

2. Vom Programm zum lauffähigen Code

2.1. Lader und Binder

2.2. Übersetzung höherer Programmiersprachen

2.3. Übersetzerstruktur

2.4. Kontextfreie Grammatiken

2.5. Lexikalische Analyse

2.6. Syntaktische Analyse

2.7. Semantische Analyse

2.8. Code-Erzeugung

2.9. Zusammenfassung (Kapitel 2)

2.1. Lader und Binder

2.1.1. Funktionalität eines Laders

2.1.2. Übersetzen und Laden

2.1.3. Die allgemeine Lademethode

2.1.4. Der Absolutprogramm-Lader

2.1.5. Herstellung externer Referenzen

2.1.6. Verschiebende Lader (Relocating Loaders)

2.1.7. Bindende Lader

2.1.8. Trennung von Binde- und Ladevorgang

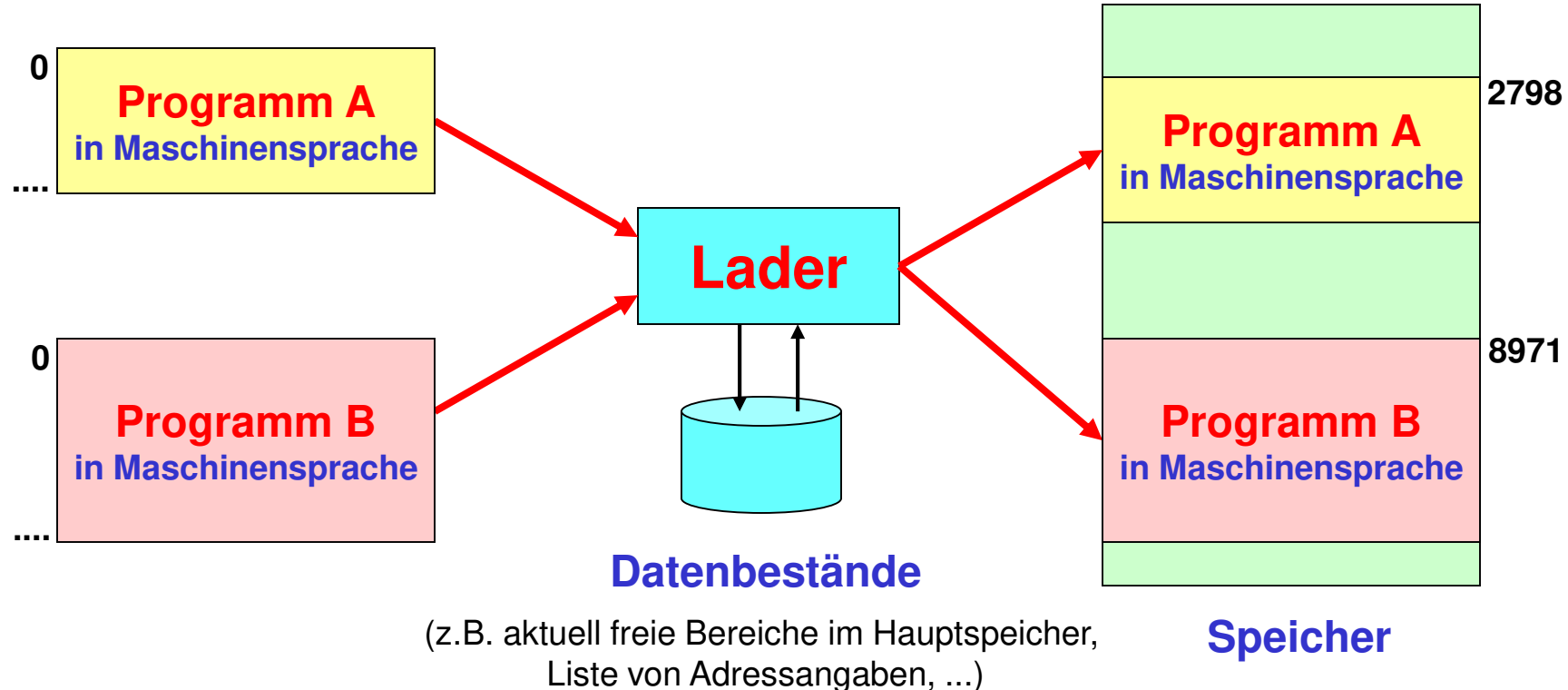
2.1.9. Dynamisches Binden und Laden

2.1.1. Funktionalität eines Laders

Die vom Assembler bzw. vom Compiler erzeugten **Programme in Maschinsprache** sind **noch nicht unmittelbar lauffähig**:

➡ Es entsteht lediglich ein Programm, das **lauffähig gemacht werden kann**.

Ein „**Lader**“ bereitet das Programm in Maschinsprache für die Ausführung vor und leitet die Ausführung ein.



Funktionalität eines Laders (2)

Für den Lader ergeben sich vier wesentliche Funktionen:

1. **Allocation** (Zuweisen)

Zuweisung von Speicherplatz.

2. **Relocation** (Verschieben)

Umrechnung aller Adressangaben entsprechend der zugewiesenen Speicherbereiche.

3. **Linking** (Binden)

Herstellung der externen Referenzen (Adressbezüge) zwischen den Programmteilen.

4. **Loading** (Laden)

Speicherung von Programm und Daten.

Anschließend wird das geladene Programm gestartet.

Im Folgenden werden einige Lademethoden betrachtet.

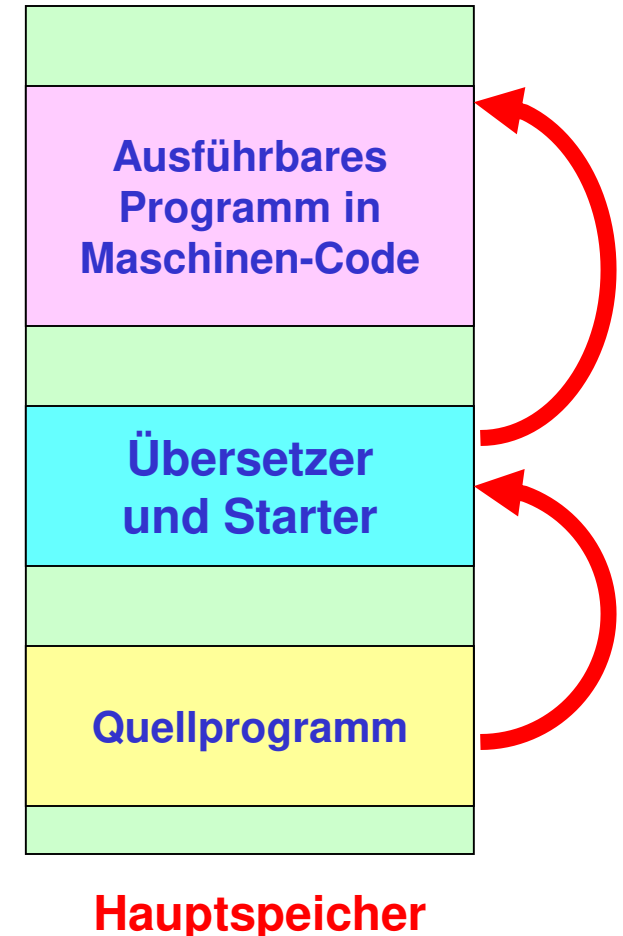
2.1.2. Übersetzen und Laden

Bei dieser Variante schreibt der **Übersetzer** bzw. der Assembler den **Objekt-Code unmittelbar in den Hauptspeicher**. Nach Abschluss dieses Vorgangs erfolgt ein Sprung zum Programm.

Der „**Lader**“ reduziert sich hier zum „**Starter**“.

Nachteile:

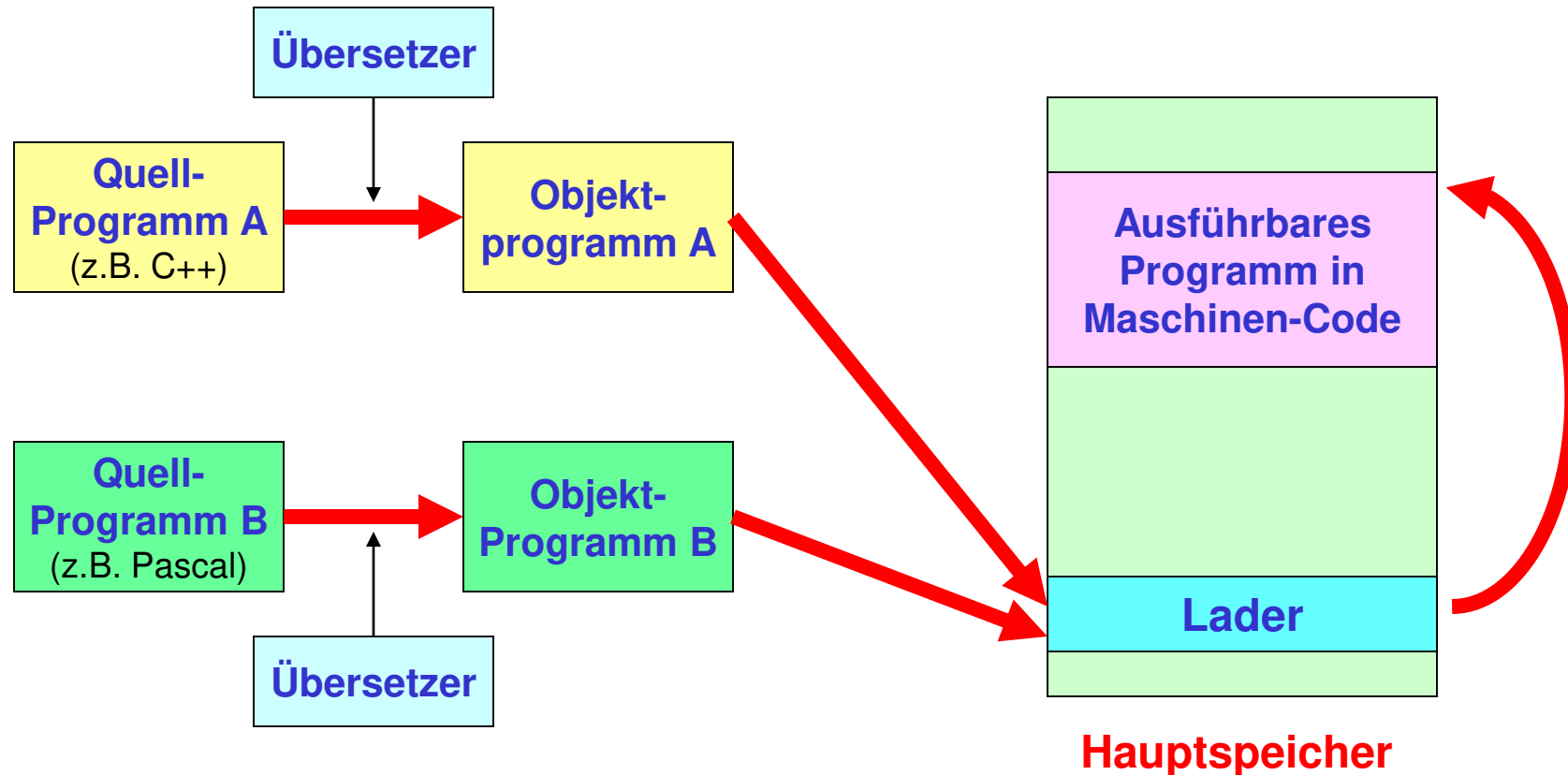
- Blockierung von Speicherzellen durch **gleichzeitige Einlagerung des Übersetzers**.
- Erneute **Übersetzung für jeden Programmlauf**.
- Kaum lösbare Probleme bei der **Verarbeitung mehrerer Segmente** (ggf. mit unterschiedlichen Quellsprachen).
Koordination mehrerer Übersetzer erweist sich in der Praxis meist als unlösbares Problem.



2.1.3. Die allgemeine Lademethode

Die **Trennung von Übersetzen und Laden** kann alle genannten Nachteile beheben.

Die Kooperation von Programmen in verschiedenen Quellsprachen setzt aber **„kompatible“ Behandlung externer Referenzen** (= Zugriffe auf Daten oder Programmcode anderer Programme) **VORAUSS**.



2.1.4. Der Absolutprogramm-Lader

Der **einfachste Typ eines Laders**

- **liest** das Objektprogramm
- **speichert** es **in** die **vom Übersetzer vorbestimmte Position**.
(Adressen wurden schon vom Übersetzer bzw. Assembler festgelegt)

Vorteile:

- Großer verfügbarer Hauptspeicher durch Auslagerung des **Übersetzers**.
(sehr kleiner Lader).
- **Einfache** Implementierung.

Nachteile:

- **Keine Flexibilität** in der **Speicherverwaltung**.
Ggf. ist der vorgesehene Speicherplatz schon belegt. Lösbares Problem bei Adressierung mit Basisregister.
- Schwierige Realisierung **externer Referenzen**.
Diese müssen bereits vom Übersetzer aufgelöst werden.

2.1.5. Herstellung externer Referenzen

Externe Referenzen werden durch **Pseudo-Operationen** möglich:

- **Angabe einer Liste der Symbole**, die

- zwar **referiert**,
- aber **nicht definiert** werden.

Die Definition dieser Symbole erfolgt in anderen Segmenten.

„Fremde
Symbole“

- **Angabe einer Liste der Symbole**, die

- im betreffenden Segment **definiert** werden und
- aus anderen Segmenten **referiert werden können**.

„Eigene
Symbole“
(für andere
Programme
zugänglich)

Eine Realisierung kann dadurch erfolgen, dass der Übersetzer dem Lader die **Symbol-Listen der jeweiligen Segmente** mitteilt.

Der Lader kann dann die externen Referenzen herstellen.

2.1.6. Verschiebende Lader (Relocating Loaders)

Bei Einsatz eines verschiebenden Laders wird die **Ablage-Adresse** des Objektprogramms **erst zum Ladezeitpunkt bestimmt** (i.d.R. vom Betriebssystem).

Folge:

Der Lader muss **Speicheradressen** im Objektprogramm **ggf. neu berechnen**.

Achtung:

Der Assembler bzw. der Übersetzer muss dann eine „**Verschiebetabelle**“ anlegen, welche die Position relativer (= vom Programmanfang abhängiger) **Adressen** angibt.



An diesen Stellen muss korrigiert werden.

Beispiel:

Der Assembler habe alle relativen Adressen ab Zelle 0 berechnet.

Die **wirkliche Anfangsadresse** (z.B. 1024) wird später an allen relevanten Stellen **hinzuaddiert**.

Bei direkter (= absoluter Adressierung) müssen alle Adressen umgerechnet werden.

Bei Basisregister und Displacement wird der **Inhalt des Basisregisters modifiziert**.

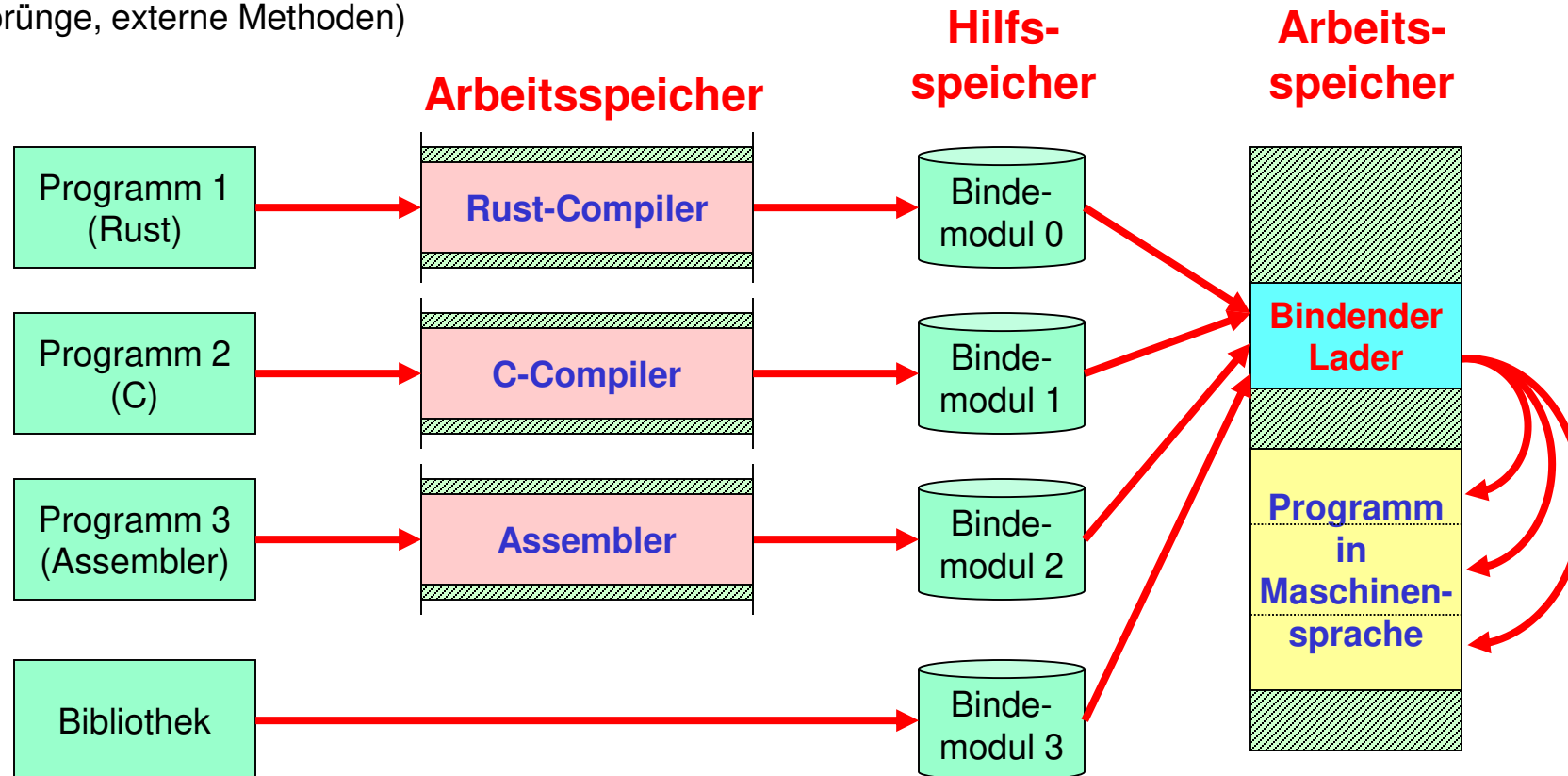
In diesem Fall entfällt auch die Verschiebetabelle.

