Fehlermeldungen für Grammatiken, für die kein deterministisches Bottom-up Parsing möglich ist. Dabei werden Situationen des Parsers beschrieben, in der

- eine shift- und eine reduce-Aktion durchgeführt werden könnte (**shift/reduce-Konflikt**). Dieser Konflikt tritt z.B. auf, wenn die rechte Seite einer Produktion gleichzeitig *Präfix* einer anderen Produktion mit gleichen Symbol auf der linken Seite ist.
- reduce-Aktionen für verschiedene Produktionen ausgeführt werden könnten (**reduce/reduce-Konflikt**). Dieser Konflikt tritt z.B. auf, wenn die rechte Seite einer Produktion zugleich *Suffix* einer anderen Produktion mit gleichem Symbol auf der linken Seite ist.

**Beispiel 5.38:** In der folgenden mehrdeutigen Grammatik tritt ein shift/reduce-Konflikt auf:

```
<stmt> → if <expr> then <stmt>
        | if <expr> then <stmt> else <stmt>
        | ...
```

Man bevorzugt in diesem Fall die shift-Operation, wodurch der else-Teil der zuletzt bearbeiteten if-Anweisung zugeordnet wird.

```
if <expr> then if <expr> then <stmt> else <stmt>
```

Besser ist natürlich die Vermeidung der Mehrdeutigkeit durch eine schließende Klammer *fi*.

**Beispiel 5.39:** Bei folgender Produktion tritt ein reduce/reduce-Konflikt auf:

```
<liste> → <liste> '+' <liste> '+' <liste>
        | <liste> '+' <liste>
```

Parser-Generatoren lösen diese Mehrdeutigkeiten mit zwei zusätzlichen Regeln, die man auch bei der Konstruktion einer Grammatik ausnutzen kann:

- bei shift/reduce-Konflikten wählt man die shift-Operation.
- bei reduce/reduce-Konflikten reduziert man mit der ersten anwendbaren Produktion – in der Reihenfolge der Grammatik-Beschreibung.

Mehrdeutigkeiten treten z.B. bei Grammatiken für arithmetische Ausdrücke auf, bei denen man auf die Konstruktion mit *<term>* und *<factor>* verzichtet. In *Yacc* bzw. *Bison* kann man sogar mehrdeutige Grammatiken zur Beschreibung arithmetischer Ausdrücke verwenden. Man muss dem Parser-Generator nur die **Priorität** und die **Assoziativität** der verwendeten Operatoren mitteilen. Dann löst der Generator shift/reduce-Konflikte entsprechend der gemachten Angaben.

Um einen shift/reduce-Konflikt zu lösen, schauen wir uns die letzte Operation auf dem Keller sowie die Operation in der Vorschau an:

- Ist die Priorität der Operation auf dem Keller größer als die der Operation in der Vorschau, so soll zuerst die Operation auf dem Keller ausgeführt werden, dazu wird reduziert;
- ist die Priorität der Operation auf dem Keller kleiner als die Priorität der Operation in der Vorschau, so soll zuerst die Operation der Vorschau ausgeführt werden, dazu muss der zweite Operand eingelesen werden, also ein shift durchgeführt werden.
- Bei gleichen Prioritäten entscheidet die Assoziativität: Sind beide Operationen
  - linksassoziativ, so wird zuerst die Operation auf dem Keller ausgeführt, also reduziert;
  - rechtsassoziativ, so wird zuerst die Operation der Vorschau ausgeführt, also geshiftet.

Dadurch kann man mehrdeutige, aber meist deutlich kleinere Grammatiken zum Parsen verwenden und die Syntaxanalyse-Tabellen kleiner halten.

**Beispiel 5.40:** Betrachte die Grammatik

$G_5 = (\{E\}, \{a, -, /, (, )\}, P, E)$

mit den Produktionen

- (1)  $E \rightarrow E - E$
- (2)  $E \rightarrow E / E$
- (3)  $E \rightarrow (E)$
- (4)  $E \rightarrow a$

Die Grammatik  $G_5$  ist mehrdeutig.

**Beispiel 5.41:** Die Grammatik  $G_5$  führt zu einer Syntaxanalyse-Tabelle mit shift-reduce Konflikten:

Gibt man  $/$  eine höhere Priorität als  $-$  und berücksichtigt, dass beide Operationen linksassoziativ sind, so werden die shift/reduce-Konflikte im Zustand 7 durch r1 und s5, die im Zustand 8 durch r2 und r2 gelöst.

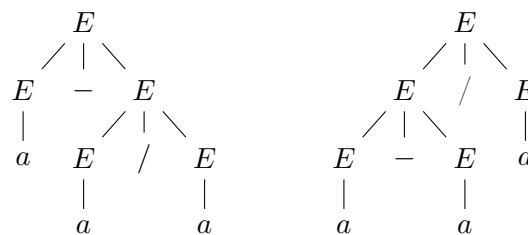
Analysiert man z.B. den Ausdruck  $a - a/a - a$ , so erhält man folgende Parserschritte:

```

[ 0          , a - a / a - a $ ]
├── [ 0 3      , - a / a - a $ ]
├── (4) [ 0 1    , - a / a - a $ ]
├── [ 0 1 4    , a / a - a $ ]
├── [ 0 1 4 3  , / a - a $ ]
├── (4) [ 0 1 4 7 , / a - a $ ]
s5 oder r1? shift 5, denn
a-a soll noch nicht berechnet werden, da
das folgende / höhere Priorität als - hat.
├── [ 0 1 4 7 5 , a - a $ ]
├── [ 0 1 4 7 5 3 , - a $ ]
├── (4) [ 0 1 4 7 5 8 , - a $ ]

```

Betrachte z.B. die Ableitungsbäume zu  $a - a/a$ :



Zust.	aktion					sprung
	a	-	/	( )	\$	E
0	s3			s2		1
1		s4	s5		acc	
2	s3			s2		6
3		r4	r4		r4	
4	s3			s2		7
5	s3			s2		8
6		s4	s5		s9	
7		s4/r1	s5/r1		r1	r1
8		s4/r2	s5/r2		r2	r2
9		r3	r3		r3	r3

**s4 oder r2?** reduce 2, denn  $a/a$  soll berechnet werden, da das folgende  $-$  kleinere Priorität als  $/$  hat.

├── (2) [ 0 1 4 7 , - a \$ ]

**s4 oder r1?** reduce 1, denn die erste Subtraktion soll ausgeführt werden, da  $-$  linksassoziativ ist.

├── (1) [ 0 1 , - a \$ ]

├── [ 0 1 4 , a \$ ]

├── [ 0 1 4 3 , \$ ]

├── (4) [ 0 1 4 7 , \$ ]

├── (1) [ 0 1 , \$ ]

├── accept

Alternativ kann man versuchen, eine Grammatik mit Hilfe der Informationen über Prioritäten und Assoziativitäten zu einer eindeutigen Grammatik umzuformen.

**Beispiel 5.42:** Betrachte die Grammatik  $G_5$  aus Beispiel 5.40. Folgende Prioritäten werden vergeben, um die Grammatik eindeutig zu machen. Außerdem sind die Operatoren  $/$  und  $-$  linksassoziativ.

Prioritäten:                      eindeutige Grammatik

0.  $E - E$                        $E_0 \rightarrow E_1 \mid E_0 - E_1$
1.  $E / E$                        $E_1 \rightarrow E_2 \mid E_1 / E_2$
2.  $a \mid (E)$                        $E_2 \rightarrow a \mid (E_0)$

Jede Prioritätsstufe kennzeichnen wir durch ein eigenes Nichtterminal und reichen die Nichtterminale durch die Prioritätsstufen (von der niedrigsten zur höchsten) durch.

Die Linksassoziativität bilden wir ab, indem der *linke* Operand einer binären Operation auf der gleichen Prioritätsstufe bleibt. Bei einer rechtsassoziativen binären Operation bleibt der rechte Operand auf der gleichen Prioritätsstufe.

### 5.4.4 Vermeidung von Sackgassen

Natürlich soll ein Bottom-up-Parser **deterministisch** arbeiten, also auch Sackgassen vermeiden.

Wir wissen bereits, dass es zu einem Wort einer eindeutigen Grammatik genau einen Ableitungsbaum (eine Rechtsableitung, eine Linksreduktion) gibt, d.h. es gibt bei jedem Reduktionsschritt nur eine „richtige“ zu ersetzende rechte Seite. Aber wie finden wir diese?

Um Sackgassen zu vermeiden, müssen wir raffiniert vorgehen! Leider nützt uns dabei weder eine verlängerte Vorschau noch ein tieferer, aber (auf  $k$  Zeichen) begrenzter Blick in den Keller:

**Beispiel 5.43:** Die Grammatik  $G_{ab}$  mit den Produktionen

$$S \rightarrow aA \mid bB, \quad A \rightarrow d \mid cAc, \quad B \rightarrow d \mid cBc.$$

erzeugt die Sprache  $L(G_{ab}) = \{a, b\}^* \{c^n d c^n \mid n \geq 0\}$ .

Für den oberen Kellerinhalt  $c^k d$  und die Vorschau  $c^k$  ( $k$  beliebig!) sind beide Produktionen  $A \rightarrow d$  und  $B \rightarrow d$  anwendbar, aber eine Produktion führt in eine Sackgasse!

Idee: Wir bestimmen alle **erfolgreichen Kellerinhalte** des Parsers, also die, die während einer *akzeptierenden* Berechnung entstehen können und erlauben nur noch **gültige Parseroperationen**, also solche, die wieder zu einem erfolgreichen Kellerinhalt führen.

**Beispiel 5.44:** Erfolgreiche Kellerinhalte (ohne  $\$$ ) für den Parser der Grammatik  $G_{ab}$  sind:

$$\begin{aligned} & \{\varepsilon\} \cup \{ac^n \mid n \geq 0\} \cup \{ac^n d \mid n \geq 0\} \cup \{ac^n A \mid n \geq 0\} \cup \{ac^n Ac \mid n \geq 1\} \\ & \cup \{bc^n \mid n \geq 0\} \cup \{bc^n d \mid n \geq 0\} \cup \{bc^n B \mid n \geq 0\} \cup \{bc^n Bc \mid n \geq 1\} \cup \{S\} \end{aligned}$$

Auf jeden erfolgreichen Kellerinhalt sind (evtl. mehrere) Parseroperationen anwendbar. Nur einige davon liefern einen erfolgreichen Kellerinhalt als Ergebnis! Z.B. können die erfolgreichen Kellerinhalte  $ac^n d$  und  $bc^n d$  mit den Produktionen  $A \rightarrow d$  bzw.  $B \rightarrow d$  reduziert werden, aber von den Ergebnissen  $ac^n A$ ,  $bc^n B$ ,  $ac^n B$ ,  $bc^n A$  sind nur die ersten beiden erfolgreiche Kellerinhalte.

Die Menge der erfolgreichen Kellerinhalte ist –wie bei  $G_{ab}$ – i.Allg. unendlich, so dass es unmöglich scheint festzustellen, welche Operation für welchen erfolgreichen Kellerinhalt gültig ist. Glücklicherweise können wir die Menge der erfolgreichen Kellerinhalte stets in *endlich viele* Äquivalenzklassen aufteilen: Zwei erfolgreiche Kellerinhalte gehören zur gleichen Äquivalenzklasse, wenn sie dieselbe Menge gültiger Operationen haben. Eine kontextfreie Grammatik  $G = (N, T, P, S)$  hat  $|T|$  shift- und  $|P|$  reduce-Operationen, also höchstens  $2^{|T|+|P|}$  Äquivalenzklassen.<sup>15</sup>

**Beispiel 5.45:** Für  $G_{ab}$  gibt es folgende Äquivalenzklassen und zugehörige gültige Operationen:

Äquivalenzklasse		gültige Operationen
$\{\varepsilon\}$	$[\varepsilon]$	shift $a$ , shift $b$
$\{ac^n \mid n \geq 0\} \cup \{bc^n \mid n \geq 0\}$	$[ac^* \mid bc^*]$	shift $d$ , shift $c$
$\{ac^n A \mid n \geq 1\} \cup \{bc^n B \mid n \geq 1\}$	$[ac^+ A \mid bc^+ B]$	shift $c$
$\{ac^n d \mid n \geq 0\}$	$[ac^* d]$	reduce $A \rightarrow d$
$\{aA\}$	$[aA]$	reduce $S \rightarrow aA$
$\{ac^n Ac \mid n \geq 1\}$	$[ac^+ Ac]$	reduce $A \rightarrow cAc$
$\{bc^n d \mid n \geq 0\}$	$[bc^* d]$	reduce $B \rightarrow d$
$\{bB\}$	$[bB]$	reduce $S \rightarrow bB$
$\{bc^n Bc \mid n \geq 1\}$	$[bc^+ Bc]$	reduce $B \rightarrow cBc$
$\{S\}$	$[S]$	accept

<sup>15</sup> In Kapitel 5.5.5 werden wir die Äquivalenzklassen durch die Vorschausymbole verfeinern. In Kapitel 5.5.7 fassen wir alle reduce-Operationen zu *einer* Operation zusammen und vergrößern die Äquivalenzklassen entsprechend.

Etwas müssen wir die Äquivalenzrelation noch verfeinern:

- Die Äquivalenzrelation soll **rechtsinvariant** sein: Werden äquivalente, erfolgreiche Kellerinhalte  $\delta_1$  und  $\delta_2$  mit einem Symbol  $X$  zu erfolgreichen Kellerinhalten verlängert, so sollen sie äquivalent bleiben:  $[\delta_1] = [\delta_2] \Rightarrow [\delta_1 X] = [\delta_2 X]$  (bzw.  $[\delta_1 X] \neq [\delta_2 X] \Rightarrow [\delta_1] \neq [\delta_2]$ ).
- Bei einer Reduktion soll die anzuwendende Produktion eindeutig sein und nicht nur die zugehörigen Äquivalenzklassen. D.h. zu einer Produktion  $A \rightarrow X_1 \dots X_n$  soll es keine Produktion  $A' \rightarrow X'_1 \dots X'_n$  geben mit  $[\delta A] = [\delta A']$ ,  $[\delta X_1] = [\delta X'_1]$ ,  $\dots$ ,  $[\delta X_1 \dots X_n] = [\delta X'_1 \dots X'_n]$ . Dies gilt sicher, wenn zwei äquivalente Kellerinhalte stets mit dem gleichen Symbol enden:  $[\delta X] = [\delta X'] \Rightarrow X = X'$ .

#### Beispiel 5.46:

Um für die Grammatik  $G_{ab}$  diese Anforderungen zu erfüllen, teilen wir Äquivalenzklassen auf:

- die Klasse  $[ac^* \mid bc^*]$  in die Klassen  $[a]$ ,  $[ac^+]$ ,  $[b]$  und  $[bc^+]$ .
- die Klasse  $[ac^+ A \mid bc^+ B]$  in die Klassen  $[ac^+ A]$  und  $[bc^+ B]$ .

Diese Äquivalenzklassen fasst man nun als Zustände eines endlichen Automaten auf, die durch die Grammatiksymbole (in Abb. 5.17 an den Kanten notiert) in andere Äquivalenzklassen übergehen. Der Anfangszustand ist  $[\epsilon]$ , der einzige Endzustand eine (beliebige) Klasse  $[\gamma]$ . Die akzeptierte (reguläre!) Sprache ist dann gerade  $[\gamma]$ . Der (Fehler-)Zustand für die nicht-erfolgreichen Kellerinhalte und die Übergänge dorthin sind der Übersichtlichkeit halber weggelassen.

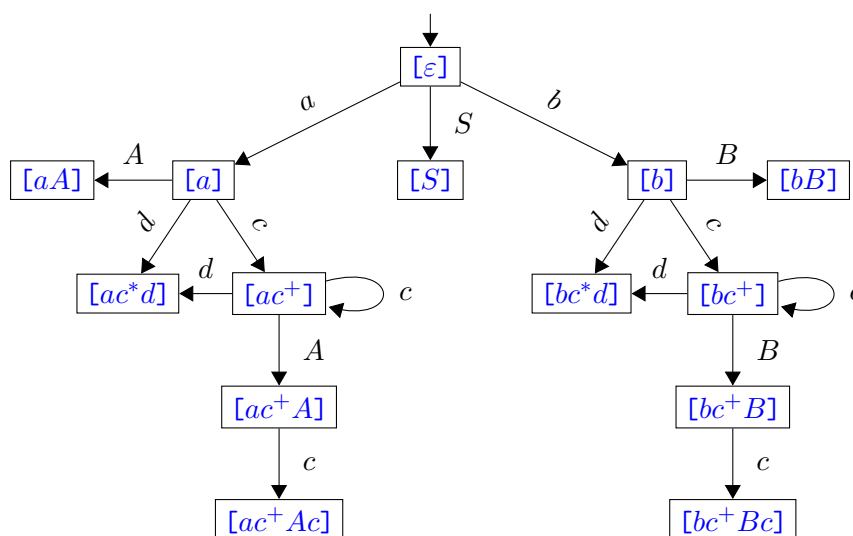


Abb. 5.17: Endlicher Automat für erfolgreiche Kellerinhalte des Parsers für Grammatik  $G_{ab}$ .

Wir wollen diese Idee, nämlich die Beschränkung auf *erfolgreiche* Kellerinhalte und *gültige* Parseroperationen sowie die Zerlegung der Menge der erfolgreichen Kellerinhalte in Äquivalenzklassen, von Kellerautomaten auf kontextfreie Grammatiken übertragen. Dazu müssen wir die shift-Operation des Kellerautomaten in den Produktionen sichtbar machen, indem wir verschiedene *Stadien* der Abarbeitung einer Produktion unterscheiden.

Wir führen dazu erfolgreiche Präfixe von Rechtsableitungen, gültige Stadien und eine Äquivalenzrelation auf der Menge der erfolgreichen Präfixe „hat gleiche Menge von gültigen Stadien“ ein. Die Äquivalenzklassen der erfolgreichen Präfixe bilden dann die Zustände eines endlichen Automaten. Es ist üblich, einen Zustand statt durch eine (reguläre) Menge von erfolgreichen Präfixen durch die Menge der für diese Präfixe gültigen Stadien zu beschreiben.

## 5.5 Ein zweiter Blick: Stadiumautomat und Parser

Schauen wir uns den Parsingprozess systematisch an. Wir beschreiben ein Verfahren, mit dem zu jeder kontextfreien Grammatik ein Kellerautomat konstruiert werden kann, der die von der Grammatik erzeugte Sprache akzeptiert. Dieser Automat ist nichtdeterministisch und für einen praktischen Einsatz daher weniger geeignet, aber wir können aus ihm effiziente Parser ableiten.

### 5.5.1 Stadiumautomat einer kontextfreien Grammatik

Eine entscheidende Rolle spielt der Begriff des **(kontextfreien Ableitungs-)Stadiums** (*item*): Wir möchten gerne markieren, wie weit eine Produktion während des Ableitungsprozesses bereits bearbeitet wurde, insbesondere die shift-Operation des Kellerautomaten sichtbar machen. Dazu ergänzen wir jede Produktion auf der rechten Seite um einen Punkt.

**Beispiel 5.47:** Wir können aus der Produktion  $S \rightarrow bAa$  folgende Stadien bilden:

$$[S \rightarrow \cdot bAa], \quad [S \rightarrow b \cdot Aa], \quad [S \rightarrow bA \cdot a] \quad \text{und} \quad [S \rightarrow bAa \cdot].$$

Aus der Produktion  $A \rightarrow \varepsilon$  entsteht das Stadium  $[A \rightarrow \cdot]$ .

Alles was links vom Punkt steht, ist bereits getan, ist „Vergangenheit“; alles was rechts vom Punkt steht, ist in „Zukunft“ noch zu tun (Reihenfolge, falls mehrere Stadien im Keller stehen?).

Das Stadium  $[A \rightarrow \alpha \cdot \beta]$  beschreibt die Situation, dass beim Versuch, aus  $A$  ein Wort  $w$  abzuleiten, aus  $\alpha$  bereits ein Präfix von  $w$  abgeleitet wurde.

**Definition:** Sei  $G = (N, T, P, S)$  eine kontextfreie Grammatik und  $A \rightarrow \alpha\beta \in P$ .

Ein **Stadium** einer Produktion von  $G$  ist ein Tripel  $(A, \alpha, \beta)$  – meist geschrieben als  $[A \rightarrow \alpha \cdot \beta]$ .

Ist  $\alpha = \varepsilon$ , also  $[A \rightarrow \cdot \beta]$ , so heißt das Stadium **initial**.

Ist  $\beta = \varepsilon$ , also  $[A \rightarrow \alpha \cdot]$ , so heißt das Stadium **final** oder **vollständig** (abgearbeitet).

Die Menge aller Stadien von  $G$  bezeichnen wir mit  $St_G$ .

Sei  $\rho = [A_1 \rightarrow \beta_1 \cdot B_1 \gamma_1] [A_2 \rightarrow \beta_2 \cdot B_2 \gamma_2] \dots [A_n \rightarrow \beta_n \cdot B_n \gamma_n] \in St_G^*$  eine Folge von Stadien.

Die **Vergangenheit** von  $\rho$  ist  $past(\rho) = \beta_1 \beta_2 \dots \beta_n$ . Die **Zukunft** von  $\rho$  ist  $fut(\rho) = \gamma_n \dots \gamma_2 \gamma_1$ .

Nun definieren wir den zu einer kontextfreien Grammatik gehörenden Stadiumautomaten. Seine Zustände und damit seine Kellersymbole sind Stadien der Grammatik. Der aktuelle Zustand ist das Stadium, an dessen rechter Seite der Automat gerade arbeitet. Im Keller darunter stehen die Stadien, deren Bearbeitung bereits begonnen, aber noch nicht beendet wurde.

Zunächst erweitern wir die Grammatik  $G$  so, dass die Terminierung des Stadiumautomaten am aktuellen Zustand abgelesen werden kann. Kandidaten für die Endzustände des Stadiumautomaten sind alle finalen Stadien  $[S \rightarrow \alpha \cdot]$  der Grammatik mit dem Startsymbol  $S$  auf der linken Seite. Tritt das Startsymbol  $S$  aber auch auf der rechten Seite einer Produktion auf, können diese Stadien oben auf dem Keller auftreten, ohne dass der Automat dann terminieren sollte, weil darunter noch unvollständig bearbeitete Stadien liegen. Darum **erweitern** wir die Grammatik  $G$  um ein neues Startsymbol  $Z$  und fügen die Produktion  $Z \rightarrow S$  zur Menge der Produktionen von  $G$  hinzu. Offensichtlich ändert sich die erzeugte Sprache durch die Erweiterung nicht. Das neue Startsymbol  $Z$  kommt dann auf keiner rechten Seite vor.

**Definition:** Sei  $G = (N, T, P, S)$  eine kontextfreie Grammatik.

Dann ist  $G^+ = (N^+, T, P^+, Z)$  mit  $N^+ = N \cup \{Z\}$  und  $P^+ = P \cup \{Z \rightarrow S\}$  die um  $Z$  **erweiterte kontextfreie Grammatik** zu  $G$ .

Als Anfangszustand des Stadiumautomaten wählen wir dann das initiale Stadium  $[Z \rightarrow \cdot S]$  und als einzigen Endzustand das finale Stadium  $[Z \rightarrow S \cdot]$ .

**Definition:** Sei  $G^+$  die um  $Z$  erweiterte kontextfreie Grammatik zu  $G = (N, T, P, S)$ .

Der Kellerautomat  $K_G = (St_{G^+}, T, \Delta, [Z \rightarrow \cdot S], \{[Z \rightarrow S \cdot]\})$  heißt **Stadiumautomat** zu  $G$ .

Die Übergangsrelation  $\Delta$  besteht aus allen Übergängen der folgenden drei Typen:

- $\Delta([X \rightarrow \beta \cdot Y\gamma], \varepsilon) = \{[X \rightarrow \beta \cdot Y\gamma] [Y \rightarrow \cdot \alpha] \mid Y \rightarrow \alpha \in P\}$  **Expansionsübergänge**
- $\Delta([X \rightarrow \beta \cdot a\gamma], a) = \{[X \rightarrow \beta a \cdot \gamma]\}$  **Leseübergänge**
- $\Delta([X \rightarrow \beta \cdot Y\gamma] [Y \rightarrow \alpha \cdot], \varepsilon) = \{[X \rightarrow \beta Y \cdot \gamma]\}$ . **Reduktionsübergänge**

**Beispiel 5.48:** Sei  $G' = (\{Z, E, T, F\}, \{+, *, (, ), a\}, P, Z)$  die (um  $Z$  als neues Startsymbol erweiterte) Grammatik von  $G_4$  mit  $P = \{Z \rightarrow E, E \rightarrow E+T \mid T, T \rightarrow T*F \mid F, F \rightarrow (E) \mid a\}$ . Die Übergangsrelation  $\Delta$  des Stadiumautomaten  $K_{G_4}$  ist in folgender Tabelle dargestellt.

altes oberes Kellerende	Eingabe	neues oberes Kellerende	Art des Übergangs
$[Z \rightarrow \cdot E]$	$\varepsilon$	$[Z \rightarrow \cdot E] [E \rightarrow \cdot E+T]$	Expansion
$[Z \rightarrow \cdot E] [E \rightarrow E+T \cdot]$	$\varepsilon$	$[Z \rightarrow E \cdot]$	Reduktion
$[Z \rightarrow \cdot E]$	$\varepsilon$	$[Z \rightarrow \cdot E] [E \rightarrow \cdot T]$	Expansion
$[Z \rightarrow \cdot E] [E \rightarrow T \cdot]$	$\varepsilon$	$[Z \rightarrow E \cdot]$	Reduktion
$[E \rightarrow \cdot E+T]$	$\varepsilon$	$[E \rightarrow \cdot E+T] [E \rightarrow \cdot E+T]$	Expansion
$[E \rightarrow \cdot E+T] [E \rightarrow E+T \cdot]$	$\varepsilon$	$[E \rightarrow E \cdot +T]$	Reduktion
$[E \rightarrow \cdot E+T]$	$\varepsilon$	$[E \rightarrow \cdot E+T] [E \rightarrow \cdot T]$	Expansion
$[E \rightarrow \cdot E+T] [E \rightarrow T \cdot]$	$\varepsilon$	$[E \rightarrow E \cdot +T]$	Reduktion
$[E \rightarrow E \cdot +T]$	$+$	$[E \rightarrow E+ \cdot T]$	Lesen
$[E \rightarrow E+ \cdot T]$	$\varepsilon$	$[E \rightarrow E+ \cdot T] [T \rightarrow \cdot T*F]$	Expansion
$[E \rightarrow E+ \cdot T] [T \rightarrow T*F \cdot]$	$\varepsilon$	$[E \rightarrow E+T \cdot]$	Reduktion
$[E \rightarrow E+ \cdot T]$	$\varepsilon$	$[E \rightarrow E+ \cdot T] [T \rightarrow \cdot F]$	Expansion
$[E \rightarrow E+ \cdot T] [T \rightarrow F \cdot]$	$\varepsilon$	$[E \rightarrow E+T \cdot]$	Reduktion
$[E \rightarrow \cdot T]$	$\varepsilon$	$[E \rightarrow \cdot T] [T \rightarrow \cdot T*F]$	Expansion
$[E \rightarrow \cdot T] [T \rightarrow T*F \cdot]$	$\varepsilon$	$[E \rightarrow T \cdot]$	Reduktion
$[E \rightarrow \cdot T]$	$\varepsilon$	$[E \rightarrow \cdot T] [T \rightarrow \cdot F]$	Expansion
$[E \rightarrow \cdot T] [T \rightarrow F \cdot]$	$\varepsilon$	$[E \rightarrow T \cdot]$	Reduktion
$[T \rightarrow \cdot T*F]$	$\varepsilon$	$[T \rightarrow \cdot T*F] [T \rightarrow \cdot T*F]$	Expansion
$[T \rightarrow \cdot T*F] [T \rightarrow T*F \cdot]$	$\varepsilon$	$[T \rightarrow T \cdot *F]$	Reduktion
$[T \rightarrow \cdot T*F]$	$\varepsilon$	$[T \rightarrow \cdot T*F] [T \rightarrow \cdot F]$	Expansion
$[T \rightarrow \cdot T*F] [T \rightarrow F \cdot]$	$\varepsilon$	$[T \rightarrow T \cdot *F]$	Reduktion
$[T \rightarrow T \cdot *F]$	$*$	$[T \rightarrow T* \cdot F]$	Lesen
$[T \rightarrow T* \cdot F]$	$\varepsilon$	$[T \rightarrow T* \cdot F] [F \rightarrow \cdot (E)]$	Expansion
$[T \rightarrow T* \cdot F] [F \rightarrow (E) \cdot]$	$\varepsilon$	$[T \rightarrow T*F \cdot]$	Reduktion
$[T \rightarrow T* \cdot F]$	$\varepsilon$	$[T \rightarrow T* \cdot F] [F \rightarrow \cdot a]$	Expansion
$[T \rightarrow T* \cdot F] [F \rightarrow a \cdot]$	$\varepsilon$	$[T \rightarrow T*F \cdot]$	Reduktion
$[T \rightarrow \cdot F]$	$\varepsilon$	$[T \rightarrow \cdot F] [F \rightarrow \cdot (E)]$	Expansion
$[T \rightarrow \cdot F] [F \rightarrow (E) \cdot]$	$\varepsilon$	$[T \rightarrow F \cdot]$	Reduktion
$[T \rightarrow \cdot F]$	$\varepsilon$	$[T \rightarrow \cdot F] [F \rightarrow \cdot a]$	Expansion
$[T \rightarrow \cdot F] [F \rightarrow a \cdot]$	$\varepsilon$	$[T \rightarrow F \cdot]$	Reduktion
$[F \rightarrow \cdot (E)]$	$($	$[F \rightarrow ( \cdot E)]$	Lesen
$[F \rightarrow ( \cdot E)]$	$\varepsilon$	$[F \rightarrow ( \cdot E)] [E \rightarrow \cdot E+T]$	Expansion
$[F \rightarrow ( \cdot E)] [E \rightarrow E+T \cdot]$	$\varepsilon$	$[F \rightarrow (E \cdot)]$	Reduktion
$[F \rightarrow ( \cdot E)]$	$\varepsilon$	$[F \rightarrow ( \cdot E)] [E \rightarrow \cdot T]$	Expansion
$[F \rightarrow ( \cdot E)] [E \rightarrow T \cdot]$	$\varepsilon$	$[F \rightarrow (E \cdot)]$	Reduktion
$[F \rightarrow (E \cdot)]$	$)$	$[F \rightarrow (E) \cdot]$	Lesen
$[F \rightarrow \cdot a]$	$a$	$[F \rightarrow a \cdot]$	Lesen

Für eine Folge von Stadien, die als Kellerinhalt in einer Berechnung des Stadiumautomaten auftreten, gilt die folgende Invariante (I):

(I) Falls der Stadiumautomat nach dem Einlesen von  $u$  den Kellerinhalt  $\rho$  hat, so kann man aus der Vergangenheit von  $\rho$  das Wort  $u$  ableiten: 
$$([Z \rightarrow \cdot S], uv) \vdash_{K_G}^* (\rho, v) \Rightarrow \text{past}(\rho) \xrightarrow{*}_G u.$$

Diese Invariante (I) ist ein wesentlicher Bestandteil des Beweises, dass der Stadiumautomat  $K_G$  nur Wörter aus  $G$  akzeptiert, also dass  $L(K_G) \subseteq L(G)$  ist.

Wir erläutern die Arbeitsweise des Automaten  $K_G$  und beweisen gleichzeitig durch Induktion (über die Länge von Berechnungen), dass die Invariante (I) für jede aus einer Anfangskonfiguration erreichbare Konfiguration gilt.

Betrachten wir für eine Eingabe  $w = uv$  zunächst die Anfangskonfiguration  $([Z \rightarrow \cdot S], w)$ .

Bereits gelesen wurde nichts ( $u = \varepsilon$ ) und  $\text{past}([Z \rightarrow \cdot S]) = \varepsilon$ . Es gilt  $\text{past}(\rho) = \varepsilon \xrightarrow{*}_G \varepsilon = u$  und damit die Invariante (I) für die Anfangskonfiguration.

Nun betrachten wir Ableitungen mit mindestens einem Übergang. Wir betrachten die Konfiguration  $(\kappa, v)$ , die aus der Anfangskonfiguration  $([Z \rightarrow \cdot S], uv)$  vor dem letzten Übergang erreicht wurde. Nach Induktionsvoraussetzung erfüllt diese Konfiguration die Invariante (I).

Fall 1: Der letzte Übergang war ein Expansionsübergang. Dann ist  $(\kappa, v) = (\rho [X \rightarrow \beta \cdot Y\gamma], v)$ . Wegen (I) gilt:  $\text{past}(\rho) \beta \xrightarrow{*}_G u$ . Als aktueller Zustand legt das Stadium  $[X \rightarrow \beta \cdot Y\gamma]$  nahe, einen Präfix von  $v$  aus  $Y$  abzuleiten. Dazu sollte der Automat eine der Alternativen für  $Y$  nichtdeterministisch auswählen. Das beschreiben gerade die Expansionsübergänge. Alle möglichen Folgekonfigurationen  $(\rho [X \rightarrow \beta \cdot Y\gamma] [Y \rightarrow \cdot \alpha], v)$  für  $Y \rightarrow a \in P$  erfüllen ebenfalls die Invariante (I), denn es gilt  $\text{past}(\rho [X \rightarrow \beta \cdot Y\gamma] [Y \rightarrow \cdot \alpha]) = \text{past}(\rho) \beta \xrightarrow{*}_G u$ .

Fall 2: Der letzte Übergang war ein Leseübergang. Dann ist  $(\kappa, v) = (\rho [X \rightarrow \beta \cdot a\gamma], av')$  mit  $v = av'$ . Wegen (I) gilt:  $\text{past}(\rho) \beta \xrightarrow{*}_G u$ . Dann erfüllt die Nachfolgekonfiguration  $(\rho [X \rightarrow \beta a \cdot \gamma], v)$  ebenfalls (I), da  $\text{past}(\rho [X \rightarrow \beta a \cdot \gamma]) = \text{past}(\rho) \beta a \xrightarrow{*}_G ua$ .

Fall 3: Der letzte Übergang war ein Reduktionsübergang.

Dann ist  $(\kappa, v) = (\rho [X \rightarrow \beta \cdot Y\gamma] [Y \rightarrow \alpha \cdot], v)$ . Wegen (I) gilt:  $\text{past}(\rho) \beta \alpha \xrightarrow{*}_G u$ . Der aktuelle Zustand ist das finale Stadium  $[Y \rightarrow \alpha \cdot]$ . Er ist das Ergebnis einer Berechnung, die mit dem Stadium  $[Y \rightarrow \cdot \alpha]$  begonnen wurde, als  $[X \rightarrow \beta \cdot Y\gamma]$  aktueller Zustand war und die Alternative  $Y \rightarrow \alpha$  für  $Y$  ausgewählt wurde. Diese Alternative wurde erfolgreich abgearbeitet. Die Nachfolgekonfiguration  $(\rho [X \rightarrow \beta Y \cdot \gamma], v)$  erfüllt ebenfalls die Invariante (I), denn aus  $\text{past}(\rho) \beta \alpha \xrightarrow{*}_G u$  folgt  $\text{past}(\rho) \beta Y \xrightarrow{*}_G u$ .

**Satz:** Für jede kontextfreie Grammatik  $G$  gilt:  $L(K_G) = L(G)$ .

Beweis: „ $\subseteq$ “:  $L(K_G) = \{w \in T^* \mid ([Z \rightarrow \cdot S], w) \vdash_{K_G}^* ([Z \rightarrow S \cdot], \varepsilon)\}$  Ist  $w \in L(K_G)$ , so besagt (I) (mit  $u = w$ ,  $v = \varepsilon$ ), dass  $\text{past}([Z \rightarrow S \cdot]) = S \xrightarrow{*}_G w$  gilt. Damit ist  $w \in L(G)$ .

„ $\supseteq$ “: Sei  $w \in L(G)$ . Dann gilt  $S \xrightarrow{*}_G w$ . Zu zeigen ist dann  $([Z \rightarrow \cdot S], w) \vdash_{K_G}^* ([Z \rightarrow S \cdot], \varepsilon)$ .

Allgemeiner gilt für jede Ableitung  $A \Rightarrow_G \alpha \xrightarrow{*}_G w$  mit  $A \in N$ , dass

$(\rho [A \rightarrow \cdot \alpha], wv) \vdash_{K_G}^* (\rho [A \rightarrow \alpha \cdot], v)$  für beliebige  $\rho \in St_{G^+}^*$  und beliebige  $v \in T^*$ . Diese Behauptung lässt sich durch Induktion über die Länge der Ableitung  $A \Rightarrow_G \alpha \xrightarrow{*}_G w$  beweisen.

### 5.5.2 Linksparser und Rechtsparser

Ein Kellerautomat ist ein Akzeptor: er entscheidet nur, ob ein Wort zur Sprache gehört oder nicht. Die Syntaxanalyse soll aber auch die syntaktische Struktur des akzeptierten Wortes liefern, z.B. als Syntaxbaum oder als Folge der in einer Rechts- bzw. Linksableitung benutzten Produktionen.



Deshalb erweitern wir Kellerautomaten um eine Ausgabe.

**Definition:** Ein **Kellerautomat mit Ausgabe**  $K = (Q, T, O, \Delta, q_0, F)$  ist ein Kellerautomat, der zusätzlich  $O$  als endliches **Ausgabealphabet** enthält.

Bei jedem Übergang *kann* der Automat ein Symbol aus  $O$  ausgeben:

$\Delta$  ist jetzt eine endliche Relation zwischen  $Q^+ \times (T \cup \{\varepsilon\})$  und  $Q^* \times (O \cup \{\varepsilon\})$ .

Eine **Konfiguration** ist ein Element aus  $Q^+ \times T^* \times O^*$  und beschreibt den aktuellen Kellerinhalt, die restliche Eingabe und die bisherige Ausgabe.

Benutzt man einen Kellerautomaten mit Ausgabe als Parser, so verwendet man als Ausgabealphabet die Menge der Produktionen der kontextfreien Grammatik (bzw. ihre Nummern).

Ein Stadiumautomat soll dann die benutzte Produktion ausgeben bei jedem

- *Expansionsübergang*, also wenn der aktuelle Zustand ein *initiales* Stadium ist. Für jede akzeptierende Berechnung ist dann die Ausgabe eine *Linksableitung* für das akzeptierte Wort. Ein Kellerautomat mit einer Linksableitung als Ausgabe heißt **Linksparser**.
- *Reduktionsübergang*, also wenn der aktuelle Zustand ein *finale*s Stadium ist. Für jede akzeptierende Berechnung ist die Ausgabe eine *gespiegelte Rechtsableitung* (Linksreduktion) für das akzeptierte Wort. Ein Kellerautomat mit dieser Ausgabe heißt **Rechtsparser**.

**Beispiel 5.49:** Wir geben zur erweiterten Grammatik von  $G_4$  aus Beispiel 5.48 die (einzige) Konfigurationsfolge des Stadiumautomaten  $K_{G_4}$  an, die zur Akzeptanz des Wortes  $a + a * a$  führt, zusammen mit der Ausgabe eines Links- bzw. eines Rechtsparsers:

Kellerinhalt	Eingabe (Rest)	Ausgabe Links- parser	Ausgabe Rechts- parser
$[Z \rightarrow \cdot E]$	$a + a * a$	$Z \rightarrow E$	
$[Z \rightarrow \cdot E] [E \rightarrow \cdot E+T]$	$a + a * a$	$E \rightarrow E+T$	
$[Z \rightarrow \cdot E] [E \rightarrow \cdot E+T] [E \rightarrow \cdot T]$	$a + a * a$	$E \rightarrow T$	
$[Z \rightarrow \cdot E] [E \rightarrow \cdot E+T] [E \rightarrow \cdot T] [T \rightarrow \cdot F]$	$a + a * a$	$T \rightarrow F$	
$[Z \rightarrow \cdot E] [E \rightarrow \cdot E+T] [E \rightarrow \cdot T] [T \rightarrow \cdot F] [F \rightarrow \cdot a]$	$a + a * a$	$F \rightarrow a$	
$[Z \rightarrow \cdot E] [E \rightarrow \cdot E+T] [E \rightarrow \cdot T] [T \rightarrow \cdot F] [F \rightarrow a \cdot]$	$+a * a$		$F \rightarrow a$
$[Z \rightarrow \cdot E] [E \rightarrow \cdot E+T] [E \rightarrow \cdot T] [T \rightarrow F \cdot]$	$+a * a$		$T \rightarrow F$
$[Z \rightarrow \cdot E] [E \rightarrow \cdot E+T] [E \rightarrow T \cdot]$	$+a * a$		$E \rightarrow T$
$[Z \rightarrow \cdot E] [E \rightarrow E \cdot +T]$	$+a * a$		
$[Z \rightarrow \cdot E] [E \rightarrow E+ \cdot T]$	$a * a$		
$[Z \rightarrow \cdot E] [E \rightarrow E+ \cdot T] [T \rightarrow \cdot T*F]$	$a * a$	$T \rightarrow T*F$	
$[Z \rightarrow \cdot E] [E \rightarrow E+ \cdot T] [T \rightarrow \cdot T*F] [T \rightarrow \cdot F]$	$a * a$	$T \rightarrow F$	
$[Z \rightarrow \cdot E] [E \rightarrow E+ \cdot T] [T \rightarrow \cdot T*F] [T \rightarrow \cdot F] [F \rightarrow \cdot a]$	$a * a$	$F \rightarrow a$	
$[Z \rightarrow \cdot E] [E \rightarrow E+ \cdot T] [T \rightarrow \cdot T*F] [T \rightarrow \cdot F] [F \rightarrow a \cdot]$	$*a$		$F \rightarrow a$
$[Z \rightarrow \cdot E] [E \rightarrow E+ \cdot T] [T \rightarrow \cdot T*F] [T \rightarrow F \cdot]$	$*a$		$T \rightarrow F$
$[Z \rightarrow \cdot E] [E \rightarrow E+ \cdot T] [T \rightarrow T \cdot *F]$	$*a$		
$[Z \rightarrow \cdot E] [E \rightarrow E+ \cdot T] [T \rightarrow T* \cdot F]$	$a$		
$[Z \rightarrow \cdot E] [E \rightarrow E+ \cdot T] [T \rightarrow T* \cdot F] [F \rightarrow \cdot a]$	$a$	$F \rightarrow a$	
$[Z \rightarrow \cdot E] [E \rightarrow E+ \cdot T] [T \rightarrow T* \cdot F] [F \rightarrow a \cdot]$			$F \rightarrow a$
$[Z \rightarrow \cdot E] [E \rightarrow E+ \cdot T] [T \rightarrow T*F \cdot]$			$T \rightarrow T*F$
$[Z \rightarrow \cdot E] [E \rightarrow E+T \cdot]$			$E \rightarrow E+T$
$[Z \rightarrow E \cdot]$			$Z \rightarrow E$

Der Linksparser erzeugt die Linksableitung

$$Z \Rightarrow E \Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow a + T \Rightarrow a + T * F \Rightarrow a + F * F \Rightarrow a + a * F \Rightarrow a + a * a$$

Der Rechtsparser erzeugt die Linksreduktion

$$a + a * a \Leftarrow F + a * a \Leftarrow T + a * a \Leftarrow E + a * a \Leftarrow E + F * a \Leftarrow E + T * a \Leftarrow E + T * F \Leftarrow E + T \Leftarrow E \Leftarrow Z$$

### 5.5.3 Charakteristischer endlicher Automat eines Stadiumautomaten

Die Übergangsrelation  $\Delta$  des Stadiumautomaten  $K_G$  kann man (statt tabellarisch wie in Beispiel 5.48) weitgehend durch einen nichtdeterministischen endlichen Automaten beschreiben, seinen **charakteristischen endlichen Automaten**  $c(G)$ . Mit einigen Übergängen von  $c(G)$  muss man allerdings Kelleroperationen verbinden.

**Definition:** Sei  $G$  eine reduzierte, kontextfreie Grammatik und  $G^+$  ihre Erweiterung.

Der nichtdeterministische endliche Automat  $c(G) = (Q_c, V_c, \Delta_c, q_c, F_c)$  heißt **charakteristischer endlicher Automat** zu  $G$ , wenn:

$$Q_c = St_{G^+}$$

Zustände: Stadien der Grammatik

$$V_c = N \cup T$$

Eingabealphabet: Nichtterminal- und Terminalsymbole

$$q_c = [Z \rightarrow \cdot S]$$

Startzustand: initiales Stadium der hinzugefügten Produktion  $Z \rightarrow S$

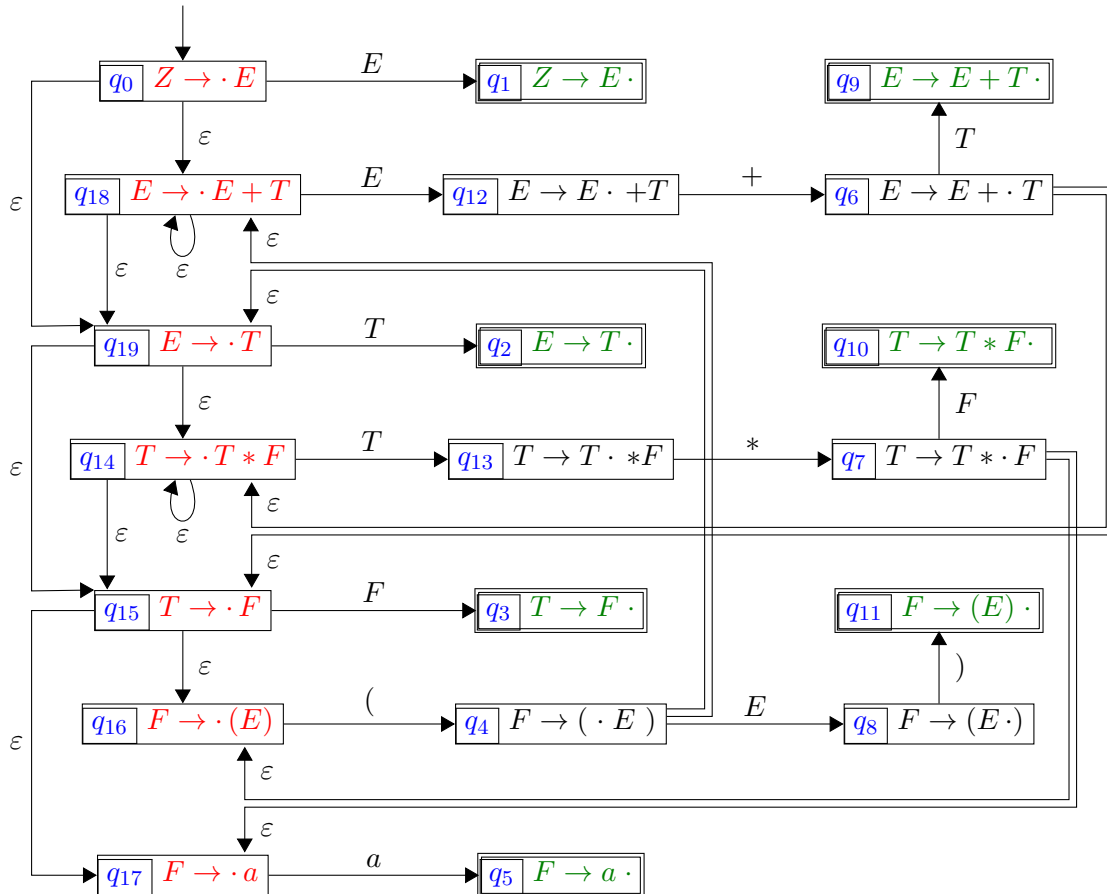
$$F_c = \{[X \rightarrow \alpha \cdot] \mid X \rightarrow \alpha \in P\}$$

Endzustände: alle finalen Stadien

$$\Delta_c = \{([X \rightarrow \alpha \cdot Y\beta], Y, [X \rightarrow \alpha Y \cdot \beta]) \mid X \rightarrow \alpha Y \beta \in P \text{ und } Y \in N \cup T\} \cup \{([X \rightarrow \alpha \cdot Y\beta], \varepsilon, [Y \rightarrow \cdot \gamma]) \mid X \rightarrow \alpha Y \beta \in P \text{ und } Y \rightarrow \gamma \in P\}$$

**Beispiel 5.50:** Der charakteristische endliche Automat  $c(G_4)$  zu den Produktionen

(0)  $Z \rightarrow E$ , (1)  $E \rightarrow E + T$ , (2)  $E \rightarrow T$ , (3)  $T \rightarrow T * F$ , (4)  $T \rightarrow F$ , (5)  $F \rightarrow (E)$ , (6)  $F \rightarrow a$ .



- $c(G)$  soll genau dann in den Zustand  $[X \rightarrow \alpha \cdot \beta]$  übergehen, wenn der Stadiumautomat einen Keller  $\rho$  erreichen kann, dessen *Vergangenheit* mit dem von  $c(G)$  gelesenen Wort übereinstimmt.  $c(G)$  liest also die abgearbeiteten Präfixe von rechten Seiten von Produktionen ein, deren Abarbeitung im Stadiumautomaten zum aktuellen Stadium führten.

- Die Menge der Eingabesymbole von  $c(G)$  ist  $N \cup T$ , weil die Produktionen Nichtterminal- und Terminalsymbole enthalten können.
- Endzustände von  $c(G)$  sind alle finalen Stadien. Im Endzustand  $[A \rightarrow \alpha \cdot]$  entspricht das gelesene Wort einem Kellerinhalt von  $K_G$ , in dem eine Reduktion mit der Produktion  $A \rightarrow \alpha$  durchgeführt werden *kann*.
- Das Lesen eines Terminalsymbols entspricht im Stadiumautomaten einem Leseübergang, das Lesen eines Nichtterminalsymbols dem Verschieben des Punkts nach einem Reduktionsübergang und die  $\varepsilon$ -Übergänge entsprechen den Expansionsübergängen.
- Einige Übergänge in  $c(G)$  verbinden wir mit Keller-Operationen in  $K_G$ :
  - Bei jedem  $\varepsilon$ -Übergang in  $c(G)$  legen wir den neuen Zustand von  $c(G)$  oben auf den Keller von  $K_G$  (Expansionsübergang).
  - Gelangt  $c(G)$  in einen Endzustand  $[X \rightarrow \alpha \cdot]$ , so wird in  $K_G$  das oberste Stadium  $[X \rightarrow \alpha \cdot]$  vom Keller entfernt und das neue oberste Kellerstadium macht einen Übergang unter  $X$  (Reduktionsübergang).

Das Verhältnis zwischen  $c(G)$  und  $K_G$  beschreibt folgender Satz im Detail:

**Satz:** Sei  $G$  eine erweiterte reduzierte kontextfreie Grammatik,  $[A \rightarrow \alpha \cdot \beta]$  ein Stadium von  $G$  und  $\gamma \in (N \cup T)^*$ .

Sei  $K_G$  der Stadiumautomat zu  $G$  und  $c(G)$  der charakteristische endliche Automat zu  $G$ . Dann sind folgende Aussagen äquivalent:

- (1) Es gibt eine Berechnung  $([Z \rightarrow \cdot S], \gamma) \xrightarrow{c(G)^*} ([A \rightarrow \alpha \cdot \beta], \varepsilon)$  in  $c(G)$ .
- (2) Es gibt eine Berechnung  $(\rho [A \rightarrow \alpha \cdot \beta], w) \xrightarrow{K_G^*} ([Z \rightarrow S \cdot], \varepsilon)$  in  $K_G$  mit  $\gamma = \text{past}(\rho) \alpha$ .
- (3) Es gibt eine Rechtsableitung  $Z \xRightarrow{*} \bar{\gamma} A w \Rightarrow \bar{\gamma} \alpha \beta w$  mit  $\gamma = \bar{\gamma} \alpha$  in  $G$ .

„(1) $\Leftrightarrow$ (2)“: Wörter, die in  $c(G)$  zu einem Stadium führen, entsprechen genau den Vergangenheiten von Kellerinhalten des  $K_G$ , deren oberstes Kellersymbol dieses Stadium ist, von denen aus –durch eine geeignete Eingabe  $w$ – der Endzustand von  $K_G$  erreicht werden kann.

„(2) $\Leftrightarrow$ (3)“: Eine akzeptierende Berechnung in  $K_G$  für ein Eingabewort  $w$ , die mit einem bestimmten Kellerinhalt beginnt, entspricht einer Rechtsableitung zur Satzform  $\gamma \beta w$ , wobei  $\gamma$  die Vergangenheit dieses Kellerinhalts ist.

Bevor wir den Satz beweisen, führen wir folgende Redeweise ein.

Für eine Rechtsableitung  $Z \xRightarrow{*} \bar{\gamma} A w \Rightarrow \bar{\gamma} \bar{\alpha} w$  und eine Produktion  $A \rightarrow \bar{\alpha}$  nennen wir (die Position von)  $\bar{\alpha}$  die **Reduktionsstelle** oder den **Griff** (*handle*) der Rechtssatzform  $\bar{\gamma} \bar{\alpha} w$ .

Ist  $\bar{\alpha} = \alpha \beta$ , dann heißt das Präfix  $\gamma = \bar{\gamma} \alpha$  ein **erfolgreiches Präfix** (*viable prefix*) von  $G$  für das Stadium  $[A \rightarrow \alpha \cdot \beta]$ . Das Stadium  $[A \rightarrow \alpha \cdot \beta]$  heißt **gültig** (*valid*) für das erfolgreiche Präfix  $\gamma$ .

Der charakteristische Automat berechnet dann gerade die erfolgreichen Präfixe. Denn der Satz besagt: Die Menge der Wörter, mit denen der charakteristische Automat ein Stadium  $[A \rightarrow \alpha \cdot \beta]$  erreicht, ist die Menge der erfolgreichen Präfixe für dieses Stadium.

**Beispiel:** Für die (um  $Z$  als neues Startsymbol erweiterte) Grammatik von  $G_4$  gilt:

Rechtssatzform	Griff	erfolgreiche Präfixe	Rechtsableitung als Begründung
$E + F$	$F$	$E, E+, E + F$	$Z \Rightarrow E \Rightarrow E + T \Rightarrow E + F$
$T * a$	$a$	$T, T*, T * a$	$Z \xRightarrow{3} T * F \Rightarrow T * a$

Wir geben zwei erfolgreiche Präfixe von  $G_4$  und einige für sie gültige Stadien an.

erfolgr. Präfix	gültiges Stadium	Rechtsableitung als Begründung	$\bar{\gamma}$	$\alpha$	$\beta$	$A$	$w$
$E+$	$[E \rightarrow E + \cdot T]$	$Z \Longrightarrow E \Longrightarrow E + T$	$\varepsilon$	$E+$	$T$	$E$	$\varepsilon$
$E+$	$[T \rightarrow \cdot F]$	$Z \xRightarrow{*} E + T \Longrightarrow E + F$	$E+$	$\varepsilon$	$F$	$T$	$\varepsilon$
$E+$	$[F \rightarrow \cdot a]$	$Z \xRightarrow{*} E + F \Longrightarrow E + a$	$E+$	$\varepsilon$	$a$	$F$	$\varepsilon$
$(E + ($	$[F \rightarrow (\cdot E)]$	$Z \xRightarrow{*} (E + F) \Longrightarrow (E + (E))$	$(E + ($	$($	$E)$	$F$	$)$
$(E + ($	$[T \rightarrow \cdot F]$	$Z \xRightarrow{*} (E + (T)) \Longrightarrow (E + (F))$	$(E + ($	$\varepsilon$	$F$	$T$	$)$
$(E + ($	$[F \rightarrow \cdot a]$	$Z \xRightarrow{*} (E + (F)) \Longrightarrow (E + (a))$	$(E + ($	$\varepsilon$	$a$	$F$	$)$

Wurde beim Versuch, eine Rechtsableitung für ein Wort zu erstellen, das bisher gelesene Präfix  $u$  des Wortes zu einem erfolgreichen Präfix  $\gamma$  reduziert, dann beschreibt jedes für  $\gamma$  gültige Stadium  $[A \rightarrow \alpha \cdot \beta]$  eine mögliche Analysesituation. Es gibt also eine Rechtsableitung, in der  $\gamma$  Präfix einer Rechtssatzform und  $A \rightarrow \alpha\beta$  eine der möglichen gerade bearbeiteten Produktionen ist. Diese Produktionen sind die Kandidaten für spätere Reduktionen.

Betrachten wir die Rechtsableitung  $Z \xRightarrow{*} \gamma Aw \Longrightarrow \gamma \alpha \beta w$ . Die nächsten Schritte der Rechtsableitung müssen erst  $\beta$  zu einem Terminalwort  $v$  ableiten und danach  $\alpha$  zu einem Terminalwort  $u$ . Insgesamt ergibt sich  $Z \xRightarrow{*} \gamma Aw \Longrightarrow \gamma \alpha \beta w \xRightarrow{*} \gamma \alpha v w \xRightarrow{*} \gamma u v w \xRightarrow{*} x u v w$ . In Linksreduktionsschritten wird erst  $x$  zu  $\gamma$  reduziert, dann  $u$  zu  $\alpha$ , dann  $v$  zu  $\beta$ . Das gültige Stadium  $[A \rightarrow \alpha \cdot \beta]$  für das erfolgreiche Präfix  $\gamma\alpha$  beschreibt die Analysesituation, in der die Reduktion von  $u$  nach  $\alpha$  bereits geschehen ist, die Reduktion von  $v$  nach  $\beta$  aber noch nicht begonnen hat. Mögliches Fernziel in dieser Situation ist die Anwendung der Produktion  $A \rightarrow \alpha\beta$ .

Wir kommen zu der Frage zurück, welche Sprache der charakteristische endliche Automat von  $K_G$  akzeptiert. Der Satz besagt, dass er unter einem erfolgreichen Präfix in einen Zustand übergeht, der ein gültiges Stadium für dieses Präfix ist. Endzustände, d.h. vollständige Stadien, sind nur gültig für erfolgreiche Präfixe, an deren Ende eine Reduktion möglich ist.

### Beweis des Satzes:

„(1)  $\Rightarrow$  (3)“: Sei  $([Z \rightarrow \cdot S], \gamma) \vdash_{c(G)}^* ([A \rightarrow \alpha \cdot \beta], \varepsilon)$ .

Mit Induktion nach der Anzahl  $n$  der  $\varepsilon$ -Übergänge konstruieren wir eine Rechtsableitung

$$Z \xRightarrow{*} \bar{\gamma} Aw \Longrightarrow \bar{\gamma} \alpha \beta w.$$

Für  $n = 0$  ist  $\bar{\gamma} = \varepsilon$  und  $[A \rightarrow \alpha \cdot \beta] = [Z \rightarrow \cdot S]$ . Wegen  $Z \xRightarrow{*} Z$  ist die Behauptung erfüllt.

Für  $n > 0$  betrachten wir den letzten  $\varepsilon$ -Übergang.

Dann lässt sich die Berechnung des charakteristischen Automaten zerlegen mit  $\gamma = \bar{\gamma}\alpha$ :

$$([Z \rightarrow \cdot S], \gamma) \vdash_{c(G)}^* ([X \rightarrow \bar{\alpha} \cdot A\bar{\beta}], \alpha) \vdash_{c(G)} ([A \rightarrow \cdot \alpha\beta], \alpha) \vdash_{c(G)}^* ([A \rightarrow \alpha \cdot \beta], \varepsilon).$$

Nach Induktionsannahme gibt es eine Rechtsableitung  $Z \xRightarrow{*} \bar{\gamma} X \bar{w} \Longrightarrow \bar{\gamma} \bar{\alpha} A \bar{\beta} \bar{w}$ .

Da die Grammatik  $G$  reduziert ist, gibt es eine Rechtsableitung  $\bar{\beta} \xRightarrow{*} v$ . Deshalb haben wir:  $Z \xRightarrow{*} \bar{\gamma} A v \bar{w} \Longrightarrow \bar{\gamma} \alpha \beta w$  mit  $\bar{\gamma} = \bar{\gamma} \bar{\alpha}$  und  $w = v \bar{w}$ .

„(3)  $\Rightarrow$  (2)“: Sei eine Rechtsableitung  $Z \xRightarrow{*} \bar{\gamma} Aw \Longrightarrow \bar{\gamma} \alpha \beta w$  gegeben.

Diese Rechtsableitung lässt sich mit  $X_n = A$  zerlegen in:

$$Z \xRightarrow{*} \alpha_1 X_1 \beta_1 \xRightarrow{*} \alpha_1 X_1 v_1 \xRightarrow{*} \dots \xRightarrow{*} \alpha_1 \dots \alpha_n X_n v_n \dots v_1 \xRightarrow{*} \alpha_1 \dots \alpha_n \alpha \beta v_n \dots v_1.$$

Mit Induktion nach  $n$  folgt, dass  $(\rho, vw) \vdash_{K_G}^* ([Z \rightarrow S \cdot], \varepsilon)$  gilt für

$\rho = [Z \rightarrow \alpha_1 \cdot X_1 \beta_1] \dots [X_{n-1} \rightarrow \alpha_n \cdot X_n \beta_n]$  und  $w = v v_n \dots v_1$ , sofern  $\beta \xRightarrow{*} v$ ,  $\alpha_1 = \beta_1 = \varepsilon$  und  $X_1 = S$ . Wegen  $\bar{\gamma} = \text{past}(\rho)$  folgt die Behauptung (2).

„(2)  $\Rightarrow$  (1)“: Sei der Kellerinhalt  $\bar{\rho} = \rho [A \rightarrow \alpha \cdot \beta]$  mit  $(\bar{\rho}, w) \vdash_{K_G}^* ([Z \rightarrow S \cdot], \varepsilon)$ .

Zuerst überzeugen wir uns mit Induktion nach der Anzahl der Übergänge in der Berechnung, dass  $\rho$  notwendigerweise von der Form  $\rho = [Z \rightarrow \alpha_1 \cdot X_1 \beta_1] \dots [X_{n-1} \rightarrow \alpha_n \cdot X_n \beta_n]$  ist für ein  $n \geq 0$  und  $X_n = A$ .

Mit Induktion nach  $n$  folgt aber, dass  $([Z \rightarrow \cdot S], \gamma) \vdash_{c(G)}^* ([A \rightarrow \alpha \cdot \beta], \varepsilon)$  gilt für  $\gamma = \alpha_1 \dots \alpha_n \alpha$ .

### 5.5.4 Kanonischer LR(0)-Automat

Bekanntlich kann man zu jedem nichtdeterministischen endlichen Automaten einen deterministischen endlichen Automaten konstruieren, der die gleiche Sprache erkennt.

Der Algorithmus arbeitet in zwei Schritten:

1.  $\varepsilon$ -Übergänge im nichtdeterministischen endlichen Automaten entfernen:  
Alle  $\varepsilon$ -Übergänge und alle Knoten, die *nur* durch  $\varepsilon$ -Übergänge erreicht werden, fallen weg. Zusätzliche Kanten entstehen für jeden alten Weg, der aus  $\varepsilon$ -Übergängen und einem anschließenden Nicht- $\varepsilon$ -Übergang besteht.  
Knoten, die durch  $\varepsilon$ -Übergänge einen alten Endknoten erreichen, werden auch Endknoten.
2. Deterministischen endlichen Automaten mit Hilfe von Knotenmengen konstruieren (**Potenzmengenkonstruktion**):  
Die Menge mit dem alten Startknoten wird neuer Knoten und neuer Startknoten.  
Für jeden neuen Knoten wird für jedes Terminalsymbol die Menge der Folgeknoten bestimmt als Vereinigung der Mengen der alten Folgeknoten ihrer Elemente unter diesem Terminalsymbol und diese Menge wird zu einem neuen Knoten.  
Jeder Knoten, der einen alten Endknoten enthält, wird ein neuer Endknoten.

Wendet man diesen Algorithmus auf  $c(G)$  an, so entsteht der deterministische endliche Automat  $(Q_d, N \cup T, \Delta_d, q_d, F_d)$ , der **kanonische LR(0)-Automat** für  $G$ , kurz  $LR_0(G)$ .

**Beispiel 5.51:** Der Automat  $c(G_4)$  nach Entfernung der  $\varepsilon$ -Übergänge (Abb. 5.18) und nach der Potenzmengenkonstruktion (Abb. 5.19):  $LR_0(G_4)$ .

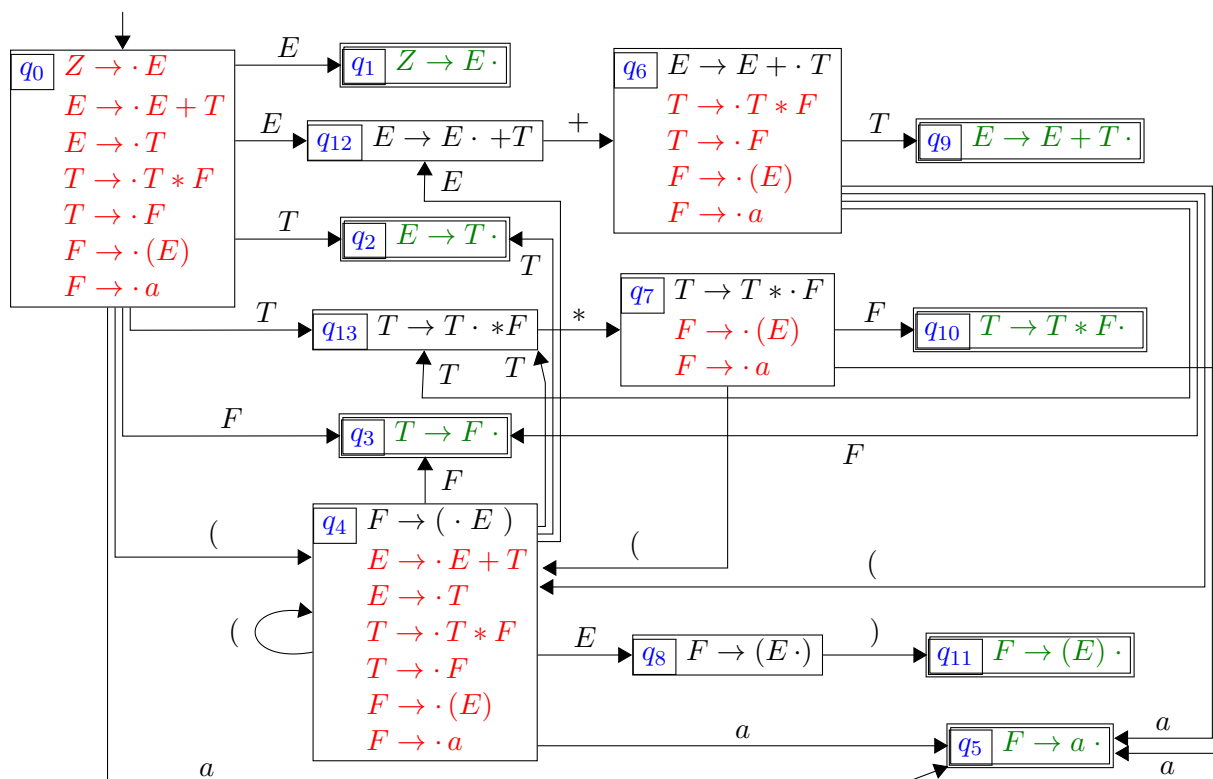


Abb. 5.18:  $c(G_4)$  nach Entfernung der  $\varepsilon$ -Übergänge.

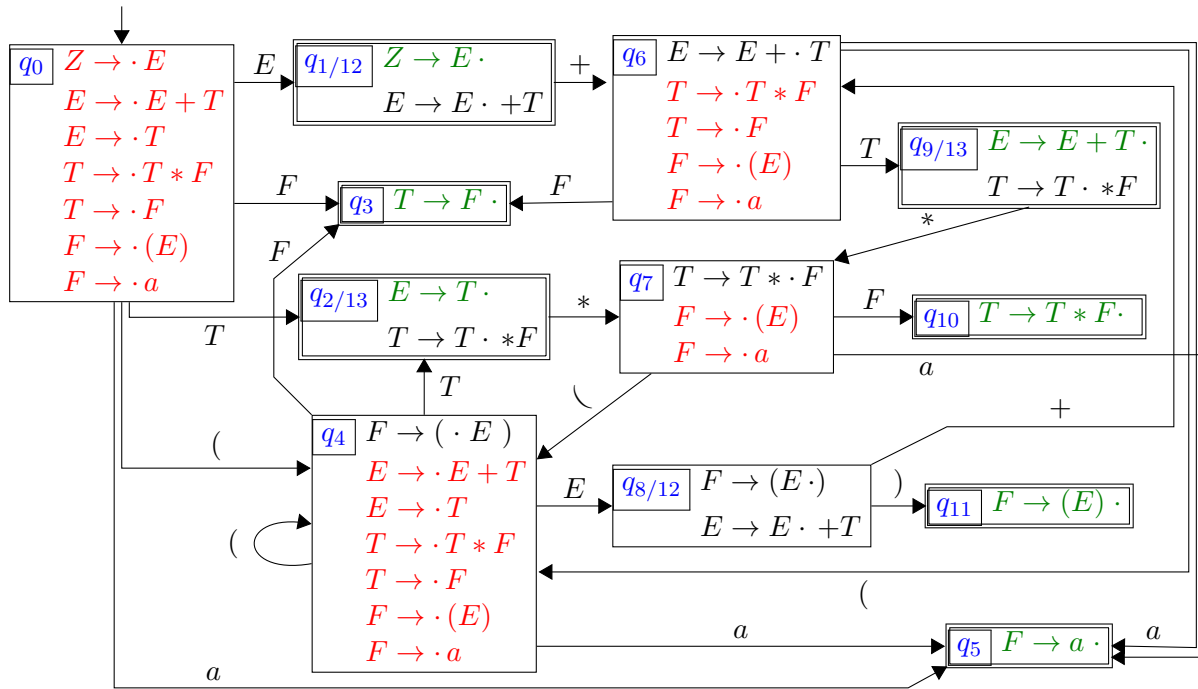


Abb. 5.19: Der  $\varepsilon$ -freie Automat  $c(G_4)$  wird nach Potenzmengenkonstruktion deterministisch, es entsteht der kanonische  $LR(0)$ -Automat  $LR_0(G_4)$  (12 Zustände).

Im folgenden notieren wir kombinierte Zustände durch die kleinste enthaltene Nummer.

Die Zustände von  $LR_0(G_4)$  stehen für folgende Äquivalenzklassen erfolgreicher Präfixe (Metazeichen in blau):

Zustand	Klasse der erfolgreichen Präfixe	gültige Operationen
0	$\varepsilon$	shift $E$ , shift $T$ , shift $F$ , shift $($ , shift $a$
4	$\alpha \equiv ( [ [ [E + ] T * ] ( ]^*$	shift $E$ , shift $T$ , shift $F$ , shift $($ , shift $a$
6	$[\alpha] E +$	shift $T$ , shift $F$ , shift $($ , shift $a$
7	$[\alpha] [E + ] T *$	shift $F$ , shift $($ , shift $a$
8	$\alpha E$	shift $+$ , shift $)$
1	$E$	shift $+$ , reduce $Z \rightarrow E \equiv \text{accept}$
2	$[\alpha] T$	shift $*$ , reduce $E \rightarrow T$
9	$[\alpha] E + T$	shift $*$ , reduce $E \rightarrow E + T$
3	$[\alpha] [E + ] F$	reduce $T \rightarrow F$
10	$[\alpha] [E + ] T * F$	reduce $T \rightarrow T * F$
11	$\alpha E)$	reduce $F \rightarrow (E)$
5	$[\alpha] [E + ] [T * ] a$	reduce $F \rightarrow a$

Die verschiedenen shift-Operationen können wir zusammenfassen, da sie rekonstruierbar sind: Beim Bottom-up-Parser wird das eingelesene Zeichen auf dem Keller gespeichert, beim LR-Parser werden stets verschiedene Folgezustände erreicht.<sup>16</sup>

In den Abbildungen lassen wir den Fehlerzustand (den Zustand mit der leeren Menge von Stadien) und alle Zustandsübergänge in den Fehlerzustand der Übersichtlichkeit halber weg.

Das Zeichen nach dem Punkt beschreibt das nächste abzuarbeitende Symbol. Nach der Abarbeitung gehen wir in einen Zustand, der durch Stadien gekennzeichnet ist, in denen der Punkt hinter das abzuarbeitende Symbol verschoben wurde.

<sup>16</sup> Abstrahiert man von der Produktion der reduce-Operationen, so verschmelzen einige Zustände (Kapitel 5.5.7).

- Die Abarbeitung eines Terminalsymbols geschieht durch Einlesen dieses Symbols.
- Die Abarbeitung eines Nichtterminalsymbols geschieht durch Anwenden einer Produktion mit diesem Nichtterminalsymbol auf der linken Seite. Dazu erweitern wir (ggf. rekursiv) den Zustand um die initialen Stadien der in Frage kommenden Produktionen. Dieser Zustand wird jetzt also Startzustand für die Abarbeitung einer neuen Produktion und nach Erreichen des finalen Stadiums dieser Produktion kann der Zustandsübergang für das Nichtterminalsymbol geschehen.
- Enthält ein Zustand ein finales Stadium, so kann mit der zugehörigen Produktion reduziert werden. Der Keller enthält als oberste Zustände die Zustände des Wegs vom initialen Stadium zum finalen Stadium dieser Produktion.

Wir fassen die wichtigsten Informationen des Übergangsdiagramms in einer Tabelle zusammen, indem wir für jeden Zustand die Nummern der Folgezustände und die Produktionen ihrer finalen Stadien notieren. Da es im Beispiel in jedem Zustand höchstens ein finales Stadium gab, gibt es keine reduce-/reduce-Konflikte.

Wir gehen also vom Zustand 0, wenn wir ein  $a$  einlesen, in den Zustand 5 und im Zustand 11 wenden wir die Produktion  $F \rightarrow (E)$  an.

In welchen Zustand gehen wir, nachdem wir die Produktion  $F \rightarrow (E)$  angewendet haben? Dadurch wurde ja das Nonterminal  $F$  im vorletzten Stadium bearbeitet! Von dessen Zustand gucken wir in die Spalte dieses Nonterminals.

Zustand	$a$	$+$	$*$	$($	$)$	$\$$	$E$	$T$	$F$	Reduktion
0	5		4				1	2	3	
1		6								$Z \rightarrow E$
2			7							$E \rightarrow T$
3										$T \rightarrow F$
4	5		4				8	2	3	
5										$F \rightarrow a$
6	5		4					9	3	
7	5		4						10	
8		6			11					
9			7							$E \rightarrow E+T$
10										$T \rightarrow T * F$
11										$F \rightarrow (E)$

Aber was machen wir im Zustand 2, falls ein  $*$  einzulesen ist: Die Produktion  $E \rightarrow T$  anwenden oder lesend in den Zustand 7 gehen? Und welche Produktion wenden wir an, falls ein Zustand mehr als ein finales Stadium enthält?

Ohne Zusatzinformation könnte es für jedes Eingabesymbol zu einer Reduktion mit jeder in dieser Zeile angegebenen Produktion kommen. Damit erhalten wir die LR(0)-Syntaxanalyse-Tabelle:

**LR(0)-Syntaxanalyse-Tabelle** in ausführlicher Schreibweise für  $G_4$

Zust.	Aktionstabelle						Sprungtabelle					
	$a$	$+$	$*$	$($	$)$	$\$$	$a$	$+$	$*$	$($	$)$	$E$ $T$ $F$
0	shift	error	error	shift	error	error	5			4	1	2 3
1	error	shift	error	error	error	accept		6				
2	$E \rightarrow T$	$E \rightarrow T$	$E \rightarrow T$ /shift	$E \rightarrow T$	$E \rightarrow T$	$E \rightarrow T$			-/7			
3	$T \rightarrow F$	$T \rightarrow F$	$T \rightarrow F$	$T \rightarrow F$	$T \rightarrow F$	$T \rightarrow F$						
4	shift	error	error	shift	error	error	5			4	8	2 3
5	$F \rightarrow a$	$F \rightarrow a$	$F \rightarrow a$	$F \rightarrow a$	$F \rightarrow a$	$F \rightarrow a$						
6	shift	error	error	shift	error	error	5			4		9 3
7	shift	error	error	shift	error	error	5			4		10
8	error	shift	error	error	shift	error		6			11	
9	$E \rightarrow E+T$	$E \rightarrow E+T$	$E \rightarrow E+T$ /shift	$E \rightarrow E+T$	$E \rightarrow E+T$	$E \rightarrow E+T$			-/7			
10	$T \rightarrow T * F$	$T \rightarrow T * F$	$T \rightarrow T * F$	$T \rightarrow T * F$	$T \rightarrow T * F$	$T \rightarrow T * F$						
11	$F \rightarrow (E)$	$F \rightarrow (E)$	$F \rightarrow (E)$	$F \rightarrow (E)$	$F \rightarrow (E)$	$F \rightarrow (E)$						

Diese Tabelle enthält zwei Konflikte (in Zustand 2 und 9; Spalte  $*$ ):  $G_4$  ist keine LR(0)-Grammatik.

In der üblichen Kurzform schreibt man den Folgezustand einer shift-Operation direkt in die Aktionstabelle und gibt bei einer reduce-Operation nur die Nummer der Produktion an. Dann sieht die Tabelle so aus:

Eine Entscheidungshilfe erhalten wir wieder durch die Vorschau-Symbole. Dazu berechnen wir mit Hilfe der Grammatik die **erwarteten Vorschau-Symbole**, jetzt bei einer Rechtsableitung.

LR(0)-Syntaxanalyse-Tabelle für  $G_4$ 

Zust.	Aktionstabelle						Sprungtab.		
	$a$	$+$	$*$	$($	$)$	$\$$	$E$	$T$	$F$
0	s5			s4			1	2	3
1		s6				acc			
2	r2	r2	r2/s7	r2	r2	r2			
3	r4	r4	r4	r4	r4	r4			
4	s5			s4			8	2	3
5	r6	r6	r6	r6	r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6				s11			
9	r1	r1	r1/s7	r1	r1	r1			
10	r3	r3	r3	r3	r3	r3			
11	r5	r5	r5	r5	r5	r5			

### 5.5.5 Kanonischer LR(1)-Parser

Dazu erweitern wir jedes Stadium um die im jeweiligen Zustand erwarteten Vorschau-Symbole (**LookAhead-Menge**), also der Menge der Terminalsymbole, die dem Nichtterminalsymbol auf der linken Seite der Produktion *in diesem Zustand* folgen können.

- Dem initialen Stadium  $[Z \rightarrow \cdot S]$  der Startproduktion wird die **LookAhead-Menge**  $\{\$ \}$  zugeordnet.
- Kommt man von einem erweiterten Stadium  $[A \rightarrow \alpha \cdot B\beta, L]$  ( $\alpha$  bzw.  $\beta$  kann auch  $\varepsilon$  sein) zu einem Stadium  $[B \rightarrow \cdot \gamma]$ , so berechnet sich dessen **LookAhead-Menge** zu  $\text{FIRST}(\beta L)$  mit  $\beta L = \{\beta x \mid x \in L\}$  und  $\text{FIRST}(L) = \bigcup_{\alpha \in L} \text{FIRST}(\alpha)$ .

Der **kanonische LR(1)-Parser** setzt seine reduce-Einträge dann *in die Spalten der Vorschausymbole, die in der LookAhead-Menge der Produktion für den jeweiligen Zustand enthalten sind*.

Da die **LookAhead-Mengen** für jedes Stadium eines Zustands erzeugt werden müssen, ist die Berechnung aufwändiger als die der FOLLOW-Mengen für jedes Nichtterminalsymbol.

LR(1)-Syntaxanalyse-Tabelle für  $G_4$ 

Zust.	Aktionstabelle						Sprungtab.		
	$a$	$+$	$*$	$($	$)$	$\$$	$E$	$T$	$F$
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7			r2			
3		r4	r4			r4			
4	s15			s14			8	12	13
5		r6	r6			r6			
6	s5			s4				9	3
7	s5			s4					10
8		s16				s11			
9		r1	s7			r1			
10		r3	r3			r3			
11		r5	r5			r5			
12		r2	s17			r2			
13		r4	r4			r4			
14	s15			s14			18	12	13
15		r6	r6			r6			
16	s15			s14				19	13
17	s15			s14					20
18		s16				s21			
19		r1	s17			r1			
20		r3	r3			r3			
21		r5	r5			r5			

**Beispiel 5.52:** Der (nicht-deterministische) charakteristische endliche Automat zur Grammatik  $G_4$  aus Beispiel 5.50 mit erweiterten Stadien. Eine Rückführung von  $q_4$  nach  $q_{18}/q_{19}$  ist nicht möglich, da die Vorschau-Menge  $\{\}$  jetzt anders ist! Denn nach einer „Klammer auf“ kann die Endmarke nicht mehr Vorschau-Symbol sein, stattdessen die „Klammer zu“. Das führt zu einer Verdoppelung der Zustände  $q_2$  bis  $q_{19}$ .



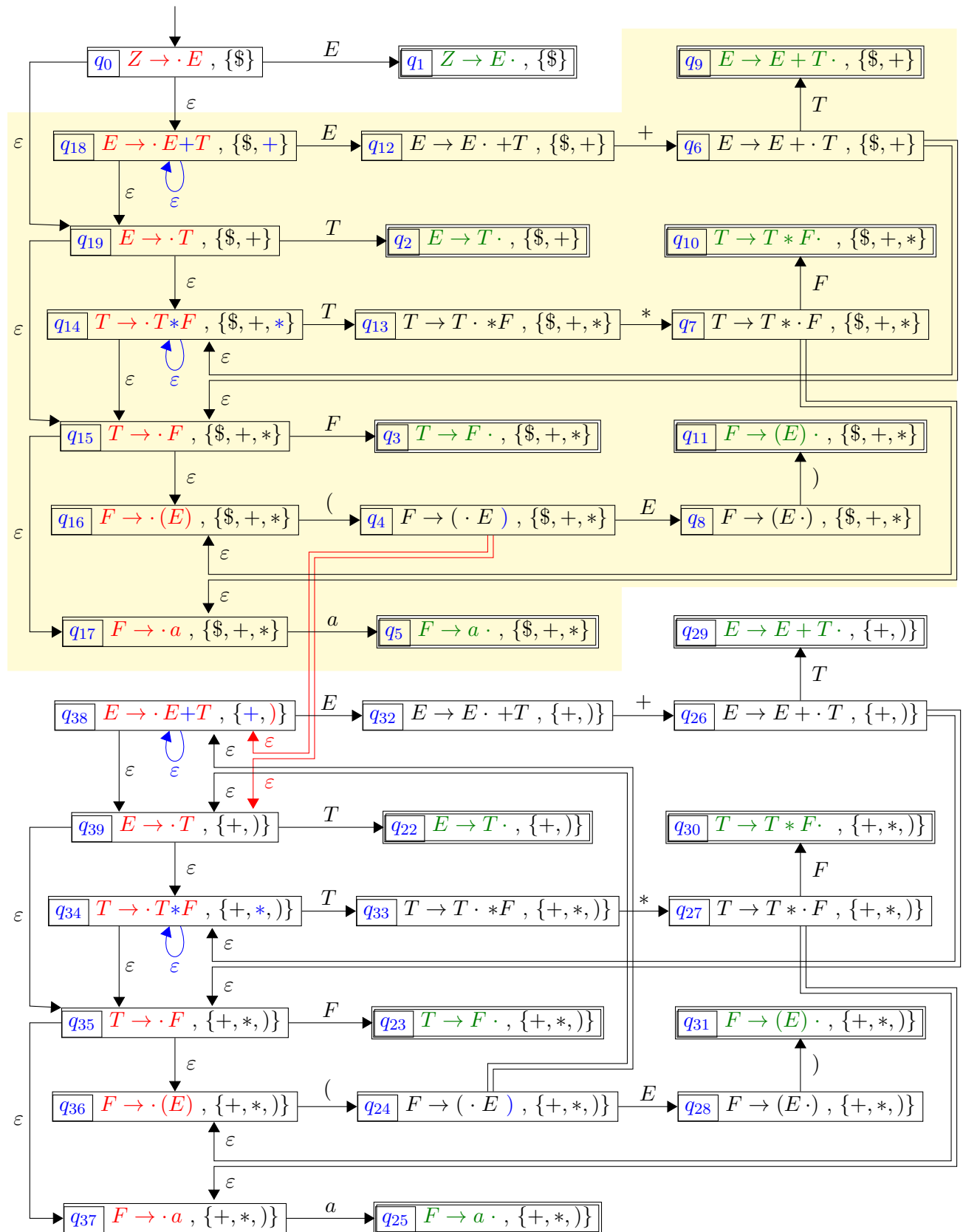


Abb. 5.20: Der (nicht-deterministische) charakteristische endliche Automat zur Grammatik  $G_4$  mit erweiterten Stadien.