2.1.7. Bindende Lader

Bindende Lader sind die am weitesten verbreitete Version des Laders.

Der bindende Lader gestattet

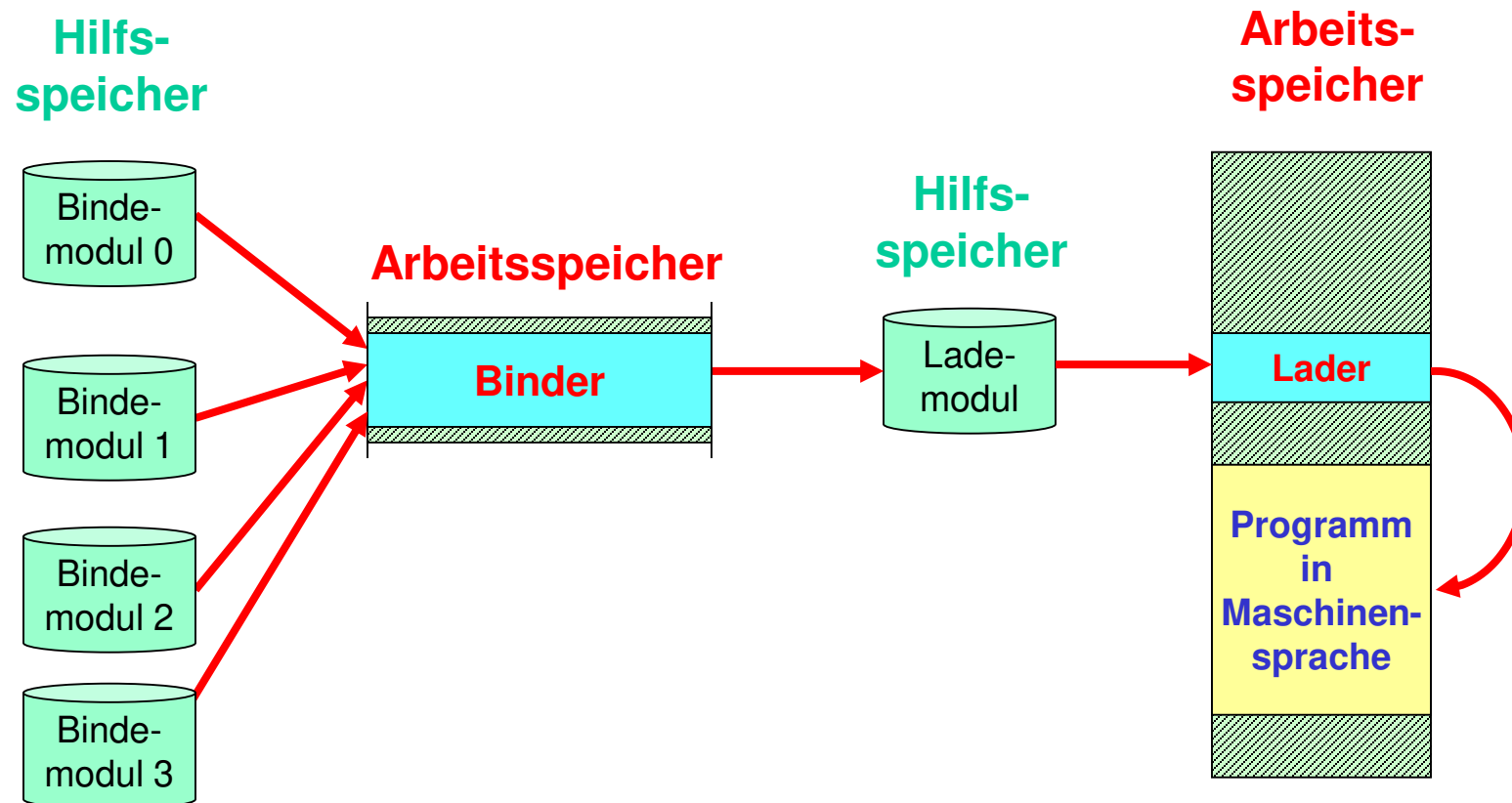
- mehrere **Programm-Segmente**,
- mehrere **Daten-Segmente**,
- **externe Referenzen auf Daten**,
- **externe Referenzen auf Befehle**
(Unterprogrammsprünge, externe Methoden)



2.1.8. Trennung von Binde- und Ladevorgang

Ein wesentlicher Nachteil des bindenden Laders besteht darin, dass **Allocation, Relocation, Linking und Loading für jeden Programmlauf** (mit recht hohem Aufwand) erneut durchgeführt werden müssen.

Für **Programme mit vielen Bindemodulen** wird daher die Trennung von Binde- und Ladevorgang sinnvoll:



Aufgaben und Funktionalität eines Binders

Der „**Binder**“ unterscheidet sich vom bindenden Lader nur dadurch, dass er den **Output**

- nicht als lauffähiges Programm in den Arbeitsspeicher schreibt,
- sondern als Datei (sog. „**Lademodul**“) speichert.

Der „**Binder**“ hat damit die folgenden Aufgaben:

1. Allocation

Zuweisung von Speicherplatz.

2. Relocation

Umrechnung aller Adressangaben entsprechend der zugewiesenen Speicherbereiche.

3. Linking

Herstellung der externen Referenzen (Adressbezüge) zwischen den Programmteilen.

Analog zu den Ladertypen sind wieder zu unterscheiden:

- Binder mit **expliziter** (= absoluter) **Adressangabe**
- Binder mit **verschiebbarem Lademodul** (Verschiebetabelle bzw. Basisregister)

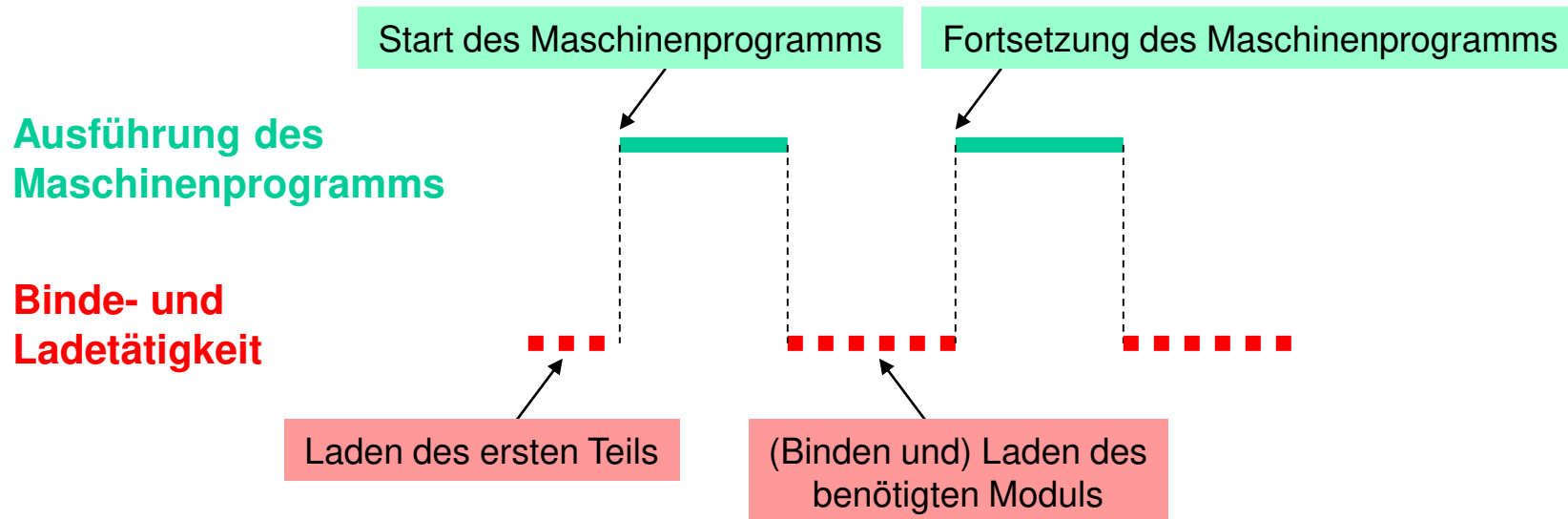
2.1.9. Dynamisches Binden und Laden

Bisher sind wir davon ausgegangen, dass stets mit dem Hauptprogramm auch alle zugehörigen „Module“ geladen werden.

Nachteile:

- Es werden auch die **Module** geladen, die vielleicht **gar nicht benötigt** werden (Aufruf nach bedingten Verzweigungen).
- Für komplexe Programme wird ein **sehr großer Hauptspeicher benötigt**. (vgl. aber auch „Paging“, Behandlung folgt im nächsten Kapitel).

Beim dynamischen Binden und Laden werden die Module (z.B. Java-Klassen) nur im Bedarfsfall hinzugefügt:



Dynamisches Binden in der Praxis

Microsoft Windows:

Alle gängigen Versionen von Windows **nutzen dynamisches Binden intensiv**.

Hierbei kommt ein Archiv, die sog. „**Dynamic Link Library**“, **DLL**, zum Einsatz. Die dort bereit gestellten Dateien können **Programm-Code und/oder Daten** enthalten. Sie haben häufig die Endungen **.dll, .ocx, .vxd, .sys, .drv**.

Es sind **Zehntausende derartige Dateien** für unterschiedlichste Einsatzgebiete bekannt und über Online-Quellen verfügbar.

Unix:

Auch bei Unix kommen intensiv umfassende Archive zum Einsatz, die dort als „**Shared Library**“ bezeichnet werden.

2.2. Übersetzung höherer Programmiersprachen

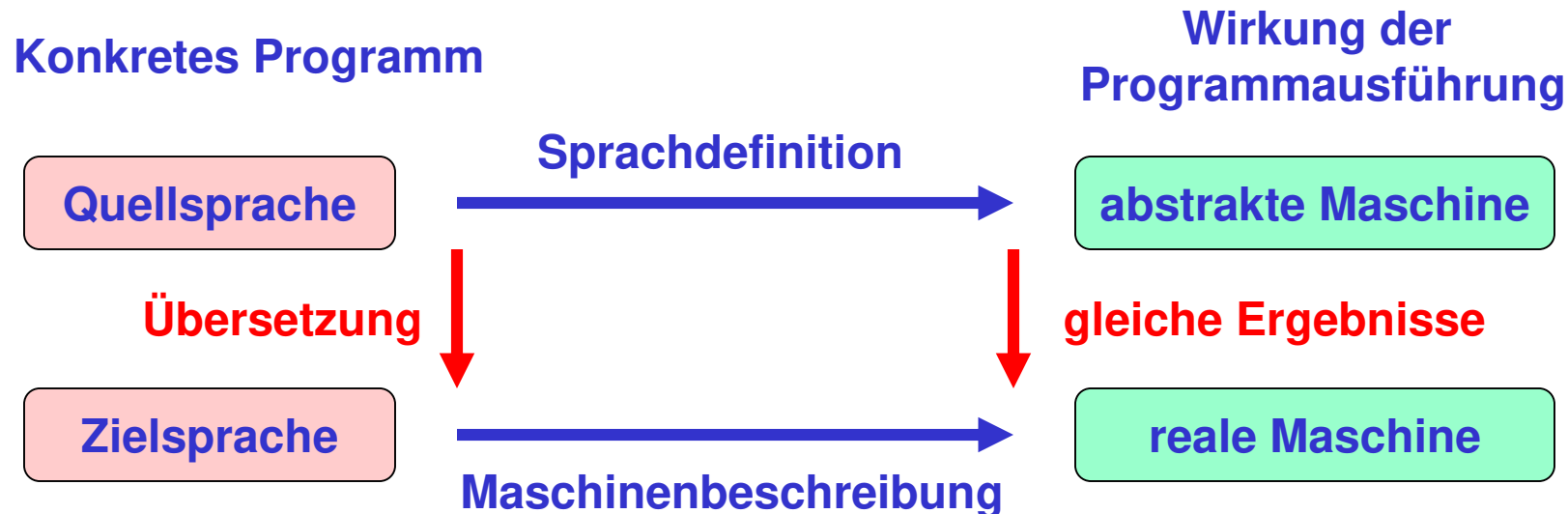
Ein **Übersetzer** erzeugt

- zu einem **Programm in einer Quellsprache**
- ein **äquivalentes Programm in einer Zielsprache**.

Quellsprache A → **Zielsprache B**

Ist die **Quellsprache mächtiger als die Zielsprache**, dann sprechen wir auch von einem „**Compiler**“ (to compile: zusammentragen).

Der wichtigste Fall ist die Übersetzung von höherer Programmiersprache in Maschinensprache:



Syntax und Semantik von Programmiersprachen


Eine automatische Übersetzung kann nur gelingen, wenn **Quellsprache und Zielsprache hinreichend formal definiert** sind.

Die **Syntax** (griech. syntaxis = Zusammenordnung, Lehre vom Satzbau)

einer Programmiersprache legt fest, welche Zeichenreihen **korrekt formulierte Programme der Sprache** sind und welche nicht.

Die **Semantik** (Semantik = Bedeutungslehre)

einer Programmiersprache legt fest, welche **Bedeutung korrekt formulierte Programme** der Sprache haben.

Syntax und Semantik von Programmiersprachen sind durch **strikte, formale Regeln** festgelegt. Daher ist es sogar möglich, **Übersetzungsalgorithmen** aus diesen Regeln **systematisch abzuleiten**:  **Automatischer Übersetzerbau**

Achtung:

Zielprogramme müssen nicht nur korrekt, sondern auch **schnell und speicher-effizient ausführbar** sein. Die **Optimierungsaufgaben** umfassen insbesondere:

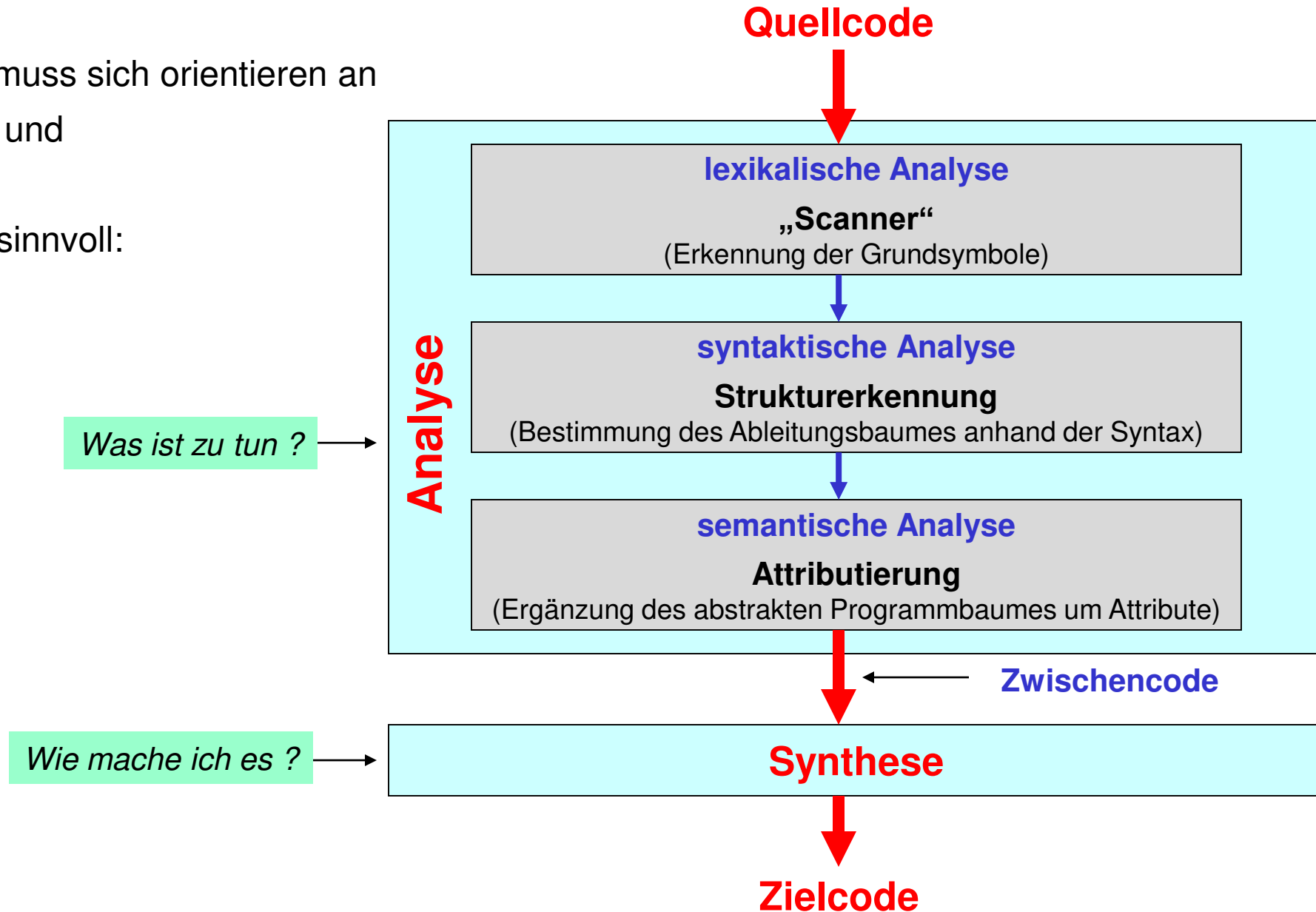
- **geschickten Einsatz der Register**,
- **geschickte Nutzung von Zwischenergebnissen**.