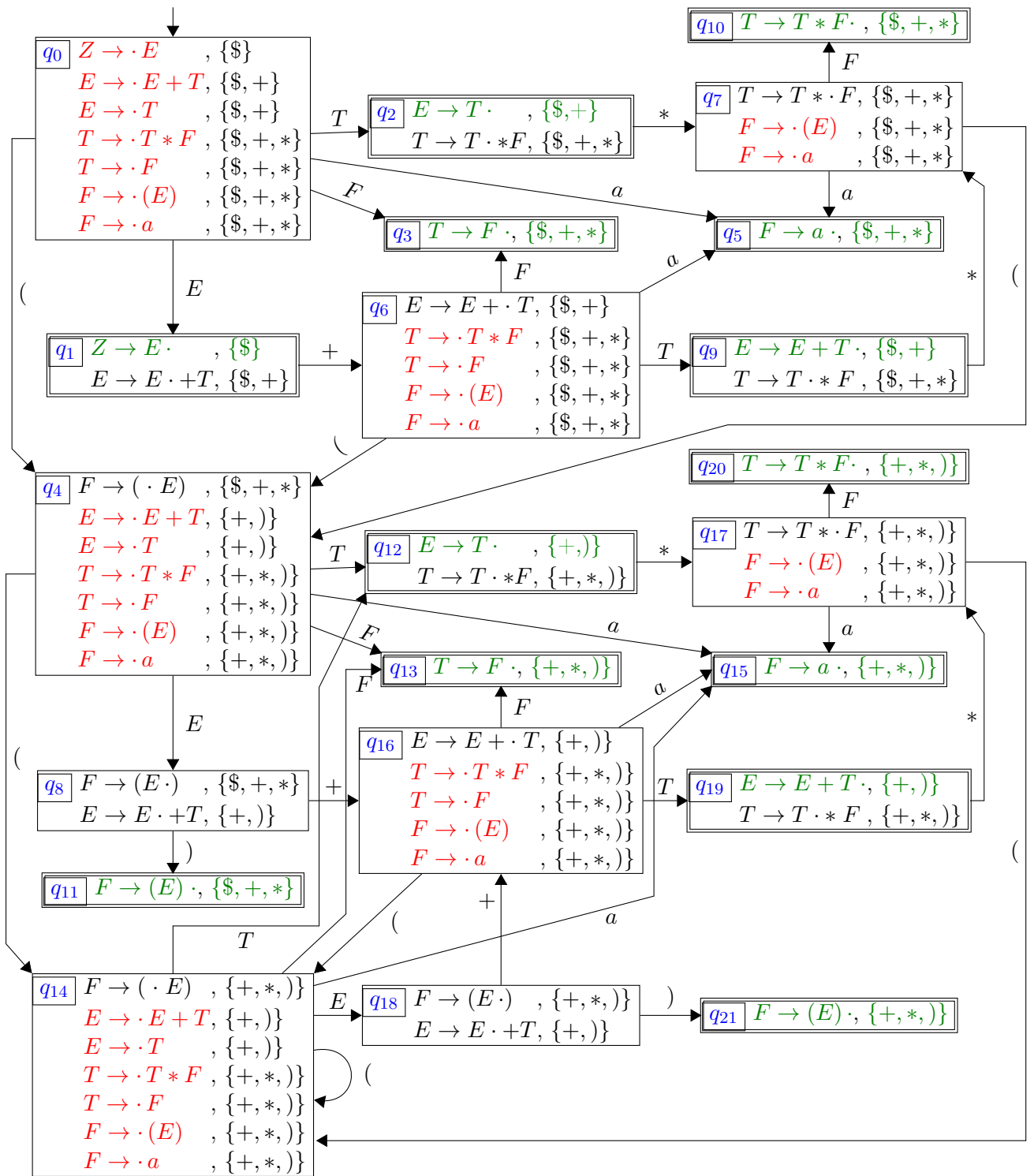
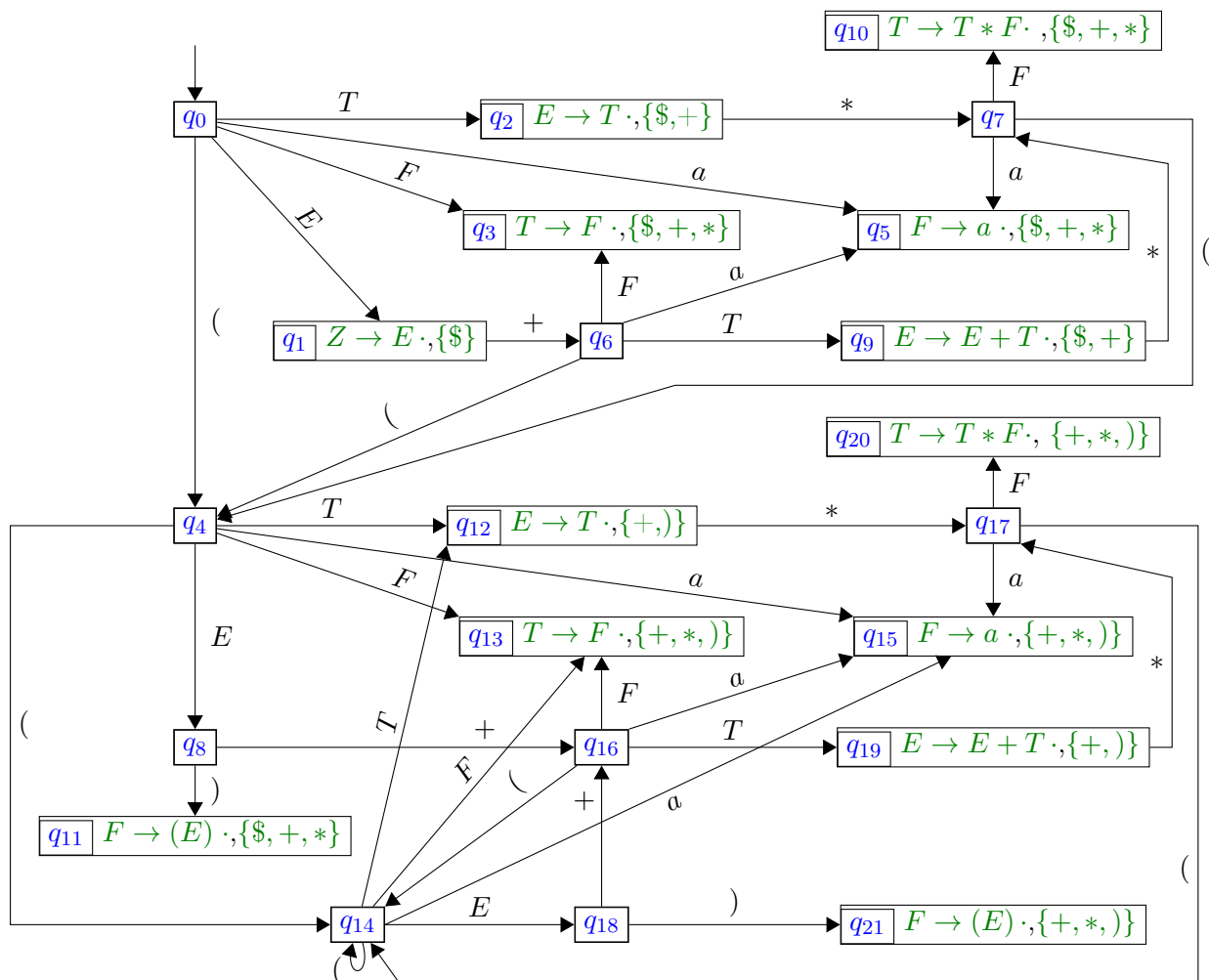
Abb. 5.21: (Deterministischer) LR(1)-Zustandsübergangsgraph für  $G_4$  (22 Zustände).

Abb. 5.22 zeigt den abgespeckten LR(1)-Zustandsübergangsgraphen für die Grammatik  $G_4$ . Dessen Zustände zeigen nur noch die (finalen Stadien der) Produktionen an, mit denen reduziert wird, falls das Vorschau-Symbol in der angegebenen Menge liegt.

Abb. 5.22: abgespeckter **LR(1)-Zustandsübergangsgraph** für die Grammatik  $G_4$ .

**Fazit:** Der Stadiumautomat  $K_G$  einer kontextfreien Grammatik  $G$  akzeptiert genau die Sprache  $L(G)$ , arbeitet aber leider nicht deterministisch. Der Grund für den Nichtdeterminismus liegt in den Expansionsübergängen:  $K_G$  muss raten, welche Produktion er für das aktuelle Nichtterminal –das Nichtterminal hinter dem Punkt– auswählen soll. Bei einer eindeutigen Grammatik ist höchstens eine Alternative richtig, um die restliche Eingabe abzuleiten.

Wir haben zwei Möglichkeiten, diesen Nichtdeterminismus zu beseitigen:

- Ein LL-Parser wählt deterministisch eine Produktion für das aktuelle Nichtterminal aus und benutzt dazu eine beschränkte Vorschau auf die restliche Eingabe. Nicht für alle kontextfreien Grammatiken, aber für  $LL(k)$ -Grammatiken, kann *genau ein Expansionsübergang* ausgewählt werden, wenn man das zu expandierende Nichtterminal und die nächsten  $k$  Eingabesymbole betrachtet. Dies gilt natürlich nicht für fehlerhafte Eingaben; dann existiert in irgendeiner Konfiguration kein Übergang. LL-Parser sind Linksparser.
- Ein LR-Parser verfolgt parallel alle Möglichkeiten, die zu einer (Rechts-)Ableitung für das Eingabewort führen können. Erst wenn ein *Reduktionsübergang* möglich ist, entscheidet der Parser, ob weiter gelesen oder ob reduziert werden soll und mit welcher Produktion. Diese Entscheidung basiert auf dem aktuellen Zustand des Automaten und  $k$  Symbolen der Vorschau. LR-Parser sind Rechtsparser. Auch LR-Parser existieren nicht für jede kontextfreie Grammatik, sondern nur für  $LR(k)$ -Grammatiken.

Der kanonische LR(1)-Parser hat leider deutlich mehr Zustände als der LR(0)-Parser. Wir können mit weniger Zuständen (und damit kleineren Syntaxanalyse-Tabellen) auskommen, wenn wir in Kauf nehmen, dass wir Fehler später erkennen (Kapitel 5.5.6) oder die anzuwendende Produktion aufwändiger zu bestimmen ist (Kapitel 5.5.7).

### 5.5.6 SLR(1)- und LALR(1)-Parser

Es gibt mehrere Ansätze, mit den Zuständen des LR(0)-Parsers auszukommen:

Wir wissen für jeden Zustand, ob und falls ja, mit welcher Produktion zu reduzieren ist. Sei  $A \rightarrow \alpha$  diese Produktion.

Und wir wissen bereits, welche Symbole auf das Nichtterminalsymbol  $A$  folgen können: die aus **Follow**( $A$ )!

Ein **SLR(1)-Parser** („S“ = „simple“) setzt seine reduce-Einträge daher nur *in die Spalten der Vorschau Symbole, die in der FOLLOW-Menge des Nichtterminalsymbols  $A$  enthalten sind*.

Die Grammatik  $G_4$  zeigt keine Konflikte mehr, sie ist eine SLR(1)-Grammatik.

Der **LALR(1)-Parser** („LA“ = „look ahead“; unglückliche Namensgebung, da SLR(1) und LR(1) auch die Vorschau nutzen) kommt mit den Zuständen des LR(0)-Parsers aus, indem alle Zustände des LR(1)-Parsers, die die gleichen Stadien enthalten, zusammengefasst werden, indem die **LookAhead**-Mengen gleicher Stadien vereinigt werden.

Der LALR(1)-Parser hat reduce-Einträge daher *in den Spalten der Vorschau Symbole, an denen die Zustände des LR(1)-Parsers mit den gleichen Stadien reduce-Einträge hatten*.

Im LALR(1)-Parser von  $G_4$  werden die Zustände  $i$  und  $i+10$  des LR(1)-Parsers für  $i = 2 \dots 11$  verschmolzen.

Nur die LL- und die LR-Parser haben folgende Eigenschaften:

- Der Parser liest das Eingabewort *einmal* von links nach rechts und baut dabei die Ableitung auf (als **Links**- bzw. als **Rechts**ableitung).
- Der Parser erkennt einen Fehler frühestmöglich, also beim *ersten* Zeichen  $a$ , das nicht zu einem Wort der Sprache gehören kann: Ist  $xy \notin L(G)$  und der Parser erkennt den Fehler beim Zeichen  $a$ , so gibt es ein Wort  $z$  mit  $xz \in L(G)$ .

Durch die zusätzlichen reduce-Einträge führt ein LALR(1)-Parser (SLR(1)- oder LR(0)-Parser) evtl. weitere Reduktionen aus, bevor er einen Fehler bemerkt (und erzeugt daher evtl. ungenauere Fehlermeldungen).

**SLR(1)-Syntaxanalyse-Tabelle** für  $G_4$

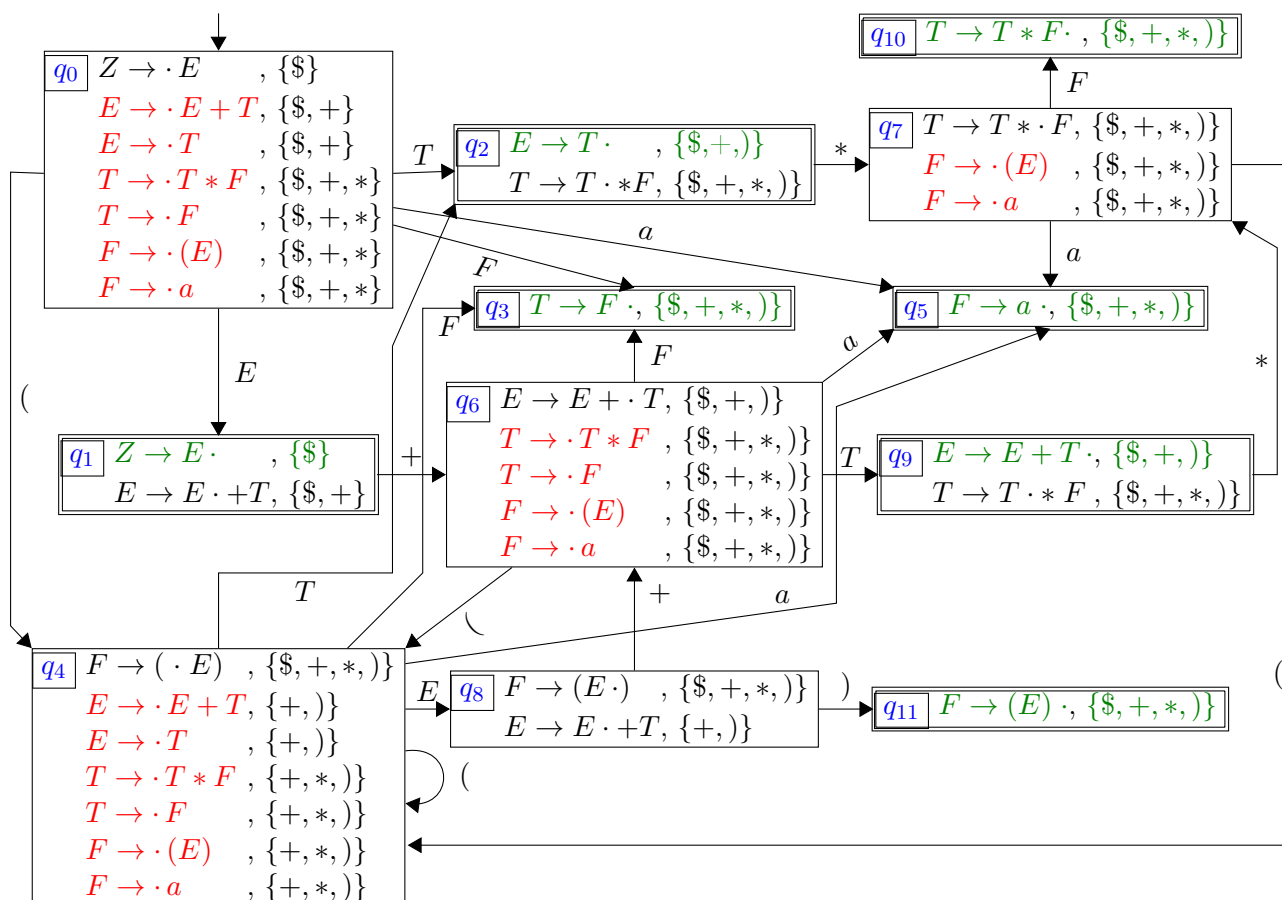
$A$	Zust.	Aktionstabelle						Sprungtab.		
		$a$	$+$	$*$	$($	$)$	$\$$	$E$	$T$	$F$
	0	s5			s4			1	2	3
$Z$	1		s6				acc			
$E$	2		r2	s7		r2	r2			
$T$	3		r4	r4		r4	r4			
	4	s5			s4			8	2	3
$F$	5		r6	r6		r6	r6			
	6	s5			s4			9	3	
	7	s5			s4					10
	8		s6			s11				
$E$	9		r1	s7		r1	r1			
$T$	10		r3	r3		r3	r3			
$F$	11		r5	r5		r5	r5			

Follow( $Z$ ) = { $\$$ }, Follow( $E$ ) = { $+$ ,  $)$ ,  $\$$ },

Follow( $T$ ) = Follow( $F$ ) = { $+$ ,  $*$ ,  $)$ ,  $\$$ }

**LALR(1)-Syntaxanalyse-Tabelle** für  $G_4$

Zust.	Aktionstabelle						Sprungtab.		
	$a$	$+$	$*$	$($	$)$	$\$$	$E$	$T$	$F$
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4			9	3	
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Abb. 5.23: **LALR(1)-Zustandsübergangsgraph** für die Grammatik  $G_4$ .

Seinen abgespeckten Graphen haben Sie schon in Abb. 5.14 kennengelernt.

Betrachten wir noch einmal die Konfigurationsübergänge für das Wort  $(a + a) * a$ .

Schauen wir uns den Übergang unter Produktion (1), also  $E \rightarrow E + T$ , an.

[ 0	, ( a + a ) * a \$ ]	[ 0 4 8	, ) * a \$ ]
— [ 0 4	, a + a ) * a \$ ]	(1) — [ 0 4 8 11	, * a \$ ]
— [ 0 4 5	, + a ) * a \$ ]	— [ 0 3	, * a \$ ]
(6) — [ 0 4 3	, + a ) * a \$ ]	(5) — [ 0 2	, * a \$ ]
(4) — [ 0 4 2	, + a ) * a \$ ]	(4) — [ 0 2 7	, a \$ ]
(2) — [ 0 4 8	, + a ) * a \$ ]	— [ 0 2 7 5	, \$ ]
— [ 0 4 8 6	, a ) * a \$ ]	(6) — [ 0 2 7 10	, \$ ]
— [ 0 4 8 6 5	, ) * a \$ ]	(3) — [ 0 2	, \$ ]
(6) — [ 0 4 8 6 3	, ) * a \$ ]	(2) — [ 0 1	, \$ ]
(4) — [ 0 4 8 6 9	, ) * a \$ ]	— accept	

Die obersten Zustände 4 8 6 9 beschreiben einen Weg vom initialen Stadium zum finalen Stadium dieser Produktion. Wird mit dieser Produktion reduziert, ist das Nichtterminalsymbol  $E$  der übergeordneten Produktion abgearbeitet und der Automat geht vom Zustand 4 unter  $E$  in den Zustand 8.

Für die Produktion  $E \rightarrow E + T$  gibt es nur die Wege 0 1 6 9 (ersetzt durch 0 1) und 4 8 6 9 (ersetzt durch 4 8).

### 5.5.7 Optimierter LR(1)- und LALR(1)-Parser

Wir wollen die volle Sprachklasse der deterministischen, kontextfreien Sprachen (der LR(1)-Sprachen) beibehalten und damit die Eigenschaft, dass jeder Fehler zum frühestmöglichen Zeitpunkt entdeckt wird. Trotzdem soll der endliche Automat mit weniger Zuständen auskommen.

Ausgehend von der Beobachtung, dass zur Berechnung der erwarteten Vorschausymbole nur die *Zukunft* des Stadiums und die bisherige erwartete Vorschau Menge nötig sind [23], entfernen wir im Stadium zur Berechnung eines **vergangenheitsreduzierten Automaten** die Vergangenheit und das aktuelle Nichtterminalsymbol.<sup>17</sup>

Wir definieren daher das **Zukunftsstadium** eines erweiterten Stadiums  $[A \rightarrow \alpha \cdot B\beta, u]$  als  $[\beta, u]$ .

Wir erhalten damit Automaten mit i.Allg. deutlich weniger Zuständen (Abb. 5.24, Abb. 5.25).

Finale Zukunftsstadien haben als erste Komponente stets das leere Wort  $\varepsilon$ . Daher können wir am obersten Zustand immer noch erkennen, ob reduziert werden kann, benötigen aber für die Entscheidung, *mit welcher Produktion* zu reduzieren ist, evtl. mehrere oberste Zustände.

**Entscheidungsbäume** zeigen dann an, mit welcher Produktion reduziert wird: Ausgehend vom letzten Zustand (oberstes Kellersymbol) und der Vorschau-Menge werden evtl. die darunterliegenden Kellersymbole (und zugehörige Eingabesymbole<sup>18</sup>) überprüft, bis eine Entscheidung für eine Produktion und den Folgezustand getroffen wird.

Diese Entscheidungsbäume entstehen dadurch, dass man alle Wege des Automaten von den initialen zu den finalen (Zukunfts-)Stadien verfolgt und anschließend ausgehend von den finalen Stadien gruppiert:

$$\begin{aligned} 0 &\xrightarrow{E} 1 \\ 0 &\xrightarrow{E} 1 \xrightarrow{+} 6 \xrightarrow{T} 2 \\ 0 &\xrightarrow{T} 2 \\ 0 &\xrightarrow{T} 2 \xrightarrow{*} 7 \xrightarrow{F} 3 \\ 0 &\xrightarrow{F} 3 \\ 0 &\xrightarrow{(} 4 \xrightarrow{E} 8 \xrightarrow{)} 3 \\ 0 &\xrightarrow{a} 3 \end{aligned}$$

$$\begin{aligned} 4 &\xrightarrow{E} 8 \xrightarrow{+} 26 \xrightarrow{T} 22 \\ 4 &\xrightarrow{T} 22 \\ 4 &\xrightarrow{T} 22 \xrightarrow{*} 27 \xrightarrow{F} 23 \\ 4 &\xrightarrow{F} 23 \\ 4 &\xrightarrow{(} 24 \xrightarrow{E} 28 \xrightarrow{)} 23 \\ 4 &\xrightarrow{a} 23 \\ 6 &\xrightarrow{T} 2 \xrightarrow{*} 7 \xrightarrow{F} 3 \\ 6 &\xrightarrow{F} 3 \\ 6 &\xrightarrow{(} 4 \xrightarrow{E} 8 \xrightarrow{)} 3 \\ 6 &\xrightarrow{a} 3 \\ 7 &\xrightarrow{(} 4 \xrightarrow{E} 8 \xrightarrow{)} 3 \\ 7 &\xrightarrow{a} 3 \end{aligned}$$

$$\begin{aligned} 24 &\xrightarrow{E} 28 \xrightarrow{+} 26 \xrightarrow{T} 22 \\ 24 &\xrightarrow{T} 22 \\ 24 &\xrightarrow{T} 22 \xrightarrow{*} 27 \xrightarrow{F} 23 \\ 24 &\xrightarrow{F} 23 \\ 24 &\xrightarrow{(} 24 \xrightarrow{E} 28 \xrightarrow{)} 23 \\ 24 &\xrightarrow{a} 23 \\ 26 &\xrightarrow{T} 22 \xrightarrow{*} 27 \xrightarrow{F} 23 \\ 26 &\xrightarrow{F} 23 \\ 26 &\xrightarrow{(} 24 \xrightarrow{E} 28 \xrightarrow{)} 23 \\ 26 &\xrightarrow{a} 23 \\ 27 &\xrightarrow{(} 24 \xrightarrow{E} 28 \xrightarrow{)} 23 \\ 27 &\xrightarrow{a} 23 \end{aligned}$$

**vergangenheitsreduzierte LR(1)-  
Syntaxanalyse-Tabelle für  $G_4$**

Zust.	Aktionstabelle					
	$a$	$+$	$*$	$($	$)$	$\$$
0	s3			s4		
1		s6				acc
2		r	s7			r
3		r	r			r
4	s23			s24		
6	s3			s4		
7	s3			s4		
8		s26			s3	
22		r	s27		r	
23		r	r		r	
24	s23			s24		
26	s23			s24		
27	s23			s24		
28		s26			s23	

<sup>17</sup> Andere Optimierungen betreffen Kettenproduktionen oder nicht-erreichbare Fehlerzustände.

<sup>18</sup> Werden **Eingabesymbole** für den Entscheidungsbaum benötigt, schreibt man sie auch auf den Keller.

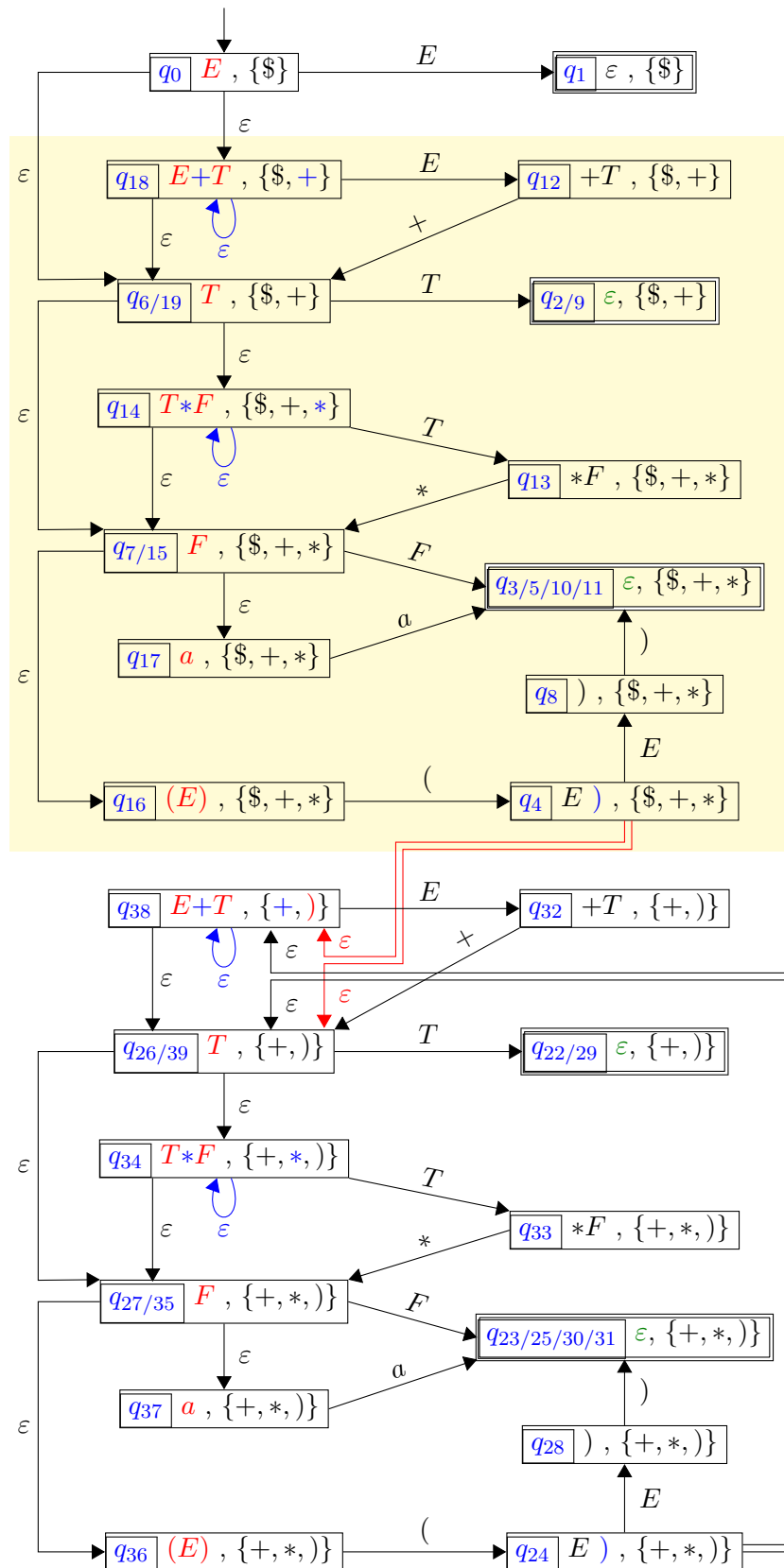
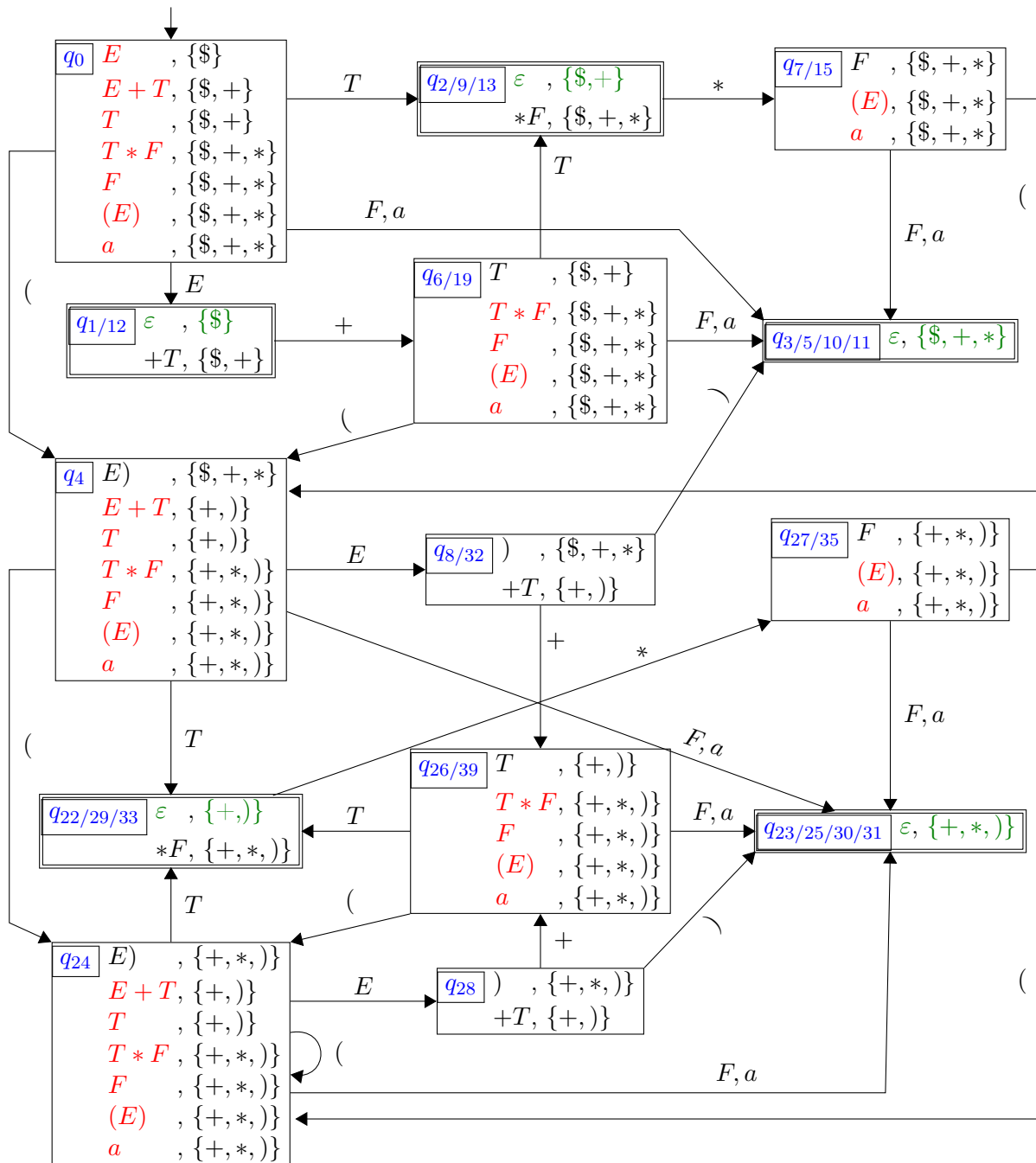
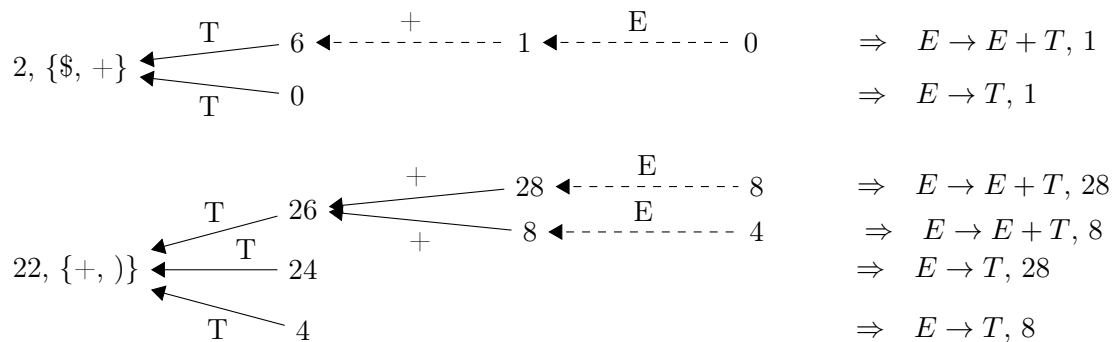
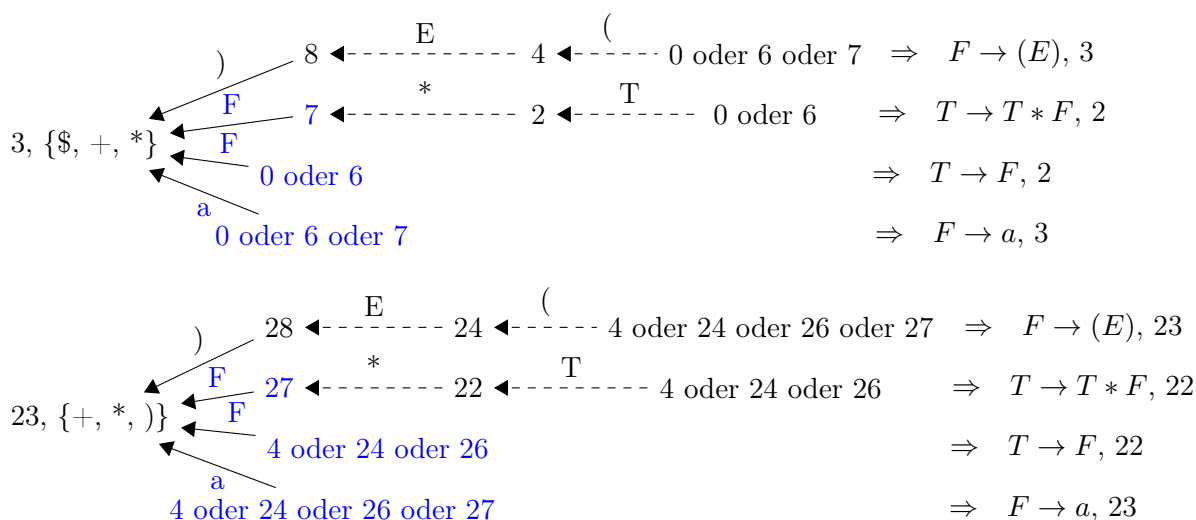


Abb. 5.24: Ausgehend von Abb. 5.20 lassen wir die Vergangenheit des Stadiums und das aktuelle Nichtterminalsymbol weg und verschmelzen entstehende gleiche Zustände.

Abb. 5.25: vergangenheitsreduzierter LR(1)-Übergangsgraph der Grammatik  $G_4$  (14 Zustände).





Die gestrichelten Teile der Entscheidungsbäume werden nicht benötigt, da die Produktion bereits feststeht. In den blau markierten Teilen werden Eingabesymbole zur Entscheidung benötigt.

Gelangt man vom Zustand 23 rückwärts in den Zustand 4, 24 oder 26, so kann man –als weitere Optimierung– nach der Reduktion mit  $F \rightarrow a$  gleich die Reduktion mit  $T \rightarrow F$  anschließen (und in den Zustand 22 gehen). Kommt man vom Zustand 23 rückwärts in den Zustand 27, so kann man nach der Reduktion mit  $F \rightarrow a$  gleich die Reduktion mit  $T \rightarrow T * F$  anschließen (und ebenfalls in den Zustand 22 gehen).

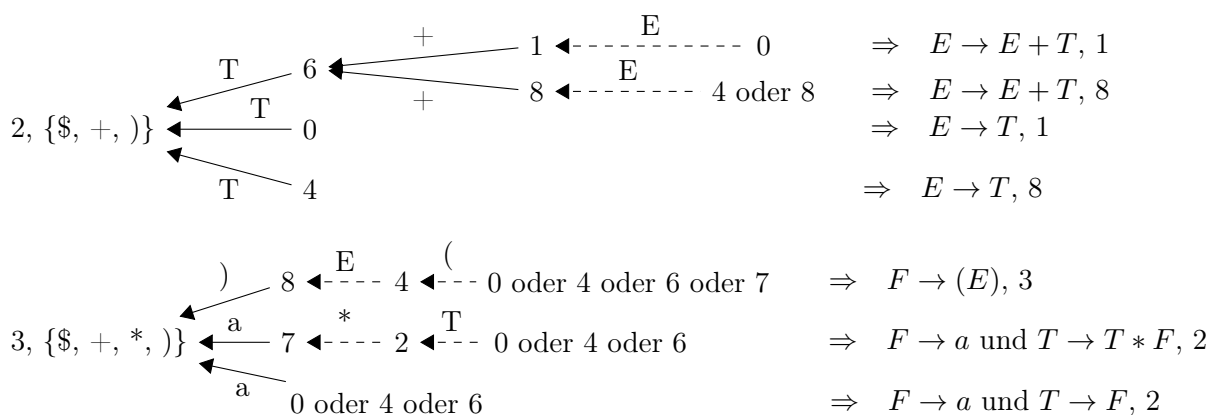
Analoges gilt für den Zustand 3, so dass man in diesem Beispiel ganz auf Eingabesymbole im Entscheidungsbaum verzichten kann.

Der **vergangenheitsreduzierte LALR(1)-Automat** für  $G_4$  entsteht durch Verschmelzung der Zustände  $i$  und  $i + 20$  für  $i = 2, 3, 4, 6, 7, 8$  und hat daher –wie der vergangenheitsreduzierte LR(0)-Automat– nur noch 8 Zustände.

Die Syntaxanalyse-Tabelle legt nahe, auch die Zustände 0, 4, 6 und 7 zusammenzufassen. Aber diese Zustände werden gebraucht, um in den Entscheidungsbäumen die anzuwendenden Produktion auszuwählen. Umgekehrt werden die Zustände 1, 2 und 8 nur nach Anwendung einer Produktion erreicht.

**vergangenheitsreduzierte LALR(1)-Syntaxanalyse-Tabelle für  $G_4$**

Zust.	Aktionstabelle					
	$a$	$+$	$*$	$($	$)$	$\$$
0	s3			s4		
1		s6				acc
2		r	s7		r	r
3		r	r		r	r
4	s3			s4		
6	s3			s4		
7	s3			s4		
8		s6			s3	



## 6 Semantische Analyse

Während der semantischen Analyse wird z.B. überprüft, ob alle verwendeten Identifier deklariert wurden, ob die Typen von Konstanten und Variablen zu den auf sie angewendeten Operationen passen und ob die Anzahl der Parameter stimmt.

### 6.1 Typprüfung

Ein Objekt einer Programmiersprache besitzt mehrere Eigenschaften:

1. Eine **Speicheradresse**, ab der dieses Objekt im Speicher abgelegt ist. Diese Speicheradresse wird vom Compiler, manchmal in Verbindung mit dem Linker, festgelegt und kann sich während des Programmlaufs durchaus ändern (z.B. bei lokalen Variablen, die in rekursiven Prozeduren auftreten).
2. Eine **Codierung** des Objektes selbst, d.h. das **Bitmuster**, das den **Wert** des Objektes darstellt. Um mit dem Wert eines Objektes arbeiten zu können, muss man wissen, wie lang diese Codierung ist und wie sie zu interpretieren ist.

Die Art der Codierung eines Objekts wird durch seinen **Typ** festgelegt, der in vielen Programmiersprachen, z.B. bei der Variablendeklaration, angegeben wird.

Jeder arithmetische Ausdruck hat ebenfalls einen Typ (oder sogar mehrere: z.B. sind nicht negative ganze Zahlen in Modula-2 vom Typ **INTEGER** oder **CARDINAL**). Die Operatoren einer Programmiersprache verlangen meist Operanden eines speziellen Typs. Ein Übersetzer kann oft prüfen, ob die durch die Sprachdefinition der Programmiersprache gegebenen Typ-Beschränkungen im vorgelegten Programm eingehalten werden (**statische Typ-Prüfung**) oder ob Programmierfehler berichtet bzw. Typen automatisch umgewandelt werden müssen. Dies kann bei einfachen Typ-Regeln während des Parsens bzw. bei der Zwischencode-Erzeugung (z.B. in Pascal) oder sonst in einem zusätzlichen Pass des Übersetzers (z.B. Ada, C++ oder Java) geschehen.

Andererseits kann diese Überprüfung ganz oder teilweise erst zur Laufzeit durchgeführt werden (**dynamische Typ-Prüfung**), indem der Übersetzer Code erzeugt, um z.B. eine Bereichsüberschreitung beim Zugriff auf ein dynamisches Feld zu erkennen oder um eine Implementation für eine virtuelle Funktion in C++ auszuwählen (**dynamische Bindung**).

In einigen Sprachen (z.B. Smalltalk, Scheme) ist keine Typ-Deklaration vorgesehen. Dann wird jedem Datenobjekt eine Kodierung des Typs (bzw. in Smalltalk der Klasse) zugeordnet. Diese Kodierung muss ebenfalls gespeichert werden (**tag-System**) und wird immer überprüft, wenn das Objekt als Operand einer Operation auftritt.

Verhindert die Typ-Prüfung des Übersetzers (fast) alle Typ-Fehler zur Laufzeit, so spricht man von einer Programmiersprache mit **starker Typ-Prüfung** (*strongly typed language*).

Eine andere Klasse von Sprachen (z.B. ML, Caml oder Haskell) erlaubt die Deklaration von Variablen ohne Angabe eines Typs, führt aber trotzdem eine starke Typ-Prüfung durch. Hier wird durch logisches Schließen vom Gebrauch der Variablen auf deren Typ geschlossen und so eine konsistente Verwendung der Variablen gewährleistet.

Den Prozess des Ableitens von Typen für Konstrukte der Programmiersprache aus verfügbaren Typ-Informationen nennt man **Typ-Inferenz**. So kann aus der Konstante der Wertzuweisung `x := 1.0` geschlossen werden, dass die Variable `x` vom Typ **real** ist; daher führt eine nachfolgende Anweisung `a[x] := 0` zu einer Fehlermeldung wegen Typ-Unverträglichkeit.

### 6.1.1 Typ-Ausdrücke

Jedem Objekt und jedem Ausdruck im Programm ordnet der Übersetzer einen **Typ-Ausdruck** zu. Typ-Ausdrücke sind Terme, die aus Typen und vorgegebenen Konstruktoren gebildet werden. Von **Basistypen** ausgehend werden rekursiv immer komplexere Typ-Ausdrücke aufgebaut.

Was ein Basistyp ist, hängt von der Programmiersprache ab; z.B. besitzt FORTRAN 90 den Basistyp *complex*, Pascal aber nicht. Das **Typ-System** einer Programmiersprache besteht aus Regeln, die jedem Objekt eines Programms einen Typ-Ausdruck zuordnen.

#### Beispiele für Typ-Ausdrücke

1. Jeder Basistyp ist ein Typ-Ausdruck.

Zu den Basistypen gehören oft: *bool*, *char*, *int*, *real*, *void*.

Aufzählungen wie 1..7 oder (*mo*, *di*, *mi*, *do*, *fr*) und Unterbereiche von einigen Basistypen sowie der spezielle (Fehler-)Typ *type-error* können auch als Basistypen angesehen werden. Aufzählungs- und Unterbereichstypen bringen jedoch viele Probleme in das Typ-System und werden in modernen Sprachen meist nicht mehr zu den Basistypen gezählt.

2. Kann man einem Typ einen Namen geben, so ist dieser Name ebenfalls ein Typ-Ausdruck.

**Beispiel:** Durch `type Zeichen = char` wird *Zeichen* ein Typ-Ausdruck.

3. Ist  $T$  ein Typ-Ausdruck, so ist  $\text{array}(T)$  ebenfalls ein Typ-Ausdruck (für einen Feld-Typ).

*array* ist ein Beispiel für einen **Typ-Konstruktor**, der einen neuen Typ erzeugt.

In einigen Programmiersprachen gehört der Indexbereich des Feldes zum Typ. Dann hätte der Typ-Ausdruck die Form  $\text{array}(I, T)$ , wobei  $I$  ein Typ-Ausdruck für einen Unterbereich von *int* oder eine Aufzählung ist. Manchmal (z.B. in der Programmiersprache C) ist der untere Index des Feldes fixiert. Dann genügt die Länge des Feldes im Typ-Ausdruck.

**Beispiel:** In `var A : array[1..10] of integer` ist  $\text{array}(\text{int})$  Typ-Ausdruck für *A*.

4. Ist  $T$  ein Typ-Ausdruck, so ist  $\text{list}(T)$  ebenfalls ein Typ-Ausdruck für einen Listen-Typ. Dieser Typ-Konstruktor wird natürlich nur dann benötigt, wenn Listen elementare Datenstrukturen der Programmiersprache sind.
5. Sind  $T_1, \dots, T_r$  Typ-Ausdrücke mit  $r \geq 1$ , dann ist  $T_1 \times \dots \times T_r$  ein Typ-Ausdruck, der Tupel mit  $r$  Komponenten vom Typ  $T_1, \dots, T_r$  beschreibt.

Dann kann man die Typen der Parameter einer Funktion zu einem Typ zusammenfassen oder den Ergebnistyp von Funktionen mit mehreren Rückgabewerten festlegen.

6. Ist  $T$  ein Typ-Ausdruck, dann ist  $\text{pointer}(T)$  ein Typ-Ausdruck, der einen Zeigertyp auf ein Objekt vom Typ  $T$  beschreibt.

**Beispiel:** `var p: ↑integer` ordnet der Variablen *p* den Typ-Ausdruck  $\text{pointer}(\text{int})$  zu.

7. Sind  $T, T_1, \dots, T_r$  Typ-Ausdrücke, dann ist  $T_1 \times \dots \times T_r \rightarrow T$  ein Typ-Ausdruck für eine Funktion mit  $r \geq 1$  Parametern vom Typ  $T_1, \dots, T_r$  und einem Ergebnis vom Typ  $T$ .

Der Operator  $\times$  ist linksassoziativ und hat eine größere Priorität als der rechtsassoziative Operator  $\rightarrow$ . Der Typ-Ausdruck

$$\text{char} \times \text{char} \rightarrow \text{int} \rightarrow \text{pointer}(\text{int} \rightarrow \text{int})$$

wird daher so geklammert:

$$(\text{char} \times \text{char}) \rightarrow (\text{int} \rightarrow \text{pointer}(\text{int} \rightarrow \text{int}))$$

**Bemerkung:** Existieren in der Programmiersprache Datenobjekte, auf deren Komponenten über den Namen zugegriffen werden kann, z.B. Recordstrukturen und Objektstrukturen, so sind weitere Konstruktionen notwendig, um deren Datentyp zu beschreiben.

**Beispiel 6.1:** Die Deklaration

ordnet dem Namen `f` folgenden Typ-Ausdruck zu:

```
function f(a,b:char): ↑integer
      char × char → pointer(int)
```

Es gibt leider viele Konstruktionen in Programmiersprachen, die implementationsabhängig sind. Seien z.B. folgende Pascal-Deklarationen gegeben:

<pre>var y: 1..20;     z: 10..50;     a: array[1..10] of integer;     b: array[0..9] of integer;</pre>	<ul style="list-style-type: none"> <li>• Soll der Term <code>y+z</code> erlaubt sein?</li> <li>• Und welchen Typ hätte das Ergebnis?</li> <li>• Ist eine Wertzuweisung <code>a := b</code> bei gleich großen Feldern erlaubt?</li> </ul>
--	--

Auch ein Überschreiten einer Feldgrenze beim Zugriff auf ein Feld kann als Typ-Fehler angesehen werden. So ein Fehler ist jedoch nicht immer bereits zur Übersetzungszeit lokalisierbar und wird bei einigen Programmiersprachen aus Effizienzgründen leider nicht einmal zur Laufzeit erkannt.

### 6.1.2 Äquivalenz von Typ-Ausdrücken

Bei der Typ-Prüfung muss z.B. getestet werden, ob der Typ-Ausdruck `typ1` eines aktuellen Operanden zum Typ-Ausdruck `typ2` für den vorgesehenen Operanden einer Operation passt, also die Typen `typ1` und `typ2` **äquivalent** sind.

Falls keine Namen als Abkürzungen für Typ-Ausdrücke auftreten, ist die Sache relativ einfach. Die offensichtliche Definition der Typ-Äquivalenz wäre in diesem Fall:

*Zwei Typ-Ausdrücke sind äquivalent, wenn sie identisch sind.*

Der Typ-Prüfer muss also nur beide Typ-Ausdrücke bzgl. ihres Aufbaus miteinander vergleichen. Dies geschieht am besten rekursiv.

Dürfen **Namen in Typ-Ausdrücken** vorkommen, gibt es zusätzliche Probleme.

**Beispiel 6.2:** Sind die Pascal-Variablen `next` und `last` vom gleichen Typ?

```
type link = ↑cell;
var next : link;
    last : ↑cell;
```

Im ersten Referenz-Handbuch zu Pascal [21] wird zwar häufig die „Typ-Gleichheit“ von Konstrukten der Programmiersprache gefordert, der Begriff aber nirgends definiert, d.h. die Antwort auf die obige Frage war implementationsabhängig!

Es gibt zwei prinzipielle Möglichkeiten der Interpretation von Namen für Typ-Ausdrücke:

**Namensäquivalenz:** Jeder Typ-Name legt einen neuen Typ fest. Zwei Typ-Ausdrücke sind dann nur äquivalent, wenn sie identisch sind (Ada, abgeschwächt in Pascal und Modula-2).

Z.B. sind in Ada folgende Typen sinnvollerweise unterschiedlich:

```
type sec is range 0..60;
type min is range 0..60;
```

Objektorientierte Sprachen mit einem strengen Klassenkonzept (z.B. Smalltalk, Eiffel, Java) nutzen die Namensäquivalenz.

**Strukturäquivalenz:** Jeder Typ-Name ist nur eine Abkürzung für den definierten Typ-Ausdruck. Dann sind –in nicht-rekursiven Definitionen– zwei Typ-Ausdrücke äquivalent, wenn man jeden auftretenden Namen durch den zugeordneten Typ-Ausdruck ersetzt und die resultierenden Typ-Ausdrücke identisch sind (z.B. Algol 60, Algol 68, FORTRAN, COBOL). Diese Form der Äquivalenz wird in modernen Sprachen nicht mehr verwendet.

Im Beispiel 6.2 sind die Typen der Variablen `next` und `last` strukturäquivalent, aber nicht namensäquivalent.

### 6.1.3 Typ-Umwandlung

Speziell für Basistypen gibt es in vielen Programmiersprachen Umwandlungsregeln, die eine automatische Anpassung der Typen von Operanden an zulässige Typen eines Operators ermöglichen. Man betrachte das folgende Beispiel:

$y := x + i$ ; mit  $x, y$  vom Typ *real* und  $i$  vom Typ *int*

Da es i.Allg. keinen Operator für diese Additionsoperation mit unterschiedlichen Operandentypen gibt, muss der Compiler oder der Programmierer etwas tun:

- Der Compiler führt eine **automatische Typ-Anpassung** (*type coercion*) durch, indem er eine Typ-Umwandlungsoperation einfügt. Dies ist meist nur erlaubt, wenn damit keine Genauigkeitsverluste verbunden sind, z.B. in C bei der Umwandlung von *char* nach *int*.
- Der Compiler signalisiert eine Fehlersituation, wie z.B. „MIXED MODE“ in FORTRAN. Dann kann der Programmierer eine **explizite Typ-Umwandlung** vornehmen durch
  - Aufruf einer Bibliotheksfunktion, z.B.  $x + \text{real}(i)$  oder
  - *casting*, z.B.  $(\text{char}) 20$ . Dadurch wird der zugehörige Typ-Ausdruck geändert, aber nicht unbedingt die interne Darstellung des Wertes.

### 6.1.4 Überlagerung von Funktionen (ad hoc Polymorphie)

Manchmal haben verschiedene Operatoren eine identische lexikale Darstellung, z.B. kann in Java  $+$  eine Ganzzahl-Addition, eine Gleitpunkt-Addition oder eine Zeichenketten-Konkatenation repräsentieren. Man bezeichnet  $+$  daher als **überlagerten Operator** (*overloading*). Der Compiler muss dann zur Übersetzungszeit aus dem Kontext den „richtigen“ Operator identifizieren.

Das Problem verschärft sich bei Programmiersprachen, die dem Programmierer mehrere Definitionen einer Funktion mit unterschiedlichen Parameter- oder Ergebnistypen erlauben.

In Ada sind z.B. folgende Definitionen gleichzeitig erlaubt:

```
function '*' (i, j: integer) return complex;
function '*' (x, y: complex) return complex;
```

Damit hat ein Name einer Funktion eine Vielzahl von Typ-Ausdrücken und auch (arithmetische) Ausdrücke haben nicht notwendigerweise nur einen Typ!

**Beispiel 6.3:** Nehmen wir an, dass der Operator  $*$  die folgenden drei Typ-Ausdrücke hat:

$int \times int \rightarrow int$                        $int \times int \rightarrow complex$                        $complex \times complex \rightarrow complex$

Haben dann die Literale 2, 3 und 5 den einzig möglichen Typ *int* und ist  $z$  eine Variable vom Typ *complex*, dann kann  $3 * 5$  den Typ *int* oder *complex* haben. Im Ausdruck  $(3 * 5) * 2$  **muss**  $3 * 5$  den Typ *int* haben, im Ausdruck  $(3 * 5) * z$  **muss**  $3 * 5$  den Typ *complex* haben.

Hier kann man auch schon eine Idee für einen Algorithmus zur Operator-Identifikation erkennen.

In einer ersten Phase wird „von unten nach oben“ jedem Teilausdruck eine Menge von möglichen Typ-Ausdrücken zugeordnet. Hat man auf diese Weise für den Gesamtausdruck einen Typ festgestellt (es darf üblicherweise für den Gesamtausdruck nur noch einen Typ geben!), so geht man dann „von oben nach unten“ durch den Ausdruck und ordnet jedem Teilausdruck den „richtigen“ Typ-Ausdruck zu.

Noch mehr Komplikationen ergeben sich, wenn –wie in C++– selbstdefinierte und automatische Typ-Konvertierung erlaubt sind.

### 6.1.5 Typvariablen für polymorphe Funktionen

Weitere Probleme treten auf, wenn die Programmiersprache (generisch) **polymorphe Funktionen** erlaubt, also Funktionen, bei denen der Typ der Parameter nicht eindeutig festgelegt werden muss. Dieses Konzept ist wichtig und hilfreich, z.B. möchte man Sortierprogramme ohne Rücksicht auf den speziellen Typ der zu sortierenden Elemente schreiben können. Ein anderes Beispiel ist eine Funktion zur Bestimmung der Länge einer Liste, hier in Haskell:

```
length(ls) = if null(ls) then 0
            else length(tail(ls)) + 1
```

Ähnliche Konstruktionen findet man in anderen funktionalen Sprachen wie Scheme oder ML oder in rein objektorientierten Sprachen wie Smalltalk, aber z.B. nicht in Pascal. Welchen Typ-Ausdruck kann man der Funktion `length` zuordnen?

Um Typ-Ausdrücke für polymorphe Funktionen angeben zu können, braucht man **Typvariable**. Eine Typvariable steht dabei für einen beliebigen Typ. Für das obige Beispiel könnte man der Funktion `length` den Typ-Ausdruck  $list(\alpha) \rightarrow int$  zuordnen, wobei  $\alpha$  eine Typvariable ist.

Die Typ-Prüfung von Programmen, in denen polymorphe Funktionen auftreten können, wird signifikant schwieriger:

- Verschiedene Vorkommen derselben polymorphen Funktion in einem Ausdruck können durchaus Argumente unterschiedlichen Typs haben.
- Da in Typ-Ausdrücken (Typ-)Variablen vorkommen können, ist die Typ-Äquivalenz neu zu definieren. Um zwei Typ-Ausdrücke mit Variablen „anzupassen“, müssen die auftretenden Variablen durch Typ-Ausdrücke (evtl. mit anderen Variablen) ersetzt werden. Dieses Vorgehen heißt **Unifikation** und wird auch in der logischen Programmierung benutzt.

**Beispiel 6.4:** Die Typ-Ausdrücke  $pointer(\alpha)$  und  $pointer(pointer(\beta))$  kann man z.B. durch  $\alpha \mapsto pointer(\gamma)$  und  $\beta \mapsto \gamma$  unifizieren.

Es gibt verschiedene Möglichkeiten, wie man die Typ-Prüfung in Programmiersprachen, die polymorphe Funktionen und Operatoren erlauben, durchführen kann, ggf. auch kombiniert:

- Man verzichtet auf die Typisierung in der Programmiersprache und auf eine Überprüfung zur Übersetzungszeit und benutzt eine dynamische Typ-Überprüfung zur Laufzeit (z.B. in Lisp, Scheme oder Smalltalk)
- Man verzichtet auf die Typisierung in der Programmiersprache, prüft aber zur Übersetzungszeit jeden Gebrauch eines Namens oder Sprachkonstrukts auf einen konsistenten Gebrauch (z.B. in ML oder Haskell)
- Man erweitert die Syntax der Programmiersprache um Typvariable oder „durchlöchert“ das übliche Typ-System durch Verwendung von Zeigern oder durch beliebiges „casting“.

**Beispiel 6.5:** Eine Programmiersprache mit Typvariablen ermöglicht die Deklarationen:

```
deref : pointer( $\alpha$ )  $\rightarrow$   $\alpha$ ;
q : pointer(pointer(integer));
```

Tritt dann im Programm der Ausdruck `deref(deref(q))` auf, so kann durch Typ-Inferenz geschlossen werden, dass dieser Ausdruck den Typ `int` hat.

### Übersetzung polymorpher Funktionen

Um z.B. eine polymorphe Funktion `swap` mit Typ-Ausdruck  $(\alpha, \alpha) \rightarrow (\alpha, \alpha)$  zu übersetzen, die die Komponenten eines Paares vertauscht,

```
swap (x,y) = (y,x)
```

müssen alle Datenstrukturen eine einheitliche Darstellung im Speicher haben. Denn bei Ausführung der Funktion `swap` muss die Anzahl der zu kopierenden Zellen unabhängig vom konkreten

Typ der Typvariablen  $\alpha$  sein. Daher stellt man jedes Datum, das mehr Platz als eine Zelle benötigt, als Zeiger dar. Diese **Zeiger-Darstellung** ist oft nicht so effizient wie eine spezialisierte Darstellung, die vom konkreten Typ abhängt, denn die Zeiger-Darstellung braucht mehr Speicherplatz und der Zugriff auf die Datenstruktur mehr Operationen.

In polymorphen Sprachen (ohne höhere Funktionen) kann man genauso effizient wie in monomorphen Sprachen sein, wenn man für jede Anwendungsstelle der polymorphen Funktion im Programm die Belegung der Typvariablen bestimmt und jede vorkommende Typinstanz mit ihrer spezialisierten Darstellung übersetzt. (Diese Technik wird oft in **Ada** benutzt.)

## 6.2 Unifikation

Bevor wir die Unifikation definieren und einen Algorithmus zur Unifikation angeben können, müssen wir (Typ-)Variablen, (Typ-)Terme und Substitutionen einführen.

Dabei umgeht man Probleme der Klammerung oder der Priorisierung der beteiligten Operatoren, indem man für die Ausdrücke (**Terme**) die Präfix-Notation verwendet.

**Definition:** Sei  $V$  ein Alphabet von **Variablen** und  $F$  ein Alphabet von **Funktionssymbolen**. Jedem Funktionssymbol  $f \in F$  ist eine natürliche Zahl  $\rho(f)$  (**Stelligkeit**) zugeordnet, die die Zahl der Parameter der Funktion  $f$  angibt. Funktionssymbole  $f$  mit  $\rho(f) = 0$  heißen **Konstante**.

Mit Hilfe dieser Alphabete konstruiert man nun Ausdrücke (**Terme**) in Präfix-Notation.

**Definition:** Seien  $V$  und  $F$  Alphabete von Variablen und Funktionssymbolen. Dann gilt:

- (a) Jede Variable  $x \in V$  ist ein Term.
- (b) Ist  $f \in F$  und sind  $t_1, \dots, t_{\rho(f)}$  Terme, so ist auch  $f(t_1, \dots, t_{\rho(f)})$  ein Term.

Bildet man einen Term aus einer Konstanten  $f$ , so schreibt man auch kurz  $f$  statt  $f()$ .

### Beispiel 6.6:

Sei  $V = \{x, y, z\}$  und  $F = \{a, b, f, g, h, l\}$  mit  $\rho(a) = \rho(b) = 0$ ,  $\rho(g) = \rho(l) = 1$ ,  $\rho(f) = 2$  und  $\rho(h) = 3$ . Dann sind  $b$ ,  $z$ ,  $f(x, y)$ ,  $g(h(z, y, a))$  und  $f(l(z), y)$  Terme bzgl. der angegebenen Alphabete.

Um den Zusammenhang mit Typ-Ausdrücken zu verdeutlichen, betrachten wir den Typ-Ausdruck

$$(char \times char) \rightarrow (int \rightarrow pointer(int \rightarrow int))$$

mit der Präfix-Notation:

$$\rightarrow (\times(char, char), \rightarrow (int, pointer(\rightarrow (int, int))))$$

Das Alphabet  $F$  der Funktionssymbole enthält  $char$ ,  $int$ ,  $pointer$  sowie  $\rightarrow$  und  $\times$  mit  $\rho(char) = \rho(int) = 0$ ,  $\rho(pointer) = 1$  und  $\rho(\rightarrow) = \rho(\times) = 2$ .

### 6.2.1 Substitution und Unifikator

Variablen können durch Terme ersetzt werden. Kommt eine Variable mehrfach vor, so soll durch eine Substitution jedes Vorkommen dieser Variablen durch den gleichen Term ersetzt werden.<sup>19</sup>

**Definition:** Eine **Substitution** ist eine Abbildung  $S$ , die jeder Variable  $x_i \in V$  einen Term  $t_i$  zuordnet. Eine Substitution wird in der Form  $\{x_1 \mapsto t_1, \dots, x_r \mapsto t_r\}$  notiert. Variablen, die nicht in der Menge vorkommen, werden auf sich selbst abgebildet.

Die Substitution  $S_{id}$  bildet jede Variable auf sich selbst ab. Offensichtlich gilt:  $S_{id} = \{\}$ .

<sup>19</sup> Das ist anders bei kontextfreien Grammatiken, wo wir gleiche Nichtterminalsymbole durch verschiedene rechte Seiten von Produktionen ersetzen dürfen.

Substitutionen werden zu Abbildungen von Termen in die Menge der Terme erweitert:

**Definition:** Ist  $S$  eine Substitution und  $t$  ein Term, so entsteht der Term  $S(t)$  aus  $t$ , indem alle Vorkommen von Variablen in  $t$  durch den durch  $S$  zugeordneten Term ersetzt werden.  $S(t)$  heißt auch **Instanz** von  $t$ .

**Beispiel 6.7:** Für  $t = f(l(z), y)$ , die Substitutionen  $S_1 = \{x \mapsto a, y \mapsto f(a, f(b, a)), z \mapsto g(x)\}$  und  $S_2 = \{x \mapsto l(y), y \mapsto b, z \mapsto f(a, x)\}$  ist

$$S_1(f(l(z), y)) = f(l(g(x)), f(a, f(b, a))) \quad \text{und} \quad S_2(f(l(z), y)) = f(l(f(a, x)), b)$$

Substitutionen können –wie alle Abbildungen– hintereinander ausgeführt werden.

Für die Substitution  $S_1 \circ S_2$  gilt:  $(S_1 \circ S_2)(t) = S_1(S_2(t))$  für alle Argumente  $t$ .

**Beispiel 6.8:** Für die Substitutionen  $S_1$  und  $S_2$  aus Beispiel 6.7 ist

$$S_1 \circ S_2 = \{x \mapsto l(f(a, f(b, a))), y \mapsto b, z \mapsto f(a, a)\} \text{ und}$$

$$S_2 \circ S_1 = \{x \mapsto a, y \mapsto f(a, f(b, a)), z \mapsto g(l(y))\}$$

Die  $\circ$  Operation ist also nicht kommutativ!

Ein Unifikator zweier Terme ist eine Substitution, die –auf beide Terme angewandt– dasselbe Ergebnis liefert:

**Definition:**

Zwei Terme  $t_1$  und  $t_2$  heißen **unifizierbar**, wenn es eine Substitution  $S$  gibt mit  $S(t_1) = S(t_2)$ .

Diese Substitution  $S$  heißt **Unifikator** von  $t_1$  und  $t_2$ .  $S(t_1)$  heißt **Unifikation** von  $t_1$  und  $t_2$ .

**Beispiel 6.9:** Sei  $t_1 = f(x, y)$  und  $t_2 = z$ . Unifikatoren für beide Terme sind dann

- $S_1 = \{x \mapsto a, y \mapsto a, z \mapsto f(a, a)\}$ , da  $S_1(t_1) = S_1(t_2) = f(a, a)$  gilt;
- $S_2 = \{y \mapsto b, z \mapsto f(x, b)\}$ , da  $S_2(t_1) = S_2(t_2) = f(x, b)$  gilt.

Wir suchen jetzt einen Unifikator, der Substitutionen „sehr zurückhaltend“ vornimmt. Er soll zwar beide Terme unifizieren, dabei aber nur die unbedingt notwendigen Substitutionen machen.

**Definition:** Ein Unifikator  $U_a$  zweier Terme  $t_1$  und  $t_2$  heißt **allgemeinster Unifikator** von  $t_1$  und  $t_2$ , wenn zu jedem Unifikator  $U$  von  $t_1$  und  $t_2$  eine Substitution  $S$  existiert, so dass  $S \circ U_a = U$  gilt, also  $U(t_1)$  eine Instanz von  $U_a(t_1)$  ist.

**Beispiel 6.10:** Für die Terme aus Beispiel 6.9 ergibt sich der allgemeinste Unifikator  $U_a$  zu  $\{z \mapsto f(x, y)\}$ . Man sieht sofort, dass man Substitutionen finden kann, die aus  $U_a(t_1) = U_a(t_2) = f(x, y)$  die Terme  $f(a, a)$  bzw.  $f(x, b)$  bilden, umgekehrt geht das aber nicht!

Wie kann man dies nun für eine Typ-Prüfung benutzen? Gegeben sind zwei Typ-Ausdrücke, die auf Typ-Äquivalenz geprüft werden sollen. Das Problem besteht also darin, für alle in beiden Ausdrücken auftretenden Typvariablen Ausdrücke zu finden, so dass nach einer Substitution der gefundenen Ausdrücke die Typ-Ausdrücke identisch sind. So kann z.B. der Typ-Ausdruck  $A = \alpha \rightarrow \alpha$  zu  $\text{int} \rightarrow \text{int}$  oder auch zu  $\text{pointer}(\beta) \rightarrow \text{pointer}(\beta)$  werden. Nun sucht man nicht irgendeine Substitution, die zwei Typ-Ausdrücke identisch macht, sondern eine möglichst allgemeine Substitution mit dieser Eigenschaft, d.h. einen allgemeinsten Unifikator.

**Bemerkung:** Wenn man von einer Unifikation zweier Typ-Ausdrücke spricht, so wird darunter meist die Unifikation mit einem *allgemeinsten* Unifikator verstanden. Der allgemeinste Unifikator ist –bis auf Variablenumbenennungen– eindeutig.

**Beispiel 6.11:** Gegeben seien die beiden Typ-Ausdrücke

$$\begin{aligned} t_1 &= ((\alpha_1 \rightarrow \alpha_2) \times \text{list}(\alpha_3)) \rightarrow \text{list}(\alpha_2) \quad \text{und} \\ t_2 &= ((\alpha_3 \rightarrow \alpha_4) \times \text{list}(\alpha_3)) \rightarrow \alpha_5 \end{aligned}$$



Der allgemeinste Unifikator  $S$  ist dann  $\{\alpha_1 \mapsto \alpha_3, \alpha_4 \mapsto \alpha_2, \alpha_5 \mapsto \text{list}(\alpha_2)\}$  mit  

$$S(t_1) = S(t_2) = ((\alpha_3 \rightarrow \alpha_2) \times \text{list}(\alpha_3)) \rightarrow \text{list}(\alpha_2)$$

Ein anderer Unifikator  $S'$  ist z.B.  $\{\alpha_2 \mapsto \alpha_1, \alpha_3 \mapsto \alpha_1, \alpha_4 \mapsto \alpha_1, \alpha_5 \mapsto \text{list}(\alpha_1)\}$  mit  

$$S'(t_1) = S'(t_2) = ((\alpha_1 \rightarrow \alpha_1) \times \text{list}(\alpha_1)) \rightarrow \text{list}(\alpha_1)$$

und natürlich ist  $S'(t_1)$  eine Instanz von  $S(t_1)$ .

### 6.2.2 Bestimmung eines allgemeinsten Unifikators

JOHN ALAN ROBINSON hat 1965 bewiesen [28], dass für zwei gegebene Terme ein allgemeinsten Unifikator existiert, falls die Terme überhaupt unifizierbar sind.

#### Algorithmus unify zur Berechnung eines allgemeinsten Unifikators:

Für zwei Terme  $t_1$  und  $t_2$  über den Alphabeten  $V$  (Variablen) und  $F$  (Funktionssymbole) liefert der Algorithmus einen allgemeinsten Unifikator, sofern diese Terme unifizierbar sind.

Es sind zwei Fälle zu unterscheiden:

**Fall 1:** Einer der beiden Terme ist eine Variable  $x$ , der andere Term sei  $t$ .

- Ist  $x = t$ , dann gebe  $S_{id}$  zurück.
- Tritt  $x$  in  $t$  auf (*occur check*), so sind  $t_1$  und  $t_2$  nicht unifizierbar. Ende.
- Sonst gib die Substitution  $\{x \mapsto t\}$  zurück.

**Fall 2:** Es sei  $t_1 = f(x_1, \dots, x_n)$  und  $t_2 = g(y_1, \dots, y_m)$ .

- Ist  $f \neq g$  oder  $n \neq m$ , so sind  $t_1$  und  $t_2$  nicht unifizierbar. Ende.
- Setze  $S = S_{id}$  und führe für alle  $k$ ,  $1 \leq k \leq n$  die folgenden Schritte aus:
  1. Berechne rekursiv einen allgemeinsten Unifikator  $S'$  von  $S(x_k)$  und  $S(y_k)$ , d.h. bestimme  $S' = \text{unify}(S(x_k), S(y_k))$ .
  2. Setze  $S = S' \circ S$ .
- Gib die Substitution  $S$  zurück.

**Bemerkung:** Der obige Algorithmus hat einen exponentiellen Zeit- und Speicherplatzbedarf. Inzwischen existieren Unifikationsalgorithmen, deren Komplexität nur noch linear ist [29].

**Beispiel 6.12:** Seien  $V$  und  $F$  wie im Beispiel 6.6. Betrachte die beiden Terme

$$\begin{aligned} t &= f(g(x), f(y, h(y, x, g(x)))) \quad \text{und} \\ u &= f(g(l(a)), f(g(z), h(g(z), l(a), z))) \end{aligned}$$

Da beide Terme keine Variablen sind, greift Fall 2 des Algorithmus. Man versucht also zunächst die Terme  $g(x)$  und  $g(l(a))$  zu unifizieren, bestimmt also  $\text{unify}(g(x), g(l(a)))$ . Dies führt auf  $\text{unify}(x, l(a))$ . Hier wenden wir Fall 1 an und bekommen die Substitution  $S' = \{x \mapsto l(a)\}$ .

Danach wird  $S$  neu bestimmt, hier auf  $S = S' \circ S_{id} = S'$  gesetzt. Nun versuchen wir, die Terme

$$\begin{aligned} S(f(y, h(y, x, g(x)))) &= f(y, h(y, l(a), g(l(a)))) \quad \text{und} \\ S(f(g(z), h(g(z), l(a), z))) &= f(g(z), h(g(z), l(a), z)) \end{aligned}$$

zu unifizieren. Dies führt zunächst auf das Teilproblem  $\text{unify}(y, g(z))$ , das (Fall 1) zur Substitution  $S' = \{y \mapsto g(z)\}$  führt.  $S$  wird dadurch neu auf  $\{x \mapsto l(a), y \mapsto g(z)\}$  gesetzt.

Das nächste Teilproblem ist  $\text{unify}(S(h(y, l(a), g(l(a))))), S(h(g(z), l(a), z)))$ , also die Bestimmung von  $\text{unify}(h(g(z), l(a), g(l(a))), h(g(z), l(a), z))$ .

Während die ersten beiden Argumente von  $h$  identisch sind, liefert das letzte Argument die Substitution  $\{z \mapsto g(l(a))\}$  zurück.

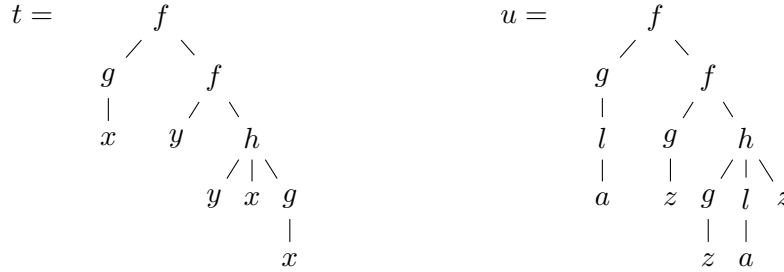
Insgesamt ergibt sich damit der allgemeinste Unifikator

$$U = \{x \mapsto l(a), y \mapsto g(g(l(a))), z \mapsto g(l(a))\}.$$

Um diesen Unifikationsprozess formaler darstellen zu können, wird die Unifikationsaufgabe in Teilaufgaben zerlegt. Man schreibt die zu unifizierenden Terme untereinander und vergleicht die Terme von links nach rechts. Stößt man beim Vergleich auf einen Unterschied, so betrachtet man die unterschiedlichen Teilterme. Lassen sich diese Teilterme unifizieren, so notiert man die gefundene Substitution und wendet diese sofort auf die Terme an und fährt mit dem Vergleich fort, bis –sofern möglich– die Terme unifiziert sind. Die bis dahin bestimmte Substitution ist der gesuchte Unifikator.

**Beispiel 6.13:** Für die Terme aus Beispiel 6.12 (hier in Präfix- und Baumdarstellung)

$$t = f(g(x), f(y, h(y, x, g(x)))) \quad \text{und} \quad u = f(g(l(a)), f(g(z), h(g(z), l(a), z)))$$



ergibt sich (zu unifizierende Teilterme sind unterstrichen):

$t = f(g(\underline{x}), f(y, h(y, x, g(x))))$	
$u = f(g(\underline{l(a)}), f(g(z), h(g(z), l(a), z)))$	$S_1 = \{x \mapsto l(a)\}$
$S_1(t) = f(g(l(a)), f(\underline{y}, h(y, l(a), g(l(a)))))$	
$S_1(u) = f(g(l(a)), f(\underline{g(z)}, h(g(z), l(a), z)))$	$S_2 = \{y \mapsto g(z)\}$
$S_2(S_1(t)) = f(g(l(a)), f(g(z), h(g(z), l(a), \underline{g(l(a))})))$	
$S_2(S_1(u)) = f(g(l(a)), f(g(z), h(g(z), l(a), \underline{z})))$	$S_3 = \{z \mapsto g(l(a))\}$
$S_3(S_2(S_1(t))) = f(g(l(a)), f(g(g(l(a))), h(g(g(l(a))), l(a), g(l(a)))))$	
$S_3(S_2(S_1(u))) = f(g(l(a)), f(g(g(l(a))), h(g(g(l(a))), l(a), g(l(a)))))$	

Der (allgemeinste) Unifikator  $U$  berechnet sich damit zu:

$$\begin{aligned}
 U &= S_3 \circ S_2 \circ S_1 = \{z \mapsto g(l(a))\} \circ \{y \mapsto g(z)\} \circ \{x \mapsto l(a)\} \\
 &= \{z \mapsto g(l(a))\} \circ \{x \mapsto l(a), y \mapsto g(z)\} \\
 &= \{x \mapsto l(a), y \mapsto g(g(l(a))), z \mapsto g(l(a))\}
 \end{aligned}$$

### 6.2.3 Typ-Bestimmung durch Unifikation

Eine interessante Anwendung erfährt dieses Konzept in der Programmiersprache **Haskell**, einer funktionalen Sprache, die polymorphe Funktionen erlaubt. Der Programmierer muss hier den Typ von Funktionen nicht deklarieren, sondern ein ausgefeiltes Typ-System berechnet den Typ eines eingegebenen Ausdrucks automatisch. Definiert man also z.B. eine Funktion **ac**

```
ac(oper,init,seq) = if null(seq) then init
                  else oper(head(seq), ac(oper,init,tail(seq)))
```

so berechnet das System den Typ als:  $ac :: ((t, t1) \rightarrow t1, t1, [t]) \rightarrow t1$

wobei dieser Typ-Ausdruck in unserer Notation so aussieht:  $(\alpha \times \beta \rightarrow \beta) \times \beta \times list(\alpha) \rightarrow \beta$

Der Typ-Ausdruck jeder Standard-Funktion ist dem Typ-System bekannt, z.B. hat die Funktion **null** den Typ-Ausdruck  $list(\alpha) \rightarrow bool$  und die Funktion **head** den Typ-Ausdruck  $list(\alpha) \rightarrow \alpha$ .

Daraus schließt das Typ-System, welcher Typ-Ausdruck der Funktion **ac** zuzuordnen ist. Bei jeder späteren Anwendung dieser Funktion kann dann der korrekte Aufruf überprüft werden.

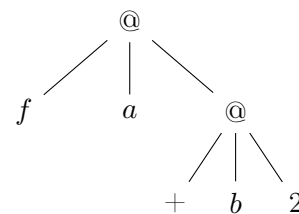
**Beispiel 6.14:**  $length(ls) = \text{if null}(ls) \text{ then } 0$   
 Typ-Inferenz für die Haskell-Funktion:  $\text{else } length(tail(ls)) + 1$

Definiert wird eine Funktion **length** mit einem Parameter **ls**, die die Länge einer Liste berechnet. Das Prädikat **null** testet, ob das Argument eine leere Liste ist. Die Funktion **tail** liefert die Ausgangsliste ohne das erste Listenelement zurück. Die Funktion **length** hat daher den Typ-Ausdruck  $list(\alpha) \rightarrow int$ .

#### Wie ermittelt das Typ-System den Typ-Ausdruck?

Zunächst wird der Ausdruck geparkt und ein entsprechender Syntaxbaum aufgebaut.

Jede **Anwendung einer Funktion** wird durch einen speziellen Knoten „@“ im Syntaxbaum dargestellt. Das erste Kind dieses Knotens stellt die Funktion, jedes weitere Kind einen Parameter der Funktion dar.



Syntaxbaum für den Ausdruck  $f(a, b+2)$

Nun beginnt die eigentliche **Typinferenz**. Den Blättern des Syntaxbaums werden Typ-Ausdrücke zugeordnet, soweit sie durch das System vorgegeben oder durch vorangegangene Definitionen bekannt sind. Einem neuen Namen wird als Typ-Ausdruck eine neue Typvariable zugeordnet.

Dann versucht das Typ-System „bottom up“, den inneren Knoten des Syntaxbaums (@-Knoten) einen Typ-Ausdruck zuzuordnen. Für die anzuwendende Funktion erhalten wir Typinformationen auf zwei Wegen:

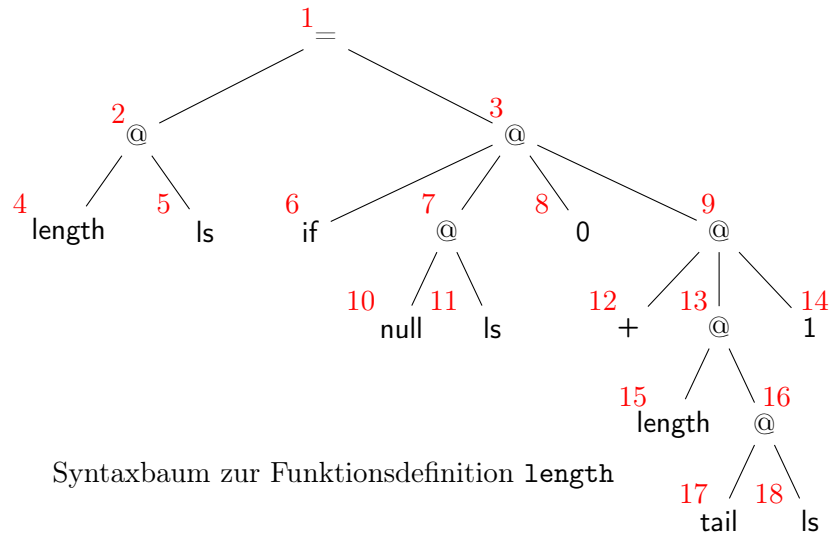
- Einerseits direkt: Der Typ-Ausdruck  $u$  des ersten Kindes (der Funktion) eines @-Knotens.
- Andererseits wird aus den Typ-Ausdrücken  $\beta_1 \dots \beta_k$  der restlichen  $k$  Kinder (der Parameter) und  $\gamma$  für das Funktionsergebnis der Typ-Ausdruck  $v = \beta_1 \times \beta_2 \dots \times \beta_k \rightarrow \gamma$  gebildet.

Nun wird versucht, die Typ-Ausdrücke  $u$  und  $v$  zu unifizieren. Die dafür notwendigen Substitutionen werden in einer Tabelle gespeichert und bei allen nachfolgenden Unifikationen für diesen Syntaxbaum berücksichtigt. Bei erfolgreicher Unifikation wird dem @-Knoten der Typ-Ausdruck für  $\gamma$  zugeordnet.

**Beispiel 6.15:** Wir betrachten den Haskell-Ausdruck von Beispiel 6.14. Wir schreiben die Typ-Information hier nicht an die Knoten des Syntaxbaumes, sondern in eine Tabelle.

Für folgende Funktionen und Konstanten ist der Typ-Ausdruck vorgegeben:

$0 : int$   
 $1 : int$   
 $+$  :  $int \times int \rightarrow int$   
 $if$  :  $bool \times \alpha \times \alpha \rightarrow \alpha$   
 $null$  :  $list(\alpha) \rightarrow bool$   
 $tail$  :  $list(\alpha) \rightarrow list(\alpha)$



Syntaxbaum zur Funktionsdefinition `length`

Nr	Sprach-Ausdruck	Typ-Ausdruck	zu unifizieren	Substitutionen
4	<code>length</code>	$\beta$	$u = \beta$	
5	<code>ls</code>	$\gamma$		
2	<code>length(ls)</code>	$\delta$	$v = \gamma \rightarrow \delta$	$\beta \mapsto \gamma \rightarrow \delta$
10	<code>null</code>	$list(\alpha_n) \rightarrow bool$	$u = list(\alpha_n) \rightarrow bool$	
11	<code>ls</code>	$\gamma$		
7	<code>null(ls)</code>	$bool$	$v = \gamma \rightarrow bool$	$\gamma \mapsto list(\alpha_n)$
17	<code>tail</code>	$list(\alpha_t) \rightarrow list(\alpha_t)$	$u = list(\alpha_t) \rightarrow list(\alpha_t)$	
18	<code>ls</code>	$\gamma = list(\alpha_n)$		
16	<code>tail(ls)</code>	$list(\alpha_t)$	$v = list(\alpha_n) \rightarrow list(\alpha_t)$	$\alpha_t \mapsto \alpha_n$
15	<code>length</code>	$\beta = list(\alpha_n) \rightarrow \delta$	$u = list(\alpha_n) \rightarrow \delta$	
16	<code>tail(ls)</code>	$list(\alpha_t) = list(\alpha_n)$		
13	<code>length(tail(ls))</code>	$\delta$	$v = list(\alpha_n) \rightarrow \delta$	—
12	<code>+</code>	$int \times int \rightarrow int$	$u = int \times int \rightarrow int$	
13	<code>length(tail(ls))</code>	$\delta$		
14	<code>1</code>	$int$		
9	<code>length(tail(ls)) + 1</code>	$int$	$v = \delta \times int \rightarrow int$	$\delta \mapsto int$
6	<code>if</code>	$bool \times \alpha_i \times \alpha_i \rightarrow \alpha_i$	$u = bool \times \alpha_i \times \alpha_i \rightarrow \alpha_i$	
7	<code>null(ls)</code>	$bool$		
8	<code>0</code>	$int$		
9	<code>length(tail(ls)) + 1</code>	$int$		
3	<code>if(...)</code>	$\alpha_i$	$v = bool \times int \times int \rightarrow \alpha_i$	$\alpha_i \mapsto int$
2	<code>length(ls)</code>	$\delta = int$		
3	<code>if(...)</code>	$\alpha_i = int$		
1	<code>=</code>	True		

Die Funktion `length` hat den Typ-Ausdruck:  $\beta = list(\alpha_n) \rightarrow int$

## 7 Einführung in die funktionale Programmierung mit Haskell

Funktionale Programmierung arbeitet mit Ausdrücken und mathematischen Funktionen. Ihre Einfachheit ermöglicht ein besseres Verständnis eines Programms und der Programmiersprache. Funktionale Programme werden meist schneller entwickelt, sind kürzer sowie leichter zu warten.

### 7.1 Konzepte funktionaler Programmiersprachen

#### 7.1.1 Funktionen und Variablen – keine Nebenwirkungen

Mathematische Funktionen liefern für gleiche Argumente –unabhängig vom Zeitpunkt der Auswertung– stets den gleichen Funktionswert (**referentielle Transparenz**), haben also keine Nebenwirkungen. Das vereinfacht viele Programmtransformationen, z.B.  $f(x) + f(x) = 2 \cdot f(x)$ , und erleichtert die Überprüfung der Korrektheit eines Programms.

Die Bedeutung eines (rein) funktionalen Programms hängt daher nicht von der Reihenfolge ihrer Definitionen ab. Funktionale Sprachen können daher eine beliebige Reihenfolge erlauben.

Der Begriff **Variable** wird hier mathematisch benutzt, also anders als bei imperativen Sprachen, wo eine Variable für einen Behälter steht, der durch eine Wertzuweisung neue Werte aufnimmt.

Funktionale Programme lassen sich –da ohne Nebenwirkungen– einfach **parallelisieren**: Alle Argumente einer Funktion können parallel ausgewertet werden. Bei einer Bedarfsauswertung kann sogar die Berechnung der Argumente einer Funktion parallel zur Berechnung des Funktionsrumpfes erfolgen. Die Möglichkeiten zur Parallelisierung sind also schnell erkannt, die anderen Probleme einer effizienten parallelen Implementierung (z.B. gute Granularität, Lokalität, Prozessorauslastung) aber ähnlich schwierig wie bei imperativen Sprachen.