2.3. Übersetzerstruktur

Ein Übersetzer muss sich orientieren an

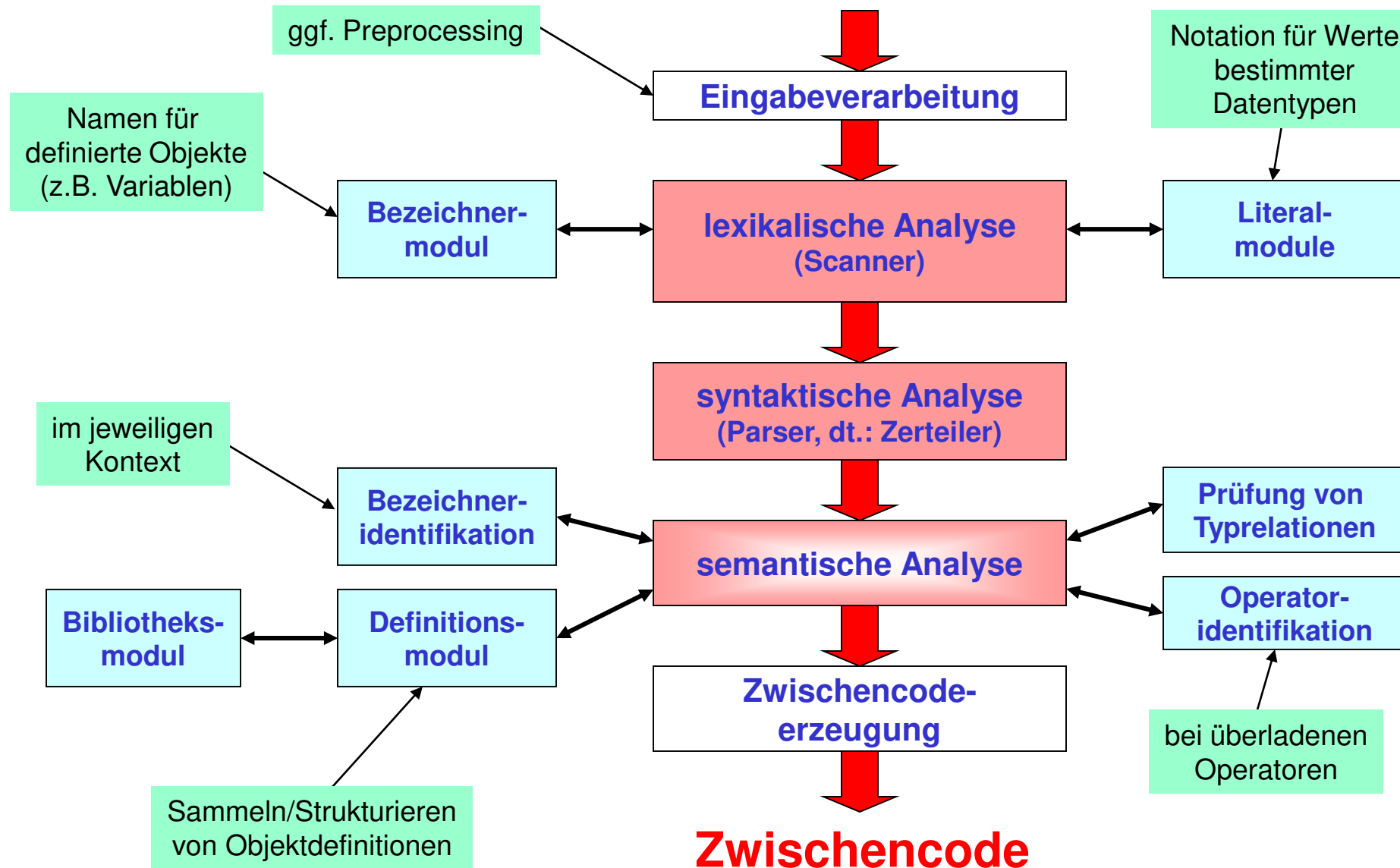
- **Quellsprache** und
- **Zielsprache**.

Eine Teilung ist sinnvoll:

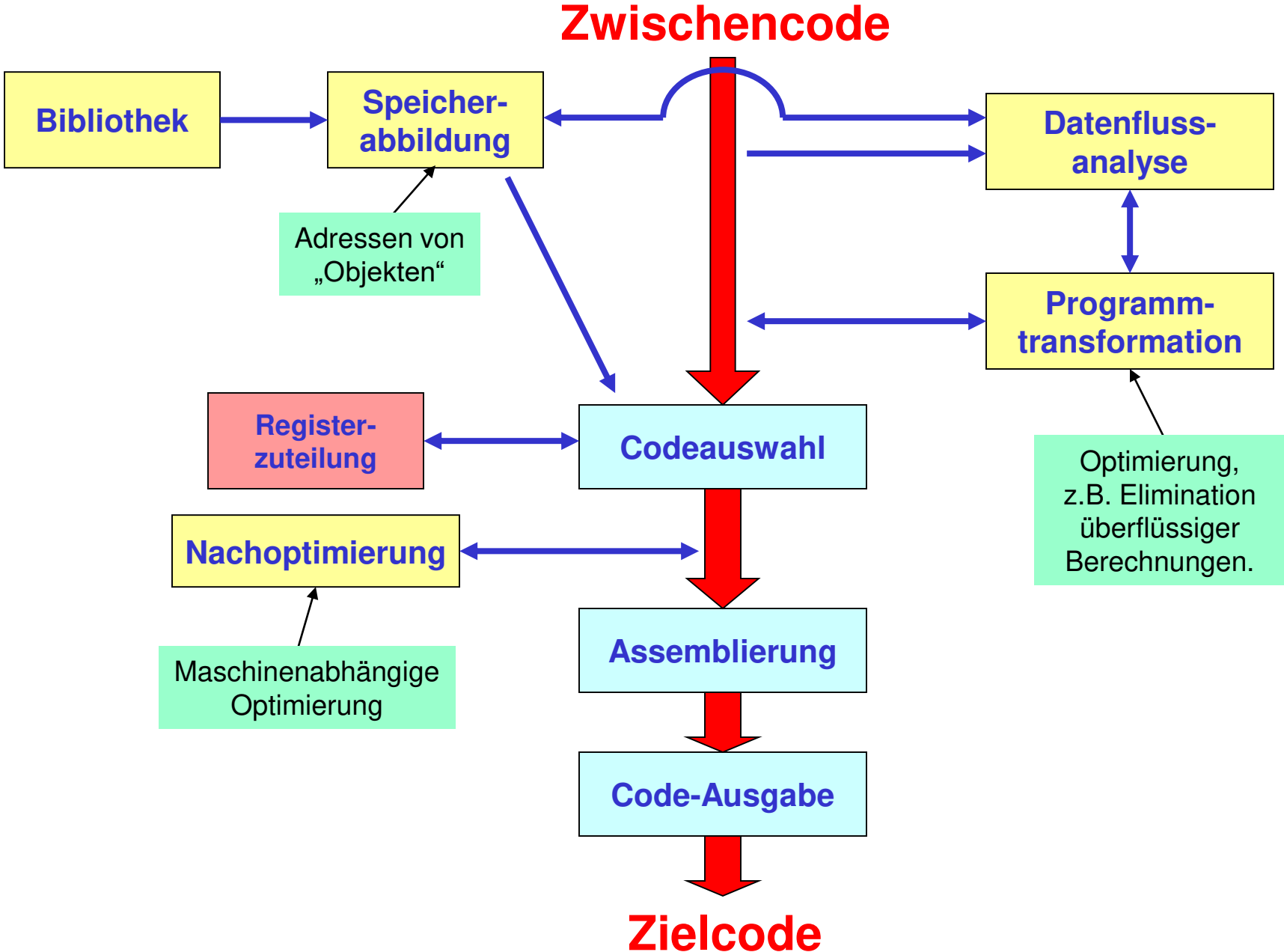


Die Analyse: Vom Quellprogramm zum Zwischencode

Quellprogramm



Die Synthese: Vom Zwischencode zum Zielcode



2.4. Kontextfreie Grammatiken

Zunächst definieren wir einige wichtige Begriffe:

Alphabet: Endliche Menge von Symbolen.
Wort, String: Aneinanderreihung endlich vieler Symbole eines Alphabets.
(formale) Sprache: Menge von Worten über einem Alphabet.

Die Spezifikation von Sprachen mit endlich vielen Elementen kann trivial durch Angabe genau dieser Elemente erfolgen (Aufzählung).

Eine Programmiersprache umfasst aber i.A. unendlich viele Elemente:
→ *Alle Programme, die den Regeln dieser Sprache entsprechen.*

Diese „Regeln“ können angegeben werden als

- Regeln zur Erzeugung von Worten der Sprache

➡ **Grammatik**

- Regeln zur Prüfung von Worten auf Zugehörigkeit zur Sprache.

➡ **Automat**

Definition: Kontextfreie Grammatik

Zentrale Bedeutung für den Entwurf von Programmiersprachen und die zugehörigen Analysealgorithmen hat die sog. „**kontextfreie Grammatik**“:

Aus bestimmten Symbolen können andere Symbole bzw. Ketten von Symbolen abgeleitet werden. Bei derartigen **Ableitungen** wird der **Kontext** des Symbols, aus dem die Ableitung erfolgt, **nicht berücksichtigt**.

Definition: Kontextfreie Grammatik (Context Free Grammar, CFG)

Eine kontextfreie Grammatik ist ein Quadrupel $G = (T, N, P, S)$ mit

T: Endliche Menge von „**Terminalsymbolen**“ (terminals).

N: Endliche Menge von „**Nicht-Terminalen**“ (nonterminals)); $N \cap T = \emptyset$.

P: Endliche Menge von „**Produktionen**“ (productions, auch: Regeln), $P \subset N \times (T \cup N)^*$

S: Eine spezielle Variable, das „**Startsymbol**“ (auch: Zielsymbol, goal symbol)

Produktionen schreiben wir oft als $A \rightarrow \alpha$, wobei

A ein **Nicht-Terminal** und

α ein **String über $(T \cup N)$** ist: $a \in (T \cup N)^*$

Aus einem Nicht-Terminal kann **kontextfrei** etwas abgeleitet werden.

Die Menge $V = T \cup N$ heißt „**Vokabular**“.

Die von einer CFG generierte Sprache

Jede Ableitung zu einer CFG ist durch einen Ableitungsbaum darstellbar.

Beispiel: $G = (T, N, P, S)$ mit

$N = \{S, B, C\},$

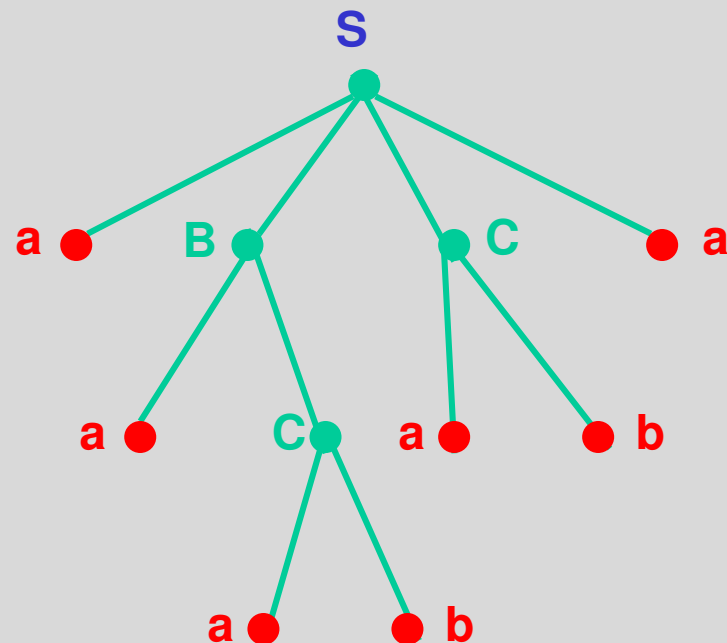
$T = \{a, b\},$

$P = \{S \rightarrow aBCa, B \rightarrow aC, C \rightarrow ab\}$

$S \rightarrow aBCa$

$B \rightarrow aC$

$C \rightarrow ab$



Das einzige in dieser Sprache ableitbare Wort ist: aaababa

Ein **String ist in der** von einer kontextfreien Grammatik G **generierten Sprache $L(G)$** genau dann, wenn

- a) der String **nur aus Terminalsymbolen** von G besteht und
- b) der String **aus dem Startsymbol abgeleitet** werden kann.

Notation einer kontextfreien Grammatik

Die **Angabe einer kontextfreien Grammatik** beschränkt sich häufig auf

- die **Angabe des Startsymbols** (bzw. des Zielsymbols, goal symbol) und
- die **Angabe der Regeln** (Produktionen).

Terminale und Nicht-Terminale sind durch die Art ihrer Verwendung in den Produktionen eindeutig unterscheidbar (nur aus den Terminalen gibt es keine Ableitungen).

Eine **spezielle Kennzeichnung (Terminal/Nonterminal)** erscheint aber wünschenswert:

- Nonterminals beginnen mit **großem Buchstaben**, Terminals mit **kleinem Buchstaben**
- Nonterminals werden **kursiv** gedruckt, Terminals **nicht-kursiv**.
- Nonterminals in spitzen Klammern, Terminals nicht.
- ...

Backus-Naur-Form (BNF)

Als **Beschreibungsform kontextfreier Grammatiken** hat die sog. Backus-Naur-Form (BNF) die größte Verbreitung.

Die BNF verwendet „**Ersetzungsregeln**“ mit einer linken und einer rechten Seite, wobei die Seiten durch **::=** **getrennt** werden.

Gibt es **mehrere rechte Seiten** für ein Nonterminal, dann werden diese **durch senkrechte Striche getrennt**. Die linke Seite wird dabei nicht wiederholt.

Beispiel:

<Ziffer> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<Buchstabe> ::= A | B | C | D | E | ... | Z

<Bezeichner> ::= <Buchstabe> | <Buchstabe> <Zeichenkette>

```
<Zeichenkette> ::= <Buchstabe> | <Ziffer> | <Buchstabe> <Zeichenkette> |  
                    <Ziffer> <Zeichenkette>
```

Mittels Rekursion sind in diesem Beispiel offenbar beliebig lange Zeichenketten möglich. Sie müssen allerdings mit einem Buchstaben beginnen.

Syntaxdiagramm

Syntaxdiagramme sind sehr anschauliche **graphische Darstellungen der Backus-Naur-Form**.

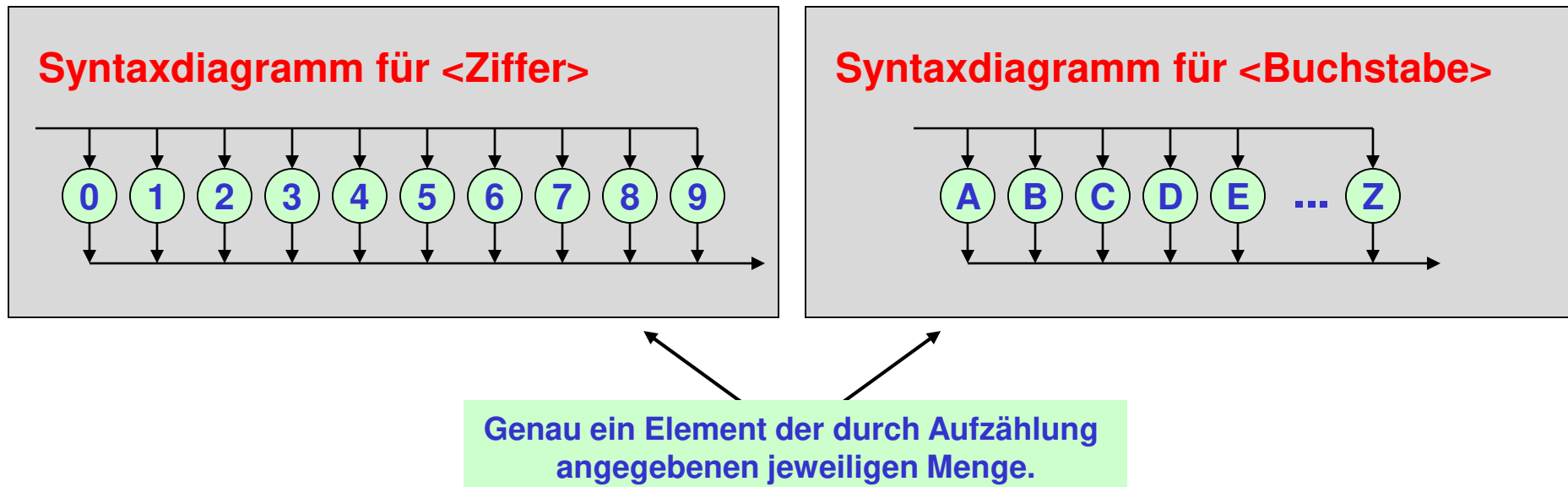
Ellipsen: repräsentieren Terminalsymbole.

Rechtecke: repräsentieren Nicht-Terminale.

Kanten: verbinden zwei Knoten genau dann, wenn die zugehörigen Zeichen in der Zeichenkette unmittelbar aufeinander folgen dürfen.

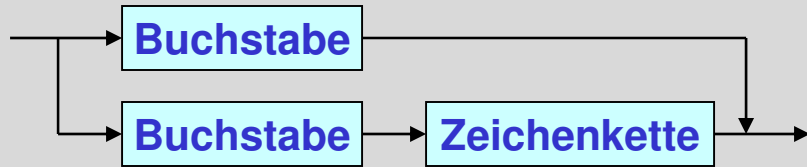
Außerdem gibt es eindeutige Eingangs- und Ausgangskanten.

Jeder Weg durch den so entstehenden **Graphen** von der Eingangskante zur Ausgangskante **beschreibt eine Ableitung aus dem zugehörigen Nicht-Terminal**.

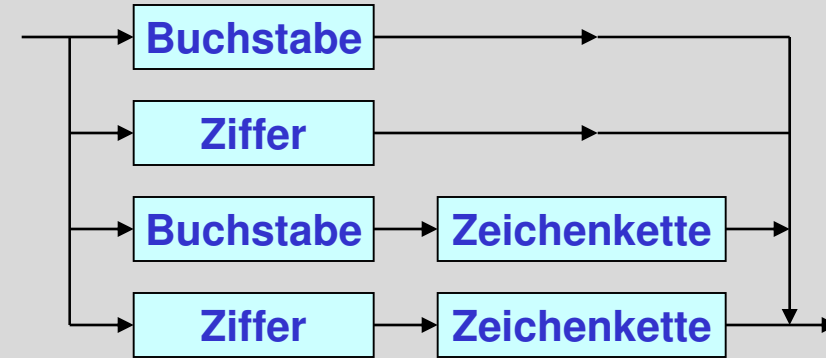


Syntaxdiagramm (2)

Syntaxdiagramm für <Bezeichner>

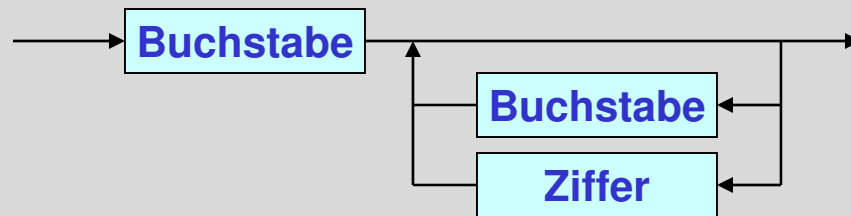


Syntaxdiagramm für <Zeichenkette>



Bei Syntaxdiagrammen sind oft Vereinfachungen möglich. So kann etwa eine zu der obigen äquivalente Spezifikation von <Bezeichner> wie folgt angegeben werden:

Vereinfachtes Syntaxdiagramm für <Bezeichner>



CFG-Notation bei der Spezifikation von Java

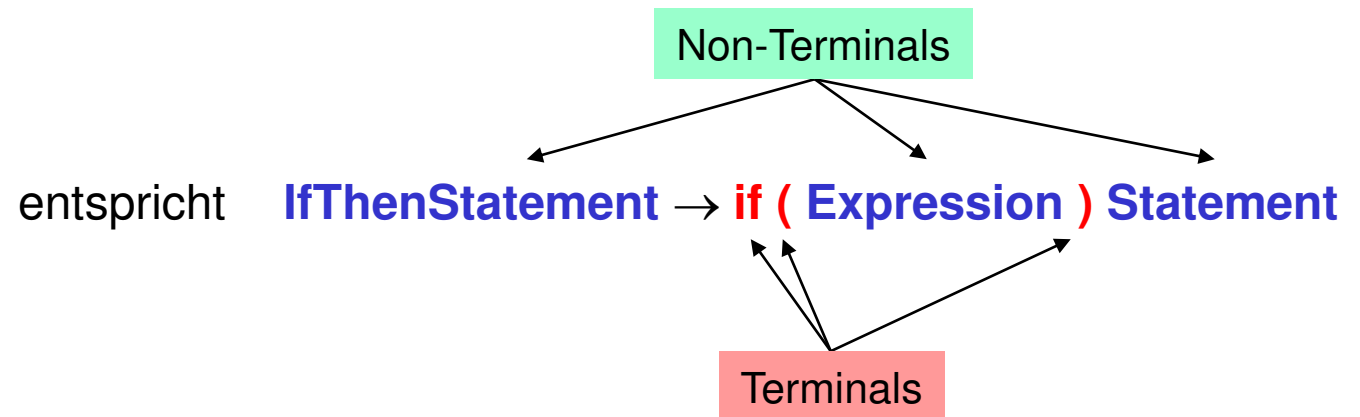
Nonterminal symbols are shown in *italic* type. The definition of a nonterminal is introduced by the name of the nonterminal being defined followed by a colon.

One or more alternative right-hand sides for the nonterminal then follow on succeeding lines.

Quelle: J. Gosling, B. Joy, G. Steele, „The Java Language Specification“, 2nd Edition.
Online verfügbar (Link-Test am 15.5.2000) unter: <http://java.sun.com/docs/books/jls/index.html>

Beispiel: Das IfThenStatement in Java

IfThenStatement :
if (*Expression*) *Statement*



CFG-Notation bei der Spezifikation von Java (2)

In der Spezifikation von Java wird die Übersichtlichkeit massiv durch den **Einsatz von optionalen Symbolen** (Terminal oder Nonterminal) verbessert.

So enthält die Spezifikation z.B.:

```
ForStatement:  
for ( ForInitopt ; Expressionopt ; ForUpdateopt ) Statement
```

Dies ist Kurzschreibweise für:

```
ForStatement:  
for ( ; Expressionopt ; ForUpdateopt ) Statement  
for ( ForInit ; Expressionopt ; ForUpdateopt ) Statement
```

Dies ist Kurzschreibweise für:

```
ForStatement:  
for ( ; ; ForUpdateopt ) Statement  
for ( ; Expression ; ForUpdateopt ) Statement  
for ( ForInit ; ; ForUpdateopt ) Statement  
for ( ForInit ; Expression ; ForUpdateopt ) Statement
```

Dies ist Kurzschreibweise für:

```
ForStatement:  
for ( ; ; ) Statement  
for ( ; ; ForUpdate ) Statement  
for ( ; Expression ; ) Statement  
for ( ; Expression ; ForUpdate ) Statement  
for ( ForInit ; ; ) Statement  
for ( ForInit ; ; ForUpdate ) Statement  
for ( ForInit ; Expression ; ) Statement  
for ( ForInit ; Expression ; ForUpdate ) Statement
```

Kurzschreibweise für 8 Produktionen

Anmerkung:

Bei der Spezifikation von Java werden **weitere Kurzschreibweisen** verwendet, auf die wir hier aber nicht näher eingehen.

Java SE Specifications


https://docs.oracle.com/javase/specs/

ORACLE
Java SE > Java SE Specifications


Java Language and Virtual Machine Specifications

Java SE 19

Released September 2022 as [JSR 394](#)

 The Java Language Specification, Java SE 19 Edition


- [HTML](#) | [PDF](#)
- Preview feature: [Pattern Matching for switch](#)
- Preview feature: [Record Patterns](#)

 The Java Virtual Machine Specification, Java SE 19 Edition


- [HTML](#) | [PDF](#)

Java SE 18

Released March 2022 as [JSR 393](#)

 The Java Language Specification, Java SE 18 Edition


- [HTML](#) | [PDF](#)
- Preview feature: [Pattern Matching for switch](#)

 The Java Virtual Machine Specification, Java SE 18 Edition


- [HTML](#) | [PDF](#)

Java SE 17

Released September 2021 as [JSR 392](#)

 The Java Language Specification, Java SE 17 Edition

- [HTML](#) | [PDF](#)
- Preview feature: [Pattern Matching for switch](#)

 The Java Virtual Machine Specification, Java SE 17 Edition

- [HTML](#) | [PDF](#)

**Die Java Language Specification
... zum Downloaden,
... zum Browsen.**

2.5. Lexikalische Analyse

Die lexikalische Analyse **zergliedert den Quelltext in** sog. „**Token**“ (= die Grundsymbole der Programmiersprache). Sie basiert auf dem **Modell des endlichen Automaten**.

[2.5.1. Grundlegende Betrachtung](#)

[2.5.2. Formale Spezifikation von Grundsymbolen](#)

[2.5.3. Endliche Automaten](#)

[2.5.4. Nicht-deterministische endliche Automaten](#)

[2.5.5. Vom Syntaxdiagramm zum endlichen Automaten](#)

[2.5.6. Die Regel des längsten Musters](#)

2.5.1. Grundlegende Betrachtung

Die lexikalische Analyse umfasst die folgenden Aktivitäten:

- **Erkennen** der Grundsymbole (Token),
- **Ausblenden** bedeutungsloser Zeichen,
- **Codieren** der Grundsymbole.

Die codierten Grundsymbole sind die Terminale der kontextfreien Grammatik, die dem Zerteiler (Parser, syntaktische Analyse) zugrunde liegt.

➡ Die lexikalische Analyse liefert die Symbole für die syntaktische Analyse.

➡ Diese Symbole sind die Terminalsymbole der Syntax-Grammatik.

Für die syntaktische Analyse sind

- die **Identität von Bezeichnern** und
- der **Wert von Literalen**

irrelevant! Sie können daher auf ein einziges Symbol codiert werden.

Bei der semantischen Analyse werden die **Identität** der Bezeichner **und** der **Wert** der Literale **als Attribute** in die Codierung der Symbole übernommen.

Beispiel Java: In 3 Schritten zum Token-Strom

Bei Java wird der **Strom von Unicode-Zeichen in drei Schritten in Java Token** umgesetzt:

1. Schritt:

Umsetzung von Unicode Escapes (`\uxxxx` mit `xxxx` = Hex-Darstellung des Unicode-Zeichens) **in Unicode**. Durch Voranschaltung dieses Schrittes wird es möglich, jedes Java-Programm vollständig in ASCII-Code darzustellen.

2. Schritt:

Umsetzung des Unicode-Stromes in einen Strom aus

- „**input characters**“ und
- „**line terminators**“ (**LF**, **CR**, **CR LF**).
ASCII LF („newline“), ASCII CR („return“) bzw. ASCII CR gefolgt von ASCII LF.

3. Schritt:

Umsetzung des in Schritt 2 erzeugten Stromes aus „input characters“ und „line terminators“ **in eine Folge von „input elements“**, aus denen sich nach Unterdrückung von „white space“ und „comments“ die **Token** ergeben, die **Terminalsymbole der Syntax-Grammatik von Java** sind.

Beispiel Java: Die Grundsymbole (Token)

Im Falle von Java gibt es Token der folgenden Arten:

Identifizier:

(Beliebig lange, aber endliche) **Folge von Buchstaben und/oder Ziffern**.
Bestimmte Folgen (z.B. true) sind unzulässig.

Keyword:

Eine in Java als **Schlüsselwort** gewählte Folge von ASCII-Zeichen.

Literal:

Darstellung des Wertes eines Grundtyps, des `String` Typs oder des `null` Typs.

Separator:

Eines der folgenden **9 ASCII-Zeichen**:

() { } [] ; , .

Operator:

Aus ASCII-Zeichen werden die folgenden 37 Operatoren gebildet:

=	>	<	!	~	?	:	==	<=	>=	!=	&&	
+	-	*	/	&		^	%	<<	>>	>>>	++	--
+=	-=	*=	/=	&=	=	^=	%=	<<=	>>=	>>>=		

Beispiel Java: Die Wortsymbole (Keywords)

ReservedKeyword: one of

abstract	continue	for	new	switch
assert	default	if	package	synchronized
Boolean	do	goto	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while
_ (underscore)				

- The keywords `const` and `goto` are reserved, even though they are not currently used. This may allow a Java compiler to produce better error messages if these C++ keywords incorrectly appear in programs.
- The keyword `strictfp` is obsolete and should not be used in new code.
- The keyword `_` (underscore) is reserved for possible future use in parameter declarations.
- `true` and `false` are not keywords, but rather boolean literals (§3.10.3).
- `null` is not a keyword, but rather the null literal (§3.10.8).

Quelle: J. Gosling, B. Joy, G. Steele, et al.: „The Java Language Specification“, Java SE 19 Edition.
Online verfügbar (Link-Test am 13.03.2023) unter: <https://docs.oracle.com/javase/specs/jls/se19/html/jls-3.html#jls-3.9>

Beispiel Java: Von „Input Elements“ zu White Space, Comment und Token

Input:

InputElements_{opt} Sub_{opt}

InputElements:

InputElement

InputElements InputElement

InputElement:

WhiteSpace

Comment

Token

Token:

Identifizier

Keyword

Literal

Separator

Operator

Sub: (the ASCII SUB character, “control-Z” wird ignoriert)

Quelle: J. Gosling, B. Joy, G. Steele, „The Java Language Specification“, 2nd Edition.
Online verfügbar (Link-Test am 15.5.2000) unter: <http://java.sun.com/docs/books/jls/index.html>

2.5.2. Formale Spezifikation von Grundsymbolen

Wortsymbole (Keywords), Operatoren und Separatoren können durch explizite Angabe der jeweiligen Zeichen bzw. Zeichenketten eindeutig spezifiziert werden.

Beispiel:

() { } [] ; , .

Ist bei Literalen und Bezeichnern eine beliebige (aber feste) Länge möglich, dann gibt es **beliebig viele zulässige Literale und Bezeichner**. Eine Spezifikation durch explizite Aufzählung ist dann offensichtlich nicht mehr möglich.

Andererseits muss zur Spezifikation aber auch **nicht die gesamte Ausdrucksstärke einer kontextfreien Grammatik** genutzt werden.

Reguläre Grammatiken

Hinreichend flexibel, präzise und vollständig sind zur Spezifikation von zulässigen Zusammensetzungen aus Zeichen die sog. „**regulären Grammatiken**“:

Definition: Rechts-lineare Grammatik

Eine kontextfreie Grammatik heißt „rechts-linear“ genau dann, wenn **alle Produktionen die Form**

$A \rightarrow w B$ oder $A \rightarrow w$

Die einzige Variable steht rechts.

haben („Rechtsableitungen“ sind), wobei A und B Nonterminals seien und w ein (ggf. leerer) String von Terminalen sei.

Analog:

„**links-linear**“ für Produktionen der Form $A \rightarrow B w$ bzw. $A \rightarrow w$ (Linksableitungen).

Definition: Reguläre Grammatik

Eine **entweder rechts- oder links-lineare** Grammatik heißt „regulär“.

Die Ausdruckskraft einer regulären Grammatik entspricht der von Syntaxdiagrammen ohne Nonterminals.

Beispiel Java: Die Bezeichner (Identifizier)

Identifizier:

IdentifizierChars but not a *Keyword* or *BooleanLiteral* or *NullLiteral*

IdentifizierChars:

JavaLetter

IdentifizierChars JavaLetterOrDigit

JavaLetter:

any Unicode character that is a Java letter (see below)

JavaLetterOrDigit:

any Unicode character that is a Java letter-or-digit (see below)

Quelle: J. Gosling, B. Joy, G. Steele, „The Java Language Specification“, 2nd Edition.
Online verfügbar (Link-Test am 15.5.2000) unter: <http://java.sun.com/docs/books/jls/index.html>

Ein Identifizier muss offensichtlich mit einem Buchstaben beginnen.
Nähere Erläuterungen („see below“) sind in Textform angegeben.

2.5.3. Endliche Automaten

Der Algorithmus zur lexikalischen Analyse basiert auf dem formalen Konzept des endlichen Automaten (engl.: finite automaton, FA).

Definition: Endlicher Automat

Ein endlicher Automat ist
ein Tupel $A = (Q, \Sigma, \delta, q_0, F)$ mit

Q : (endliche) Zustandsmenge

Σ : (endlicher) Zeichensatz (= Eingabe-Alphabet)

δ : Übergangsfunktion ($\delta: Q \times \Sigma \rightarrow Q$)

q_0 : Anfangszustand ($q_0 \in Q$)

F : Menge der Endzustände ($F \subset Q$)

Veranschaulichung:

0	1	1	0	1	...
---	---	---	---	---	-----

 Eingabe-Band
(Symbole aus Σ)

Leserichtung

endliche
Kontrolle

- endlich viele Zustände
- Zustands-Übergänge gemäß δ

Das Verhalten eines FAs wird durch die Übergangsfunktion δ beschrieben:

$\delta(q, a) = q'$ Im Zustand q akzeptiert der Automat das Zeichen a und geht in Zustand q' über.
(Wir sagen auch: Der Automat liest a und wechselt nach q')

Ein **String x** ist **in der von A akzeptierten Sprache $L(A)$** genau dann, wenn der String x den Automaten **A** vom Anfangszustand in einen Endzustand überführt.

Übergangsgraph (Transition diagram)

Ein FA kann durch einen gerichteten Graphen dargestellt werden:

Knoten: Zustände des Automaten

Intitialer Pfeil:

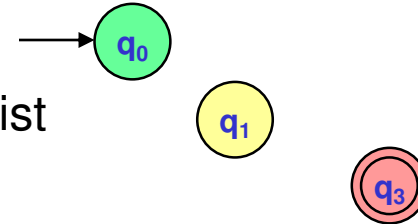
Anfangszustand (exakt 1)

Einfacher Kreis:

Zustand, der kein Endzustand ist

Doppelter Kreis:

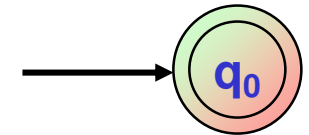
Endzustand (mehrere möglich)



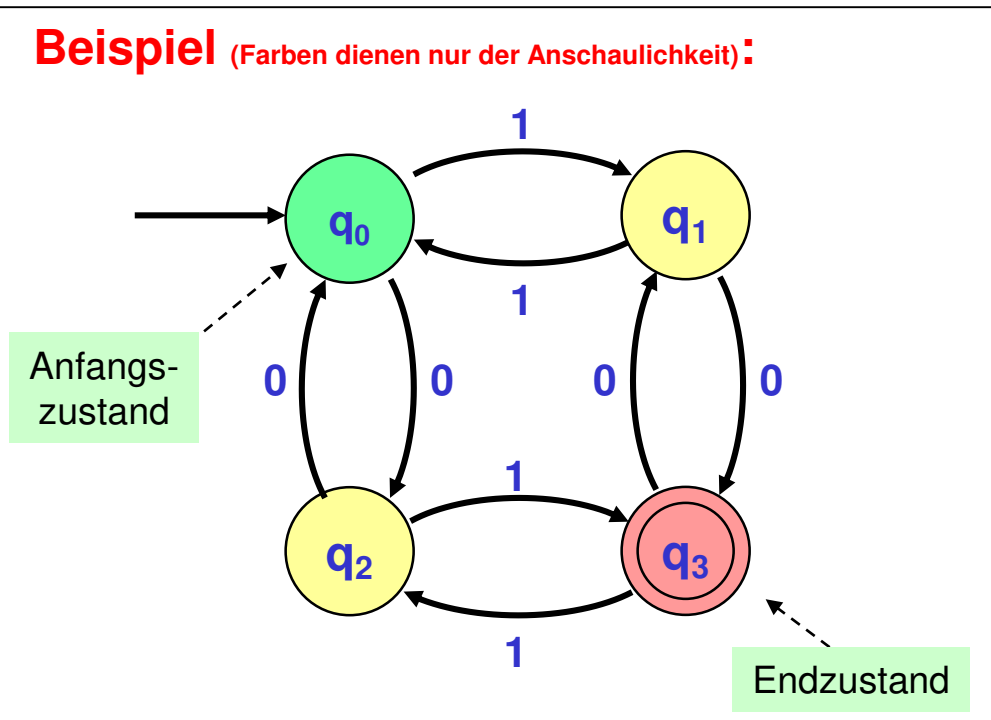
Kanten: Repräsentation der Übergangsfunktion

(Markierung durch das jeweilige Zeichen)

Ist der **Anfangszustand auch Endzustand**, zeigt der initiale Pfeil auf einen Endzustand:



Beispiel (Farben dienen nur der Anschaulichkeit):



Erweiterung der Übergangsfunktion auf Strings

Die formale Darstellung des FAs beschränkt sich bisher auf die **Bearbeitung einzelner Zeichen**: $\delta: Q \times \Sigma \rightarrow Q$.

Wir erweitern diese formale Beschreibung nun auf Strings:

$\hat{\delta}: Q \times \Sigma^* \rightarrow Q$ mit

a) $\hat{\delta}(q, \varepsilon) = q$

b) für alle Strings $w \in \Sigma^*$ und Zeichen $a \in \Sigma$ gilt: $\hat{\delta}(q, wa) = \delta(\hat{\delta}(q, w), a)$.

Zustand nach
Bearbeitung von w.

a) sagt uns, dass der Automat den Zustand nicht wechseln kann, ohne ein Zeichen zu lesen.

b) sagt uns, was zu tun ist, nachdem ein String w vollständig gelesen wurde.