#### 7.1.2 Ändern von Datenstrukturen

Funktionale Programmiersprachen betrachten eine **Datenstruktur als Ganzes**. Imperative Sprachen erlauben **selektive Änderungen**, z.B. das Überschreiben einer Feldkomponente oder das Umsetzen eines Zeigers:  $A[i] := x$  oder  $p^{\wedge}.next := q$ . Dies begünstigt Fehler wie das vorzeitige Überschreiben von Feldelementen oder das Kappen noch benötigter Zeiger.

In funktionalen Sprachen wird dagegen eine komplett neue Datenstruktur gebildet:

$\text{let } A' = \text{update}(A, i, x)$  oder  $\text{let } L' = q:L$

Der hohe Speicherverbrauch kann durch automatische Speicherbereinigung in Grenzen gehalten werden, aber der Zeitaufwand für das Kopieren bleibt – als Preis für die geringere Fehleranfälligkeit. Für eine Optimierung ist es daher wichtig zu erkennen, ob eine Datenstruktur nur einmal benötigt wird und deshalb selektiv geändert werden darf, oder ob sie mehrfach benötigt wird und daher unverändert erhalten bleiben muss.

#### 7.1.3 Rekursive Datenstrukturen

Funktionale Programmiersprachen bieten **rekursiv definierte Typen** an und verweisen damit **Zeiger** auf die Ebene der Implementierung. Während in imperativen Sprachen die Datenstruktur **Feld** häufig ist, benutzt man in funktionalen Sprachen oft den rekursiven Datentyp **Liste**.

Listen haben zwei Datenkonstruktoren: `[]` erzeugt die leere Liste und `:` verlängert eine Liste am Anfang um ein Element. `[1,2,3]` beschreibt dann kurz die Liste `1:(2:(3:[]))`.

### 7.1.4 Automatische Speicherverwaltung

In funktionalen Sprachen muss sich der Programmierer nicht um die Anforderung und Freigabe von Speicherplatz kümmern; die Speicherverwaltung wird durch die Implementierung der Sprache übernommen. Da für jede Anwendung eines  $k$ -stelligen Konstruktors mit  $k > 0$  und jeden verzögerten Ausdruck Speicherplatz angefordert wird, ist eine gute Speicherverwaltung der Halde Voraussetzung für eine effiziente funktionale Sprache.

### 7.1.5 Muster in Funktionsdefinitionen

Der **Mustervergleich** (*pattern matching*) ist eine kompakte, gut lesbare Variante der Fallunterscheidung. Dazu werden die formalen Parameter durch Muster ersetzt. Ein **Muster** (*pattern*) ist ein Ausdruck mit Variablen und Datenkonstruktoren. Die Werte der aktuellen Parameter werden dann mit diesen Mustern verglichen. Passt der Wert jedes aktuellen Parameters auf sein Muster, so werden die Variablen im Muster an die entsprechenden Komponenten des Werts gebunden.

**Beispiel 7.1:** Wertet man den Ausdruck `f [2, 3]` aus bzgl. folgender einstelliger Funktion `f`

```
f [1, x, y] = x + y
f [2, z]    = z + 7 ,
```

so passt der Wert `[2, 3]` auf das Muster `[2, z]` der zweiten Alternative; daher wird die Variable `z` an die Komponente 3 gebunden und das Funktionsergebnis ist 10.

### 7.1.6 Funktionale: Funktionen höherer Stufe

Viele funktionale Programmiersprachen erlauben **Funktionale** (Funktionen höherer Stufe, *higher-order function*). Sprachen werden danach unterteilt in **Sprachen erster Stufe** und **Sprachen höherer Stufe**. Sprachen höherer Stufe betrachten Funktionen als Werte und damit als **gleichberechtigte Datenobjekte** (*first class citizens*). Dann können Funktionen auch als Ergebnis eines Ausdrucks oder einer Funktion, als Parameter oder als Teil einer Datenstruktur auftreten.

Z.B. kann man eine Funktion definieren, die zwei Funktionen  $f$ ,  $g$  als Parameter hat und die Komposition beider Funktionen,  $(f \circ g)$ , als Wert zurückgibt.

Mit höheren Funktionen (wie `map`, `filter`, `zip`) kann man häufige Rekursionsmuster beschreiben. Daher werden diese –z.T. vordefinierten– Funktionen oft (wieder)verwendet, was zu besserer Lesbarkeit von Programmen und guter Modularisierung führt.

Als Parameter eines Funktionals müssen Funktionen nicht unbedingt einen Namen bekommen. Man erzeugt dann eine anonyme Funktion durch einen Lambda-Ausdruck.

### 7.1.7 Bedarfsauswertung

Auch nach der **Auswertungsstrategie** können funktionale Sprachen eingeteilt werden:

- Bei **striktter Auswertung** (*strict evaluation, call by value*) wird jedes Argument einer Funktion ausgewertet, bevor die Funktion aufgerufen wird – wie oft in imperativen Sprachen.
- Bei **Bedarfsauswertung** (*lazy evaluation, call by need*) wird jedes Argument der Funktion unausgewertet als Ausdruck übergeben und erst berechnet, wenn der Wert benötigt wird.

Strikte Auswertung braucht weniger Speicherplatz und ist leichter mit imperativen Erweiterungen kombinierbar. Die Bedarfsauswertung ermöglicht **zyklische** oder **unendliche Datenstrukturen**.

### 7.1.8 Statische Typisierung

Schließlich unterscheidet man funktionale Sprachen nach ihrer **Typisierung**. Zwar sind alle verbreiteten funktionalen Sprachen stark typisiert, d.h. alle Typfehler werden erkannt. Einige Sprachen überprüfen die Typen aber erst zur Laufzeit (**dynamische Typprüfung**); andere Sprachen erledigen die vollständige Typprüfung bereits zur Übersetzungszeit (**statische Typprüfung**).

Typen werden in funktionalen Programmiersprachen meist als Terme dargestellt. Basistypen sind atomare Terme, wie `Int`, `Bool`. Strukturierte Typen werden durch Typterme beschrieben, z.B. eine Liste von `Int`-Werten durch `[Int]`. Der Typ eines Ausdrucks kann auch durch ein Typschema mit Typvariablen gegeben sein. Eine **Typvariable** kann bei verschiedenen Benutzungen mit unterschiedlichen Typen instantiiert werden (**parametrische Polymorphie**). Ein Typ(schema) kann durch Deklaration angegeben oder durch **Typinferenz** aus den Anwendungsstellen der Funktion abgeleitet werden.

In Haskell brauchen Typen (wie in ML oder OCaml) nicht deklariert zu werden. Stattdessen schließt das System aus dem Gebrauch der Variablen auf mögliche Typen und prüft, ob alle Vorkommen dieser Variablen konsistent sind.

Basierend auf den Kriterien Funktionale, Auswertungsstrategie und Typisierung erhalten wir folgende Klassifikation funktionaler Sprachen:

		Strikte Auswertung	Bedarfsauswertung
Dynamische Typprüfung	Erste Stufe	Mathematica, Erlang	
	Höhere Stufe	Lisp, Scheme, APL, FP	SASL
Statische Typprüfung	Erste Stufe	SISAL	Id
	Höhere Stufe	ML, SML, Caml, Hope	Miranda, Haskell

Abb. 7.1: Klassifikation funktionaler Programmiersprachen

Einige funktionale Sprachen orientieren sich an bestimmten Anwendungen: ML, SML und Hope werden für Theorembeweiser eingesetzt, Erlang für verteilte Telekommunikationsanwendungen (Ericsson); SISAL und Id für paralleles Hochleistungsrechnen. Mathematica ist Teil eines Systems für symbolische Algebra. XSLT beschreibt und transformiert XML-Daten.

## 7.2 Benutzung von Haskell

Beispielsprache hier ist Haskell (<http://www.haskell.org/onlinereport/haskell2010>).

Haskell ist eine funktionale Sprache mit einem ausgereiften Typ-System und Bedarfsauswertung.

Der **Glasgow Haskell Compiler** (GHC) kann von <http://www.haskell.org/platform> heruntergeladen werden. Er enthält u.a. den interaktiven Interpretierer `GHCi`<sup>20</sup>.

Wichtige Kommandos im `GHCi` sind (oft reicht das erste Zeichen nach dem `:`):

```
<expression>    wertet einen (einzeiligen) Ausdruck aus
:?              liefert Liste aller Kommandos
:info <name>     informiert über das genannte Objekt
:load <filename> lädt den angegebenen Modul, Haskell-Dateien enden auf .hs
:reload         lädt den (geänderten) Modul erneut
:quit           beendet GHCi
```

<sup>20</sup> Achtung: Definitionen muss im `GHCi` ein `let` vorangehen (im Gegensatz zum `GHC`).

<http://book.realworldhaskell.org/read/> oder <http://learnyouahaskell.com/chapters/> liefern ausführliche Anleitungen zum Erlernen der Sprache.

Oder nutzen Sie <http://tryhaskell.org> ! Beim Verstehen von Fehlermeldungen hilft [42].

Haskell unterscheidet Groß- und Kleinschreibung;

insbesondere beginnen Namen für Typen und Module mit Großbuchstaben;

Namen für Werte (auch Funktionen) und Typvariablen mit Kleinbuchstaben (oder `_`).

**Schlüsselwörter** dürfen nicht für Namen benutzt werden; in Haskell sind dies:

`case`, `class`, `data`, `default`, `deriving`, `do`, `else`, `foreign`, `if`, `import`, `in`, `infix`, `infixl`, `infixr`, `instance`, `let`, `module`, `newtype`, `of`, `then`, `type`, `where`, `_`

Ein **Kommentar** beginnt mit `--` und endet mit der Zeile.

Oder er beginnt mit `{-` und endet mit `-}`; diese Version ist schachtelbar.

**Leerzeichen** am Zeilenbeginn haben in Haskell eine Bedeutung!

In Haskell gilt die **Abseitsregel**, wenn nach einem der Schlüsselwörter `where`, `let`, `do` oder `of` keine öffnende Klammer `{` folgt. Dann wird diese eingefügt sowie bei einer kürzeren Einrückung der nächsten Zeile eine schließende Klammer `}` bzw. bei gleicher Einrückung ein Semikolon. Bei einer längeren Einrückung wird nichts eingefügt.<sup>21</sup>

### 7.3 Datentypen in Haskell

Zu den **Werten** einer Programmiersprache gehören Wahrheitswerte, Zahlen und Zeichen.

Diese Werte können auf Gleichheit (`==`) oder Ungleichheit (`/=`) geprüft werden.

Die **Wahrheitswerte** `True` und `False` bilden den Datentyp `Bool` mit den Operationen `not` (Negation), `&&` (Konjunktion, *and*), `||` (Disjunktion, *or*).

**Zahlen** gibt es ganzzahlig oder in Gleitpunktdarstellung.

Für ganze Zahlen gibt es die Datentypen `Int` und `Integer` mit den Operationen `+`, `-`, `*`, `div` und `mod`. Eine Zahl des Datentyps `Int` wird durch 32 Bit beschrieben und liegt daher im Intervall `[-2.147.483.648...2.147.483.647]`; der Datentyp `Integer` hat keine Größenbeschränkung.

Für Gleitpunktzahlen gibt es die Datentypen `Float` (32 Bit) und `Double` (64 Bit). Der Operator `/` erzeugt einen Wert vom Typ `Float`.

Für **Zeichen** gibt es den Datentyp `Char`. Alle Symbole von Unicode können bearbeitet werden. Zeichen werden in einfachen Anführungsstrichen notiert, z.B. `'a'`.

Neue Typen werden in Haskell nur auf der obersten Programmebene definiert.

#### 7.3.1 Tupel (kartesisches Produkt)

Mehrere Werte können zu einem **Tupel** zusammengefasst werden, z.B. `(False, 2013)`. Runde Klammern machen aus mehreren –durch Komma getrennten– Werten *ein* Tupel. Ein Tupel hat eine feste Stelligkeit. Ein Tupel kann Werte unterschiedlicher Typen enthalten.

Auf die Komponenten eines Paares kann mit den Funktionen `fst` und `snd` zugegriffen werden.

**Beispiel 7.2:** Ein Punkt in der Ebene wird durch zwei Koordinaten beschrieben.

Für den entsprechenden Typ (`Double, Double`) kann man ein prägnantes Synonym einführen:

```
type Punkt = (Double, Double)
```

Es gibt auch ein null-stelliges Tupel: `()`

Ein-stellige Tupel werden in Haskell mit ihrer Komponente identifiziert.

<sup>21</sup> Vorsicht mit Tabulatoren! Einige Editoren wandeln Tabulatoren in Leerzeichen um.



### 7.3.2 Algebraische Datentypen: Aufzählungstypen und direkte Summe

Ein neuer **algebraischer Datentyp** wird durch Aufzählung der Werte definiert:

```
data Color = Red | Green | Blue
```

Der **Typ-Konstruktor** `Color` erzeugt einen Typ mit allen Werten, die durch Daten-Konstruktoren auf der rechten Seite beschrieben werden. Jedem **Daten-Konstruktor** folgen die Typen der Werte der Komponenten. Alle Konstrukturen beginnen mit Großbuchstaben. Der Daten-Konstruktor `Red` hat keine Parameter, ist also eine Konstante.

**Beispiel 7.3:** Ein Datentyp für Kreise (beschrieben durch Mittelpunkt und Radius) und Rechtecke (beschrieben durch zwei Eckpunkte, die auf einer Diagonalen liegen):

```
data Geometrie = Kreis Punkt Double | Rechteck Punkt Punkt
```

Die Daten-Konstrukturen `Kreis` und `Rechteck` haben je zwei Parameter.

Auch rekursive Datentypen sind möglich:

```
data IntTree = Empty | Branch IntTree Int IntTree
```

Hier werden binäre Bäume mit ganzzahligen Knoten definiert als Typ `IntTree` mit den Werten `Empty` und `(Branch l z r)` für beliebige ganze Zahlen `z` und Werte `l` und `r` vom Typ `IntTree`.

### 7.3.3 Listen

Mehrere Daten können in **Haskell** zusammengefasst werden.

Eine **Liste** hat keine feste Anzahl von Elementen; sie kann verlängert oder verkürzt werden.

Haskell kennt nur **homogene Listen**, d.h. alle Elemente einer Liste haben den gleichen Typ.

```
[1,2,3]           -- eine Liste mit drei Elementen
[ ]               -- die leere Liste mit null Elementen
```

Das Einfügen eines Elements am Anfang einer Liste erledigt der `:`-Operator (*cons* für *construct*). Der `:`-Operator ist rechtsassoziativ, also `1:2:3:[ ] = 1:(2:(3:[ ]))`.

Die Liste `[1,2,3]` hat daher mehrere Darstellungen: `[1,2,3]` `1:[2,3]` `1:2:[3]` `1:2:3:[ ]`

Listen haben also die beiden Daten-Konstrukturen `[ ]` und `:`.

Umgekehrt kann man mit dem `:`-Operator eine Liste zerlegen in einen **Kopf** (*head*, das erste Element der Liste) und **Rumpf** (*tail*, die Liste mit allen restlichen Elementen).

Vordefinierte Operationen auf Listen sind z.B.:

- **head** `x` liefert das erste Element einer Liste `x`
- **tail** `x` liefert den Rest der Liste `x`, ohne das erste Element
- **last** `x` liefert das letzte Element einer Liste `x`
- **init** `x` liefert den Anfang der Liste `x`, ohne das letzte Element
- `x ++ y` verbindet zwei Listen `x` und `y` gleichen Typs zu einer Liste (**Konkatenation**).

Das  $n$ -te Element einer Liste `x` erhält man mit `x !! n`.

Achtung: Die Indexzählung beginnt bei Null!

`[1,2,3,4] !! 2` liefert also 3.

Der Typ einer Liste mit Elementen von Typ `a` wird mit `[a]` bezeichnet.

Für **Listen von Zeichen** gibt es eine abkürzende Schreibweise:

```
['H','a','s','k','e','l','l']      schreibt man kurz      "Haskell" .
```

Für den zugehörigen Typ `[Char]` ist der Name `String` vordefiniert.

Der leere String `""` ist ein Synonym für eine leere Liste `[ ]` von Zeichen.

## 7.4 Fallunterscheidung in Haskell

Verschiedene Fälle können mit dem `case`-Konstrukt unterschieden werden. Alle Ergebnisse einer Fallunterscheidung müssen vom gleichen Typ sein!

**Beispiel 7.4:** Das Vorzeichen einer Zahl `x` wird mit nebenstehender Fallunterscheidung ermittelt:

```
case x of
  x>0    -> 1
  x==0   -> 0
  x<0    -> -1
```

Das traditionelle  
`if e1 then e2 else e3`  
ist dann nur eine Abkürzung für:

```
case e1 of
  True   -> e2
  False  -> e3
```

In funktionalen Sprachen gibt es das `if-then-else` nur für *Ausdrücke*, d.h. hinter `then` bzw. `else` steht keine Anweisung, sondern ein Ausdruck, der einen Wert liefert. Der `else`-Teil kann nicht weggelassen werden! Ein `if-then-else`-Ausdruck kann Teil eines größeren Ausdrucks sein.

## 7.5 Funktionen in Haskell

Eine Funktion in Haskell besteht aus Definitionen und einem Ausdruck, dessen Wert als Funktionsergebnis zurückgeliefert wird. (Wir schreiben Funktionsdefinitionen in eine Datei, starten GHCi aus diesem Verzeichnis (oder ändern den Suchpfad mit `:cd`) und laden dann diese Datei.)

Werden **Parameter** in imperativen Programmiersprachen durch Komma getrennt und mit runden Klammern eingeschlossen<sup>22</sup>, z.B. `max(2,1)`, so werden sie in funktionalen Programmiersprachen nur durch Zwischenraum getrennt hinter den Funktionsnamen geschrieben: `max 2 1`.

Klammern werden nötig, wenn ein Parameter erst berechnet werden muss: `max (min 2 3) 1`.

**Beispiel 7.5:** Die Funktionsanwendung hat die höchste Priorität:

```
fac n = if n==0 then 1      > fac 5      > fac (5-1)      > fac 5-1
        else n * fac (n-1)  120          24              119
```

Eine Funktion wird durch (eine oder mehrere) Gleichungen definiert.

Nach dem Funktionsnamen folgen –durch Zwischenraum getrennt– die Parameter und das Gleichheitszeichen, danach die Berechnungsvorschrift. Ist der Parameter vom Typ `t` und die Berechnungsvorschrift vom Typ `u`, so hat die Funktion den Typ `t -> u`.

**Beispiel 7.6:** Definition einer Funktion `dec` (lies = als „ist definiert als“): `dec x = x-1`

Funktionen ohne Parameter definieren **Konstanten**: `e = 2.71828`

### 7.5.1 Mustervergleich und Wächter in Haskell

Eine Funktionsdefinition kann übersichtlich in mehrere definierende Gleichungen zerlegt werden. Dabei kann eine Fallunterscheidung für die Parameter auf der *linken* Seite der Definition vorgenommen werden. Bei einer Funktionsanwendung wird die erste<sup>23</sup> Definition verwendet, die für das aktuelle Argument passt (**Mustervergleich**, *pattern matching*)<sup>24</sup>.

**Beispiel 7.7:** Die boolesche Funktion `xor` (*exclusive or*) ist definiert als `xor x y = x /= y`

<sup>22</sup> Man kann funktional alle Parameter genauso schreiben, bekommt dann aber *einen* Tupel-Parameter.

<sup>23</sup> Reihenfolgeabhängig wird die erste passende Definition (*first fit*) gewählt, nicht eine speziellste (*best fit*).

<sup>24</sup> Ein Mustervergleich erlaubt –im Gegensatz zur Unifikation– keine freien Variablen im zweiten Term.

Mit Mustervergleich beschreiben wir explizit die Wahrheitstabelle in mehreren Gleichungen:

```
xor True  True  = False
xor True  False = True
xor False True  = True
xor False False = False
```

Ein Parameter, der zur Berechnung einer Funktion nicht benötigt wird, bekommt keinen Namen (**anonymer Parameter**) und man markiert seine Stelle mit dem Symbol `_`.

Mit anonymen Parametern können wir die restlichen Fälle kürzer beschreiben:

```
xor True  False = True
xor False True  = True
xor _      _      = False
```

Für eine Fallunterscheidung innerhalb der Funktionsdefinition kann man **Wächter** (*guard*) benutzen, die mit `|` beginnen. **otherwise** beschreibt den Restfall (als syntaktischer Zucker für `True`).

```
xor x y  -- kein = vor dem 1. Wächter!
| x && not y = True
| not x && y = True
| otherwise = False
```

Die **if-then-else**-Fallunterscheidung für bedingte Ausdrücke ist nicht so übersichtlich:

```
xor x y =
  if x && not y then True
  else if not x && y then True
  else False
```

Bei Listenoperationen, z.B. um die Länge einer Liste zu bestimmen, zeigen sich die Vorteile rekursiver Definitionen besonders gut.

```
length [ ]      = 0
length (_,xs) = 1 + length xs

length :: [a] -> Int
length [3..6]  = 4
```

**Beispiel 7.8:** Die Funktion `zip` macht aus zwei Listen eine einzige, indem sie aus den beiden Listenelementen auf gleicher Position ein Paar macht und partnerlose Elemente ignoriert.

```
zip (x:xs) (y:ys) = (x,y) : zip xs ys
zip _ _          = [ ]
```

**Beispiel 7.9:** Strukturelle Mustervergleiche, die die leere Liste ausschließen: `head` und `tail` sind –nicht für leere Listen– definiert als

```
head (x:_) = x
tail (_,xs) = xs
```

**Beispiel 7.10:** Summe der Knotenmarkierungen eines Baums. Wir unterscheiden, ob der Wert des Parameters `tree` durch den Datenkonstruktor `Empty` oder `Branch` erzeugt wurde.

```
sum tree = case tree of
  Empty      -> 0
| Branch l z r -> z + sum l + sum r

sum Empty      = 0
sum (Branch l z r) = z + sum l + sum r
```

Übersichtlicher ist eine Definition mit zwei Gleichungen. Klammern, weil *ein* Parameter!

Haskell erlaubt nur **lineare Muster**, d.h. jede Variable darf nur einmal im Muster vorkommen.

### 7.5.2 Präfix- und Infixnotation bei Funktionen

Ein Funktionsname, der nur aus Sonderzeichen besteht, heißt **Operator**. Operatoren werden standardmäßig in Infixnotation verwendet, andere Funktionen in Präfixnotation.

- Um einen Operator in Präfixnotation verwenden zu können, wird das Operationszeichen in runde Klammern eingeschlossen: aus `3+4` wird `(+) 3 4`.
- Um eine binäre Funktion in Infixnotation zu verwenden, schließt man den Funktionsnamen mit accent grave (*backtick*) ein: aus `xor a b` wird `a `xor` b`.

## 7.6 Typvariablen und Typklassen

Den Typ eines Namens oder die **Signatur** einer Funktion (also die Typen der Parameter und des Ergebnisses) kann **Haskell** selbst ermitteln; sie kann mit `:t` bzw. `:type` abgefragt werden.

Auf die Frage `:type dec` hätten wir folgende Signatur erwartet: `dec :: Int -> Int`  
 Stattdessen erhalten wir (lies `::` als „hat Typ“): `dec :: Num a => a -> a`  
 Dabei ist `a` eine Typvariable, die hier an die Typklasse `Num` gebunden ist.<sup>25</sup>

Eine **Typvariable** steht für einen beliebigen Typ; allerdings müssen alle Vorkommen einer Typvariablen durch denselben Typ ersetzt werden. Bei der Funktion `dec` muss also der Typ des Parameters mit dem Typ des Ergebnisses übereinstimmen. Typvariablen beginnen mit Kleinbuchstaben und sind in **Haskell** stets implizit allquantifiziert.

Eine **Typklasse** beschreibt eine Menge von Datentypen; zur Klasse `Num` gehören die numerischen Typen `Int`, `Integer`, `Float` und `Double`. Für jeden Datentyp dieser Typklasse ist die Subtraktion definiert, so dass mehrere Signaturen möglich sind: Die Funktion ist **überlagert**.

- Die Typklasse `Show` enthält die Typen, deren Elemente anzeigbar sind.
- Die Typklasse `Eq` enthält alle Typen, deren Elemente man auf (Un-)Gleichheit testen kann.
- Die Typklasse `Ord` enthält die Typen, deren Elemente zusätzlich total geordnet sind (`<`).

Um den Knotentyp für binäre Bäume frei wählen zu können, definieren wir einen **polymorphen Typ** mit einer Typvariable `a`:

```
data Tree a = Empty | Branch (Tree a) a (Tree a)
```

Dann ist `Branch (Branch Empty 5 Empty) 2 Empty` ein Wert des Typs `Tree Int`,  
 aber `Branch (Branch Empty False Empty) 2 Empty` ist nicht vom Typ `Tree`,  
 da ein Knoten vom Typ `Int` ist, ein anderer vom Typ `Bool`.

Hier sind Typen über gemeinsame Typvariablen gekoppelt (**parametrische Polymorphie**).

## 7.7 Gemeinsame Teilausdrücke: Lokale Definitionen in Haskell

Benötigt eine Funktion eine Hilfsfunktion oder gemeinsame Teilausdrücke, die nur in dieser Funktion Verwendung finden, können wir diese hinter dem Schlüsselwort **where** lokal definieren.

**Beispiel 7.11:** Eine quadratische Gleichung  $ax^2 + bx + c = 0$  hat keine, eine oder zwei Lösungen. Da der **then** und **else**-Teil vom gleichen Typ sein müssen, wird eine *Liste* der Lösungen erzeugt:

```
quadeq a b c =
  if diskrim == 0.0 then [middle]
  else if diskrim > 0.0 then [middle+radix, middle-radix]
  else [ ]
  where diskrim = b*b - 4.0*a*c
        radix   = (sqrt diskrim)/(2.0*a)
        middle  = -b/(2.0*a)

> quadeq 1.0 4.0 4.0      > quadeq 1.0 (-6.0) 8.0      > quadeq 1.0 0.0 4.0
[-2.0]                  [4.0, 2.0]                  [ ]
```

Eine andere Möglichkeit ist das **let-in**-Konstrukt. Nach dem **let** werden Namen für z.B. lokale Funktionen oder gemeinsame Teilausdrücke definiert, die dann im Ausdruck nach dem Schlüsselwort **in** verwendet werden können. Das **let-in**-Konstrukt ist ein Ausdruck: es liefert einen Wert.

<sup>25</sup> Die Idee ist, jedem Ausdruck seinen allgemeinsten polymorphen Typ zuzuordnen ([40], [41]).

```
quadeq' a b c =
  let diskrim = b*b - 4.0*a*c
      radix   = (sqrt diskrim)/(2.0*a)
      middle  = -b/(2.0*a)
  in  if diskrim == 0.0 then [middle]
      else if diskrim > 0.0 then [middle+radix, middle-radix]
      else [ ]
```

Die Definitionen werden nicht in der notierten Reihenfolge ausgeführt, sondern bei Bedarf. Die Interpretation beginnt hier mit dem `if`.

Die **where**-Klausel steht am Ende einer Funktionsdefinition und gilt für die gesamte Funktionsdefinition (und alle ihre Wächter). Die **let**-Klausel gilt nur für den folgenden **in**-Ausdruck.

## 7.8 Die implizite Bildung von Listen

### 7.8.1 Listenbildung durch charakterisierende Eigenschaften

Listen können kompakt durch charakterisierende Eigenschaften (*list comprehension*) beschrieben werden. Die Syntax hat Analogien zur mathematischen Beschreibung der Bildung von Mengen.

Für die ersten geraden Quadratzahlen  $\{x^2 \mid x \in \{1, 2, 3, 4, 5\} \wedge x \text{ ist eine gerade Zahl}\}$  schreibt man in Haskell:

```
[x*x | x <- [1,2,3,4,5], mod x 2 == 0]
```

Auf einen Ausdruck folgen **Generatoren** für „Grundmengenlisten“ und Bedingungen als **Filter**:

```
[x*y | x <- [2,5,10], y <- [8,10,11], x*y > 50, x*y < 100] liefert [55,80]
```

Achtung: Die Reihenfolge der Generatoren spielt eine Rolle:

```
[(a,b) | a<-[1..3], b<-[1..2]] ergibt [(1,1), (1,2), (2,1), (2,2), (3,1), (3,2)],
[(a,b) | b<-[1..2], a<-[1..3]] liefert [(1,1), (2,1), (3,1), (1,2), (2,2), (3,2)].
```

**Beispiel 7.12:** Quicksort lässt damit kurz und übersichtlich formulieren:

```
quicksort([ ]) = [ ]
quicksort(a:x) = quicksort([b | b<-x, b<=a]) ++ [a] ++ quicksort([b | b<-x, b>a])
```

Die leere Liste bleibt unverändert; das erste Element `a` einer nicht-leeren Liste dient als Pivot-Element. Die restlichen Elemente aus `x` werden in zwei Listen aufgeteilt: alle Elemente  $\leq a$  kommen in die vordere Liste, alle Elemente  $> a$  in die hintere Liste. Diese *kürzeren* Listen werden ihrerseits sortiert und schließlich die sortierten Listen konkateniert.

### 7.8.2 Listenbildung durch arithmetische Folgen

Eine Liste `[a, b .. c]` lässt sich festlegen durch den **Startwert** `a`, die **Schrittweite** `b-a` und die **Obergrenze** `c`. Fehlt `b`, so wird die Schrittweite 1 benutzt.

Die Obergrenze ist nicht unbedingt das letzte Element der Liste: Für natürliche Zahlen kommen alle Elemente  $x \leq c$  in der Folge vor. Warnung: Für Gleitkommazahlen gilt:  $x \leq c + \frac{b-a}{2}$  ☹.

```
> [2,4..8]    > [2,4..9]    > [0,-1..-4]    > [4.1..6.5]    > [4.1..6.7]
[2,4,6,8]     [2,4,6,8]     [0,-1,-2,-3,-4]  [4.1,5.1,6.1]  [4.1,5.1,6.1,7.1]
```

**Beispiel 7.13:** Pythagoräische Tripel enthalten natürliche Zahlen  $x < y < z$  mit  $x^2 + y^2 = z^2$ .

```
pT n = [(x,y,z) | x <- [1..n-2], y <- [x+1..n-1], z <- [y+1..n], x*x+y*y==z*z]
```

Ohne Obergrenze wird eine **unendlich lange Liste** (*stream*) definiert. Durch die Bedarfsauswertung berechnet Haskell stets nur den benötigten Teil der Liste.

**Beispiel 7.14:** Eine Liste aller Quadratzahlen erhält man durch: `[n*n | n <- [0..] ]`

**Beispiel 7.15:** Ermittlung der Liste aller Primzahlen, angelehnt an das „Sieb des Eratosthenes“:

```
primzahlen = sieb [2..]
```

```
where sieb (p:xs) = p : sieb [n | n <- xs, n `mod` p > 0]
```

Ausgehend von einer Liste der natürlichen Zahlen ab 2 kommt jeweils die erste Zahl  $p$  der Liste als Primzahl in die Ergebnisliste und alle Vielfachen von  $p$  werden aus der Liste entfernt.

Die Funktion `take` liefert die ersten  $n$  Elemente einer Liste, `drop` die restlichen Elemente:

```
take _ [] = []           drop 0 xs = xs
take 0 _ = []           drop _ [] = []
take n (x:xs) = x:take (n-1) xs    drop n (_:xs) = drop (n-1) xs
```

Die ersten 10 Primzahlen erhält man also durch

```
> take 10 primzahlen
[2,3,5,7,11,13,17,19,23,29]
```

## 7.9 Funktionale

In Funktionalen kommen Funktionen als Parameter oder als Ergebnis vor.

Viele Algorithmen der Funktionalen Programmierung basieren auf dem **Map-Filter-Reduce**-Prinzip. In Haskell sind entsprechende Funktionale vordefiniert.

Das Funktional `map` wendet die Funktion  $f$  auf jedes Element einer Liste an:

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

`map f [x1, x2, ..., xn]` liefert also `[f x1, f x2, ..., f xn]`.

Das Funktional `filter` belässt nur die Elemente in einer Liste, die das Prädikat  $p$  erfüllen:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) | p x = x : rest
                  | otherwise = rest
                  where rest = filter p xs
```

Das Funktional `reduce`, das alle Elemente einer Liste mit Hilfe einer binären Funktion  $f$  zusammenfasst, gibt es in verschiedenen Versionen.

Das Funktional `foldr1` wertet die Listenelemente von rechts nach links aus:

```
foldr1 :: ((a, a) -> a) -> [a] -> a
foldr1 f [x] = x
foldr1 f [x:xs] = f x (foldr1 f xs)
```

Beispiele für **Aggregationsfunktionen**  $f$  sind Summe, Produkt oder Maximum der Elemente.

```
foldr1 (+) [a,b,c,d,e] = a + (b + (c + (d + e)))
```

Die Reduce-Operation ist für die leere Liste zunächst nicht definiert. Meist nimmt man das neutrale Element der Operation  $f$  als Ergebnis der Operation `foldr1 f []`.

Die Auswertung von links nach rechts erledigt ein Funktional `foldl1`.

```
foldl1 (-) [a,b,c,d,e] = (((a - b) - c) - d) - e
```

### 7.9.1 Currying und partielle Applikation

Um die Anzahl der Klammern in einem Ausdruck zu reduzieren, ersetzt man oft ein *Tupel* von Argumenten durch eine *Folge* von Argumenten. Denn man kann jeder Funktion des Typs

$f : (X \times Y) \rightarrow Z$  eineindeutig eine entsprechende Funktion des Typs  $\text{curry}(f) : X \rightarrow (Y \rightarrow Z)$  zuordnen. Dann braucht man nur noch Funktionen mit *einem* (einfachen) Parameter betrachten und kann bekannte Theorien wie den  $\lambda$ -Kalkül verwenden. Diese Idee von MOSES SCHÖNFINKEL wurde von HASKELL B. CURRY populär gemacht und heißt nach ihm *currying*.

Betrachten wir die Additionsfunktion:

- Natürlich können wir beide Parameter zu *einem* Tupel zusammenfassen:  
 $\text{add } (x, y) = x + y$  ist vom Typ  $\text{add} :: (\text{Int}, \text{Int}) \rightarrow \text{Int}$
- Die übliche Variante in funktionalen Sprachen ist aber (ohne Klammern und Komma!):  
 $\text{add } x \ y = x + y$  ist vom Typ  $\text{add} :: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})$

Eine Funktionsanwendung  $\text{add } e1 \ e2$  ist linksassoziativ, also äquivalent zu  $(\text{add } e1) \ e2$ .

Die Anwendung der Funktion **add** auf *ein* Argument liefert eine neue Funktion für einen (den zweiten) Parameter. Auf diese Weise können wir auch eine Inkrement-Operation definieren und erhalten so eine Funktion, die eine Funktion zurückliefert:  $\text{inc} = \text{add } 1$

Damit können wir uns auf den Standpunkt stellen, dass jede Funktion nur *einen* Parameter hat. Funktionen mit mehreren Parametern sehen wir als Funktionen höherer Stufe an: **add** verarbeitet nur den ersten Parameter (**partielle Applikation**) und liefert als Ergebnis eine (namenlose) Funktion, die den zweiten Parameter verarbeiten kann. Den Operator  $\rightarrow$  definieren wir dann als rechtsassoziativ: der Typ  $\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$  ist also gleich  $\text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})$ .

**Beispiel 7.16:** Zwei Funktionen, wo eine Funktion als Parameter vorkommt:

$\text{applyTwice } f \ x = f (f \ x)$  vom Typ  $\text{applyTwice} :: (a \rightarrow a) \rightarrow a \rightarrow a$

Die Klammern in der Typdeklaration sind notwendig, da der erste Parameter eine Funktion ist. Der zweite Parameter ist –als Argument für diese Funktion **f**– auch vom Typ **a**.

**Beispiel 7.17:** Die Funktion  $\text{iterate } f \ x$  liefert eine unendliche Liste zurück, die durch wiederholte Anwendung der Funktion **f** auf den Anfangswert **x** entsteht:  $[x, f(x), f(f(x)), \dots]$   
 $\text{iterate } f \ x = x : \text{iterate } f (f \ x)$  vom Typ  $\text{iterate} :: (a \rightarrow a) \rightarrow a \rightarrow [a]$

## 7.9.2 Anonyme Funktionen ( $\lambda$ -Ausdrücke)

Funktionen ohne Namen werden durch  $\lambda$ -Notation definiert: Statt des Funktionsnamens wird ein  $\lambda$  vor die Parameter geschrieben. In Haskell schreibt man statt  $\lambda$  einfach  $\backslash$  (*backslash*). Anonyme Funktionen werden z.B. als Parameter benutzt.

**Beispiel 7.18:** Eine anonyme Funktion, die drei Zahlen addiert, ist  $\backslash x \ y \ z \rightarrow x+y+z$

$\text{dec } x = x-1$  ist also nur eine Abkürzung für die Namensdefinition  $\text{dec} = \backslash x \rightarrow x-1$ .

**Beispiel 7.19:** Nullstellenbestimmung durch das Newton-Verfahren.

Um z.B.  $\sqrt{z}$  zu berechnen, bestimmen wir eine Nullstelle der Funktion  $f(x) = x^2 - z$ , indem wir den Grenzwert der Folge  $x_0 = 1, x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} = \frac{1}{2} \left( x_i + \frac{z}{x_i} \right)$  berechnen.

Da wir nicht wissen, wann die Folge den Grenzwert erreicht, benutzen wir eine unendliche Liste.

$\text{limes}(a:b:x) = \text{if } a == b \text{ then } a$  -- Vorsicht: Rundungsfehler beim Vergleich möglich  
 $\text{else } \text{limes}(b:x)$

$\text{wurzel } z = \text{limes } (\text{iterate } g \ 1)$   
 $\text{where } g(x) = (x + z/x)/2.0$

Mit anonymer Funktion:  $\text{wurzel } z = \text{limes } (\text{iterate } (\backslash x \rightarrow (x + z/x)/2.0) \ 1)$

## 7.10 Monaden

Für manche Probleme ist sogar in funktionalen Sprachen ein imperatives –vom Zustand abhängiges– Programmkonstrukt geeigneter, z.B. für Ein-/Ausgabe, zur Erzeugung eindeutiger Marken, an der Schnittstelle zu imperativen Sprachen oder bei zerstörerischen Änderungen einer Datenstruktur. Dazu müssen **Nebenwirkungen simuliert** werden, ohne die Funktionalität der Sprache zu zerstören. Haskell, als funktionale Sprache mit Bedarfsauswertung, benutzt dafür **Monaden**.

## 7.11 Funktionale Programmierstile am Beispiel der Fakultät

- |   |   |
|---|---|
| <ul style="list-style-type: none"> <li>• Ein bedingter Ausdruck ersetzt die bedingte Anweisung der imperativen Lösung:</li> </ul>   | <pre>fac n = if n == 0       then 1       else n * fac (n-1)</pre>                    |
| <ul style="list-style-type: none"> <li>• Die Funktion wird durch mehrere Gleichungen definiert:</li> </ul>  | <pre>fac 0 = 1 fac n = n * fac (n-1)</pre>  |
| <ul style="list-style-type: none"> <li>• Ein zusätzlicher Parameter <i>a</i> einer Hilfsfunktion <i>facAcc</i> speichert das Zwischenergebnis (Akkumulator) und wird mit 1 initialisiert.</li> </ul>  | <pre>facAcc a 0 = a facAcc a n = facAcc (n*a) (n-1) fac = facAcc 1</pre>              |
| <ul style="list-style-type: none"> <li>• Ein zusätzlicher Parameter <i>k</i> einer Hilfsfunktion <i>facCps</i> merkt sich die noch zu erledigten Funktionen (<i>continuation passing style</i>), initialisiert mit dem neutralen Element <i>id</i> der Funktionskomposition.</li> </ul> | <pre>facCps k 0 = k 1 facCps k n =     facCps (k . (n *)) (n-1) fac = facCps id</pre> |
- Die verbleibenden Aufrufe sind endrekursiv und benötigen daher nur *einen* Kellerrahmen.
- Mit Listen und Listenfunktionen:
- ```
fac n = product [1..n]
```

## 7.12 Bemerkungen zur Übersetzung funktionaler Programmiersprachen

Der auffälligste Unterschied gegenüber imperativen Sprachen ist, dass es in funktionalen Sprachen keine Wertzuweisung gibt.

### 7.12.1 Übersetzung höherer Funktionen

Für die Berechnung eines Funktionswerts benötigt man

- den Code für die Funktion,
- die Werte der –innerhalb der Funktion vorkommenden– nicht-lokalen Variablen,
- die Werte der Parameter, soweit sie für die Auswertung des Rumpfes benötigt werden.

In einer Sprache erster Stufe ist eine Funktion eine Konstante. Insbesondere ist die Anfangsadresse des Codes dieser Funktion zur Übersetzungszeit bekannt.

In höher-stufigen Sprachen können *Werte* vorkommen, die selbst **Funktionen** sind. Da dieser Wert –eine Funktion!– oft erst zur Laufzeit ermittelt wird, wird er durch einen **Funktionszeiger** (*code pointer*) implementiert, einen Zeiger auf den Code der Funktion, der bei einer Funktionsanwendung auszuführen ist. Die Lebensdauer einer Funktion kann dadurch die Lebensdauer ihres Sichtbarkeitsbereichs überdauern! Daher muss der Funktionszeiger mit allen von der Funktion benutzten Variablenbindungen auf der Halde abgelegt werden.

Die *first-class*-Eigenschaft von Funktionen führt auch dazu, dass die Anzahl der Parameter von Funktionsdefinition und Funktionsaufruf nicht mehr übereinstimmen muss:



- **unterversorgte Funktionsaufrufe:** Enthält ein Funktionsaufruf weniger Parameter (z.B. 3) als die Funktionsdefinition (z.B. 5), so ist das Ergebnis eine Funktion mit den restlichen (hier:  $5-3 = 2$ ) Parametern.
- **übersorgte Funktionsaufrufe:** Enthält ein Funktionsaufruf mehr Parameter (z.B. 5) als die Funktionsdefinition (z.B. 3), so muss das Ergebnis zu einer Funktion führen, die dann die restlichen (hier: 2) Parameter als Argumente konsumieren kann.

### 7.12.2 Striktheitsanalyse: Kann der Call-by-Need durch den Call-by-Value ersetzt werden?

Eine naive Implementierung der Bedarfsauswertung (ständige Tests, ob ein Ausdruck schon berechnet wurde; viele Konstruktionen zur Darstellung unausgewerteter Ausdrücke) führt zu ineffizienten Programmen. Dieser Aufwand kann stark verringert werden, wenn man für viele Ausdrücke bereits zur Übersetzungszeit feststellt, dass sie zur Berechnung des Programm-Ergebnisses *immer* benötigt werden und sie dann durch Call-by-Value auswertet. Eine **Striktheitsanalyse** (*strictness analysis*) soll diese Ausdrücke finden.

Eine  $n$ -stellige Funktion  $f$  heißt strikt im  $i$ -ten Argument, falls der Wert dieses Arguments auf jeden Fall, also für alle möglichen Werte der Parameter von  $f$ , zur Berechnung des Funktionsergebnisses benötigt wird. Falls zur Berechnung des Ergebnisses einer Funktion  $f$  der Wert des  $i$ -ten Arguments sicher benötigt wird, bei einer Applikation von  $f$  die Berechnung des  $i$ -ten Arguments jedoch nicht terminiert, so kann auch die Berechnung der gesamten Applikation nicht terminieren. Stellt man einen undefinierten Wert –z.B. eine nicht-terminierende Berechnung– durch  $\perp$  dar, wird Striktheit formal definiert als:

Eine  $n$ -stellige Funktion  $f$  heißt **strikt im  $i$ -ten Argument** ( $1 \leq i \leq n$ ) genau dann, wenn gilt:

$$x_i = \perp \Rightarrow f\ x_1 \dots x_n = \perp$$

Wird der Wert dieses Arguments auf jeden Fall benötigt, so kann der  $i$ -te aktuelle Parameter bei einer Applikation der Funktion  $f$  schon beim Aufruf der Funktion  $f$  berechnet werden.

Ähnlich kann man bei Bindungen in *nicht-rekursiven* **let**-Ausdrücken verfahren: Wird der Wert eines Bezeichners  $x$  aus der Definitionsliste eines **let**-Ausdrucks zur Berechnung des Wertes des **let**-Ausdrucks auf jeden Fall benötigt, so braucht die Berechnung der rechten Seite der Definition von  $x$  nicht verzögert werden.

Leider ist die Striktheit einer Funktion unentscheidbar; jeder Algorithmus approximiert daher nur das Striktheitsverhalten. Die Approximation muss aber „von der richtigen Seite“ kommen: Ist eine Funktion strikt, so darf das Analyseergebnis „nicht strikt“ lauten – dann wird Optimierungspotential verschenkt. Eine nicht-strikte Funktion darf aber nie als strikt klassifiziert werden – eine Optimierung könnte dann die Bedeutung des Programms verändern.

Eine Striktheitsanalyse nutzt daher vor allem bei Funktionen erster Ordnung und flachen Wertebereichen (Datenstrukturen ohne Subkomponenten).

### 7.12.3 Übersetzung des Call-by-Need

Das Prinzip der **Bedarfsauswertung** (*lazy evaluation*) besteht darin, dass bei Funktionen die Auswertung der Argumente solange verzögert wird, bis die Auswertung nötig wird. Es genügt, einen Ausdruck soweit zu berechnen, dass der äußerste Daten-Konstruktor des Ergebniswertes bestimmt werden kann (*weak head normal form*); Unterkomponenten des Wertes können noch unausgewertet sein.

Nur in Sprachen ohne Nebenwirkungen lässt sich der **Call-By-Need** ohne allzu großen Mehraufwand implementieren: Man sorgt dafür, dass eine spätere Auswertung eines Ausdrucks den gleichen Wert liefert, indem man die Bindungen aller nicht-lokalen („freien“) Variablen (**Umgebung**),



Unser Ausdruck vereinfacht sich zu `member 2 (STOP (1 : STOP (2 : STOP (3 : [ ])))`

Wir erhalten dadurch ein Programm, in dem einige Argumente mit Call-By-Need, andere mit Call-By-Value ausgewertet werden.

#### 7.12.4 Übersetzung von unendlichen Listen

Wir veranschaulichen die Realisierung von unendlichen Listen mit Hilfe der Schlüsselwörter `STOP` und `GO`: ein `STOP` unterdrückt die Auswertung solange, bis ein `GO` die Auswertung erzwingt.

**Beispiel 7.20:** Die Funktion `map` wendet den ersten Parameter (eine Funktion) auf alle Elemente des zweiten Parameters (einer Liste) an:

```
map :: (a -> b) -> [a] -> [b]
map f [ ] = [ ]
map f (x:xs) = f x : map f xs
```

Eine Liste könnten wir definieren als:

```
data List a = [ ] | (:) a (List a)      -- (:) ist der Konstruktor für den NotEmpty-Fall
```

In der Implementierung mit den neuen Schlüsselwörtern würde daraus:

```
map f [ ] = [ ]
map f (x:xs) = f x : STOP (map f GO (xs))

data List a = [ ] | (:) a (STOP (GO (List a)))
```

Wenden wir jetzt einen Ausdruck wie `head(tail(map f L))` auf eine unendliche Liste `L = a:b:c:...` an, so müssen wir automatisch den Ausdruck um ein `GO` und die Liste um die `STOP`s ergänzen: `head(GO(tail(map f L)))` mit `L = a:STOP(b:STOP(c:...))`

Dann erhalten wir wie gewünscht:

```
head(GO(tail(map f L)))      -- Definition von map (mit head L für x und tail L für xs)
= head(GO(tail(f (head L) : STOP(map f GO(tail L)))))      -- tail auswerten
= head(GO(STOP(map f GO(tail L))))      -- GO hebt STOP auf
= head(map f GO(tail L))      -- konkrete Liste verwenden
= head(map f GO(tail (a:STOP(b:STOP(c:...)))))      -- tail auswerten
= head(map f GO(STOP(b:STOP(c:...))))      -- GO hebt STOP auf
= head(map f (b:STOP(c:...)))      -- Definition von map
= head(f b : map f STOP(c:...))      -- head auswerten
= f b
```

#### 7.12.5 Übersetzung von Haskell-Programmen

Ein Haskell-Programm wird in mehreren Schritten übersetzt:

- Das Programm wird durch Scanner und Parser in einen Syntaxanalyse-Baum überführt. Hier wird bereits die Typprüfung durchgeführt, da sich Fehlermeldungen sonst nur schwierig auf den ursprünglichen Programmtext beziehen können.
- Alle Konstrukte von Haskell können durch wenige, einfache **Kernkonstrukte** (*core*) von Haskell ausgedrückt werden (*desugaring*), vgl. Anhang F. Auf dieser Ebene werden viele Optimierungen durchgeführt.
- Die Kern-Sprache wird in Befehle einer virtuellen Maschine übersetzt, der STG-Maschine (*spineless tagless graph reduction machine*).
- Die eigentliche Codegenerierung erzeugt dann Code für C—.
- Daraus wird ein C-Programm, ein Assembler-Programm oder ein LLVM-Programm erzeugt.

## 8 Logische Programmierung mit Prolog

Anfänge der logischen Programmierung sind in den 1960er Jahren zu finden, als man Programme entwickeln wollte, die automatisch Beweise zu vorgelegten Aussagen konstruieren. In den 1980er Jahren spielte logische Programmierung eine wichtige Rolle beim Bau von Expertensystemen. Heute wird sie in den Bereichen Computerlinguistik und Künstliche Intelligenz verwendet.

### 8.1 Konzepte logischer Programmiersprachen

Funktionale Sprachen basieren auf Funktionen, die ja stets *einen* Funktionswert liefern. Der Funktionswert ist also durch die Argumente der Funktion determiniert. Logische Sprachen nutzen (endliche) Relationen und probieren daher verschiedene Alternativen durch (**Backtracking**).

#### 8.1.1 Fakten, Regeln und Anfragen

Logische Programme bekommen als Datenbasis **Fakten** (*facts*, *axioms*) und **Regeln** (*rules*). Wird dem Programm dann eine **Frage** (*query*, *goal*) gestellt, so versucht das Programm festzustellen, ob sich die Frage durch logische Schlüsse aus der Datenbasis **ableiten** lässt oder nicht.

**Beispiel 8.1:** Die Datenbasis besteht hier aus vier Fakten und einer Regel.

Datenbasis: Ein Hund ist ein Säugetier.  
Ein Mensch ist ein Säugetier.  
Ein Hund hat keine Arme.  
Ein Mensch hat zwei Arme.  
Ein Säugetier hat vier Beine und keine Arme oder  
es hat zwei Beine und zwei Arme.

Fragen: Hat ein Mensch zwei Beine? → ja  
Hat ein Hund zwei Beine? → nein  
Wieviel Beine hat ein Hund? → 4

Es wird nirgendwo erklärt, wie dieses Programm zu seiner Antwort kommt, d.h. man gibt nur Fakten, Regeln und eine Frage vor! Wie der Beweis durchzuführen ist, wird nicht beschrieben. Die Reihenfolge, in der die Fakten und Regeln auftreten, ist logisch nicht relevant. Programme, die derartige Schlussfolgerungen ziehen können, heißen **Deduktionssysteme**.

Programmiersprachen, die nur das Ergebnis beschreiben (*Was ist gesucht?*), aber nicht den Berechnungsweg (*Wie wird es gefunden?*), heißen **nicht-prozedural** (*high level programming*). Ein anderes Beispiel für eine nicht-prozedurale Sprache ist die Datenbanksprache SQL.

Da das Beantworten der gestellten Frage logisches Schließen erfordert, bildet die mathematische Logik die Grundlage dieser Systeme.

Logische Ausdrücke werden aufgebaut aus **Prädikaten**, d.h. Funktionen, die Wahrheitswerte **true** oder **false** liefern. Man benötigt Konstanten (z.B. Zahlen und Symbole) und Variablen, um die Argumente der Prädikate zu bilden. Prädikate können über boolesche Funktionen wie **and**, **or** und **not** verknüpft werden. Außerdem gibt es die **Implikation** „ $\Rightarrow$ “. Dabei bedeutet  $A \Rightarrow B$  (aus  $A$  folgt  $B$ ) das gleiche wie (**not**  $A$ ) **or**  $B$ .

Dabei wird stets angenommen, dass die Prädikate für die angegebenen Argumente den Wert **true** liefern und für alle anderen, nicht mit Hilfe der Regeln herleitbaren Werte, den Wert **false** liefern (**Closed World Assumption**).

Weiter gibt es Quantifizierer: den Allquantor  $\forall$  (mit der Bedeutung: Für alle  $X$  gilt ...) und den Existenzquantor  $\exists$  (mit der Bedeutung: Es gibt ein  $X$ , für das gilt ...), die jeweils eine Variable einführen, die in den Prädikaten als Argument auftreten kann.

Zu den Konstruktionsregeln für logische Ausdrücke gehören Schlussregeln, die angeben, wie man aus einer Menge von (wahren) Fakten und Regeln neue (wahre) Aussagen erzeugen kann.

**Beispiel 8.2:** Das Beispiel 8.1 in Prädikat-Schreibweise:

```
säugetier(hund).
säugetier(mensch).
arme(hund,0).
arme(mensch,2).
Für alle X gilt: säugetier(X)  $\Rightarrow$  (beine(X,4) and arme(X,0)) or
                                   (beine(X,2) and arme(X,2)).
```

Die Fragen lauten jetzt:

```
beine(hund,4)?            $\rightarrow$  ja
beine(hund,2)?            $\rightarrow$  nein
Existiert ein X mit beine(hund,X)?  $\rightarrow$  X=4
```

Die Fakten und Regeln bilden das logische Programm, die Eingabe ist die Frage, ob sich diese Aussage aus den Fakten und Regeln herleiten lässt, und die Ausgabe ist entweder **ja** oder **nein** oder –falls die Frage Variablen enthält– die **Variablenbelegungen**, für die die Ausgabe **ja** hergeleitet werden kann.

Prolog ist nicht typisiert, d.h. beliebige Terme können an eine Variable gebunden werden bzw. an Parameterpositionen von Prädikaten stehen.

Leider ist der bisher vorgestellte Prädikatenkalkül noch zu umfangreich und zu mächtig für ein effizientes automatisches Beweisprogramm. Daher beschränkt man sich auf eine Teilmenge aller logischen Ausdrücke, die **Horn-Klauseln**.

Horn-Klauseln haben die Form  $a_1 \text{ and } a_2 \text{ and } \dots \text{ and } a_n \Rightarrow b$ , wobei die  $a_i$  genügend „einfache“ Ausdrücke sein müssen (sie dürfen z.B. kein **or** und keine Quantoren enthalten). Man bezeichnet  $b$  auch als Kopf und „ $a_1 \text{ and } a_2 \text{ and } \dots \text{ and } a_n$ “ als Rumpf der Klausel.

Jede im Kopf auftretende Variable ist implizit mit einem Allquantor, jede *nur* im Rumpf auftretende Variable mit einem Existenzquantor verbunden.

**Beispiel 8.3:** Im Ausdruck  $\text{parent}(X,Y) \text{ and } \text{parent}(Y,Z) \Rightarrow \text{grandparent}(X,Z)$  sind **parent** und **grandparent** Prädikate und  $X$ ,  $Y$  und  $Z$  Variable. Links vom  $\Rightarrow$  Zeichen ist der Rumpf der Regel, rechts der Kopf der Regel. Man interpretiert den Ausdruck wie folgt: Für alle  $X$  und  $Z$  gilt: Wenn es ein  $Y$  gibt, so dass  $X$  Elternteil von  $Y$  ist und  $Y$  Elternteil von  $Z$  ist, dann ist  $X$  Großelternteil von  $Z$ .

Horn-Klauseln werden üblicherweise „anders herum“ geschrieben. In Prolog wird außerdem das **and** durch ein Komma und das Implikationszeichen  $\Leftarrow$  durch **:-** ersetzt:

$$b \text{ :- } a_1, a_2, \dots, a_n.$$

Fakten werden durch Horn-Klauseln mit leerem Rumpf dargestellt. In Prolog beginnen Variablen immer mit Großbuchstaben, Prädikate und Symbole mit einem kleinen Buchstaben.

**Beispiel 8.4:** Das Beispiel 8.2 lautet in Prolog-Notation (die Regel wurde durch zwei Horn-Klauseln ersetzt; für jeden oder-Zweig eine Klausel):

```

saeugetier(hund).
saeugetier(mensch).
arme(hund,0).
arme(mensch,2).
beine(X,4) :- saeugetier(X) , arme(X,0).
beine(X,2) :- saeugetier(X) , arme(X,2).

```

### 8.1.2 Arbeitsweise des Prolog-Systems: Resolution und Unifikation

Wie kann man nun aus Fakten und Regeln, die als Horn-Klauseln notiert sind, andere Aussagen herleiten? Die prinzipielle Arbeitsweise besteht aus der Anpassung von Variablen (**Unifikation**) und der Kombination von Klauseln (**Resolution**).

So kann man aus den beiden Klauseln

```

b :- a1, a2, ... , an.      und
c :- d1, d2, ... , dk.      mit  c = ai

```

die folgende Klausel erzeugen:

```

b :- a1, ... , ai-1, d1, ... , dk, ai+1, ... , an.

```

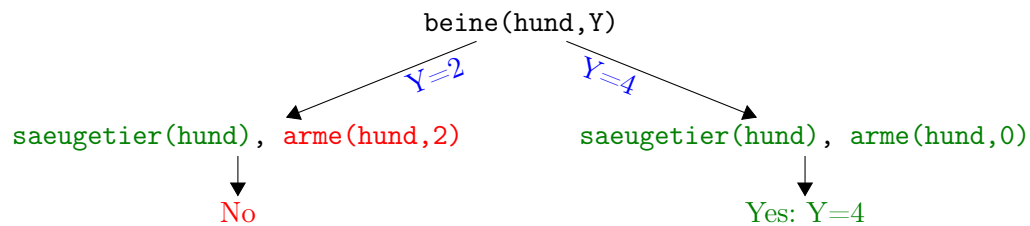
Enthalten  $a_i$  und  $c$  Variablen, so muss zunächst festgestellt werden, ob es möglich ist, diese Variablen so an (möglichst allgemeine) Werte zu binden, dass beide Teilausdrücke identisch werden (**Unifikation**). Es werden hier also Zeichenketten verglichen.

Man kann jetzt die prinzipielle Arbeitsweise dieses deduktiven Systems erahnen. Aus Effizienzgründen versucht man nicht, ausgehend von den Fakten mit Hilfe der Regeln die Frage (das Ziel) zu konstruieren, sondern man geht von der eingegebenen Frage aus, zerlegt diese durch die Regeln in einfachere Teilziele, um letztlich bei den Fakten zu enden.

Also wird zunächst die Datenbasis durchsucht, ob es eine Horn-Klausel mit „passendem“ Kopf gibt. Dann werden durch Unifikation evtl. Variablen mit Werten belegt. Diese Werte werden dann in den Rumpf der entsprechenden Regel substituiert, und man muss nun versuchen, die einzelnen Teile des Rumpfes (Teilziele) auf die gleiche Weise zu bearbeiten. Dies endet, wenn man nur noch Teilziele hat, die Fakten der Datenbasis sind. Gelangt man jedoch in eine Sackgasse, z.B. weil ein Teilziel nicht weiter verarbeitet werden kann oder keine Unifikation möglich ist, so wird die Berechnung soweit rückgängig gemacht, dass eine andere Alternative verfolgt werden kann (**Rücksetzverfahren** (*backtracking*)).

**Beispiel 8.5:** Wir stellen jetzt Fragen an die Datenbasis aus Beispiel 8.4.

- Auf die Frage `?- beine(mensch,2).` ist eine Unifikation nur mit der 6. Regel möglich, wobei  $X = \text{mensch}$  gesetzt muss. Dies führt zu den beiden neuen Fragen `saeugetier(mensch)` und `arme(mensch,2)` und, da beide Fakten in der Datenbasis sind, zur Antwort **ja**.
- Die Frage `?- arme(hund,X).` enthält eine **freie Variable**  $X$ . Unifikation liefert mit der 3. Regel die Antwort **ja** mit der Bindung  $X = 0$ .
- Auf die Frage `?- beine(hund,Y)` ist eine Unifikation mit der 6. Regel möglich, wobei man  $X = \text{hund}$  und  $Y = 2$  setzen muss. Damit ergeben sich die neuen Teilziele (Fragen) `saeugetier(hund)` und `arme(hund,2)`, die zu beweisen sind. Während das erste Teilziel als Fakt in der Datenbasis steht, ist das zweite Ziel kein Fakt und auch nicht herleitbar. Also muss man zurück zu einem Punkt, an dem eine Alternative möglich ist. In unserem Fall ist das der Versuch einer Unifikation mit der 5. Regel und den Zuordnungen  $X = \text{hund}$  und  $Y = 4$ . Damit ergeben sich die beiden Teilziele `saeugetier(hund)` und `arme(hund,0)`. Beide Teilziele sind Fakten und die Antwort auf die Frage damit  $Y = 4$ .



```

1   ?- beine(hund, Y).
2   -> Y = 4

```

Am Beispiel sieht man, dass zwei Punkte bei der logischen Programmierung nicht festgelegt sind:

- In welcher Reihenfolge werden passende Regeln angewendet?
- In welcher Reihenfolge werden die Teilziele bearbeitet?

In Prolog sind diese Punkte eindeutig festgelegt. Die Datenbasis wird in der Eingabereihenfolge durchsucht („von oben nach unten“) und die Teilziele werden von links nach rechts abgearbeitet. Dabei werden neue Teilziele sofort berücksichtigt, d.h. man kann sich die Arbeitsweise auch als **Tiefensuche** (*depth first search*) in einem Baum aller vom Ziel aus möglichen Schritte vorstellen.

**Beispiel 8.6:** Man betrachte folgende Fakten und Regeln:

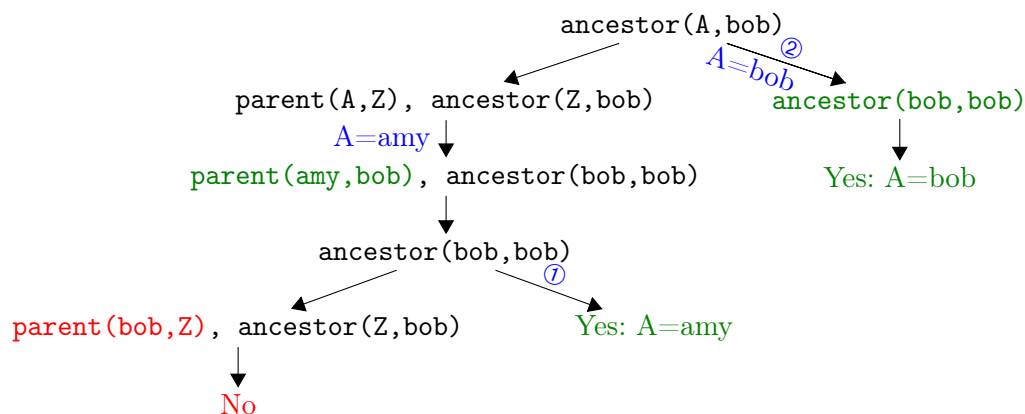
```

ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
ancestor(X,X).
parent(amy,bob).

```

Stellt man die Frage `?- ancestor(A,bob).`, so müsste man bei einem idealen System die Antworten `A=amy` und `A=bob` bekommen, wobei die Reihenfolge unbestimmt ist.

In Prolog ist die Reihenfolge der Berechnungsschritte fest vorgegeben und die Antworten erscheinen stets in der Reihenfolge `A=amy` und `A=bob`.



(Die Reihenfolge des Backtrackings ist mit ①, ②, ③, ④ usw. gekennzeichnet. Literale, die nicht mit Regeln der Datenbasis unifiziert werden können, sind rot gezeichnet; Literale, die als wahr erkannt wurden, also mit Fakten der Datenbasis unifiziert werden konnten, sind grün gezeichnet.)

```

1   ?- ancestor(A, bob).
2   -> A = amy
3   -> A = bob

```

Vertauscht man die Teilziele der ersten Regel, also

```

ancestor(X,Y) :- ancestor(Z,Y), parent(X,Z).

```

so dürfte das bei einem idealen System keine Änderung hervorrufen, bei Prolog erhält man jedoch keine Antwort mehr, da bei der Abarbeitung der obigen Regel das erste Teilziel `ancestor(Z,Y)`

wieder auf die Anwendung der gleichen Regel führt. Man befindet sich also in einer Endlosschleife und das System wird wegen Speichermangels anhalten.

**Beispiel 8.7:** Eine Datenbasis für Verwandtschaftsbeziehungen:

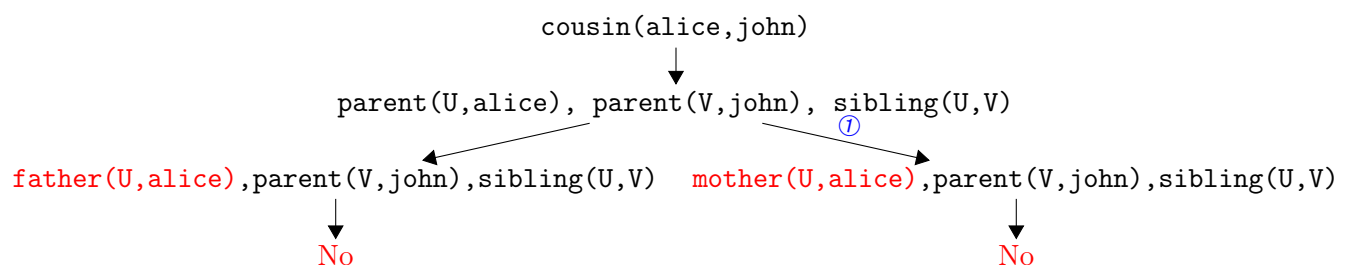
```
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
grandparent(X,Z) :- parent(X,Y), parent(Y,Z).
ancestor(X,Z) :- parent(X,Z).
ancestor(X,Z) :- parent(X,Y), ancestor(Y,Z).
sibling(X,Y) :- mother(M,X), mother(M,Y), X\=Y, father(F,X), father(F,Y).
cousin(X,Y) :- parent(U,X), parent(V,Y), sibling(U,V).

father(bert, jeff).
mother(alice, jeff).
mother(alice, george).
father(bert, george).
father(john, mary).
mother(sue, mary).
father(george, cindy).
mother(mary, cindy).
mother(mary, vic).
father(george, vic).
```

Da eine Unifikation nur gelingt, wenn das Prädikatensymbol sowie die Stelligkeit übereinstimmen, kann das Prolog-System zur Optimierung die Regeln und Fakten der Datenbasis nach Prädikatensymbol und Stelligkeit des Kopfes sortieren (z.B. nach `father/2`), um dann schnell auf alle Regeln eines Prädikats zugreifen zu können. Die Sortierung darf natürlich die Reihenfolge der Regeln des gleichen Prädikats nicht verändern.

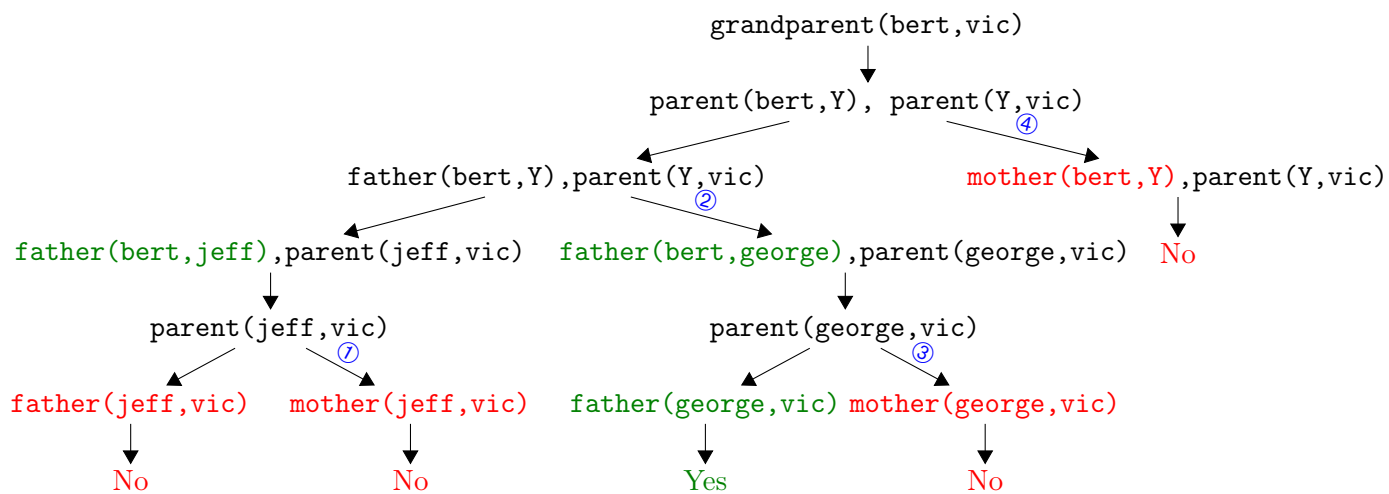
Man kann dann folgende Fragen stellen:

```
?- father(george,vic). -> yes           -- unmittelbar aus der Datenbasis
?- cousin(alice,john). -> no
```



```
?- grandparent(bert,vic). -> yes
```





Anfragen mit Variablen sind möglich:

```
?- father(X,jeff). -> X = bert
```

Eine Anfrage kann mehrere Lösungen liefern:

```
?- sibling(A,B). -> A = jeff    B = george
                  A = george   B = jeff
                  A = cindy    B = vic
                  A = vic      B = cindy
```

Eine Anfrage kann aus mehreren Prädikaten zusammengesetzt sein:

```
?- grandparent(john,X), parent(Y,X), sibling(Y,jeff).
                                     -> X = cindy  Y = george
                                     X = vic    Y = george
```

**Beispiel 8.8:** Eine formale Differentiation (ohne Termvereinfachung) ist in Prolog nicht schwer:

```
d(X, U+V, DU + DV) :- d(X,U,DU), d(X,V,DV).
d(X, U-V, DU - DV) :- d(X,U,DU), d(X,V,DV).
d(X, U*V, DU*V + U*Dv) :- d(X,U,DU), d(X,V,DV).
d(X, U/V, (DU*V - U*Dv)/V^2) :- d(X,U,DU), d(X,V,DV).
d(X, U^C, C*U^(C-1)*DU) :- d(X,U,DU), atomic(C), C\=X.
d(X,X,1).
d(X,C,0) :- atomic(C), C\=X.
```

und man erhält z.B.

```
?- d(x,x^2,Ans). -> Ans = 2*x^(2-1)*1
?- d(x, x^2+b*x+c, Ans). -> Ans = 2*x^(2-1)*1+(0*x+b*1)+0
?- d(y,y^n/(x+y),Dy). -> Dy = (n*y^(n-1)*1*(x+y)-y^n*(0+1))/(x+y)^2
?- d(x, Int, 2*x^(2-1)*1). -> Int = x^2
```

## 8.2 Computer-Arithmetik in Prolog

In Prolog werden Datenstrukturen i.d.R. implizit durch ihre Eigenschaften definiert. So kann man natürliche Zahlen und die Summe definieren durch:

```
sum(succ(X), Y, succ(Z)) :- sum(X,Y,Z).
sum(0, X, X).
```

Das Prädikat `sum` legt die Eigenschaften der Summe fest. Das Funktionssymbol `succ` beschreibt dabei für den Nachfolger einer Zahl. Die Summe  $2 + 3 = A$  wird dann so formuliert:

```
?- sum(succ(succ(0)), succ(succ(succ(0))), A).
    -> A = succ(succ(succ(succ(succ(0)))))
```

```
sum(succ(succ(0)), succ(succ(succ(0))), A)
    1. Klausel ↓ A=succ(Z)
sum(succ(0), succ(succ(succ(0))), Z)
    1. Klausel ↓ Z=succ(Z1)
sum(0, succ(succ(succ(0))), Z1)
    2. Klausel ↓ Z1=succ(succ(succ(0)))
Yes: A=succ(succ(succ(succ(succ(0)))))
```

Dabei sind die Rollen als Ein- bzw. Ausgabeparameter nicht festgelegt, z.B.  $2 + B = 5$ :

```
?- sum(succ(succ(0)), B, succ(succ(succ(succ(succ(0))))) .
    -> B = succ(succ(succ(0)))

?- sum(A, B, succ(succ(succ(succ(succ(0))))) .
    -> A = succ(succ(succ(succ(succ(0))))) B = 0
    A = succ(succ(succ(succ(0)))) B = succ(0)
    A = succ(succ(succ(0))) B = succ(succ(0))
    A = succ(succ(0)) B = succ(succ(succ(0)))
    A = succ(0) B = succ(succ(succ(succ(0))))
    A = 0 B = succ(succ(succ(succ(succ(0)))))
```

Wir können die Differenz als Umkehrfunktion also definieren durch:

```
dif(X,Y,Z) :- sum(Z,Y,X).
```

Diese Arithmetik ist natürlich viel zu langsam, so dass alle Prolog-Systeme üblicherweise die Computer-Arithmetik direkt verwenden. Dies hat jedoch einige unerfreuliche Folgen. Zum einen kann man Regeln, die arithmetische Ausdrücke enthalten, nicht mehr so einfach interpretieren, und zum anderen lässt sich die Umkehrfunktion nicht durch Parametertauschen berechnen.

Man betrachte z.B. die Fibonacci-Zahlen  $F_i$ , die durch  $F_0 = 0$ ,  $F_1 = 1$  und  $F_i = F_{i-1} + F_{i-2}$  für  $i \geq 2$  definiert sind.

Eine naheliegende Umsetzung in eine logische Programmiersprache wäre:

```
fib(0,0).
fib(1,1).
fib(N,F) :- N = M+1, M = K+1, fib(M,G), fib(K,H), F = G+H, N > 1.
```

In Prolog funktioniert das nicht! Ein Gleichheitszeichen steht für Unifikation der beiden Seiten. Ein Aufruf von `?- fib(2,A)` findet daher keine Lösung, denn 2 ist nicht mit  $M+1$  unifizierbar!

Prolog sieht für einen derartigen Fall die `is`-Relation vor, z.B. funktioniert

```
?- M is 2 + 3.    -> M = 5
```

Also schreibt man die dritte Zeile des obigen Fibonacci-Programms um zu

```
fib(N,F) :- N is M+1, M is K+1, fib(M,G), fib(K,H), F is G+H, N > 1.
```

Leider löst das unser Problem auch nicht, denn  $M+1$  kann nicht berechnet werden (aber  $N-1$ ).

```
fib(N,F) :- M is N-1, K is M-1, fib(M,G), fib(K,H), F is G+H, N > 1.
```

Dieses Mal findet Prolog die Lösung; sucht danach aber nach weiteren Lösungen und das System stoppt wegen Speichermangels. Die Abbruchbedingung muss vor den rekursiven Aufrufen stehen! Die endgültige Version ist daher

```
fib(0,0).
fib(1,1).
fib(N,F) :- N > 1, M is N-1, K is M-1, fib(M,G), fib(K,H), F is G+H.
```

Auch die folgende Frage funktioniert nicht, da die Computer-Arithmetik nicht umkehrbar ist:

```
?- fib(N,2).
```

### 8.3 Listen in Prolog

In Prolog lassen sich auch Listen durch Eigenschaften definieren<sup>26</sup>:

```
hd(cons(X, _), X).      list(nil).      null(nil).
tl(cons(_, L), L).      list(cons(_, L)) :- list(L).
```

Dabei bezeichnet „\_“ eine neue („anonyme“) Variable, deren Name nicht weiter interessiert.

`nil` bezeichnet die leere Liste. Die Operation `cons` fügt ein Element vor die Elemente einer existierenden Liste ein. `hd` liefert das erste Element einer Liste, `tl` die Restliste.

Listen sind wichtige Strukturen, daher hat man in Prolog ein wenig „syntaktischen Zucker“ um ihre äußere Darstellung gelegt. Listen werden durch  $[a_1, \dots, a_n]$  bezeichnet. Die Notation  $[X|L]$  entspricht dem `cons(X, L)`. Damit kann man folgende alternative Definition der Prädikate bilden:

```
hd([X|_], X).
tl([_|L], L).
?- hd([a,b],c,d,X). -> X = [a,b]
```

**Beispiel 8.9:** Das folgende `append`-Prädikat verkettet zwei Listen:

```
append([],L,L).
append([X|L],M,[X|N]) :- append(L,M,N).

?- append([a,b],[c,d],[a,b,c,d]). -> yes
?- append([a,b],[c,d],Ans). -> Ans = [a,b,c,d]
```

Das `append`-Prädikat kann auch zur Ermittlung des Anfangsstücks einer Liste oder zur Aufteilung einer Liste verwendet werden:

```
?- append(L,[c,d],[a,b,c,d]). -> L = [a,b]
?- append(X,Y,[a,b,c,d]).      ->
                                X = []   Y = [a,b,c,d]
                                X = [a]   Y = [b,c,d]
                                X = [a,b] Y = [c,d]
                                X = [a,b,c] Y = [d]
                                X = [a,b,c,d] Y = []
```

<sup>26</sup>Mehrfaches Vorkommen von Variablen in Mustern stört in logischen Programmiersprachen nicht.

**Beispiel 8.10:** Ein einfaches Prolog-Programm ermittelt, ob ein Wort  $w$  zu einer von einer kontextfreien Grammatik erzeugten Sprache gehört. Dabei stellen wir Wörter durch eine *Liste* von Zeichen dar. Für jedes Nichtterminal der Grammatik definieren wir ein zweistelliges Hilfsprädikat `nt_`, das als Parameter die –bei einer Linksableitung– noch nicht verarbeiteten Zeichen *vor* und *nach* der Abarbeitung dieses Nichtterminals enthält. Jede Produktion wird durch eine Hornklausel dargestellt.

Sei z.B. die kontextfreie Grammatik  $G = (N, T, P, S)$  mit  $N = \{S, A, B\}$ ,  $T = \{a, b, c, d\}$  und  $P = \{S \rightarrow aAdB, A \rightarrow bcA \mid \varepsilon, B \rightarrow c\}$  gegeben.

```

gehoertzuLG(X)      :- ntS(X, [ ]).                -- S ist Startsymbol
ntS([a | U], V)    :- ntA(U, [d | W]), ntB(W, V).
ntA([b, c | U], V) :- ntA(U, V).
ntA(U, U).
ntB([c | U], U).

```

Die Anfrage `gehoertzuLG([a,b,c,b,c,d,c]).` ermittelt dann, ob das Wort *abcbedc* in der Sprache  $L(G)$  liegt.

## 8.4 Verkleinerung des Suchraums: Das Prädikat cut

Zunächst definieren wir die Fakultätsfunktion:

```

fac(0,1).
fac(N,F) :- N1 is N-1, fac(N1,F1), F is F1*N.

```

Problem: Nach dem Finden der Lösung läuft das System in eine Endlos-Schleife und stoppt schließlich mit einem Speicherüberlauf. Das Problem kommt noch stärker zum Vorschein, wenn man als Ziel z.B.

```
?- fac(4,A), A=10.
```

vorgibt.

Nachdem das Teilziel `fac(4,A)` bewiesen wurde und  $A$  an  $4! = 24$  gebunden wurde, wird jetzt das zweite Teilziel angegangen, das aber mit der vorgegebenen Bindung von  $A$  nicht zu beweisen ist. Also wird ein Backtracking durchgeführt, und der einzige Punkt, an dem eine andere Klausel angewendet werden kann, ist das Teilziel `fac(0,1)` von `fac(4,A)`. Also wird dort die zweite Klausel probiert, die dazu führt, das ein neues Teilziel `fac(-1,F1)` eingeführt wird. Dies führt jedoch zu einer Endlos-Schleife.

Zur Lösung derartiger Probleme (und zur gezielten Effizienzsteigerung) gibt es in Prolog das **cut-Prädikat** (Schreibweise „!“), das immer den Wert `true` hat. Ein `cut` friert die einmal gemachte Auswahl der Klausel ein, d.h. wird beim Backtracking ein `cut` erreicht, dann werden weitere Alternativen ausgeschlossen, also der Baum aller möglichen Lösungswege wird beschnitten!

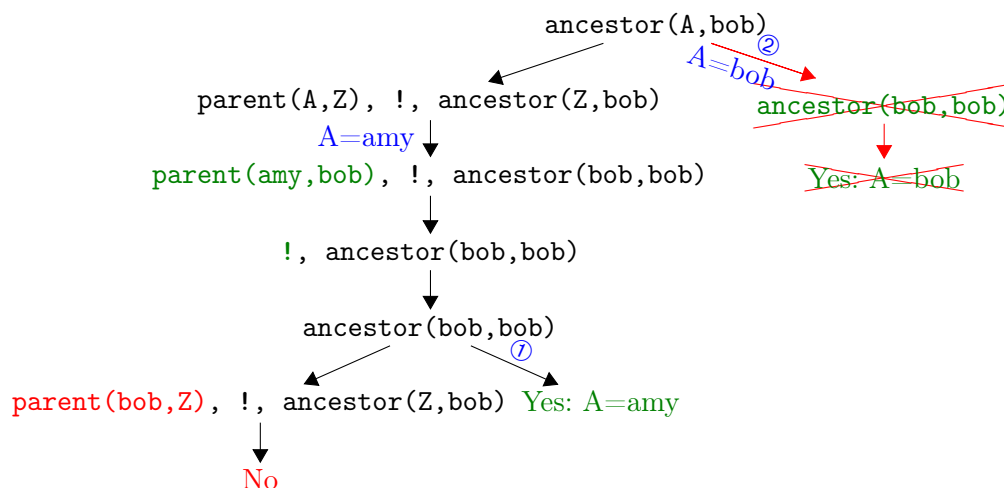
**Beispiel 8.11:** Mit den Regeln und Fakten aus dem Beispiel 8.6 liefert das Programm auf die Frage `?- ancestor(A,bob).` als Antwort `amy und bob`.

Mit einem `cut` in der Mitte des Rumpfes der ersten Regel erhält man nur eine Lösung:

```

ancestor(X,Y) :- parent(X,Z), !, ancestor(Z,Y).
ancestor(X,X).
parent(amy,bob).
?- ancestor(A,bob).    ->    A = amy

```



Mit einem `cut` am Anfang des Rumpfes der ersten Regel erhält man gar keine Lösung:

```
ancestor(X,Y) :- !, parent(X,Z), ancestor(Z,Y).
ancestor(X,X).
parent(amy,bob).
?- ancestor(A,bob).    ->    no
```

Durch Abarbeitung des Prädikats „!“ werden alle noch offenen Alternativen für diese Klausel beim Backtracking gelöscht.

## 8.5 Kurze Bemerkungen zur Übersetzung von Prolog

Die Übersetzung eines Prolog-Programms kann man sich –grob– stufenweise vorstellen:

- Übersetzung einer atomaren Formel bzw. eines Literals (als Prämisse),
- Übersetzung einer Horn-Klausel,
- Übersetzung eines Prädikats.

Ein **Literal** ist eine atomare Formel oder ihre Negation. Ein Literal  $q(t_1, \dots, t_k)$  kann man wie einen Prozeduraufruf übersetzen: Es wird ein Kellerrahmen angelegt, die Parameter werden gespeichert und es wird das Prädikat  $q/k$  aufgerufen.

Eine **Horn-Klausel**  $r \equiv q(X_1, \dots, X_k) \leftarrow g_1, \dots, g_n$  enthalte die Variablen  $\{X_1, \dots, X_m\}$ ,  $m \geq k$ .

Der Code für eine Klausel muss zuerst Platz für diese lokalen Variablen der Klausel allokalieren. Hier wird die **Unifikation** durchgeführt. Anschließend benötigen wir Code zur Auswertung des Rumpfs. Am Ende sollte der Kellerrahmen –wenn möglich– freigegeben werden.

Ein ( $k$ -stelliges) **Prädikat**  $q/k$  wird definiert durch eine *Folge* von Klauseln  $rr \equiv r_1 \dots r_f$ .

Falls  $q/k$  durch mehrere Klauseln definiert ist, muss die erste Alternative angesprungen werden. Schlägt diese fehl, werden der Reihe nach die weiteren Alternativen probiert. Wir benötigen dazu eine Implementierung, die einzelne Alternativen versuchsweise ausführt und bei Fehlschlag ihren gesamten Effekt wieder rückgängig macht (*backtracking*).

Wird das Backtracking aktiviert, soll die gesamte Berechnung bis zum *dynamisch* letzten Ziel rückgängig gemacht werden, an dem eine alternative Klausel gewählt werden kann. Den zugehörigen Kellerrahmen auf dem Laufzeitkeller nennen wir den aktuellen **Rücksetzpunkt**. Ein neues Register, der **Rücksetzzeiger** *BP* (*backtrack pointer*), zeigt auf den aktuellen Rücksetzpunkt.

Um zwischenzeitlich eingegangene Variablenbindungen aufzuheben, werden die eingegangenen neuen Bindungen protokolliert, indem die zugehörigen Variable in einer speziellen Datenstruktur, der **Spur** (*trail*) gespeichert werden. Die Spur ist ein weiterer Keller *T*, auf dem Haldenadressen von Referenzobjekten gemerkt werden, die eine neue Bindung erhalten. Ein weiteres Register, der **Spurzeiger** *TP* (*trail pointer*), zeigt stets auf die oberste belegte Spurzelle.

Die Organisation eines Kellerrahmes (Abb. 8.1) wird dadurch aufwändiger:

Wie bisher nutzen wir eine organisatorische Zelle zur Rettung des aktuellen Werts des *PC*, der **positiven Fortsetzungsadresse**, d.h. der Stelle im Code, an der bei erfolgreicher Abarbeitung des Literals fortgefahren werden soll. Dies entspricht der Rücksprungadresse. Ebenso retten wir wieder den Inhalt des Registers *FP*, den Verweis auf den **dynamischen Vorgänger**.

Im Kellerrahmen benötigen wir zusätzlich die **Codeadresse für die nächste Alternative**, d.h. die **negative Fortsetzungsadresse** sowie den alten Wert des *BP*. Weiterhin retten wir an einem Rücksetzpunkt die alten Werte der Register *TP* und *HP*.

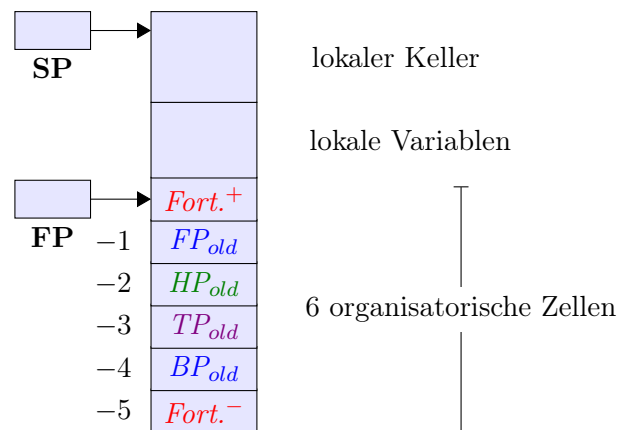


Abb. 8.1: Ein Kellerrahmen in Prolog.

## 8.6 Ausblick: Funktional-logische Sprachen

Funktionale Sprachen bieten durch Funktionen mit ihren ein-elementigen Ergebnissen deterministische Effizienz. Geschachtelte Ausdrücke und **Funktionale** ermöglichen knappe, elegante Programme und die Bedarfsauswertung erlaubt nicht nur **rekursive**, sondern sogar **unendliche Datenstrukturen**. Ausdrücke einer funktionalen Sprache werden deterministisch reduziert; am Ende einer Berechnung enthält der Ausdruck nur noch Konstruktoren.

Ausdrücke in logischen Sprachen sind Suchziele mit Unbekannten, also freien Variablen. Das Ergebnis einer Berechnung sind **Bindungen** für diese Unbekannten; die Berechnung ist eine Suche nach diesen Bindungen. Logische Sprachen arbeiten relational (mehrere Ergebnisse sind möglich), also potentiell nicht-deterministisch, was zu Effizienzverlusten durch **Backtracking** führen kann. Aber:

- Die **Unifikation** ist allgemeiner als der **Mustervergleich** funktionaler Sprachen.
- Eine Berechnung funktioniert in logischen Sprachen i.d.R. auch bei unvollständigen (nicht-instantiierten) Daten, also von **Ausdrücken mit freien Variablen**.
- Die –in logischen Sprachen übliche– **Funktionsinvertierung** spart viele Funktionsdefinitionen ein, die in funktionalen Programmen nötig wären.  
So kann z.B. das **append**-Prädikat nicht nur zwei Listen zusammenfügen, sondern auch eine Liste aufteilen oder ein Anfangsstück einer Liste bestimmen (siehe Beispiel 8.9).
- Die in logischen Sprachen eingebauten **Suchstrategien** sind bequem (es muss ja nicht immer –wie in Prolog– nur eine Tiefensuche sein).

Kurz: Man hätte gern das Beste aus beiden Welten.

Die Sprache Curry erweitert Haskell um **freie Variablen** und **Bedingungen** (*constraint*) an diese, um deren Wertebereich einzuschränken, sowie um **nicht-deterministische „Funktionen“**.

Die freien Variablen werden mit Werten belegt, so dass der Ausdruck auswertbar wird und zu

einem Ergebnis führt:

```
x && (y || (not x))    where x,y free
{x=True, y=True} True
{x=True, y=False} False
{x=False, y=y} False
```

Falls wir nicht an allen Lösungen interessiert sind, können wir mit dem Operator `==` eine **Gleichheitsbedingung** (*equational constraint*) formulieren wie

```
(x && (y || (not x))) == True    where x,y free
```

mit der Lösung

```
{x=True, y=True} success
```

Zwischen mengenwertigen Relationen und einwertigen Funktionen vermittelt z.B. die **nicht-deterministische „Funktion“** `choose`, die eines ihrer Argumente als Ergebnis auswählt und –wie bei logischen Sprachen üblich– weitere Ergebnisse auf Anforderung liefert.

```
choose 1 (choose 2 3)    liefert einen der Werte 1, 2 oder 3.
```

Zwei *Bedingungen* lassen sich mit `&` konjunktiv verknüpfen:

```
x == choose 1 (choose 2 3)    &    x+x == x*x    where x free
```

liefert

```
{x=2} success
```

Um freie Variablen in Ausdrücken bearbeiten zu können, gibt es zwei Auswertungsmechanismen:

- Beim **Narrowing** wird der Auswertungsmechanismus funktionaler Sprachen (**Reduktion**) um den der logischen Sprachen (**Resolution**) erweitert. Dabei wird der **Mustervergleich** durch die **Unifikation** der formalen und aktuellen Parameter eines Funktionsaufrufs ersetzt. Eine Operation, die durch Narrowing berechnet wird, heißt **flexibel** (*flexible*). Alle benutzerdefinierten Funktionen sind flexibel.

```
x == 2+2 where x free
```

-- Narrowing

```
{x=4} success
```

- Die **Residuation** verzögert Funktionsaufrufe mit uninstantiierten Variablen als Parameter solange, bis alle Variablen von parallel laufenden Prozessen gebunden wurden bzw. zumindest eine eindeutige Regelauswahl möglich ist. Leider berechnet die Residuation für manche Terme keine Lösung, selbst wenn die im Term enthaltenen freien Variablen nicht zur Lösung beitragen. Eine Operation, die durch Residuation berechnet wird, heißt **starr** (*rigid*). Die meisten vordefinierten Funktionen sind starr, z.B. die arithmetischen Funktionen.

```
x == 2+2 where x free
```

-- Residuation

```
*** Warning: there are suspended constraints
```

Das Narrowing ermöglicht **Muster mit Funktionen**; im Muster dürfen also außer Datenkonstruktoren und Variablen auch *flexible* Funktionen benutzt werden. Damit kann das letzte Element einer nicht-leeren Liste mit dem flexiblen Konkatenations-Operator `++` definiert werden als:

```
last (_ ++ [e]) = e
```

**Beispiel 8.12:** Bestimme alle Teillisten einer Liste, so dass das letzte Element `z` der Teilliste gerade das Doppelte des ersten Elements `x` der Teilliste ist:

```
doublesublist (_ ++ [x] ++ y ++ [z] ++ _) | 2*x == z    =    [x] ++ y ++ [z]
```

`doublesublist [3,6,2,1,4,5]` liefert also `[3,6]` und `[2,1,4]`.

# Anhang

## A Funktionen zur Codeerzeugung

### A.1 Berechnung von Adressen: $\text{code}_A$

$$\begin{aligned} \text{code}_A^\rho x &= \begin{cases} \text{loadc } j & , \text{ falls } \rho(x) = (G, j) & // \text{ Zugriff auf globale Variable } x \\ \text{loadrc } j & , \text{ falls } \rho(x) = (L, j) & // \text{ Zugriff auf lokale Variable oder Wert-Parameter } x \\ \text{loadr } j & , \text{ falls } \rho(x) = (R, j) & // \text{ Zugriff auf Referenz-Parameter } x \\ \text{loadmc } j & , \text{ falls } \rho(x) = (A, j) & // \text{ Zugriff auf Attribut } x \end{cases} \\ \text{code}_A^\rho f &= \begin{cases} \text{loadc } \_f & , \text{ falls } \rho(f) = (G, \_f) \text{ oder } \rho(f) = (N, \_f) & // \text{ Funktion bzw. Methode } f \\ \text{loadv } b & , \text{ falls } \rho(f) = (V, b) & // \text{ überschreibbare Methode } f \end{cases} \\ \text{code}_A^\rho (*e) &= \text{code}_W^\rho e \\ \text{code}_A^\rho (e.c) &= \text{code}_A^\rho e, \text{loadc } m, \text{add} \quad , \text{ wobei } m = \text{offsets}(t, c). & // \text{ Record-Komponente} \\ \text{code}_A^\rho (e \rightarrow c) &= \text{code}_A^\rho ((*e).c) = \text{code}_W^\rho e, \text{loadc } m, \text{add} \\ \text{code}_A^\rho e_1[e_2] &= \text{code}_A^\rho e_1, \text{code}_W^\rho e_2, \text{loadc } |t|, \text{mul}, \text{add} \end{aligned}$$

### A.2 Berechnung von Werten: $\text{code}_W$

$$\begin{aligned} \text{code}_W^\rho q &= \text{loadc } q & // q \text{ Konstante} \\ \text{code}_W^\rho x &= \text{code}_A^\rho x, \text{load} & // \text{ Variable auf der rechten Seite einer Wertzuweisung} \\ \text{code}_W^\rho f &= \text{code}_A^\rho f & // f \text{ Funktionsname} \\ \text{code}_W^\rho (e_1 + e_2) &= \text{code}_W^\rho e_1, \text{code}_W^\rho e_2, \text{add} & // \text{ binäre Operatoren} \\ \text{code}_W^\rho (-e) &= \text{code}_W^\rho e, \text{neg} & // \text{ unäre Operatoren} \\ \text{code}_W^\rho (x \leftarrow e) &= \text{code}_W^\rho e, \text{code}_A^\rho x, \text{store} & // \text{ Zuweisung eines Werts} \\ \text{code}_W^\rho (e_1 \leftarrow e_2) &= \text{code}_W^\rho e_2, \text{code}_A^\rho e_1, \text{store } |t| & // \dots \text{ vom Verbundtyp } t \\ \text{code}_W^\rho e &= \text{code}_A^\rho e, \text{load } m & // \text{ Typ von } e \text{ ist Verbund der Größe } m \\ \text{code}_W^\rho e &= \text{code}_A^\rho e & // e \text{ Feld} \\ \text{code}_W^\rho (\text{malloc}(e)) &= \text{code}_W^\rho e, \text{new} \\ \text{code}_W^\rho (\&e) &= \text{code}_A^\rho e \\ \text{code}_W^\rho f(e_1, \dots, e_n) &= \text{alloc } q, \quad \text{code}_W^\rho e_n, \dots, \text{code}_W^\rho e_1, \quad \text{mark}, \text{code}_W^\rho f, \text{call}, \text{slide } d \quad |t| \\ &\quad \text{mit } q = \max(|t| - |par|, 0), \quad d = \text{if } |t| \leq |obl| \text{ then } |par| - |obl| \text{ else } \max(|par| - |t|, 0), \\ &\quad \text{wobei } t \text{ der Rückgabetypp von } f, |obl| \text{ der Platzbedarf für die obligatorischen Parameter} \\ &\quad \text{und } |par| \geq |obl| \text{ der Platzbedarf für alle Parameter.} \\ \text{code}_W^\rho (\text{this}) &= \text{loadr } -3 \\ \text{code}_W^\rho e_1.f(e_2, \dots, e_n) &= \text{code}_W^\rho e_n, \dots, \text{code}_W^\rho e_2, \text{code}_A^\rho e_1, \text{mark}, \text{code}_A^\rho f, \text{call} \\ \text{Sei } |par| &\text{ der Platzbedarf für die aktuellen Parameter } e_2, \dots, e_n, |K| \text{ Platzbedarf für ein Objekt} \\ &\text{der Klasse } K, \_K \text{ die Anfangsadresse des Codes für den Konstruktor und } q = |par| + |K| - 1: \\ \text{code}_W^\rho \text{new } K(e_2, \dots, e_n) &= \text{loadc } |K|, \text{new}, \text{code}_W^\rho e_n, \dots, \text{code}_W^\rho e_2, \text{loads } |par|, \text{calld } \_K \\ \text{code}_W^\rho K(e_2, \dots, e_n) &= \text{alloc } |K|, \text{code}_W^\rho e_n, \dots, \text{code}_W^\rho e_2, \text{loadsc } q, \text{calld } \_K \end{aligned}$$



### A.3 Anweisungen: code

$\text{code}^\rho(e;) = \text{code}_W^\rho e, \text{alloc } -|e|$   
 $\text{code}^\rho(s\ ss) = \text{code}^\rho s, \text{code}^\rho ss \quad //\ s\ \text{Anweisung, } ss\ \text{Folge von Anweisungen}$   
 $\text{code}^\rho \varepsilon = \varepsilon \quad //\ \varepsilon\ \text{leere Folge von Anweisungen}$   
 $\text{code}^\rho(\text{if } e\ \text{then } s_1\ \text{else } s_2) = \text{code}_W^\rho e, \text{jumpz } A, \quad \text{code}^\rho s_1, \text{jump } B, \quad A: \text{code}^\rho s_2, B: \dots$   
 $\text{code}^\rho(\text{while } e\ \text{do } s) = A: \text{code}_W^\rho e, \text{jumpz } B, \text{code}^\rho s, \text{jump } A, B: \dots$   
 $\text{code}^\rho(\text{for } e_1; e_2; e_3\ \text{do } s) =$   
 $\quad \text{code}_W^\rho e_1, \text{alloc } -1, A: \text{code}_W^\rho e_2, \text{jumpz } B, \text{code}^\rho s, C: \text{code}_W^\rho e_3, \text{alloc } -1, \text{jump } A, B: \dots$   
 $\text{code}^\rho(\text{switch}(e)\{\text{case } 0: ss_0\ \text{break; case } 1: ss_1\ \text{break; } \dots \text{case } k-1: ss_{k-1}\ \text{break; default: } ss_k\})$   
 $\quad = \text{code}_W^\rho e, \quad \text{dup, loadc } 0, \text{geq, jumpz } A, \text{dup, loadc } k, \text{leq, jumpz } A, \text{jumpi } B,$   
 $\quad A: \text{alloc } -1, \text{loadc } k, \text{jumpi } B, \quad C_0: \text{code}^\rho ss_0, \text{jump } D, \dots, \quad C_k: \text{code}^\rho ss_k, \text{jump } D,$   
 $\quad B: \text{jump } C_0, \dots, \text{jump } C_k, \quad D: \dots$   
 $\text{code}^\rho \text{free}(e); = \text{code}_W^\rho e, \text{alloc } -|e|$

Adressumgebung und die nächste freie Relativadresse:

$\text{elab\_global}(\rho, n)(t\ x) = (\rho \oplus \{x \mapsto (G, n)\}, n + |t|) \quad //\ \text{Deklaration } t\ x$   
 $\text{elab\_formal}(\rho, z)(t\ x) = (\rho \oplus \{x \mapsto (L, z - |t|)\}, z - |t|)$   
 $\text{elab\_local}(\rho, n)(t\ x) = (\rho \oplus \{x \mapsto (L, n)\}, n + |t|)$

Sei  $k$  der maximale Speicherbedarf für Zwischenergebnisse der Funktion  $f$  auf dem Keller;  
 $l$  die Anzahl Speicherzellen für die lokalen Variablen,  $|obl|$  der Platzbedarf für die obligatorischen Parameter,  $|t|$  die Größe des Ergebnisses und  $r = 3 + \max\{|obl| - |t|, 0\}$ .

$\text{code}^\rho \text{return}; = \text{return } (|obl| + 3)$

$\text{code}^\rho \text{return } e; = \text{code}_W^\rho e, \text{storer } \rho(\text{ret})\ |t|, \text{return } r$

$\text{code}^\rho t\ f\ (\text{params})\ \{\text{locals } ss\} = \_f: \text{alloc } l, \text{enter } k, \text{code}^{\rho_f} ss, \text{return } r \quad //\ \text{Funktionsdef.}$

wobei  $\rho_f$  die Adressumgebung für  $f$  ist (ggf. um  $\text{ret}$  zu ergänzen):

$\rho_f = \quad \text{let } \rho = \rho \oplus \{f \mapsto (G, \_f)\} \quad //\ \_f\ \text{ist die Anfangsadresse des Codes von } f$   
 $\quad \text{in let } (\rho, \_) = \text{elab\_formals}(\rho, -2)\ \text{params}$   
 $\quad \quad \text{in let } (\rho, \_) = \text{elab\_locals}(\rho, 1)\ \text{locals}$   
 $\quad \quad \text{in } \rho$

$\text{code } p = \text{alloc } k, \text{enter } 3, \text{mark, loadc } \_f_n, \text{call, slide } (k-1)\ 1, \text{halt,}$

$\text{code}^{\rho_1} df_1, \dots, \text{code}^{\rho_n} df_n \quad //\ p\ \text{Programm}$

mit  $(\rho_0, k) = \text{elab\_globals}(1, \emptyset)\ dd \quad \text{für globale Deklarationen } dd$

$\text{code}^\rho K(t_2\ x_2, \dots, t_m\ x_m) : O(e_2, \dots, e_n)\{ss\} = \_K: \text{alloc } l, \text{enter } k, \text{code}_W^{\rho_1} e_n, \dots, \text{code}_W^{\rho_1} e_2,$   
 $\quad \text{loadr } -3, \text{mark, loadc } \_O, \text{call, initVirtual } K, \text{code}^{\rho'} ss, \text{return } r$

wobei  $\rho_1$  die Adressumgebung zur Auswertung der aktuellen Parameter des Konstruktors der Oberklasse  $O$  und  $\rho'$  die Adressumgebung innerhalb des Konstruktors  $K$  sind und

$\text{initVirtual } K \equiv \text{loadc } \_üM_K, \text{storem } 0, \text{alloc } -1$

## B Befehlssatz der virtuellen C-Maschine

| Befehl                | Implementierung                                                                                                                            |
|-----------------------|--------------------------------------------------------------------------------------------------------------------------------------------|
| <b>loadc</b> $q$      | $SP++$ ; $S[SP] \leftarrow q$ ;                                                                                                            |
| <b>load</b>           | $S[SP] \leftarrow S[S[SP]]$ ;                                                                                                              |
| <b>load</b> $m$       | <b>for</b> $i \leftarrow m-1$ ; $i \geq 0$ ; $i--$ <b>do</b> $S[SP+i] \leftarrow S[S[SP]+i]$ ; $SP \leftarrow SP+m-1$ ;                    |
| <b>loada</b> $q$ $m$  | <b>loadc</b> $q$ , <b>load</b> $m$ <span style="float:right">-- für <math>m=1</math> wird <math>m</math> weggelassen</span>                |
| <b>dup</b>            | $S[SP+1] \leftarrow S[SP]$ ; $SP++$ ;                                                                                                      |
| <b>loadrc</b> $j$     | $SP++$ ; $S[SP] \leftarrow FP+j$ ;                                                                                                         |
| <b>loadr</b> $j$ $m$  | <b>loadrc</b> $j$ , <b>load</b> $m$ <span style="float:right">-- für <math>m=1</math> wird <math>m</math> weggelassen</span>               |
| <b>loadmc</b> $q$     | <b>loadr</b> $-3$ , <b>loadc</b> $q$ , <b>add</b>                                                                                          |
| <b>loadm</b> $q$ $m$  | <b>loadmc</b> $q$ , <b>load</b> $m$ <span style="float:right">-- für <math>m=1</math> wird <math>m</math> weggelassen</span>               |
| <b>loadv</b> $b$      | $S[SP+1] \leftarrow S[S[S[SP-2]]+b]$ ; $SP++$ ;                                                                                            |
| <b>loadsc</b> $q$     | $S[SP+1] \leftarrow SP-q$ ; $SP++$ ;                                                                                                       |
| <b>loads</b> $q$      | <b>loadsc</b> $q$ , <b>load</b>                                                                                                            |
| <b>store</b>          | $S[S[SP]] \leftarrow S[SP-1]$ ; $SP--$ ;                                                                                                   |
| <b>store</b> $m$      | <b>for</b> $i \leftarrow 0$ ; $i < m$ ; $i++$ <b>do</b> $S[S[SP]+i] \leftarrow S[SP-m+i]$ ; $SP--$ ;                                       |
| <b>storea</b> $q$ $m$ | <b>loadc</b> $q$ , <b>store</b> $m$ <span style="float:right">-- für <math>m=1</math> wird <math>m</math> weggelassen</span>               |
| <b>storer</b> $j$ $m$ | <b>loadrc</b> $j$ , <b>store</b> $m$ <span style="float:right">-- für <math>m=1</math> wird <math>m</math> weggelassen</span>              |
| <b>storem</b> $q$ $m$ | <b>loadmc</b> $q$ , <b>store</b> $m$ <span style="float:right">-- für <math>m=1</math> wird <math>m</math> weggelassen</span>              |
| <b>jump</b> $A$       | $PC \leftarrow A$ ;                                                                                                                        |
| <b>jumpz</b> $A$      | <b>if</b> $S[SP] = 0$ <b>then</b> $PC \leftarrow A$ ; $SP--$ ;                                                                             |
| <b>jumpi</b> $A$      | $PC \leftarrow A + S[SP]$ ; $SP--$ ;                                                                                                       |
| <b>add</b>            | $S[SP-1] \leftarrow S[SP-1] + S[SP]$ ; $SP--$ ;                                                                                            |
| <b>sub</b>            | $S[SP-1] \leftarrow S[SP-1] - S[SP]$ ; $SP--$ ;                                                                                            |
| <b>mul</b>            | $S[SP-1] \leftarrow S[SP-1] \cdot S[SP]$ ; $SP--$ ;                                                                                        |
| <b>div</b>            | $S[SP-1] \leftarrow S[SP-1] / S[SP]$ ; $SP--$ ;                                                                                            |
| <b>mod</b>            | $S[SP-1] \leftarrow S[SP-1] \bmod S[SP]$ ; $SP--$ ;                                                                                        |
| <b>neg</b>            | $S[SP] \leftarrow -S[SP]$ ;                                                                                                                |
| <b>eq</b>             | $S[SP-1] \leftarrow S[SP-1] = S[SP]$ ; $SP--$ ;                                                                                            |
| <b>neq</b>            | $S[SP-1] \leftarrow S[SP-1] \neq S[SP]$ ; $SP--$ ;                                                                                         |
| <b>le</b>             | $S[SP-1] \leftarrow S[SP-1] < S[SP]$ ; $SP--$ ;                                                                                            |
| <b>leq</b>            | $S[SP-1] \leftarrow S[SP-1] \leq S[SP]$ ; $SP--$ ;                                                                                         |
| <b>gr</b>             | $S[SP-1] \leftarrow S[SP-1] > S[SP]$ ; $SP--$ ;                                                                                            |
| <b>geq</b>            | $S[SP-1] \leftarrow S[SP-1] \geq S[SP]$ ; $SP--$ ;                                                                                         |
| <b>and</b>            | $S[SP-1] \leftarrow S[SP-1] \text{ and } S[SP]$ ; $SP--$ ;                                                                                 |
| <b>or</b>             | $S[SP-1] \leftarrow S[SP-1] \text{ or } S[SP]$ ; $SP--$ ;                                                                                  |
| <b>not</b>            | $S[SP] \leftarrow \text{not } S[SP]$ ;                                                                                                     |
| <b>mark</b>           | $S[SP+1] \leftarrow EP$ ; $S[SP+2] \leftarrow FP$ ; $SP \leftarrow SP+2$ ;                                                                 |
| <b>call</b>           | $FP \leftarrow SP$ ; <b>var</b> $tmp \leftarrow PC$ ; $PC \leftarrow S[SP]$ ; $S[SP] \leftarrow tmp$ ;                                     |
| <b>calld</b> $a$      | <b>mark</b> , <b>loadc</b> $a$ , <b>call</b>                                                                                               |
| <b>callv</b> $b$      | <b>mark</b> , <b>loadv</b> $b$ , <b>call</b>                                                                                               |
| <b>enter</b> $k$      | $EP \leftarrow SP + k$ ; <b>if</b> $EP \geq HP$ <b>then</b> <b>error</b> („Stack Overflow“);                                               |
| <b>slide</b> $d$ $z$  | <b>if</b> $d > 0$ <b>then</b> <b>if</b> $z = 0$ <b>then</b> $SP \leftarrow SP-d$ ; <b>else</b> $\{SP \leftarrow SP-d-z$ ;                  |
|                       | <b>for</b> $i \leftarrow 0$ ; $i < z$ ; $i++$ <b>do</b> $\{SP++$ ; $S[SP] \leftarrow S[SP+d]\}$ <b>}</b>                                   |
| <b>return</b> $r$     | $PC \leftarrow S[FP]$ ; $EP \leftarrow S[FP-2]$ ; <b>if</b> $EP \geq HP$ <b>then</b> <b>error</b> („Stack Overflow“);                      |
|                       | $SP \leftarrow FP-r$ ; $FP \leftarrow S[FP-1]$ ;                                                                                           |
| <b>alloc</b> $q$      | $SP \leftarrow SP + q$ ;                                                                                                                   |
| <b>new</b>            | <b>if</b> $HP - S[SP] > EP$ <b>then</b> $\{HP \leftarrow HP - S[SP]$ ; $S[SP] \leftarrow HP$ ; <b>}</b> <b>else</b> $S[SP] \leftarrow 0$ ; |
| <b>halt</b>           |                                                                                                                                            |

## C Eine einfache Programmiersprache

### C.1 Grammatik nach Entfernung gemeinsamer Präfixe

```

1  <program>      → begin <statementlist> end
2  <statementlist> → <statement> <statementRest>
3.1 <statementRest> → ; <statementlist>
3.2                  | ε
4.1 <statement>   → ident ← <expression>
4.2                  | print <expression>
5  <expression>   → <sign> <termlist>
6.1 <sign>        → +
6.2                  | -
6.3                  | ε
7  <termlist>     → <term> <termRest>
8.1 <termRest>    → + <termlist>
8.2                  | - <termlist>
8.3                  | ε
9  <term>         → <factorlist>
10 <factorlist>   → <factor> <factorRest>
11.1 <factorRest> → * <factorlist>
11.2                  | / <factorlist>
11.3                  | ε
12.1 <factor>     → ident
12.2                  | number
12.3                  | ( <expression> )

```

### C.2 Berechnung der First- und Follow-Funktion

| X               | First(X)                   | Follow(X)                 |
|-----------------|----------------------------|---------------------------|
| <program>       | { begin }                  | { \$ }                    |
| <statementlist> | { ident, print }           | { end }                   |
| <statementRest> | { ;, ε }                   | { end }                   |
| <statement>     | { ident, print }           | { end, ; }                |
| <expression>    | { +, -, ident, number, ( } | { end, ;, ) }             |
| <sign>          | { +, -, ε }                | { ident, number, ( }      |
| <termlist>      | { ident, number, ( }       | { end, ;, ) }             |
| <termRest>      | { +, -, ε }                | { end, ;, ) }             |
| <term>          | { ident, number, ( }       | { end, ;, +, -, ) }       |
| <factorlist>    | { ident, number, ( }       | { end, ;, +, -, ) }       |
| <factorRest>    | { *, /, ε }                | { end, ;, +, -, ) }       |
| <factor>        | { ident, number, ( }       | { end, ;, +, -, *, /, ) } |

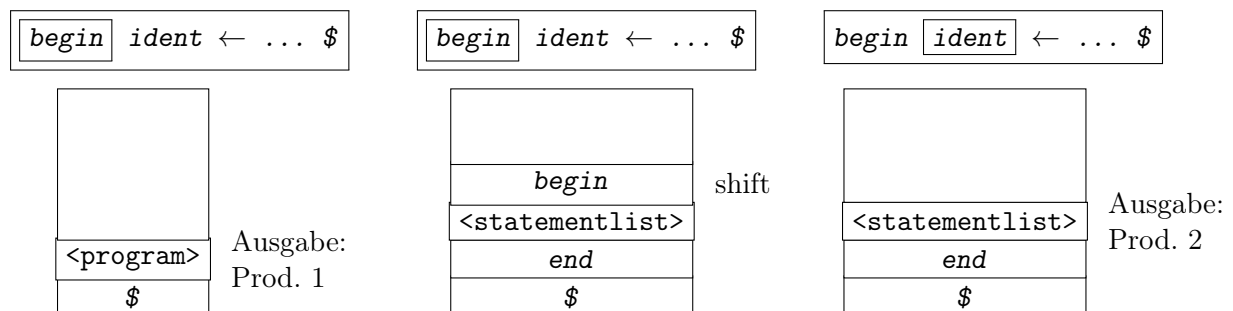
### C.3 Syntaxanalyse-Tabelle

|                 | <i>begin</i> | <i>print</i> | <i>ident</i> | <i>number</i> | (    | *    | /    | +    | -    | )    | ;    | <i>end</i> |
|-----------------|--------------|--------------|--------------|---------------|------|------|------|------|------|------|------|------------|
| <program>       | 1            |              |              |               |      |      |      |      |      |      |      |            |
| <statementlist> |              | 2            | 2            |               |      |      |      |      |      |      |      |            |
| <statementRest> |              |              |              |               |      |      |      |      |      |      | 3.1  | 3.2        |
| <statement>     |              | 4.2          | 4.1          |               |      |      |      |      |      |      |      |            |
| <expression>    |              |              | 5            | 5             | 5    |      |      | 5    | 5    |      |      |            |
| <sign>          |              |              | 6.3          | 6.3           | 6.3  |      |      | 6.1  | 6.2  |      |      |            |
| <termlist>      |              |              | 7            | 7             | 7    |      |      |      |      |      |      |            |
| <termRest>      |              |              |              |               |      |      |      | 8.1  | 8.2  | 8.3  | 8.3  | 8.3        |
| <term>          |              |              | 9            | 9             | 9    |      |      |      |      |      |      |            |
| <factorlist>    |              |              | 10           | 10            | 10   |      |      |      |      |      |      |            |
| <factorRest>    |              |              |              |               |      | 11.1 | 11.2 | 11.3 | 11.3 | 11.3 | 11.3 | 11.3       |
| <factor>        |              |              | 12.1         | 12.2          | 12.3 |      |      |      |      |      |      |            |

Die Spalten für  $\leftarrow$  und  $\$$  haben keine Einträge.

### C.4 Beispiel für eine Syntaxanalyse

Eingabe: *begin ident  $\leftarrow$  number \* ident + number; print (-ident) end \$*



Insgesamt wird nachstehende Folge von Produktionen ausgegeben bzw. ihre Nummern:

1, 2, 4.1, 5, 6.3, 7, 9, 10, 12.2, 11.1, 10, 12.1, 11.3, 8.1, 7, 9, 10, 12.2, 11.3, 8.3, 3.1, 2, 4.2, 5, 6.3, 7, 9, 10, 12.3, 5, 6.2, 7, 9, 10, 12.1, 11.3, 8.3, 11.3, 8.3, 3.2

## D Klassen von formalen Sprachen

| Sprachklasse                       | Verfahren                                                                                                                                                                                                                                                                                                                                                                                                  |                                                                                                                                                            |                    |
|------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------|
|                                    | erzeugend                                                                                                                                                                                                                                                                                                                                                                                                  | erkennend                                                                                                                                                  | algebraisch        |
| rekursiv aufzählbare Sprachen      | Grammatik<br>$P \subseteq V^* \times V^*$<br>$A \rightarrow a \mid BC \mid \varepsilon, AB \rightarrow AC$                                                                                                                                                                                                                                                                                                 | nichtdeterministische Turingmaschine<br>deterministische Turingmaschine<br>nichtdeterministischer Zweikellerautomat<br>deterministischer Zweikellerautomat |                    |
| kontextsensitive Sprachen          | monotone Grammatik <sup>ε</sup><br>$l \rightarrow r \in P \Rightarrow  l  \leq  r $<br>Kuroda-Normalform <sup>ε</sup><br>$A \rightarrow a, A \rightarrow BC, AB \rightarrow CD$<br>kontextsensitive Grammatik<br>$A \rightarrow a, A \rightarrow BC, AB \rightarrow AC$                                                                                                                                    | nichtdeterministischer linear-beschränkter Automat                                                                                                         |                    |
| wachsend kontextsensitive Sprachen | streng monotone Grammatik <sup>ε</sup><br>$l \rightarrow r \in P \Rightarrow  l  <  r $                                                                                                                                                                                                                                                                                                                    | nichtdeterministischer schrumpfender Zweikellerautomat<br>nichtdeterministischer längenreduzierender Zweikellerautomat                                     |                    |
| kontextfreie Sprachen              | kontextfreie Grammatik<br>$P \subseteq N \times V^*$<br>Chomsky-Normalform <sup>ε</sup><br>$P \subseteq N \times (T \cup N^2)$<br>$A \rightarrow a, A \rightarrow BC$<br>Greibach-Normalform <sup>ε</sup><br>$P \subseteq N \times TN^*$<br>quadratische Greibach-NF <sup>ε</sup><br>$P \subseteq N \times (T \cup TN \cup TN^2)$<br>Operator-Normalform<br>$P \subseteq N \times (T^* \cup (T^*NT)^*T^*)$ | nichtdeterministischer Kellerautomat                                                                                                                       |                    |
| deterministische kfr. Sprachen     | LR(k)-Grammatik                                                                                                                                                                                                                                                                                                                                                                                            | deterministischer Kellerautomat                                                                                                                            |                    |
| reguläre Sprachen                  | linkslineare Grammatik<br>$P \subseteq N \times (T^* \cup NT^*)$<br>rechtslineare Grammatik<br>$P \subseteq N \times (T^* \cup T^*N)$                                                                                                                                                                                                                                                                      | nichtdeterministischer endlicher Automat<br>deterministischer endlicher Automat                                                                            | reguläre Ausdrücke |

$V := N \cup T, \quad \varepsilon$  Sonderregel für Sprachen mit dem leeren Wort  $\varepsilon$



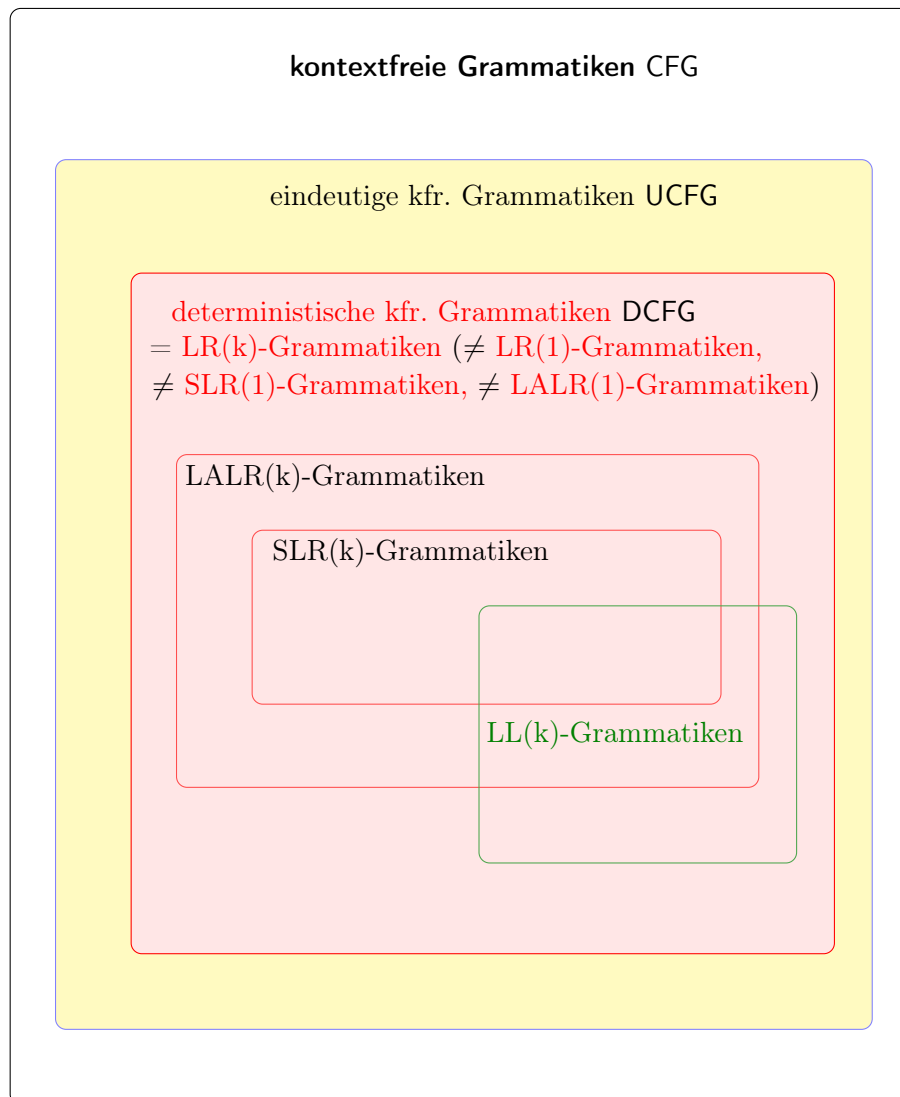


Abb. D.2: Mengeninklusionsdiagramm von Grammatikklassen.

- $P = \{S \rightarrow Aa \mid Bb, \quad A \rightarrow c, \quad B \rightarrow c\}$   
beschreibt eine SLR(1)-Grammatik, aber keine LR(0)-Grammatik.
- $P = \{S \rightarrow aAb \mid adc \mid bAc, \quad A \rightarrow d\}$   
beschreibt eine LALR(1)-Grammatik, aber keine SLR(1)-Grammatik.
- $P = \{S \rightarrow Aa \mid Bb \mid bAb \mid bBa, \quad A \rightarrow c, \quad B \rightarrow c\}$   
beschreibt eine LR(1)-Grammatik, aber keine LALR(1)-Grammatik.
- $P = \{S \rightarrow a \mid Sa\}$   
beschreibt eine LR(k)-Grammatik, aber keine LL(k)-Grammatik.
- $P = \{S \rightarrow a \mid aSa\}$   
beschreibt eine eindeutige kontextfreie Grammatik, aber keine LR(k)-Grammatik.

Ist folgendes Problem entscheidbar, wenn die Sprachen durch Grammatiken gegeben sind?

| Entscheidungsproblem                   |                                    | rek. aufzählbare Sprachen | kontextsensitive Sprachen | kontextfreie Sprachen | deterministische kfr. Spr. | reguläre Sprachen |
|----------------------------------------|------------------------------------|---------------------------|---------------------------|-----------------------|----------------------------|-------------------|
|                                        |                                    | RE                        | CSL                       | CFL                   | DCFL                       | REG               |
| Wortproblem                            | $w \in L(G) ?$                     | nein                      | ja                        | ja                    | ja                         | ja                |
| Endlichkeitsproblem                    | $L(G)$ endlich ?                   | nein                      | nein                      | ja                    | ja                         | ja                |
| Leerheitsproblem                       | $L(G) = \emptyset ?$               | nein                      | nein                      | ja                    | ja                         | ja                |
| Universalitäts-(Ausschöpfungs-)problem | $L(G) = T^* ?$                     | nein                      | nein                      | nein                  | ja                         | ja                |
| Mehrdeutigkeitssproblem                | $G$ mehrdeutig ?                   | nein                      | nein                      | nein                  | ja                         | ja                |
| Äquivalenzproblem                      | $L(G_1) = L(G_2) ?$                | nein                      | nein                      | nein                  | ja                         | ja                |
| Inklusionsproblem                      | $L(G_1) \subseteq L(G_2) ?$        | nein                      | nein                      | nein                  | nein                       | ja                |
| Schnittproblem                         | $L(G_1) \cap L(G_2) = \emptyset ?$ | nein                      | nein                      | nein                  | nein                       | ja                |

Ist eine Sprachklasse abgeschlossen gegenüber der angegebenen Operation?

| Abschlusseigenschaft               | rekursiv aufzählbare Sprachen | entscheidbare (rekursive) Sprachen | kontextsensitive Sprachen | wachsend kontextsensitive Spr. | kontextfreie Sprachen | eindeutige kfr. Sprachen<br>(nicht inhärent mehrdeutige Spr.) | deterministische kfr. Sprachen | reguläre Sprachen | endliche Sprachen |
|------------------------------------|-------------------------------|------------------------------------|---------------------------|--------------------------------|-----------------------|---------------------------------------------------------------|--------------------------------|-------------------|-------------------|
|                                    | $\Sigma_1$<br>RE              | $\Sigma_0$<br>REC                  | CSL                       | GCSL                           | CFL                   | UCFL                                                          | DCFL                           | REG               | FIN               |
| Homomorphismen $h$                 | ja                            | ja                                 | nein <sup>ε</sup>         | nein <sup>ε</sup>              | ja                    |                                                               | nein                           | ja                | ja                |
| inverse Homomorphismen $h^{-1}$    | ja                            | ja                                 | ja                        | ja                             | ja                    | ja                                                            | ja                             | ja                | nein              |
| Vereinigung $\cup$                 | ja                            | ja                                 | ja                        | ja                             | ja                    | nein <sup>*,•</sup>                                           | nein                           | ja                | ja                |
| Verkettung (Konkatenation) $\circ$ | ja                            | ja                                 | ja                        | ja                             | ja                    | nein                                                          | nein                           | ja                | ja                |
| Hülle $*$                          | ja                            | ja                                 | ja                        | ja                             | ja                    | nein                                                          | nein                           | ja                | nein              |
| Spiegelung (reversal) $R$          | ja                            | ja                                 | ja                        | ja                             | ja                    |                                                               | nein                           | ja                | ja                |
| Durchschnitt $\cap$                | ja                            | ja                                 | ja                        | nein <sup>*</sup>              | nein <sup>*</sup>     | nein <sup>*</sup>                                             | nein <sup>*</sup>              | ja                | ja                |
| Komplement $-$                     | nein                          | ja                                 | ja                        | nein                           | nein                  | nein                                                          | ja                             | ja                | nein              |
| Differenz $-$                      | nein                          | ja                                 | ja                        |                                | nein                  | nein <sup>*</sup>                                             | nein                           | ja                | ja                |

\* aber abgeschlossen, wenn die zweiten Operanden reguläre Sprachen sind

<sup>ε</sup> aber abgeschlossen, wenn die Homomorphismen  $\varepsilon$ -frei sind

• aber abgeschlossen, wenn die Operanden disjunkt sind



## E Arbeitsweise von Parser-Generatoren am Beispiel von yacc

yacc steht für *yet another compiler compiler* (STEPHEN C. JOHNSON, 1975) [22]. Ein yacc-Quellprogramm enthält Deklarationen von Tokenklassen, Prioritäten von Operatoren und die Grammatik in BNF mit zugeordneten semantischen Aktionen (Übersetzungsregeln). yacc konstruiert daraus ein C-Programm, das letztlich den Parser realisiert. Außerdem wird eine numerisch codierte Liste von Tokennamen (`y.tab.h`) erzeugt, die dann vom Scannergenerator `lex` zusammen mit der Definition der Tokenklassen zur Konstruktion eines Scanners benutzt wird. Beide Programme liefern C-Programme, die mit einem C-Compiler in einen Scanner und Parser übersetzt werden.

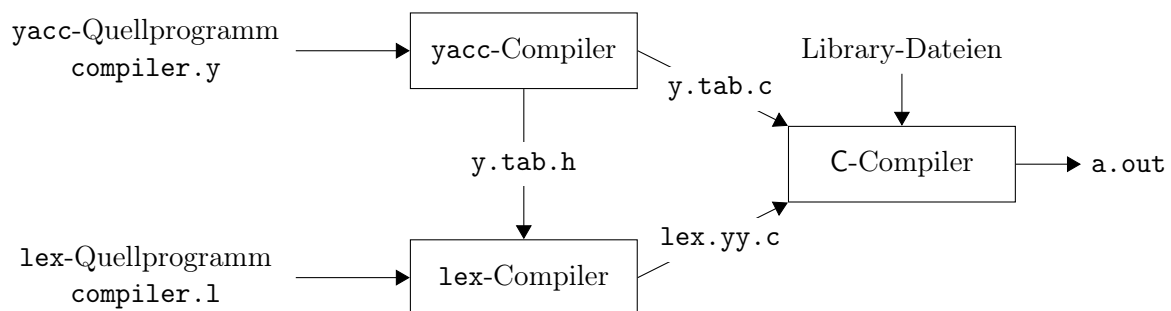


Abb. E.1: Zusammenspiel von `lex` und `yacc`.

Ein yacc-Programm hat folgenden strukturellen Aufbau:

```
%{
Deklarationen von Konstanten, Variablen usw.
}%
Deklarationen von Tokenklassen, Prioritäten usw.
%%
Grammatik in BNF mit zugeordneten semantischen Aktionen (Übersetzungsregeln)
%%
Hilfsfunktionen
```

Aus diesen Informationen stellt yacc den C-Code für einen Parser mit Namen `yyparse` her. Dieser Parser ruft zunächst den (evtl. mit `lex` erstellten) Scanner `yylex` auf und erhält als Rückgabewert die Tokenklasse des nächsten Tokens in der Eingabe. Die globale Variable `yylval` wird ebenfalls im Scanner gesetzt und enthält dann den Tokenwert.

Die Kommunikation zwischen Scanner und Parser beruht wesentlich auf gemeinsamen Token-Definitionen und auf der Variablen `yylval` vom Typ `YYSTYPE`. Dieser Typ wird festgelegt durch

```
%union { Liste von Deklarationen der Tokenwerte }
```

Folgender Typ fasst z.B. die möglichen Tokenwerte Integer und Floating-Point zusammen.