Aufgrund von $\hat{\delta}(q, a) = \delta(\hat{\delta}(q, \varepsilon), a) = \delta(q, a)$ kann es nicht zu „Unstimmigkeiten“ für Argumente kommen, für die sowohl δ als auch $\hat{\delta}$ definiert ist.

Zur Vereinfachung unterscheiden wir nachfolgend nicht zwischen δ und $\hat{\delta}$.

Offenbar kann die gedachte Maschine „Endlicher Automat“ **trivial durch eine praktische Realisierung implementiert** werden, z.B. in Java.

2.5.4. Nicht-deterministische endliche Automaten

Ein **endlicher Automat** gemäß unserer Definition arbeitet deterministisch:

In jedem Zustand gibt es für jedes Zeichen höchstens einen Folgezustand.

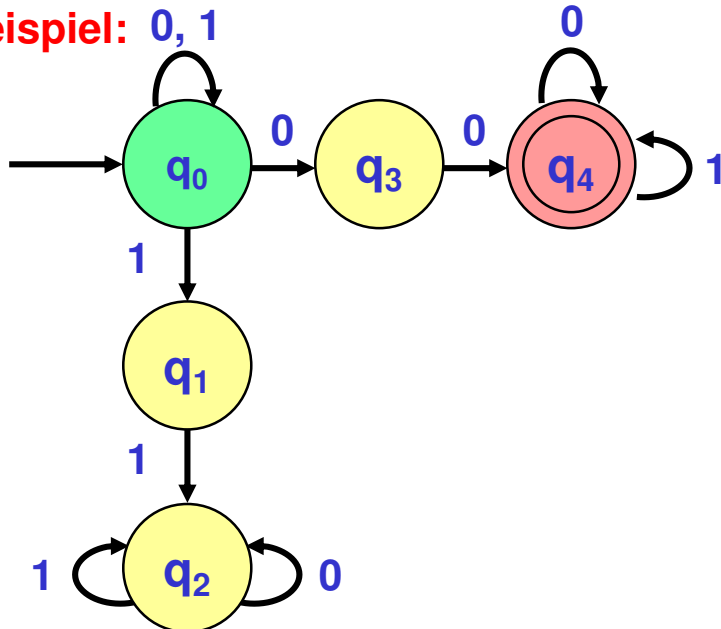
Zur Erinnerung:

Eine **Funktion** $f: M \rightarrow N$ ist eine Relation $f \subseteq M \times N$ so, dass
aus $(x,y) \in f$ und $(x,z) \in f$ folgt $y = z$.

Im Allgemeinen ist die **Übergangsfunktion partiell**: Es gibt Eingaben, für die diese Funktion nicht definiert ist. Tritt eine solche Eingabe auf, dann stoppt der Automat.

Lassen wir die Forderung nach deterministischer Arbeitsweise fallen, dann gelangen wir zum nicht-deterministischen endlichen Automaten (NFA).

Beispiel: 0, 1



Anmerkungen:

- In der **theoretischen Informatik** sind NFAs **nützlich** für das Beweisen von Sätzen.
- In der **praktischen Informatik** sind NFAs weitgehend **unbrauchbar**.
- **Äquivalenz:** Jede Sprache, die von einem NFA akzeptiert wird, wird auch von einem FA akzeptiert. Der **Beweis** dieses Satzes ist **konstruktiv** (= zeigt, wie man den FA gewinnt).
- **Näheres hierzu vgl. Literatur**, z.B. J.E. Hopcroft, J.D. Ullman, „Introduction to Automata Theory, Languages and Computation“, Addison-Wesley

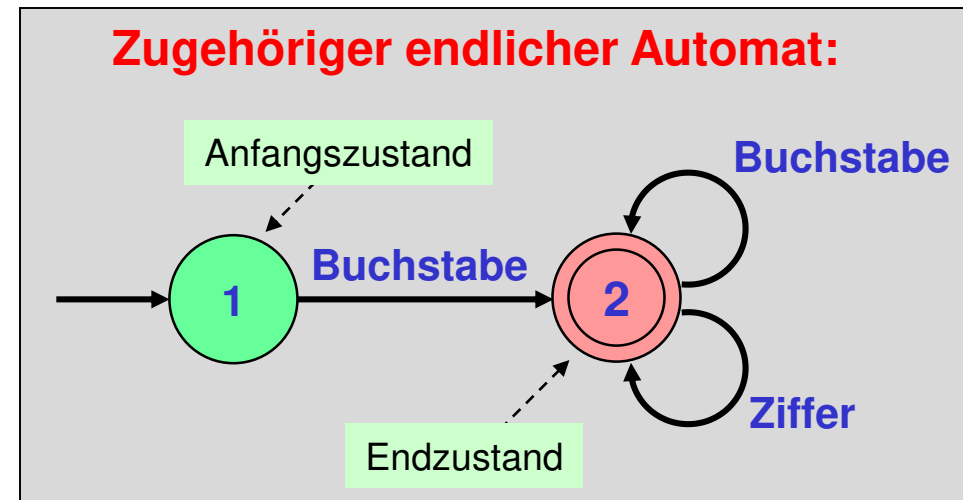
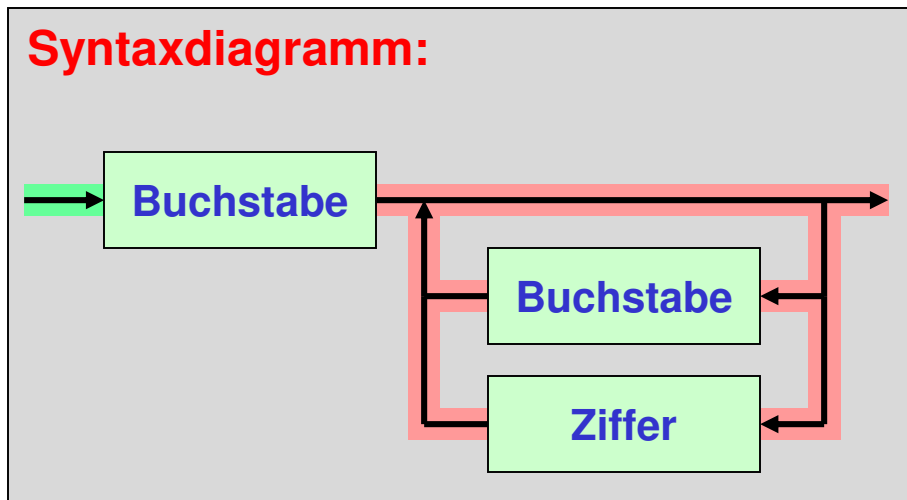
2.5.5. Vom Syntaxdiagramm zum endlichen Automaten

Zu einem vorgegebenen Syntaxdiagramm einer regulären Sprache kann **systematisch ein endlicher Automat konstruiert** werden, der genau die im Syntaxdiagramm spezifizierte Sprache akzeptiert:

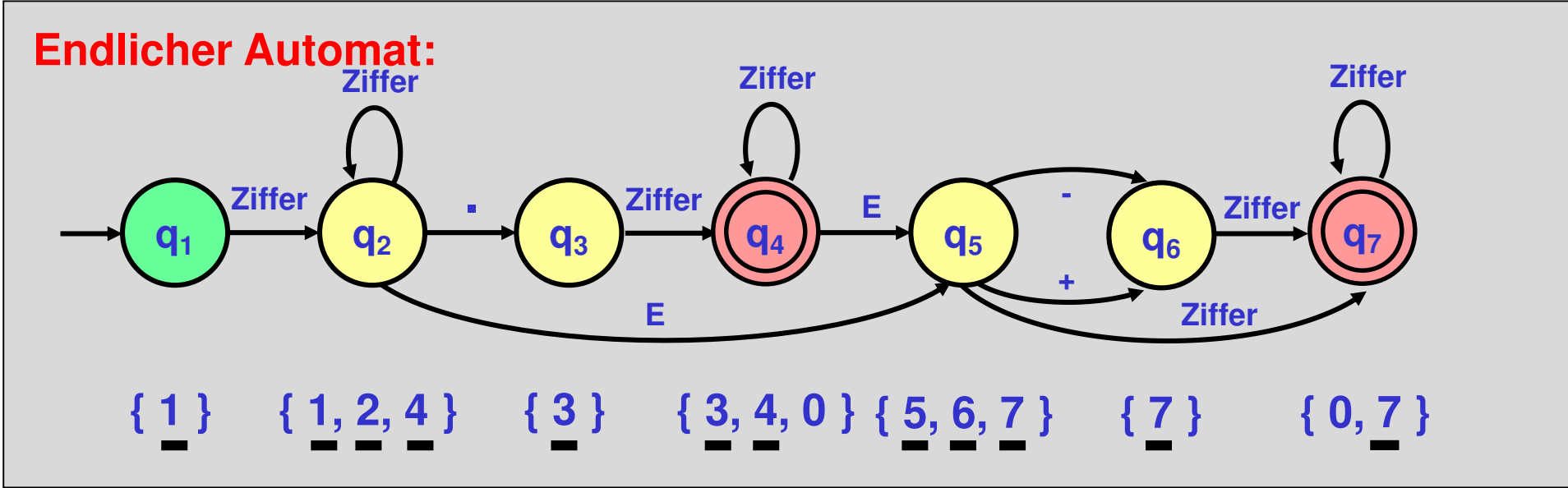
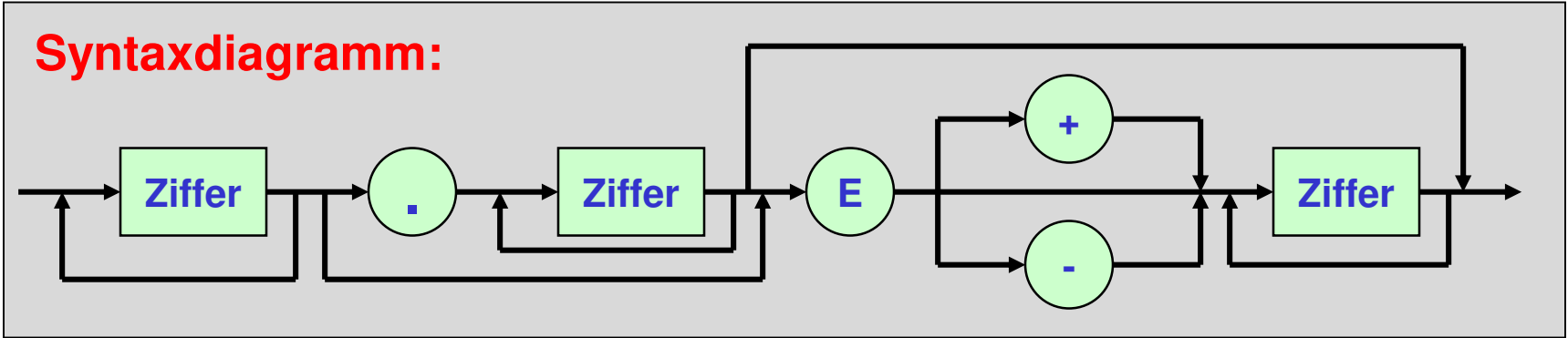
Knoten im Syntaxdiagramm  werden zu **Übergängen** des Automaten
(das angegebene Zeichen wird akzeptiert)

Kanten im Syntaxdiagramm  werden zu **Zuständen** des Automaten

Das nachfolgende Beispiel zeigt einen endlichen Automaten, der genau die im Syntaxdiagramm spezifizierten Bezeichner akzeptiert.



Ein komplexeres Beispiel



Das Non-Terminal <Ziffer> könnte unmittelbar durch je 10 Knoten (mit zugehörigen Kanten) für die 10 Ziffern ersetzt werden. Da hierdurch die Übersichtlichkeit nicht erhöht würde, arbeiten wir im Folgenden mit der hier gezeigten **Kurzschreibweise für genau ein Element aus der Menge der Ziffern**.

Algorithmus: Konstruktion des endlichen Automaten

Die im Beispiel betrachtete Umsetzung eines Syntaxdiagramms in einen endlichen Automaten, der genau die gemäß Syntaxdiagramm zulässigen Strings akzeptiert, kann dadurch geschehen, dass

- **sukzessive Mengen von Marken bestimmt** werden, die
- **jeweils einen Zustand des Automaten** repräsentieren.

1. Markierung des Syntaxdiagramms

Markiere alle Knoten des Syntaxdiagramms eindeutig **mit natürlichen Zahlen**, den **Ausgang** markiere mit **0**.

2. Bestimmung des Anfangszustandes

Bestimme (im Syntaxdiagramm) die Knoten, die **vom Eingang aus direkt erreichbar** sind (= erreichbar, ohne einen Knoten zu passieren).

Die Menge der zugeordneten Marken bildet den **Anfangszustand q_1** .

3. Wahl eines Zustandes und eines Zeichens

Wähle einen schon konstruierten **Zustand q** mit **Markenmenge m** .

Wähle ferner ein **Zeichen a** , das als Inschrift mindestens einer Marke in m zugeordnet ist.

Algorithmus: Konstruktion des endlichen Automaten (2)

4. Bestimmung der bei Wahl gemäß 3. direkt erreichbaren Zustände

Bestimme (im Syntaxdiagramm) die Menge k von **Knoten mit Inschrift a** , deren **Marke in m** ist. Dann bilde die **Menge m'** von Marken, die (im Syntaxdiagramm) von irgendeinem Knoten **aus k direkt erreichbar** sind. Den m' zugeordneten bzw. neu zuzuordnenden Zustand nennen wir q' .

Wichtige Hinweise zu Schritt 4:

- Bilde in diesem Schritt **für alle Knoten in k mit der Inschrift a** die erreichbaren Marken-Mengen und vereinige diese!
- Es genügt **nicht**, **nur für einen ausgewählten Knoten** aus k diese Marken-Menge zu bilden!
- **Falsche Interpretation** dieser Regel liefert einen **nicht-deterministischen Automaten**. Dieser ist für uns als praktisch einsetzbarer Spracherkenner unbrauchbar und daher (in Übungen und Klausuren) **falsch**!
- Gehe erst **zu Schritt 5**, **nachdem alle Knoten aus k** entsprechend **bearbeitet** wurden.

Algorithmus: Konstruktion des endlichen Automaten (3)

5. Prüfung, ob ein neuer Zustand gefunden wurde

Gibt es noch keinen Zustand q' mit (nicht-leerer) Markenmenge m' , so **füge q'** dem Automaten **hinzu**.

6. Einfügung einer neuen Transition in den Automaten

Verbinde den gewählten Zustand **q** und den gefundenen Zustand **q'** **durch eine Kante**, die **mit a markiert** ist (Übergangsfunktion: $\delta(q,a) = q'$). Dies gilt auch, wenn q' kein neuer Zustand ist.

7. Iteration

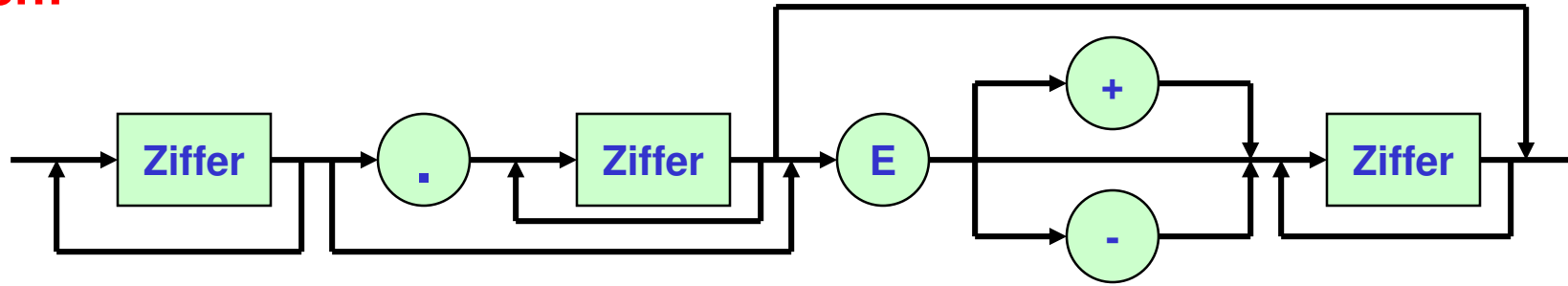
Wiederhole die Schritte 3 bis 6, bis alle Paare von konstruierten Zuständen und Zeichen betrachtet wurden. Wähle aber nicht mehrfach dasselbe Paar.

8. Bestimmung des Endzustandes / der Endzustände

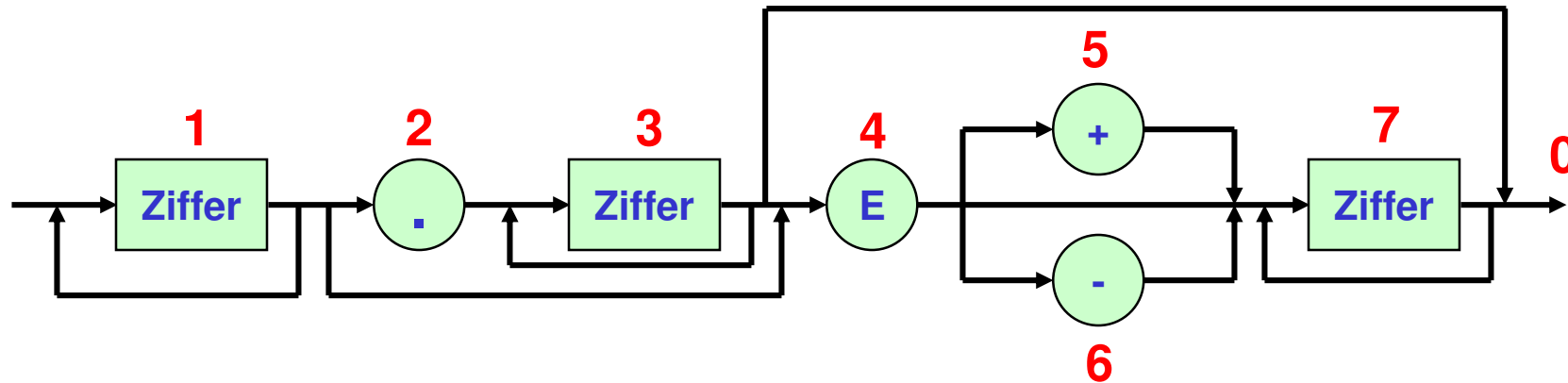
Ein Zustand mit Markenmenge m ist genau dann ein Endzustand, wenn $0 \in m$ ist.