```
%union {
    int i;
    float f;
};
```

Danach können Tokenklassen definiert werden, z.B. durch

```
%token ADD MUL
%token <i> NUMBER LABEL    , wobei durch <i> der Typ von yylval spezifiziert wird.
```

Die Priorität der Operatoren wird durch die Reihenfolge der Deklarationen festgelegt, die Assoziativität der Operatoren durch verschiedene Schlüsselwörter:

|                                 |                                                        |
|---------------------------------|--------------------------------------------------------|
| <code>%right ASSIGN</code>      | ASSIGN ist rechtsassoziativ, niedrige Priorität        |
| <code>%left ADD SUB</code>      | ADD, SUB sind linksassoziativ, haben gleiche Priorität |
| <code>%left MUL DIV</code>      | MUL und DIV haben höhere Priorität als ADD und SUB     |
| <code>%nonassoc P_ASSIGN</code> | P_ASSIGN ist nicht assoziativ, hohe Priorität          |

Das Startsymbol der Grammatik wird definiert durch

`%start startsymbol`      sonst wird die linke Seite der ersten Produktion Startsymbol

Die Produktionen der Grammatik werden üblicherweise in BNF angegeben, z.B.

```
expr:
    expr '+' expr
    | expr '*' expr
    | '(' expr ')'
    | 'a'
;
```

Um ein ausführliches Protokoll der einzelnen Schritte des von yacc erzeugten Parsers zu erhalten, definiert man die globale Variable `yydebug` und setzt ihren Wert auf 1.

## E.1 Eine Beispieleingabe für yacc

```
%{
    /* In diesem Teil stehen alle Definitionen, die unverändert in
       das von yacc gelieferte C-Programm uebernommen werden sollen */
#include <math.h>
void out(char*); /* die Ausgabefunktion */
%}

%token NUM

%left '+' '-'
%left '*' '/'
%left NEG /* Negation - Vorzeichen Minus */
%right '^' /* Exponentiation ist rechts-assoziativ */

%start prog /* prog ist das Startsymbol der Grammatik */
%%

prog:
    /* leeres Wort */ { out("prog -> "); }
    | prog line      { out("prog -> prog line"); }
    ;

line:
    '\n' { out("line -> \\n"); }
    | exp '\n' { out("line -> exp"); }
    ;
```

```

exp:
    NUM                { out("exp -> NUM"); }
  | exp '+' exp        { out("exp -> exp + exp"); }
  | exp '-' exp        { out("exp -> exp - exp"); }
  | exp '*' exp        { out("exp -> exp * exp"); }
  | exp '/' exp        { out("exp -> exp / exp"); }
  | '-' exp %prec NEG  { out("exp -> -exp"); }
  | exp '^' exp        { out("exp -> exp ^ exp"); }
  | '(' exp ')'        { out("exp -> (exp)"); }
;

%%

/* Es wird angenommen, dass der Scanner beim Erkennen einer Zahl die
   Tokenklasse NUM und als Tokenwert eine Floating-Point Zahl im double Format
   zurueckgibt. Leerzeichen und Tabulatoren werden ueberlesen.
   Alle anderen ASCII-Zeichen werden direkt uebergeben. */

main()
{
    yyparse();
}

yyerror(s)                /* wird von yyparse bei Fehlern aufgerufen */
char *s;
{
    printf("%s\n", s);
}

void out(r)                /* Ausgabeprozedur */
char *r;
{
    printf("Reduziert mit Regel: %s\n", r);
}

```

## E.2 Semantische Aktionen und syntax-gesteuerte Übersetzung

Da bei der Konstruktion von Grammatiken für Programmiersprachen jedes nichtterminale Symbol für einen Begriff steht (z.B. E für einen arithmetischen Ausdruck (*expression*)), liegt es nahe, für die Übersetzung wichtige Eigenschaften dieses Begriffs an das nichtterminale Symbol zu heften. Diese Eigenschaften heißen **Attribute**.

Bestimmen sich die Werte eines Attributs eines nichtterminalen Symbols *A* nur aus Attributwerten von Symbolen, die im Ableitungsbaum *A* folgen, nennt man dieses Attribut **synthetisch**. Die Werte synthetischer Attribute werden von unten nach oben im Ableitungsbaum berechnet.

Eine Regel, nach denen diese Attribute mit Werten versehen werden, nennt man **semantische Regel** oder **semantische Aktion**. Diese Regeln werden an die Produktionen geheftet. Für jedes synthetische Attribut des Symbols auf der linken Seite einer Produktion gibt es eine Regel, die festlegt, wie der Wert dieses Attributs in Abhängigkeit der Werte anderer Attribute von Symbolen dieser Produktion bestimmt werden kann.

**Beispiel E.1:**

```

E → E(1) + E(2)    { E.Wert = E(1).Wert + E(2).Wert }
E → E(1) * E(2)    { E.Wert = E(1).Wert * E(2).Wert }
E → ( E(1) )        { E.Wert = E(1).Wert }
E → num            { E.Wert = lexValue }

```

Eine kontextfreie Grammatik mit solchen Attributen und Regeln heißt **syntax-gesteuertes Übersetzungsschema**. Dieses Übersetzungsschema arbeitet besonders gut mit einem LR-Parsingverfahren zusammen, daher hat man diesen Mechanismus auch in **yacc** eingebaut.

Jedes nichtterminale Zeichen hat genau *ein* synthetisches Attribut. Die synthetischen Attribute sind für jede Produktion durchnummeriert; \$\$ bezeichnet das Attribut des Symbols auf der linken Seite einer Produktion, \$1 das Attribut des ersten Symbols, \$2 das des zweiten Symbols auf der rechten Seite einer Produktion. Die semantische Aktion wird in **yacc** durch ein C-Codefragment beschrieben, in dem die „Namen“ \$\$, \$1, \$2 usw. vorkommen dürfen.

**Beispiel E.2:** (Fragment der yacc-Eingabe)

```

exp:
    exp '+' exp      { $$ = $1 + $3; }
  | exp '*' exp      { $$ = $1 * $3; }
  | '(' exp ')'      { $$ = $2; }
  | NUM              { $$ = $1; }           der Tokenwert wird ebenso übergeben

```

Die semantischen Regeln werden bei der Reduktion ausgeführt, dann steht die rechte Seite dieser Produktion oben auf dem Keller.

**Beispiel E.3:** Als Beispiel –auch für die Zusammenarbeit von **lex** und **yacc**– wird ein einfaches Taschenrechnerprogramm vorgestellt. Das Programm liest arithmetische Ausdrücke ein, prüft die Syntax und gibt den Wert des arithmetischen Ausdrucks zurück.

Hier zunächst die Eingabedatei **rechner.y** für **yacc**:

```

%{
#define YYDEBUG 1                                /* erlaubt erweiterte Fehlersuche */
#include <math.h>
int lineno = 1;
}%

%union {
    double d;                                     /* fuer Zahlen soll der Tokenwert vom Typ double sein */
};

%token <d> NUM
%type <d> exp                                     /* Datentyp des Attributs von exp ist double */
%left ADD SUB
%left MUL DIV
%left NEG   /* Negation - Vorzeichen Minus */
%right EXP   /* Exponentiation ist rechts-assoziativ */

%start prog                                       /* prog ist das Startsymbol der Grammatik */

%%

```

```

prog:
    | prog line
    ;

line:
    '\n' /* leere Eingabezeile */
    | exp '\n' { printf("\n\t%.10g\n", $1); }
    ;

exp:
    NUM { $$ = $1; }
    | exp ADD exp { $$ = $1 + $3; }
    | exp SUB exp { $$ = $1 - $3; }
    | exp MUL exp { $$ = $1 * $3; }
    | exp DIV exp { $$ = $1 / $3; }
    | SUB exp %prec NEG { $$ = -$2; }
    | exp EXP exp { $$ = pow($1, $3); }
    | '(' exp ')' { $$ = $2; }
    ;

%%

extern int yydebug; /* Fuer debug-Zwecke */

main(int argc, char **argv)
{ /* Hier kommt das Hauptprogramm hin */
    /* yydebug = 1; */ /* Wenn die debug-Option gewuenscht wird */
    yyparse();
    printf("\nEnde\n");
}

yyerror(s) /* wie oben angegeben */

```

Und die Eingabedatei `rechner.l` für `lex`:

```

%{
#include <stdio.h>
#include <stdlib.h>
#include "rechner.tab.h"

extern int lineno;
%}

%%

[ \t] { /* space und tab ueberlesen */ }
"_" { return SUB; }
"*" { return MUL; }
"/" { return DIV; }
"\^" { return EXP; }
"+" { return ADD; }
[0-9]+|[0-9]*\.[0-9]+ { sscanf(yytext, "%lf", &yyval.d); return NUM; }
\n { lineno++; return '\n'; }
. { return yytext[0]; }

```

## F Syntaktischer Zucker in Haskell

### F.1 Funktionsdefinitionen

Eine Gleichung mit Wächtern entspricht einem bedingten Ausdruck:

$$\begin{array}{l} f \ p \mid C_1 = E_1 \\ \mid C_2 = E_2 \\ \vdots \\ \mid C_n = E_n \\ \mid \text{otherwise} = E_{n+1} \end{array} \equiv \begin{array}{l} f \ p = \text{if } C_1 \text{ then } E_1 \\ \text{else if } C_2 \text{ then } E_2 \\ \vdots \\ \text{else if } C_n \text{ then } E_n \\ \text{else } E_{n+1} \end{array}$$

Mehrere Gleichungen mit Mustern entsprechen einer Gleichung mit einer Fallunterscheidung:

$$\begin{array}{l} f \ p_1 = E_1 \\ \vdots \\ f \ p_n = E_n \end{array} \equiv \begin{array}{l} f \ x = \text{case } x \text{ of} \\ \quad p_1 \rightarrow E_1 \\ \vdots \\ \quad p_n \rightarrow E_n \end{array}$$

Eine elementweise Definition einer Funktion entspricht einer Funktionsdefinition mit  $\lambda$ -Ausdruck.

$$f \ x = E \equiv f = \backslash x \rightarrow E$$

### F.2 Ausdrücke

Ein bedingter Ausdruck ist eine Fallunterscheidung über Wahrheitswerten:

$$\begin{array}{l} \text{if } C \\ \text{then } E_1 \\ \text{else } E_2 \end{array} \equiv \begin{array}{l} \text{case } C \text{ of} \\ \quad \text{True} \rightarrow E_1 \\ \quad \text{False} \rightarrow E_2 \end{array}$$

Hilfsdefinitionen mit **let** und **where** entsprechen einander:

$$\begin{array}{l} \text{let } D_1 \\ \vdots \\ D_n \\ \text{in } E \end{array} \equiv \begin{array}{l} E \\ \text{where } D_1 \\ \vdots \\ D_n \end{array}$$

Operatoren können wie Funktionen benutzt werden und umgekehrt:

$$\begin{array}{l} E_1 \otimes E_2 \equiv (\otimes) E_1 E_2 \\ E_1 \text{ `f` } E_2 \equiv f E_1 E_2 \end{array}$$

Listengeneratoren sind Kombinationen der Funktionen *map* und *filter*:

$$[T \mid x \leftarrow G, C] \equiv \text{map } (\backslash x \rightarrow T) (\text{filter } (\backslash x \rightarrow C) G)$$

Listenausdrücke mit Ellipsen entsprechen vordefinierten Aufzählungsfunktionen:

$$\begin{array}{l} [u..] \equiv \text{from } u \\ [u..o] \equiv \text{fromTo } u \ o \\ [u, n..] \equiv \text{fromBy } u \ (n-u) \\ [u, n..o] \equiv \text{fromByTo } u \ (n-u) \ o \end{array}$$

Listenausdrücke sind geschachtelte Aufrufe des Listenkonstruktors:

$$[e_1, \dots, e_n] \equiv e_1 : \dots : e_n : []$$

Ein Zeichenketten-Literal ist eine Liste von Einzelzeichen:

$$"z_1 \dots z_n" \equiv [z_1', \dots, z_n']$$

### F.3 Typ-Ausdrücke

Die Typkonstruktoren für Funktionsräume, Listen und Tupel können infix bzw. „mixfix“ benutzt werden:

$$\begin{array}{l} t \rightarrow s \equiv (\rightarrow) t \ s \\ [t] \equiv ([\ ]) t \\ (t_1, \dots, t_n) \equiv ((, \dots, )) t_1 \dots t_n \end{array}$$

# Literatur

## allgemeine Literatur zur Vorlesung:

- [1] K.C. Louden, *Programming Languages Principles and Practice*, PWS-Kent Publishing, 1993.  
Deutsch: *Programmiersprachen - Grundlagen, Konzepte, Entwurf*, Thomson Publishing, 1994.
- [2] B.J. MacLennan, *Principles of Programming Languages*, 2<sup>nd</sup> ed., Holt, Rinehart and Winston, 1987.
- [3] J.C. Mitchell, *Concepts in Programming Languages*, Cambridge University Press, 2003.
- [4] R. Sethi, *Programming Languages - Concepts and Constructs*, Addison Wesley, 1990.
- [5] P. van Roy und S. Haridi, *Concepts, Techniques, and Models of Computer Programming*, MIT Press, 2004.
- [6] R. W. Sebesta, *Concepts of Programming Languages*, 7<sup>th</sup> ed., Pearson International, 2006.
- [7] D.A. Watt, *Programmiersprachen, Konzepte und Paradigmen*, Hanser, 1996.

## zu Kapitel 1 – Historische Entwicklung der Programmiersprachen

- [8] J. Backus, *The History of FORTRAN I, II and III*, Annals of the History of Computing, 1 (1979), 21–37.
- [9] T.J. Bergin, R.G. Gibson (Eds.), *History of Programming Languages*, ACM Press, 1996.
- [10] P.E. Ceruzzi, *A History of Modern Computing*, The MIT Press, 1998.
- [11] D.E. Knuth und L. Trabb Pardo, *The Early Development of Programming Languages*, in: *A History of Computing in the Twentieth Century*, N. Metropolis, J. Howlett und G.-C. Rota (Eds.), ACM Press, 1980, 0-12-491650-3, 197–273.
- [12] J. McCarthy, *History of Lisp*, in R.L. Wexelblat (Ed.), *History of Programming Languages*, Academic Press, 1981.
- [13] W.F. Schmitt, *The UNIVAC Short Code*, Annals of History of Computing, 10 (1988), 7–18.

## zu Kapitel 3 und 4 – Zwischencodeerzeugung

- [14] R. Wilhelm, H. Seidl, *Übersetzerbau - Virtuelle Maschinen*, Springer, 2007.
- [15] B. Stroustrup, *The C++ Programming Language*, 2<sup>nd</sup> ed., Addison Wesley, 1993.

## zu Kapitel 2, 5 und 6 – Übersetzer und Interpretierer

- [16] A.V. Aho, M.S. Lam, R. Sethi und J.D. Ullman, *Compilers - Principles, Techniques, and Tools*, Pearson-Addison Wesley, 2<sup>nd</sup> Edition, 2007.  
Deutsche Ausgabe: *Compiler - Prinzipien, Techniken und Werkzeuge*, Pearson, 2008.
- [17] H. Albas, A. Nymeyer, *Practice and Principles of Compiler Building with C*, Prentice Hall, 1996.
- [18] A.A. Appel, *Modern Compiler Implementation in Java*, Cambridge University Press, 2002 (2. Auflage). (Das Buch gibt es auch in einer C- und einer ML-Version!)
- [19] B. Bauer und R. Höllerer, *Übersetzung objektorientierter Programmiersprachen*, Springer, 1998.
- [20] D. Grune, H.E. Bal, C.J.H. Jacobs und K.G. Langendoen, *Modern Compiler Design*, Wiley, 2000.
- [21] K. Jensen, N. Wirth, *PASCAL User Manual and Report*, Springer, 1976.
- [22] S.C. Johnson, *Yacc: Yet Another Compiler-Compiler*, Technical Report, 1975, <http://dinosaur.compilertools.net/yacc/>.
- [23] S. Kannapinn, *Eine Rekonstruktion der LR-Theorie zur Elimination von Redundanz mit Anwendung auf den Bau von ELR-Parsern*, Dissertation, TU Berlin, 2001.

- [24] O. Lecarme, M.-C. Peyrolle-Thomas, *Self-compiling Compilers: An Appraisal of their Implementation and Portability*, Software-Practice and Experience, 8 (1978), 149–170.
- [25] T. Mason und D. Brown, *lex & yacc*, O'Reilly & Associates Inc., 1991.
- [26] T.W. Parsons, *Introduction to Compiler Construction*, Computer Science Press, 1992.
- [27] T. Pittman und J. Peters, *The Art of Compiler Design*, Prentice Hall, 1992.
- [28] J. A. Robinson. *A machine-oriented logic based on the resolution principle*, J. ACM, 12 (1965), 23–41.
- [29] M.S. Paterson und M.N. Wegman, *Linear unification*, STOC'76: Proceedings of the eighth annual ACM symposium on theory of computing, ACM Press, 1976, 181–186.
- [30] S. Sippu, E. Soisalon-Soininen, *Parsing Theory, Vol. II: LR(k) and LL(k) Parsing*, Springer, 1990.
- [31] B. Teufel, S. Schmidt und T. Teufel, *C<sup>2</sup> Compiler Concepts*, Springer 1993.
- [32] R. Wilhelm, H. Seidl, S. Hack, *Übersetzerbau - Syntaktische und semantische Analyse*, Springer, 2012.
- [33] N. Wirth, *The Design of a PASCAL Compiler*, Software-Practice and Experience, 1 (1971), 309–333.

#### zu Kapitel 7 und 8 – Programmierparadigmen

- [34] T.W. Pratt und M.V. Zelkowitz, *Programming Languages - Design and Implementation*, Prentice-Hall, 2001.
- [35] D.P. Friedman, M. Wand und C.T. Haynes, *Essentials of Programming Languages*, MIT Press / McGraw-Hill Book Company, Cambridge, 1992.
- [36] W.F. Clocksin und C.S. Mellish, *Programming in Prolog*, Springer, 1987.
- [37] L. Sterling und E. Shapiro, *The Art of Prolog*, MIT Press, Cambridge, 1986.
- [38] M. Odersky, *Funktionale Programmierung*, in: P. Rechenberg, G. Pomberger (eds.), Informatik-Handbuch, Hanser, 4. Auflage, 2005, 599–612.
- [39] M. Block, A. Neumann, *Haskell Intensivkurs*, Springer, 2011.
- [40] R. Hindley. *The principal type scheme of an object in combinatory logic*, Transactions of the American Mathematical Society, 146 (1969), 29–60.
- [41] R. Milner. *A theory of type polymorphism in programming*, Journal of Computer and System Sciences, 17 (1978), 348–375.
- [42] J. Stolarek, *Understanding Basic Haskell Error Messages*, The Monad.Reader, Issue 20, 2012, 21–41, <https://themonadreader.files.wordpress.com/2012/08/issue20.pdf>
- [43] S. Antoy, M. Hanus, *Curry – A Tutorial Introduction*, 13.08.2014, [http://www-ps.informatik.uni-kiel.de/currywiki/\\_media/documentation/tutorial.pdf](http://www-ps.informatik.uni-kiel.de/currywiki/_media/documentation/tutorial.pdf)



# Index

- BACKUS, JOHN W. (1924–2007), 6, 8
- BAUER, FRIEDRICH L. (1924–2015), 8
- CHOMSKY, NOAM (1928–), 59
- CHURCH, ALONSO (1903–1995), 9
- COCKE, JOHN (1925–2002), 67
- COLMERAUER, ALAIN (1941–), 10
- CURRY, HASKELL B. (1900–1982), 12, 131
- DAHL, OLE-JOHAN (1931–2002), 10
- ECKERT, JOHN P. (1919–1995), 6
- FARBER, DAVID J. (1935–), 9
- GOLDBERG, ADELE (1945–), 11
- GOSLING, JAMES (1955–), 13
- GRISWOLD, RALPH E. (1934–2006), 9
- HOPPER, GRACE M. (1906–1992), 7
- HUDAK, PAUL R. (1952–2015), 12
- HUET, GÉRARD (1947–), 12
- JOHNSON, STEPHEN C., 157
- KASAMI, TADAO (1930–2007), 67
- KAY, ALAN C. (1940–), 11
- KEMENY, JOHN G. (1926–1992), 9
- KING, ADA (1815–1852), 12
- KLEENE, STEPHEN C. (1909–1994), 9
- KNUTH, DONALD E. (1938–), 5
- KURTZ, THOMAS E. (1928–), 9
- LEROY, XAVIER (1968–), 12
- MAUCHLY, JOHN W. (1907–1980), 6
- MCCARTHY, JOHN (1927–2011), 8, 9
- MILNER, ROBIN (1934–2010), 12
- NAUR, PETER (1928–2016), 8
- NYGAARD, KRISTEN (1926–2002), 10
- ODERSKY, MARTIN (1958–), 13
- PASCAL, BLAISE (1623–1662), 10
- PERLIS, ALAN J. (1922–1990), 8
- PEYTON JONES, SIMON (1958–), 12
- POLONSKY, IVAN P., 9
- RITCHIE, DENNIS (1941–2011), 11
- ROBINSON, JOHN ALAN (1928–), 117
- ROSSUM, GUIDO VAN (1956–), 13
- RUTISHAUSER, HEINZ (1918–1970), 8
- SAMELSON, KLAUS (1918–1980), 8
- SCHÖNFINKEL, MOSES (1889–1942), 131
- SCHMITT, WILLIAM F., 6
- STROUSTRUP, BJARNE (1950–), 12
- TRABB PARDO, LUIS, 5
- TURNER, DAVID (1946–), 12
- WADLER, PHILIP (1956–), 12
- WIJNGAARDEN, ADRIAAN VAN (1916–1987), 10
- WIRTH, NIKLAUS (1934–), 10, 20
- YOUNGER, DANIEL H., 67
- ZUSE, KONRAD (1910–1995), 6
- ANSI-C, 12, 36
- APL, 123
- Ada, 3, 9, 12, 52, 110, 112, 113, 115
- Algol 60, 3, 8–10, 34, 35, 112
- Algol 68, 3, 10, 112
- BASIC, 2, 9, 14
- C++, 4, 10–13, 34, 41, 50–54, 56–58, 110, 113
- CLOS, 9, 58
- COBOL, 3, 7–9, 112
- C#, 10, 50
- Caml, 12, 110, 123
- Common Lisp, 9
- Curry, 12, 146
- C, 3, 9, 11–13, 15, 17, 19, 24, 25, 27, 28, 30, 33–37, 39–47, 50, 53, 56, 57, 111, 113, 135, 150, 157, 160
- Dart, 13
- Delphi, 10
- Eiffel, 4, 10, 50, 51, 58, 112
- Erlang, 123
- Escher, 12
- FORTRAN, 3, 6, 7, 9, 10, 19, 20, 111–113
- FP, 4, 123
- Flavors, 10
- Go, 13
- Haskell, iv, 2, 4, 9, 12, 22, 34, 35, 52, 66, 110, 114, 119–121, 123–132, 135, 146, 162
- Hope, 4, 123
- ICON, 9
- ISWIM, 9, 66
- Id, 123
- Java, 2, 4, 9, 10, 13, 17, 22, 50–52, 54, 58, 59, 74, 110, 112, 113
- KRC, 12
- Lisp, 4, 9, 10, 14, 35, 114, 123
- Loops, 10
- ML, 9, 12, 110, 114, 123
- Mathematica, 123
- Miranda, 4, 9, 12, 66, 123
- Modula-2, 3, 9, 10, 110, 112
- Modula-3, 4
- NPL, 9
- OCaml, 4, 9, 12, 52, 123
- Oak, 13
- Oberon, 10
- Objective-C, 10
- Occam, 66
- PHP, 50
- PL/I, 9, 12, 35
- Pascal, 3, 5, 9, 10, 12, 17, 20–22, 34, 36, 41, 43, 50, 59, 65, 74, 110–112, 114
- Perl, 9, 58
- Plankalkül, 6
- Postscript, 4
- Prolog, iv, 4, 10, 11, 22, 136–146
- Python, 10, 12, 13, 17, 50, 58, 66
- Ruby, 50
- Rust, 13
- SASL, 12, 123
- SGML, 4
- SISAL, 123
- SML, 12, 123
- SNOBOL, 9
- SQL, 4, 14, 136
- Scala, 12, 13, 34, 58
- Scheme, 4, 9, 110, 114, 123
- Short Code, 6
- Simula, 4, 10, 11
- Smalltalk, 4, 10, 11, 17, 50, 110, 112, 114
- Standard ML, 4, 12
- Swift, 13
- Tcl, 9
- Turbo-Pascal, 10
- TypeScript, 13
- VHDL, 4
- XML, 4, 123
- XSLT, 123
- shell, 4, 14
- $\Rightarrow$ , 60
- $\lambda$ -Kalkül, 9
- $\rightarrow$ , 59
- $\vdash$ , 63
- $\varepsilon$  (leeres Wort), 59
- $\varepsilon$ -Übergang, 63
- A-Produktion, 59
- abgeleitete Klasse, 50
- ableitbares Wort, 60
- Ableitung, 60
- Ableitungsbaum, 60
- Ableitungsschritt, 60
- Abschluss, 134
- Abseitsregel, 66, 124
- abstrakte Klasse, 49, 51
- Abstraktion, 52
- accept-Operation, 82
- Adressaufruf, 34
- Adresse, 24, 25, 28
- Adressoperator, 47
- Adressumgebung, 18, 25
- Aggregationsfunktion, 130
- Aktionstabelle, 84, 85
- aktueller Zustand, 63
- akzeptierende Berechnung, 63
- akzeptierte Sprache, 66
- Akzeptor, 65
- algebraischer Datentyp, 125
- allgemeinster Unifikator, 116
- Alphabet, 59
- Alternative, 24, 30
- Analyse
  - , lexikale, 16
  - , semantische, 17

- , syntaktische, 17
- Analysephase, 16
- Anfangsadresse, 46
- Anfangskonfiguration, 63
- Anfangszustand, 63, 65
- Anfrage, 136
- angewandte Vorkommen, 35
- anonymer Parameter, 127
- ANTLR, 81
- Anweisung, 3, 24, 30
- Assemblersprachen, 3
- Attribut, 159
  - , synthetisches, 159
- Attribute, 49, 53
- Aufruf, 25, 39
- Aufrufbaum, 32
- Ausdruck, 4, 24
  - , logischer, 136
  - , mit freien Variablen, 146
- Ausführungszeit, 15
- Ausgabealphabet, 93
- Auslassung, 61, 65
- Auswertungsstrategie, 33, 122
  - , nicht-strikte, 33
  - , strikte, 33
  - , verzögerte, 33
- automatische Typ-Anpassung, 113
- axiom, 136
  
- Backtracking, 136, 146
- Backus-Naur-Form, 8, 61
- Basisklasse, 50
- Basistypen, 111
- Bedarfsauswertung, 33, 122, 133
- bedingter Sprung, 30
- Bedingungen, 146
- Befehl, 23
- Befehlsregister, 23
- Befehlszähler, 23
- binäre Fallunterscheidung, 30
- Bindungen, 146
- Bison, 84, 86
- Block, 35
- BNF, 61
  - , erweiterte, 61
- Bootstrapping, 19
- Bottom-up-Parser, 81
  - , tabellengesteuerter, 82
- Bottom-up-Parsing
  - , deterministisches, 81
- break, 31
  
- C-Maschine, 24
- Call-By-Name, 34
- Call-By-Need, 34, 122, 133
- Call-By-Reference, 34, 41
- Call-By-Value, 34, 41, 122
- charakteristischer endlicher Automat, 94
- Church-Rosser-Sprachen, 81
- Closed World Assumption, 136
- Code-Erzeugung, 18
  
- Codeoptimierung, 18
- Compiler, 2
- continue, 31
- Cross-Compiler, 15, 21
- cut-Prädikat, 144
  
- Daten-Konstruktor, 125
- Datenabstraktion, 4, 44, 49
- Datenbanksprachen, 4
- Datenfeld, 3, 44, 45
- Datenstruktur als Ganzes, 121
- Datenverbund, 3, 44
- Debugger, 15
- Deduktionssysteme, 136
- definierendes Vorkommen, 35
- Deklaration, 3, 32
- deklarierte Daten, 46
- dynamische Bindung, 35, 51, 110
- dynamische Eigenschaft, 15
- dynamischer Vorgänger, 36
  
- Effekt, 50
- Ein-/Ausgabeschnittstelle, 32
- eindeutige Grammatik, 61
- Eingabealphabet, 63, 65
- eingebettetes System, 15
- Einrückung, 66
- Endkonfiguration, 63
- endlicher Automat, 65
  - , deterministischer, 65
  - , nichtdeterministischer, 66
- Endmarke, 69, 72
- Endzustand, 63, 65
- Entscheidungsbaum, 106
- erfolgreicher Kellerinhalt, 88
- erfolgreiches Präfix, 95
- Ergebnis, 25
- erreichbar, 62
- erwartete Vorschau-Symbole, 69, 100
- erweiterte BNF, 61
- erzeugte Sprache, 60
- Expansionsübergänge, 91
- explizite Typ-Umwandlung, 113
  
- fact, 136
- Fakten, 136
- Fallunterscheidung, 24
- Fehlerbehandlung, 18
- Fehlermeldung, 18, 73, 76
- Feld, 45, 121
  - , statisch, 45
- finale Stadium, 90
- First-Funktion, 68
  - , Berechnung, 70, 71
- flex, 66
- Follow-Funktion, 69
  - , Berechnung, 70, 71
- formale Sprache, 59
- Formelaufruf, 34
- Formelaufruf mit Ergebnisteilhabe, 34
  
- Frage, 136
- freie Variable, 35, 138, 146
- Funktion, 3, 25
  - , überlagert, 113
  - , flexible, 147
  - , rekursive, 25
  - , starre, 147
- Funktional, 122
- Funktionale, 146
- funktionale Abstraktion, 3, 25, 49
- funktionale Programmierung, 121
- Funktionsaufruf, 41
  - , überversorgt, 133
  - , unterversorgt, 133
- Funktionsinvertierung, 146
- Funktionsymbol, 115
- Funktionszeiger, 132
  
- gebundene Variable, 35
- gemischte Berechnung, 14
- Generizität, 49, 52
- gleichberechtigte Datenobjekte, 122
- Gleichheitsbedingung, 147
- globale Variable, 35, 36
- goal, 136
- Grammatik
  - , LALR(1), 84
  - , LL(1), 69
  - , LR(1), 83
  - , kontextfreie, 59
  - , eindeutige, 61
  - , mehrdeutige, 61
  - , reduzierte, 62
  - , erweiterte, 90
  - , reguläre, 63
- Griff, 81, 95
- gültige Parseroperation, 88
- gültiges Stadium, 95
- Gültigkeitsregel, 35
  
- Halde, 23, 46
- Haldenüberlauf, 47
- Haldenzeiger, 23, 46
- hängender Zeiger, 48
- happy, 85
- Hardware-Beschreibungssprachen, 4
- Haskell
  - , Liste, 125
  - , Tupel, 124
  - , Wächter, 127
  - , ++, 125
  - , :, 125
  - , head, 125
  - , init, 125
  - , last, 125
  - , tail, 125
- Hauptausführungszyklus, 23
- Hindley-Milner-Typsystem, 12
- Horn-Klausel, 4, 137, 145
  
- Implikation, 136
- Index, 45
- Informationskapselung, 49

- initiales Stadium, 90
- Inkarnation, 25, 32, 35
- Inkarnationsweg, 32
- Instantiierung, 52
- Instanz, 116
- Instanzen einer Objektklasse, 50
- Interpretierer, 2, 14, 78
- item, 90
- JavaCC, 81
- jflex, 66
- Keller, 23, 25
- Kellerüberlauf, 31, 46
- Kellerrahmen, 36
- Kellerautomat, 62
  - , deterministischer, 63
  - , mit Ausgabe, 93
- Kellerzeiger, 23
- Klasse, 4, 10, 50, 53
- Komponente, 44
- Konfiguration, 63, 74, 82, 93
- Konfigurationsübergang, 63
- Konflikt
  - , reduce/reduce, 86
  - , shift/reduce, 86
- Konstante, 115
- Konstantentabelle, 43
- Konstruktor, 53
- kontextfreie Grammatik, 59
- kontextfreie Sprache, 60
- $L(G)$ , 60
- LALR(1)-Grammatik, 84
- LALR(1)-Parser, 104
- Länge eines Wortes, 59
- Laufzeit, 15
- Laufzeitkeller, 36
- Laufzeitsystem, 23
- Lebende Objekte kopieren, 48
- lebendige Inkarnation, 33
- Lebensdauer, 35, 46
- leere Liste, 125
- leeres Wort, 59
- Leseübergänge, 91
- lex, 66
- Lexem, 16
- lexikale Analyse, 16
- linksableitende Syntaxanalyse, 67
- Linksableitung, 60, 67
- Linksparser, 93
- Linksreduktion, 61, 81
- linksreduzierende Syntaxanalyse, 81
- linksrekursive Produktion, 68
- Liste, 121
- Literal, 145
- LL(1)-Grammatik, 69
- LL-Parser, 72
- logische Ausdrücke, 136
- logische Programmierung, 136
- lokale Variable, 35, 36
- lookahead, 67
- LR(1)-Grammatik, 83
- LR(1)-Parser, 100
- LR-Parser, 83
- Map-Filter-Reduce, 130
- Marke, 3, 30
- Markieren und Löschen, 48
- Maschine
  - , reale, 22
  - , virtuelle, 22
- Maschinensprache, 2
- mehrdeutige Grammatik, 61
- mehrfache Beerbung, 58
- Merkmale, 49
- Methoden, 53
- Monaden, 132
- Multimethode, 51
- Muster, 122
  - , mit Funktionen, 147
- Mustervergleich, 9, 122, 126, 146, 147
- Mutterspracheneffekt, 2
- Name, 3, 24, 32, 35
- namenlose Daten, 46
- Namensäquivalenz, 112
- Narrowing, 147
- nativer Code, 22
- Nebenwirkung, 3, 28
- nicht-prozedurale Sprache, 136
- nicht-strikte Auswertung, 33
- Nichtterminalsymbol, 59
  - , erreichbar, 62
  - , produktiv, 62
- Normalform, 4
- Oberklasse, 50
- Objekt, 49
- Objekt-Methoden, 49
- Objekterzeugung, 57
- Objektklasse, 49, 50
- occur check, 117
- Operandenteil, 2
- Operationsteil, 2
- Operator, 127
- organisatorische Zellen, 36, 146
- Parameter, 25
- Parameterübergabe, 34
  - , by name, 34
  - , by need, 34, 122
  - , by reference, 34, 41
  - , by value, 34, 41
- parametrische Polymorphie, 123
- Parchmann-Graphen, 70
- Parser, 17
  - , LL, 72
  - , LR, 83
- Parser-Generator, 81, 83
- partielle Applikation, 131
- partielle Auswertung, 14
- plattformunabhängig, 22
- polymorphe Funktion, 114
- Polymorphie, 51
  - , Inklusions-, 51
  - , Teiltyp-, 51
  - , ad hoc, 113
  - , parametrische, 51, 128
- Portierbarkeit, 22
- Portierung, 21
- Potenzmengenkonstruktion, 97
- Prädikat, 136, 145
- Prädikatenlogik, 4
- Präfix, 67
- präfixfrei, 69
- Pragmatik, 1
- Produktion, 59
  - , linksrekursive, 68
- produktiv, 62
- Programmablauf, 24
- Programmiersprache, 3
- Programmierung
  - , funktionale, 121
  - , logische, 136
- Programmspeicher, 23
- Prozedur, 3, 32
- Prozesse, 4
- Quellprogramm, 15
- Quellsprache, 15
- query, 136
- Rücksetzverfahren, 138
- Rahmenzeiger, 23, 36
- reale Maschine, 22
- Rechtsableitung, 60
- rechtsinvariant, 89
- Rechtsparser, 93
- Redex, 33
- reduce-Operation, 82
- reduce/reduce-Konflikt, 86
- Reduktion, 4, 147
- Reduktionsschritt, 61
- Reduktionsübergang, 91
- referentielle Transparenz, 121
- Regel, 59, 136
- Register
  - , COP, 54
  - , EP, 32
  - , FP, 23, 36
  - , HP, 23, 46
  - , IR, 23
  - , PC, 23
  - , SP, 23
- reguläre Grammatik, 63
- reguläre Sprache, 63
- regulärer Ausdruck, 64
- rekursive Datenstrukturen, 44, 121, 146
- Relativadresse, 26
- Residuation, 147
- Resolution, 4, 138, 147
- Rückgabewert, 37
- Rücksetzpunkt, 145
- Rücksetzzeiger, 145
- Rücksprungadresse, 37

- rule, 136
- Rumpf, 32
- s-Grammatik, 67
- Sackgasse, 61, 83, 88
- Sandkasten-Prinzip, 22
- Satzform, 60
- Scanner, 16, 66
- Scanner-Generator, 66
- Schleife, 24
- Schlussregeln, 137
- Schnittstelle, 52
- Schrankenzeiger, 32
- selektive Änderungen, 121
- Semantik, 1
- semantische Aktion, 159
- semantische Analyse, 17, 110
- semantische Regel, 159
- shift-Operation, 82
- shift/reduce-Konflikt, 86
- Sichtbarkeitsregel, 35
- Signatur, 128
- SLR(1)-Parser, 104
- Speicherbelegungsfunktion, 25
- Speicherbereinigung, 9, 46, 48
- Sprache, 60
  - , formale, 59
  - , kontextfreie, 60
  - , reguläre, 63
- Sprachprozessor, 2
- Sprung, 24
  - , bedingter, 30
  - , unbedingter, 30
- Sprungtabelle, 84, 85
- Spur, 146
- Spurzeiger, 146
- Stadium, 90
  - , gültiges, 95
  - , finales, 90
  - , initiales, 90
  - , vollständiges, 90
- Stadiumautomat, 91
- Startsymbol, 59
- statische Bindung, 35
- statische Eigenschaft, 15
- statischer Vorgänger, 36
- Stelligkeit, 115
- strikt im *i*-ten Argument, 133
- strikte Auswertung, 33, 122
- Striktheitsanalyse, 133, 134
- Struktur, 44
- Strukturäquivalenz, 112
- Substitution, 115
  - , zusammengesetzte, 116
- Substitutionsregel, 75
- Suchstrategie, 146
- Symbol
  - , nichtterminales, 59
  - , terminales, 59
- Symboltabelle, 16, 18
- syntaktische Analyse, 17
- Syntax, 1
- Syntaxanalysator, 17
- Syntaxanalyse, 61
  - , linksableitende, 67
  - , linksreduzierende, 81
- Syntaxanalyse-Tabelle, 73, 82
- Syntaxbaum, 17
- Syntaxgraph, 74
  - , deterministischer, 76
- Synthesephase, 16
- synthetisches Attribut, 159
- T-Diagramm, 19
- Teiltyp, 49, 51
- Teiltypregel, 51
- Terme, 115
- Termersetzung, 33
- Terminalsymbol, 59
  - , nützlich, 62
- Tiefensuche, 139
- Token, 16
- Tokenizer, 16
- Tokenklasse, 16
- Tokenwert, 16
- Top-down-Parser
  - , tabellengesteuerter, 72
- Top-down-Parsing
  - , deterministisches, 67
- Typ, 50, 110
  - , Anpassung, 51, 113
  - , Ausdruck, 111
  - , Basis, 111
  - , Größe, 25
  - , Inferenz, 110
  - , Konstruktor, 111
  - , Prüfung, 110
    - , dynamische, 110
    - , starke, 110
    - , statische, 110
  - , System, 111
  - , Variable, 114
  - , Äquivalenz, 112
    - , Namensäquivalenz, 112
    - , Strukturäquivalenz, 112
    - , mit Typvariablen, 114
  - , überlagert, 113
  - , polymorph, 114
  - , unmittelbarer, 51
- Typ-Ausdruck, 111
- Typ-Konstruktor, 111, 125
- Typ-System, 111
- Typinferenz, 119, 123
- Typklasse, 128
- Typvariable, 114, 123, 128
- Übergangsfunktion, 65
- Übergangsrelation, 63
- Übersetzer, 2, 14
- Übersetzungsfunktion, 27
- Übersetzungsschema
  - , syntax-gesteuertes, 160
- Übersetzungszeit, 14
- Umgebung, 133
- unbedingter Sprung, 30
- unendliche Datenstruktur, 129, 146
- Unifikation, 114–116, 138, 145–147
- Unifikator, 116
  - , Berechnung, 117
  - , allgemeinsten, 116
- unifizierbar, 116
- unmittelbare Objekte, 51
- unmittelbarer Typ, 51
- Unterklasse, 50, 51
- Variable, 24, 115, 121
  - , frei, 35
  - , gebunden, 35
  - , global, 35
  - , lokal, 35
- Verbund, 44
- Vererbung, 4, 49, 50
- Vererbungshierarchie, 49, 50
- Vergangenheit, 90
- vergangenheitsreduzierter Automat, 106
- Verweiszählung, 48
- verzögerte Auswertung, 33
- virtuelle Maschine, 22
- vollständiges Stadium, 90
- von-Neumann-Rechner, 3
- Voraussetzung, 50
- Vorbereitung, 14
- Vorschau-Symbol, 67
- Wert, 24, 28
- Wertaufruf, 34
- Wertzuweisung, 28
- while-Schleife, 31
- Wiederholung, 61
- Wijngaarden-Grammatik, 10
- Wort
  - , ableitbares, 60
  - , leeres, 59
- Wortgröße, 43
- Wortproblem, 67
- yacc, 84, 86, 157
- Zeiger, 46, 121
- Zeiger-Darstellung, 115
- Zielprogramm, 15
- Zielsprache, 15
- Zukunft, 90
- Zukunftsstadium, 106
- Zustand, 3, 24, 62, 65
- Zwischencode-Erzeugung, 17