Beispiel: Konstruktion des endlichen Automaten

Gegeben:

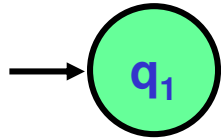


Markiere das Syntaxdiagramm



Beispiel: Konstruktion des endlichen Automaten (2)

Bestimme den Anfangszustand



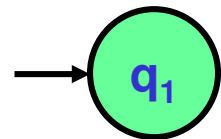
$\{1\}$

Alle Marken, die vom Eingang aus direkt erreichbar sind.

Wähle einen Zustand und ein Zeichen

Damit ein Zustand gewählt werden kann, muss er existieren.

Da zunächst nur ein Zustand existiert, ist die Wahl trivial: Wir wählen q_1 mit $m = \{1\}$



$\{1\}$

Es gibt Knoten mit Inschriften: „Ziffer“, „.“, „E“, „+“, „-“.

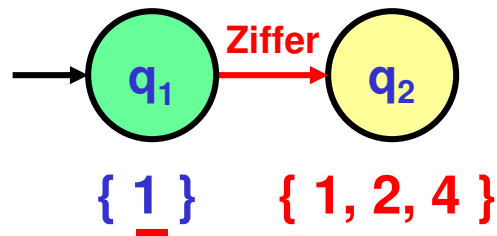
Wählen wir eines der Zeichen „.“, „E“, „+“, „-“, dann finden wir keinen Knoten mit diesem Zeichen als Inschrift und Marke in m . Damit wird Schritt 4 des Algorithmus trivial:

Wir wählen das Zeichen „Ziffer“.

Bestimme die Knotenmenge m' ; füge dem Automaten eine Transition hinzu.

Das Zeichen „Ziffer“ tritt auf als Inschrift der Knoten mit Marken 1, 3 und 7.

Aber nur die Marke 1 ist in der Menge m des gewählten Zustandes q_1 .



$\{ \underline{1} \}$

$\{1, 2, 4\}$

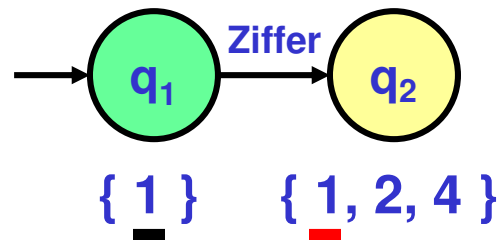
Vom Knoten mit Marke 1 aus direkt erreichbar sind die Knoten mit den Marken 1, 2, 4. Damit gilt $m' = \{1, 2, 4\}$ und es ergibt sich

- ein **neuer Zustand** für den Automaten und
- eine **neue Kante** vom gewählten Zustand q_1 zum neuen Zustand q_2 , die
- mit dem **gewählten Zeichen** „Ziffer“ **beschriftet** wird.

Beispiel: Konstruktion des endlichen Automaten (3)

Wähle einen Zustand und ein Zeichen

Wir wählen jetzt q_2 , den neu hinzu gekommenen Zustand.

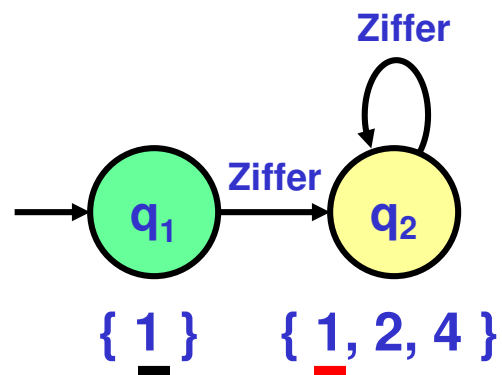


Dem Zustand q_2 ist die Markenmenge $\{1, 2, 4\}$ zugeordnet. Die zugehörigen Knoten im Syntaxdiagramm tragen die Inschriften

- „Ziffer“ (Knoten mit Marke 1),
- „.“ (Knoten mit Marke 2) und
- E (Knoten mit Marke 4).

Wähle zunächst das Zeichen „Ziffer“.

Bestimme die Knotenmenge m' ; füge dem Automaten eine Transition hinzu.

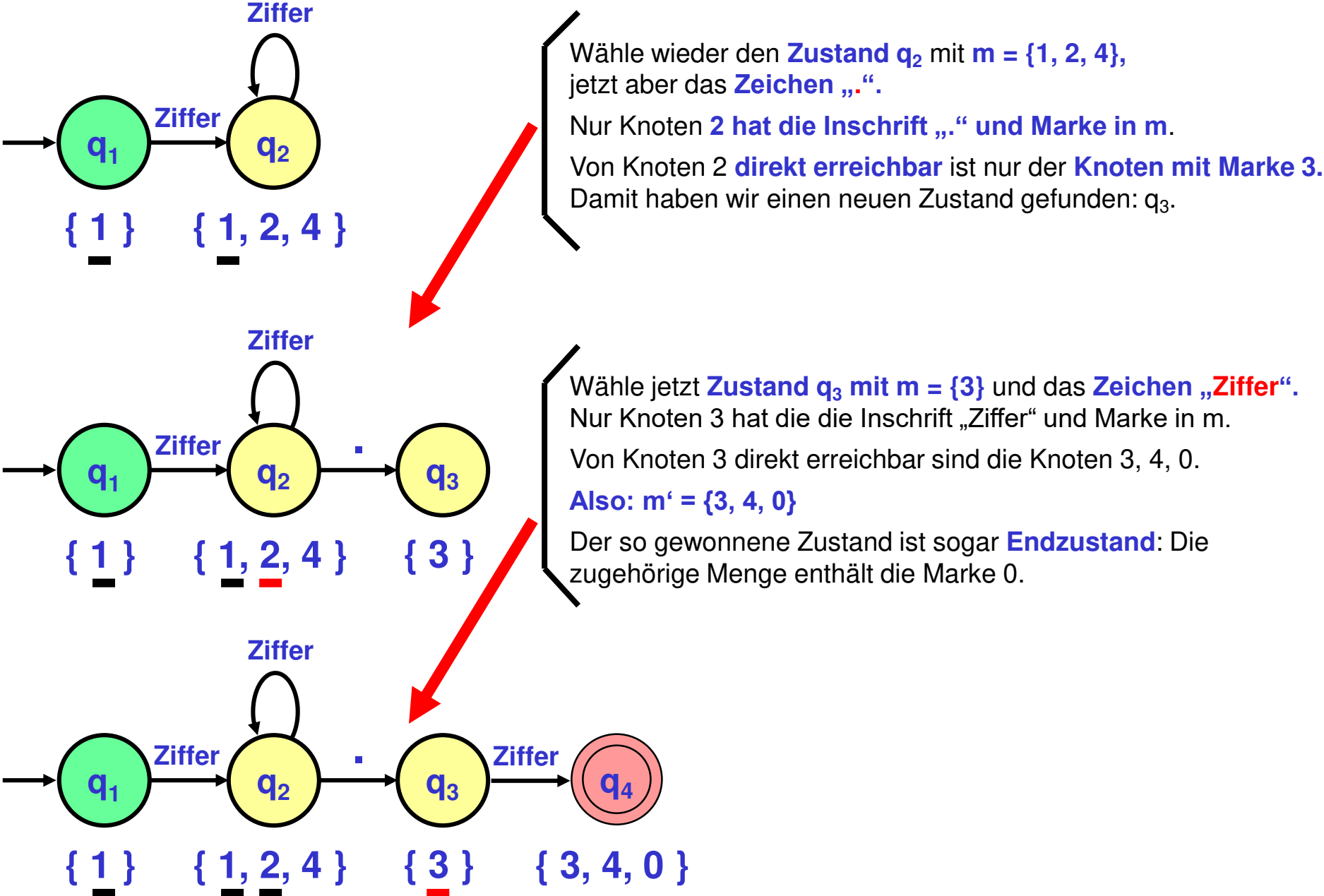


Das gewählte Zeichen „Ziffer“ ist **Inschrift der Knoten mit den Marken 1, 3 und 7**. Aber **nur die Marke 1 ist in der Menge m** des gewählten Zustandes q_2 .

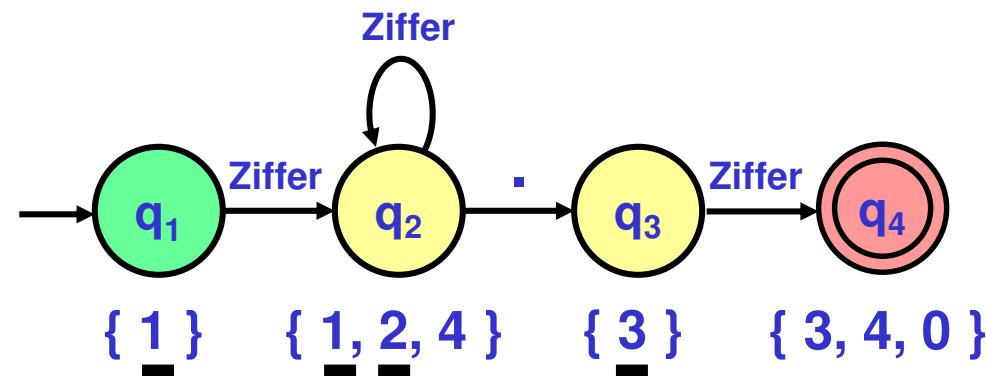
Vom Knoten mit Marke 1 aus **direkt erreichbar** sind die Knoten mit den Marken **1, 2, 4**. Damit folgt **$m' = \{1, 2, 4\}$** . Der zugehörige Zustand ist bereits als q_2 im Automaten enthalten.

Die gerade bestimmte Kante im Automatendiagramm (**$\delta(q_2, \text{Ziffer}) = q_2$**) führt also vom „alten“ Zustand q_2 zum „alten“ Zustand q_2 zurück.

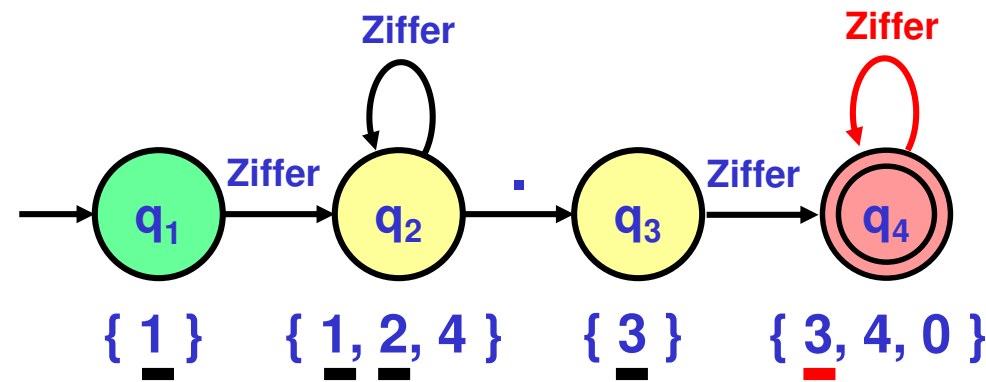
Beispiel: Konstruktion des endlichen Automaten (4)



Beispiel: Konstruktion des endlichen Automaten (5)

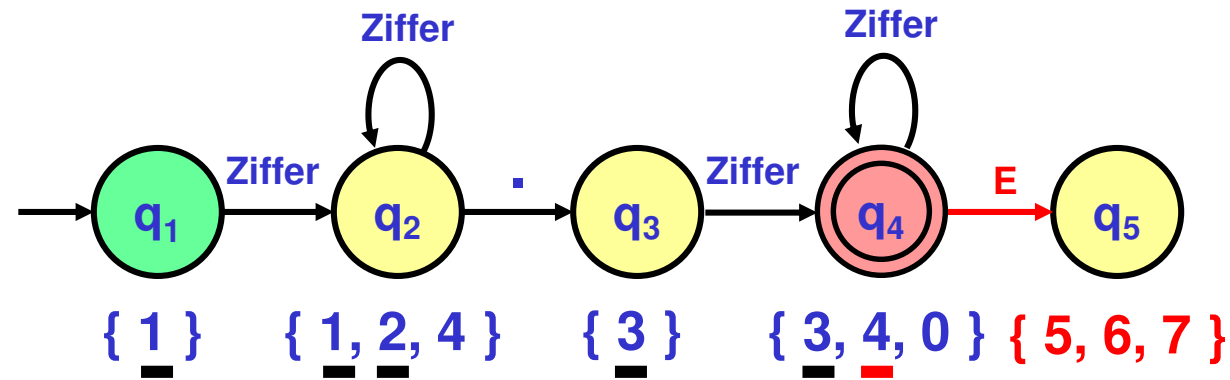


gewählter Zustand	gewähltes Zeichen	Inschrift o.k., Marke in m	m'
q_4	Ziffer	Knoten 3	$\{3, 4, 0\}$ alt!



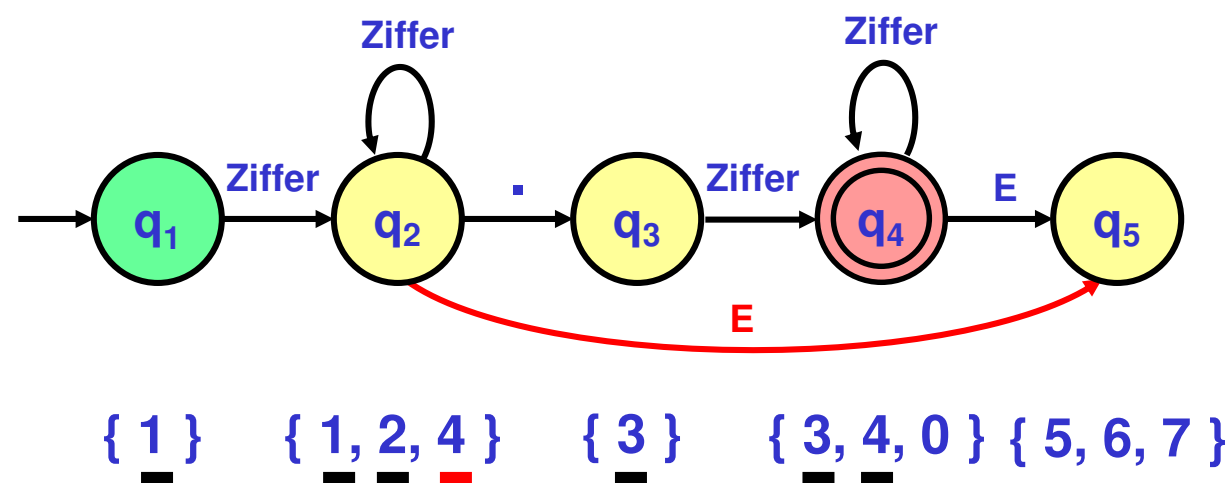
Beispiel: Konstruktion des endlichen Automaten (6)

gewählter Zustand	gewähltes Zeichen	Inschrift o.k., Marke in m	von dort direkt erreichbar
q_4	E	Knoten 4	{5, 6, 7} neu!



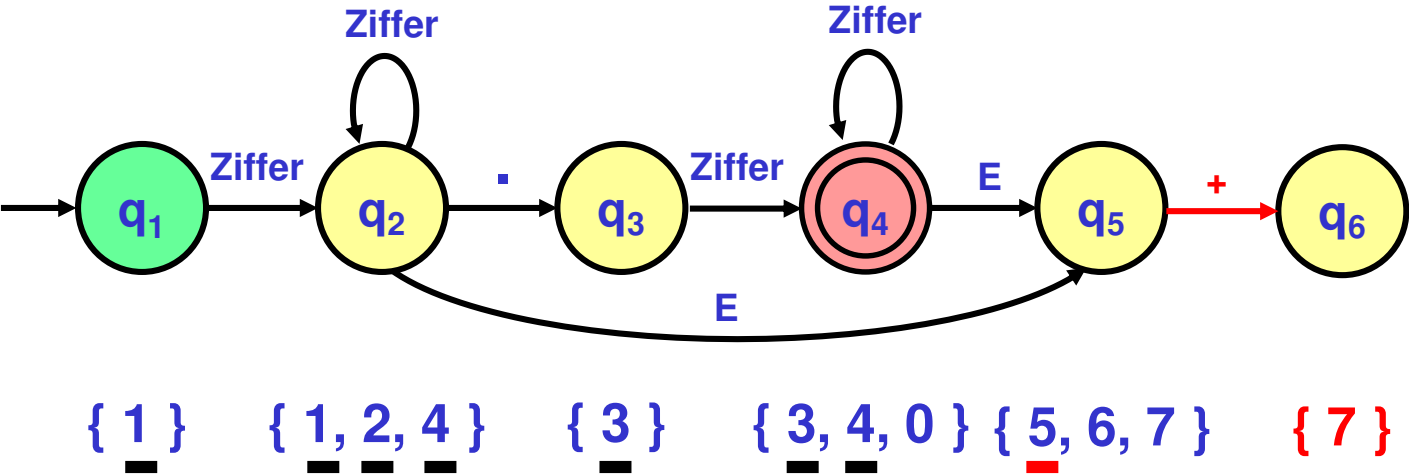
Beispiel: Konstruktion des endlichen Automaten (7)

gewählter Zustand	gewähltes Zeichen	Inschrift o.k., Marke in m	von dort direkt erreichbar
q_2	E	Knoten 4	{5, 6, 7} alt!



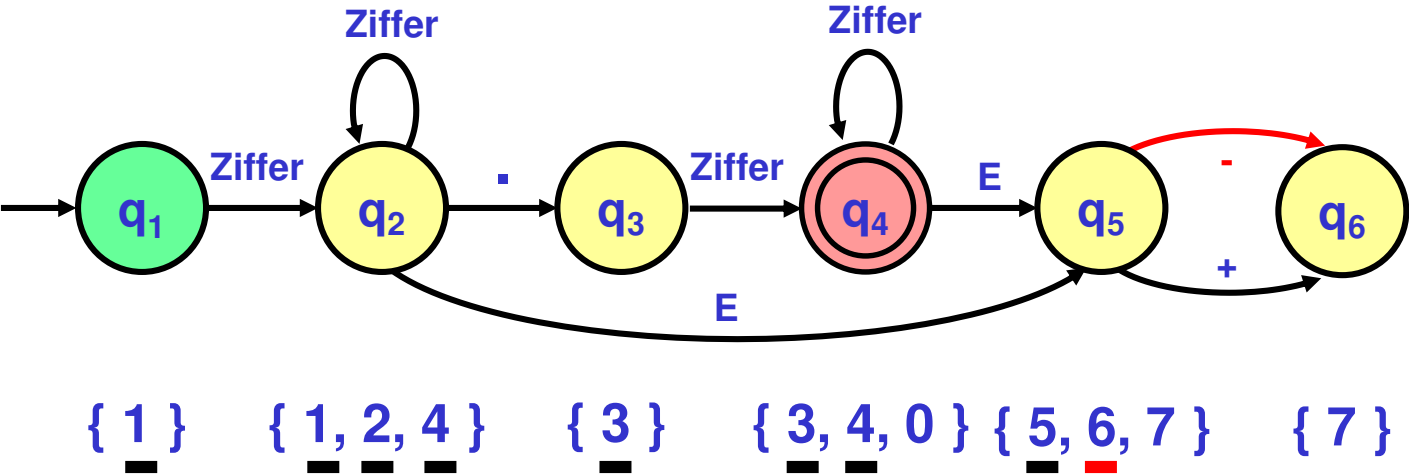
Beispiel: Konstruktion des endlichen Automaten (8)

gewählter Zustand	gewähltes Zeichen	Inschrift o.k., Marke in m	von dort direkt erreichbar
q_5	+	Knoten 5	{7} neu!



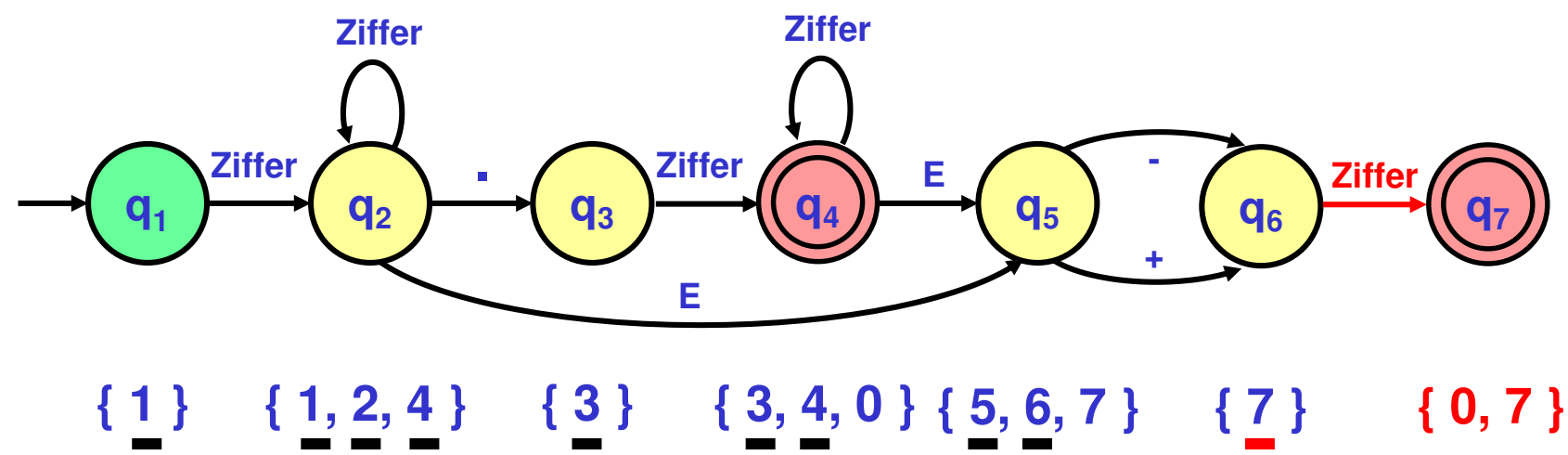
Beispiel: Konstruktion des endlichen Automaten (9)

gewählter Zustand	gewähltes Zeichen	Inschrift o.k., Marke in m	von dort direkt erreichbar
q_5	-	Knoten 6	$\{7\}$ alt!



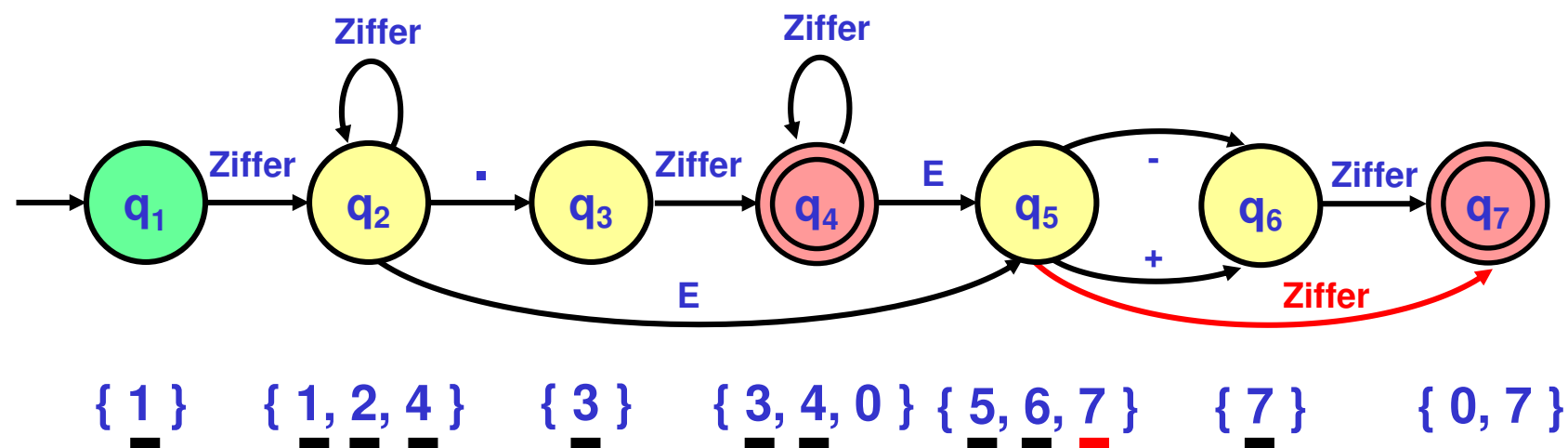
Beispiel: Konstruktion des endlichen Automaten (10)

gewählter Zustand	gewähltes Zeichen	Inschrift o.k., Marke in m	von dort direkt erreichbar
q_6	Ziffer	Knoten 7	$\{0, 7\}$ neu, Endzustand!



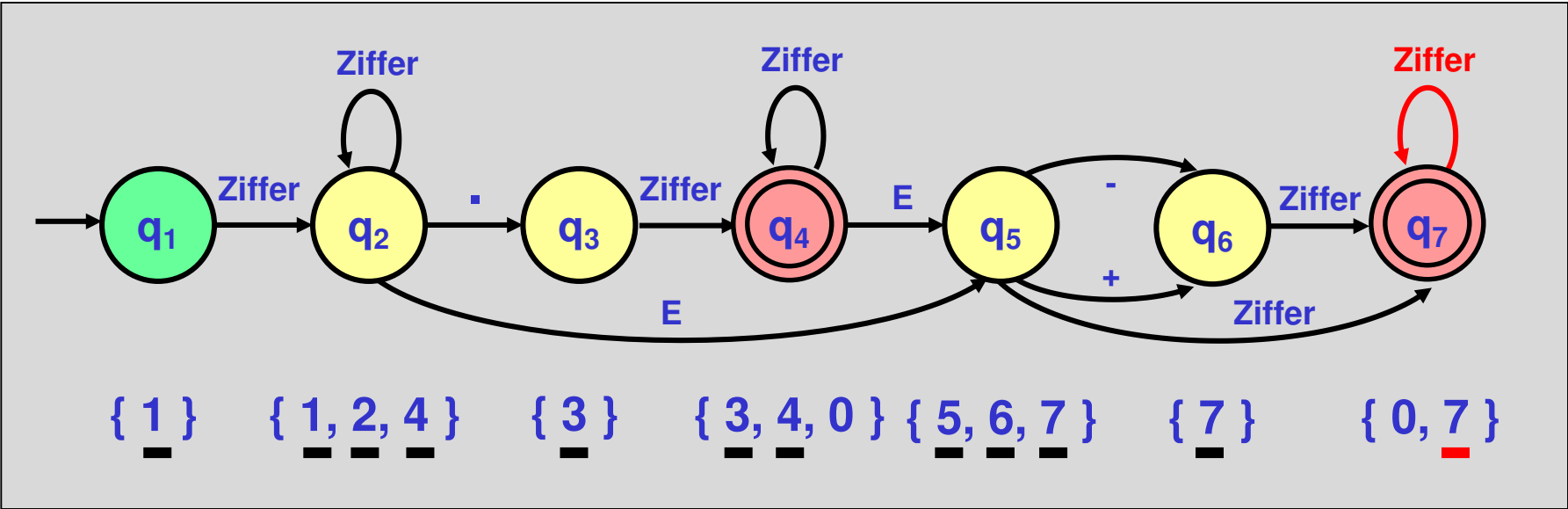
Beispiel: Konstruktion des endlichen Automaten (11)

gewählter Zustand	gewähltes Zeichen	Inschrift o.k., Marke in m	von dort direkt erreichbar
q₅	Ziffer	Knoten 7	{0, 7} alt!



Beispiel: Konstruktion des endlichen Automaten (12)

gewählter Zustand	gewähltes Zeichen	Inschrift o.k., Marke in m	von dort direkt erreichbar
q₇	Ziffer	Knoten 7	{0, 7} alt!



Für alle anderen hier nicht betrachteten Wahlen von Zuständen und Zeichen ergibt sich weder ein neuer Zustand noch eine neue Kante. Der gesuchte Spracherkenner ist damit gefunden.

2.5.6. Die Regel des längsten Musters

Frage 1: Wann hält der endliche Automat zur Grundsymbol-Erkennung ?

Antwort: Für die Erkennung von Grundsymbolen wird die „Regel des längsten Musters“ eingesetzt:

 Ist mit dem aktuellen Zeichen kein weiterer Übergang möglich, dann hält der Automat.

Frage 2: Welches ist das erkannte Grundsymbol ?

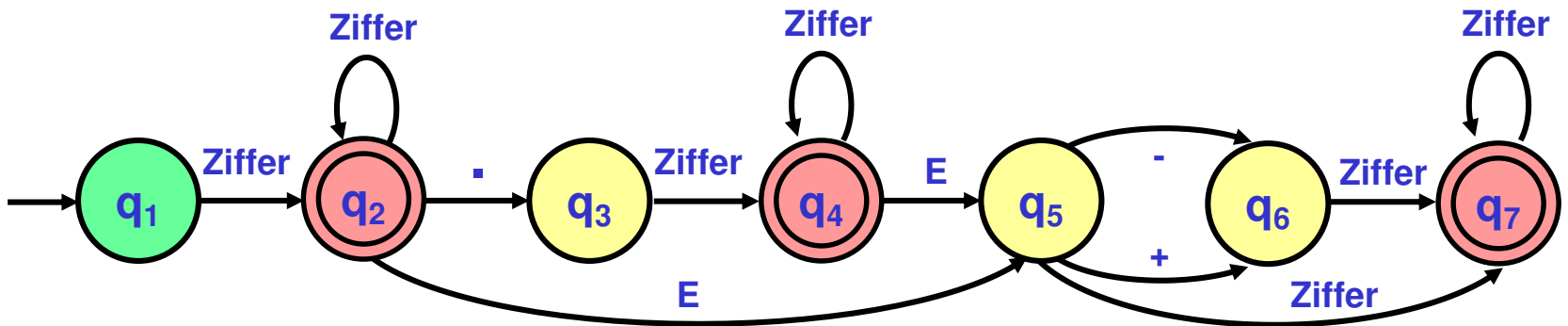
Antwort: Es ist die **bis zum** zuletzt durchlaufenen **Endzustand** akzeptierte Zeichenfolge.

 Alles, was zwischen dem letzten Endzustand und dem Anhalten des Automaten folgte, gehört zu einem nachfolgenden Grundsymbol.

Beispiele: Regel des längsten Musters

Für die nachfolgenden Betrachtungen modifizieren wir den konstruierten endlichen Automaten zu Erkennung von Gleitpunktzahlen so, dass **auch ganze Zahlen akzeptiert** werden.

➡ **Wähle auch Zustand q_2 als Endzustand.**



Der so modifizierte Automat erkennt in den Anfängen der Folgen

245-	die ganze Zahl	245	(endet in Zustand q_2)
245.62+	die Gleitpunktzahl	245.62	(endet in Zustand q_4)
10..20	die Gleitpunktzahl	10	(endet in Zustand q_3)
10END	die ganze Zahl	10	(endet in Zustand q_5)
.53	einen	Fehler	(kein Übergang, endet in q_1)

2.6. Syntaktische Analyse

Die lexikalische Analyse liefert die Folge der Grundsymbole (Token) in der Reihenfolge, in der diese in dem vorgegebenen Programm auftreten. Im nächsten Schritt wird zu dieser Tokenfolge ein „**Ableitungsbaum**“ bestimmt:

Die syntaktische Analyse

- **ermittelt** aus der Tokenfolge die **Struktur des Programms**,
- **repräsentiert** diese **als abstrakte Datenstruktur** für die weitere Übersetzung,
- meldet einen **Fehler**, wenn das Programm nicht den Regeln für zulässige syntaktische Strukturen der jeweiligen Programmiersprache entspricht.

Anschaulich:

- Die **lexikalische Analyse** ermittelt „die **Worte** des Programms“.
- Die **syntaktische Analyse** ermittelt „den **Satzbau** des Programms“.

Die **Regeln der syntaktischen Struktur** von Programmen werden durch **kontextfreie Grammatiken** formal spezifiziert.

[3.6.1. Zerteiler \(Parser\)](#)

[3.6.2. Arithmetische Ausdrücke](#)

3.6.1. Zerteiler (Parser)

Ein Programm, das die **syntaktische Struktur eines Quellprogramms bestimmt**, bezeichnet man als „**Parser**“ (dt.: „Zerteiler“).

Für die **lexikalische Analyse**

war es möglich, zu einer vorgegebenen Spezifikation (z.B. Syntaxdiagramm) den zugehörigen **Analysealgorithmus automatisch** zu **konstruieren** und in Form eines **endlichen Automaten** formal zu beschreiben.

Für die **syntaktische Analyse**

ist es auch möglich, zu einer vorgegebenen Spezifikation den zugehörigen **Analysealgorithmus automatisch** zu **konstruieren**. Da die für die Spezifikation eingesetzten kontextfreien **Grammatiken** aber i.A. **nicht regulär** sind, ist das formale Modell des endlichen Automaten nicht ausreichend: Wir benötigen einen „**Kellerautomaten**“ (**Pushdown Automaton**).

Kellerautomat (Pushdown Automaton, PDA)

Ein Kellerautomat hat - wie ein endlicher Automat - **endlich viele Zustände**.

Er verfügt aber zusätzlich über einen **Kellerspeicher** (Stack) und damit über ein unendliches „Gedächtnis“. Weitere wichtige Unterschiede zum endlichen Automaten:

- Es gibt „ **ϵ -Moves**“ (Zustandswechsel und Stack-Manipulation ohne „Verbrauch“ von Input)
- Der Automat ist **nicht-deterministisch**.

Definition: Kellerautomat

Eine Kellerautomat ist **ein Tupel $A = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$** mit

Q : (endliche) Zustandsmenge

Σ : (endlicher) Zeichensatz (genannt: Eingabe-Alphabet)

Γ : (endlicher) Zeichensatz (genannt: Stack-Alphabet)

q_0 : Anfangszustand ($q_0 \in Q$)

Z_0 : ein spezielles Stack-Symbol $Z_0 \in \Gamma$ (genannt: Startsymbol)

F : Menge der Endzustände ($Q \supset F$)

δ : eine Abbildung von $Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma$ zu einer endlichen Teilmenge von $Q \times \Gamma^*$

Aus nicht-deterministisch mach' deterministisch ?

Man kann zeigen:

- a1: Zu jeder **regulären Grammatik** gibt es einen **endlichen Automaten**, der die mittels Grammatik spezifizierte Sprache akzeptiert (und umgekehrt).
- a2: Zu jedem **nicht-deterministischen endlichen Automaten** gibt es einen (deterministischen) **endlichen Automaten**, der die gleiche Sprache akzeptiert (und umgekehrt).

- b1: Zu jeder **kontextfreien Grammatik** gibt es einen **Kellerautomaten**, der die mittels Grammatik spezifizierte Sprache akzeptiert (und umgekehrt).
- b2: **Deterministische Kellerautomaten** akzeptieren nur eine **Teilmenge der kontextfreien Sprachen**.

Detaillierte Betrachtungen zur syntaktischen Analyse setzen Kenntnisse aus dem Gebiet „Formale Sprachen“ voraus. Im Folgenden beschränken wir uns auf elementare Betrachtungen zur Strukturanalyse von arithmetischen Ausdrücken.

2.6.2. Arithmetische Ausdrücke

Elementare arithmetische Ausdrücke mit Addition und Subtraktion können mit einer sehr einfachen kontextfreien Grammatik formal beschrieben werden:

$$\begin{aligned} N &= \{E\} \\ T &= \{+, -, (,), \text{id}\} \\ P &= \{E \rightarrow E + E, E \rightarrow E - E, E \rightarrow (E), E \rightarrow \text{id}\} \end{aligned}$$

Anmerkung: „id“ sei ein Bezeichner (identifizier)

Die von dieser Grammatik generierte Sprache kann durch

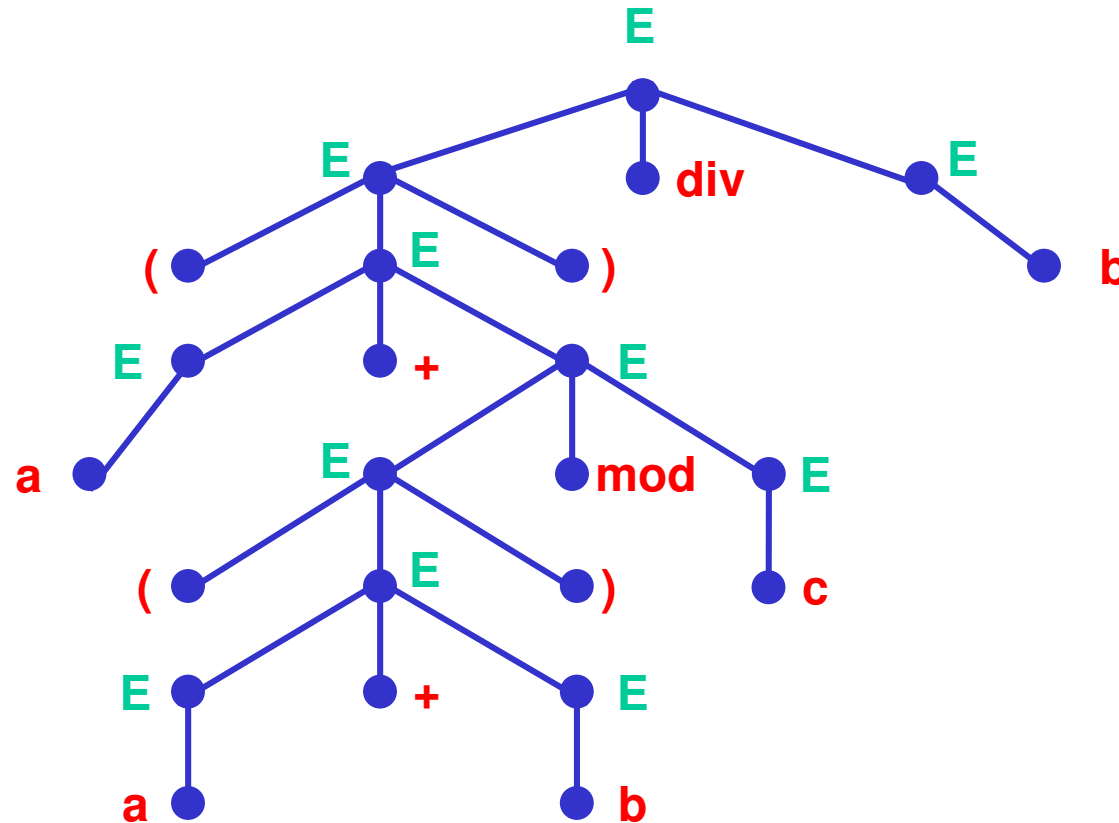
- **zusätzliche Terminalsymbole für Operationen** (\cdot , div, mod, exp, log, ...)
- **zusätzliche Terminalsymbole für beliebige Operanden** („Bezeichner“ statt „id“) und
- Hinzufügung **entsprechender Regeln**

leicht auf **alle arithmetischen Ausdrücke erweitert** werden.

Somit sind arithmetische Ausdrücke als Ableitungsbäume darstellbar.

Beispiel: Arithmetischer Ausdruck als Ableitungsbaum

Als Beispiel stellen wir nun den Ausdruck $(a + (a + b) \bmod c) \div b$ als Ableitungsbaum dar.

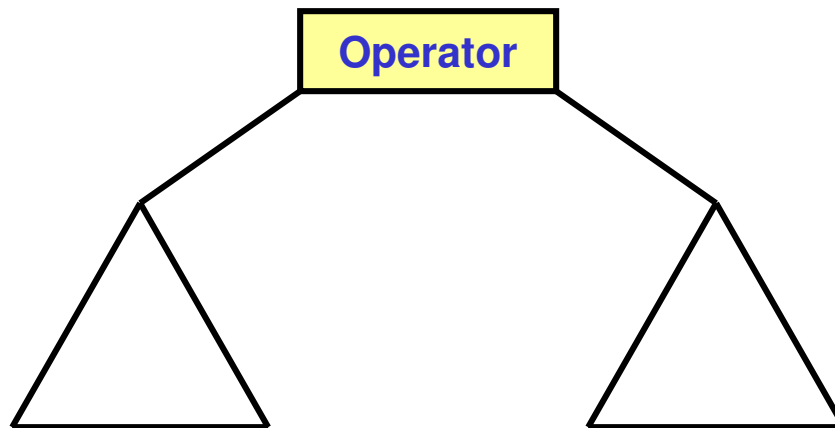


Darstellung ohne Klammerung

Die Darstellung eines arithmetischen Ausdrucks kann auch ohne Klammern erfolgen. Dann benötigen wir aber andere Konventionen zur Festlegung der Prioritäten.

Idee:

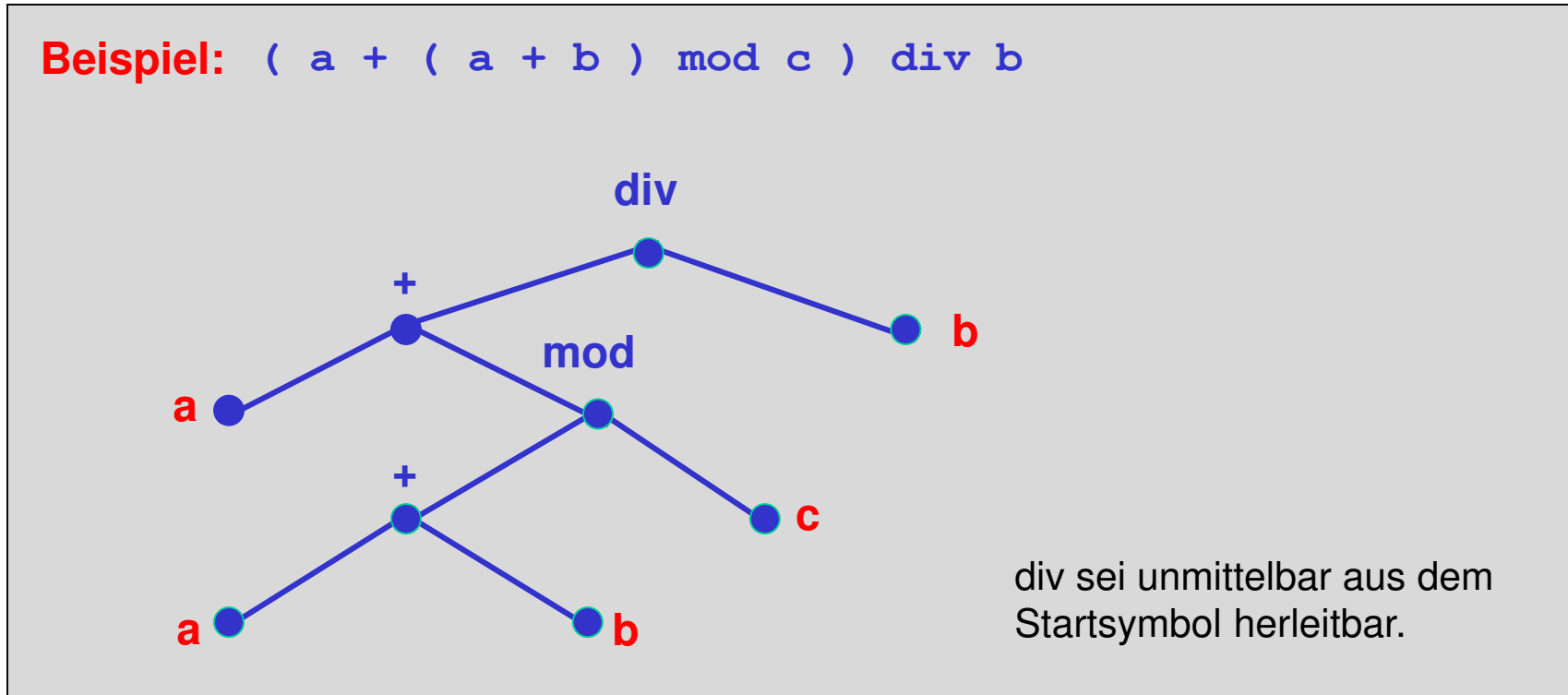
- Verwende die **Operatoren als Nicht-Terminale**, aus denen die Operanden abgeleitet werden können.
- Die Operatoren beschreiben arithmetische Operationen mit den **Teilbäumen als Operanden**.
- Bei zweistelligen Operationen beschreibt das **linke Kind den ersten Operanden**, das **rechte Kind den zweiten Operanden**.



Die durch den Operator angegebene Operation verwendet als Operanden die (Ergebnisse der) Teilbäume.

Beispiel: Darstellung ohne Klammerung

Bei Kenntnis der einschlägigen Konventionen (z.B. Punktrechnung vor Strichrechnung, Beachtung der Klammerung) ergibt sich die Strukturbeschreibung unmittelbar aus dem arithmetischen Ausdruck.



Beispiel: Bestimmung der syntaktischen Struktur

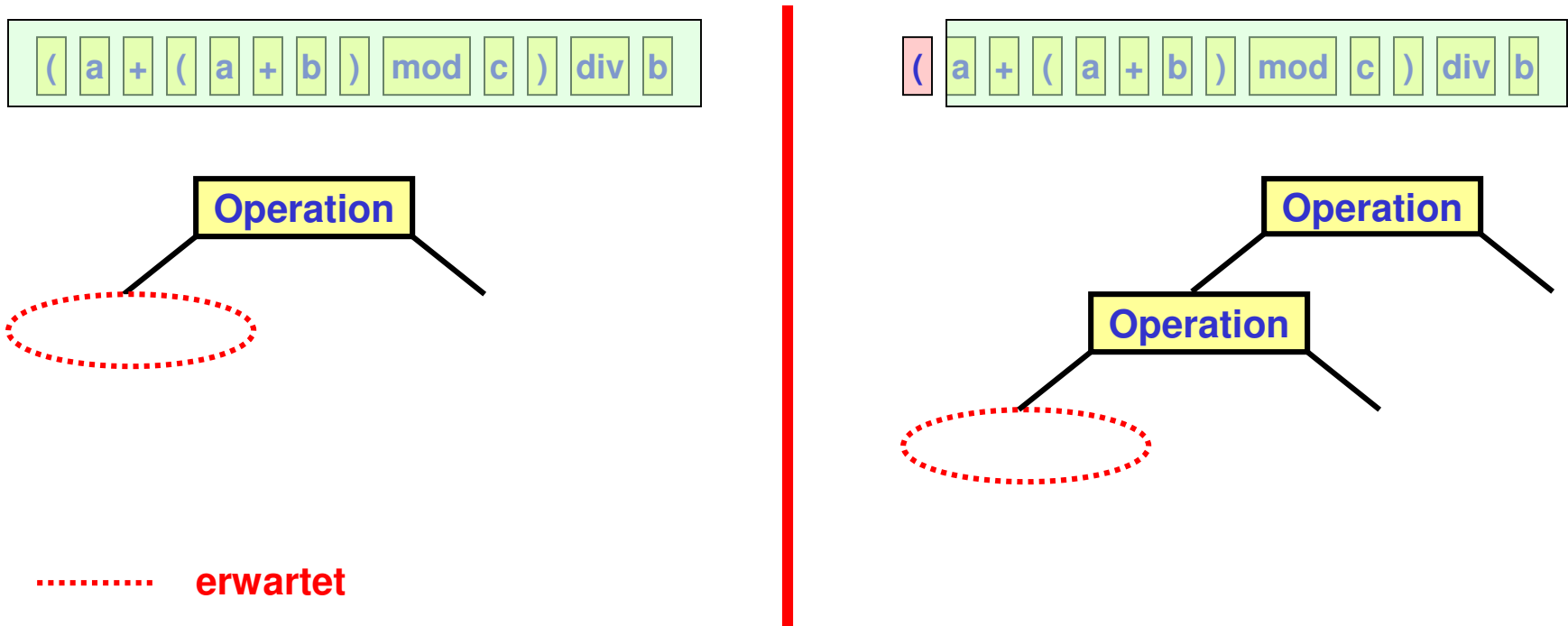
Im Folgenden betrachten wir detailliert die Gewinnung der syntaktischen Struktur eines vorgegebenen arithmetischen Ausdrucks in Infix-Notation.

Bei Vorgabe von

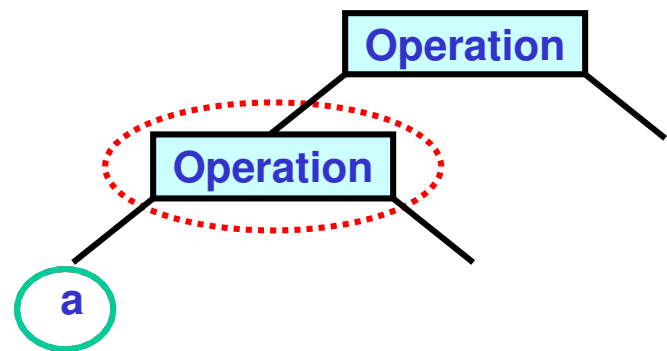
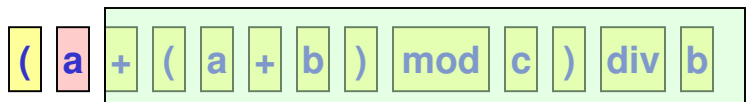
(a + (a + b) mod c) div b

habe die lexikalische Analyse geliefert:

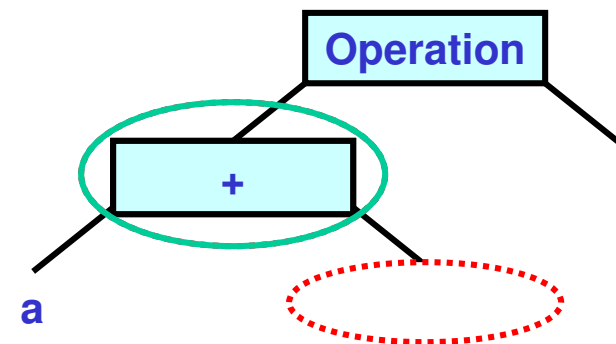
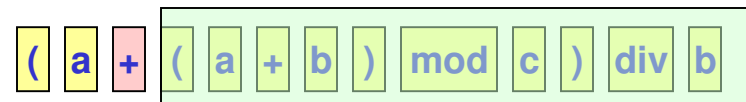
(a + (a + b) mod c) div b



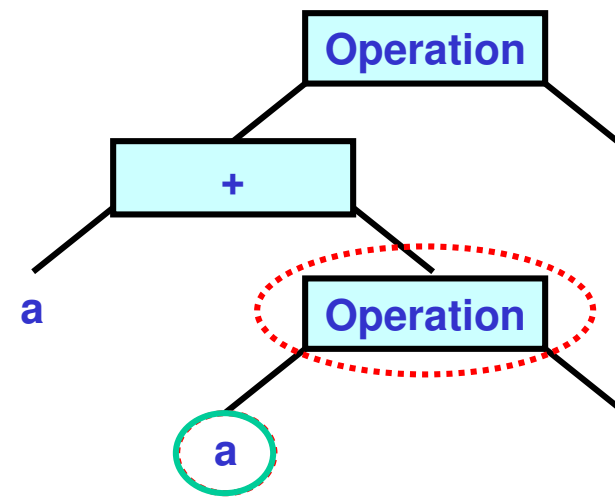
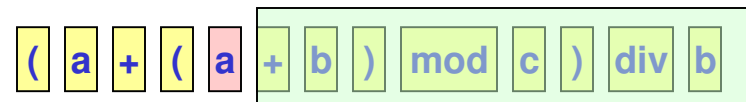
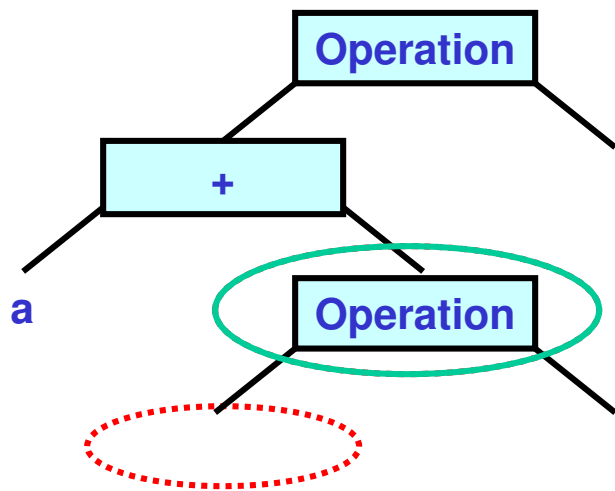
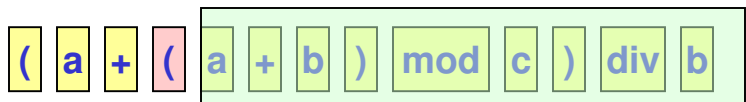
Beispiel: Bestimmung der syntaktischen Struktur (2)



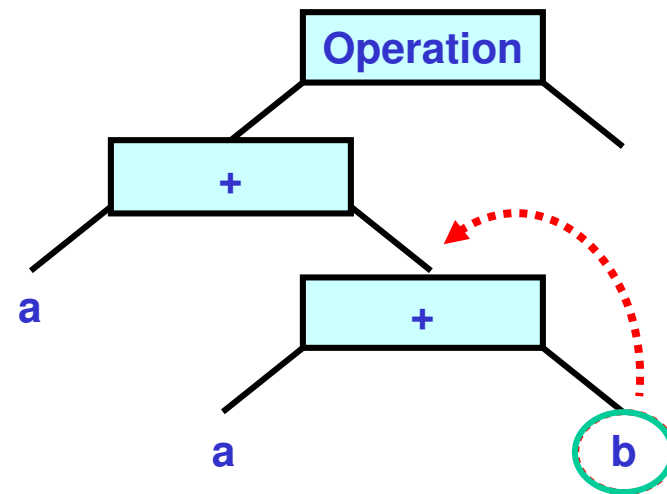
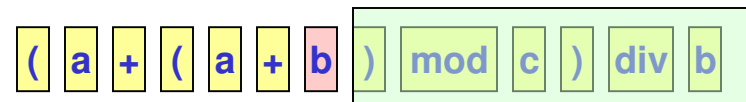
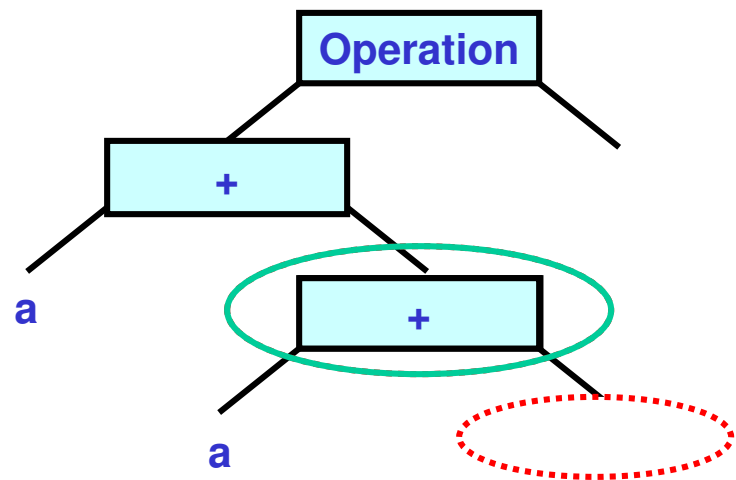
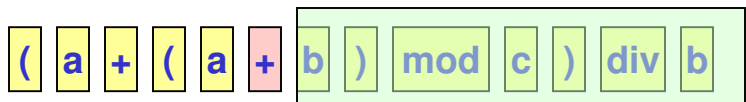
——— aktuell
..... erwartet



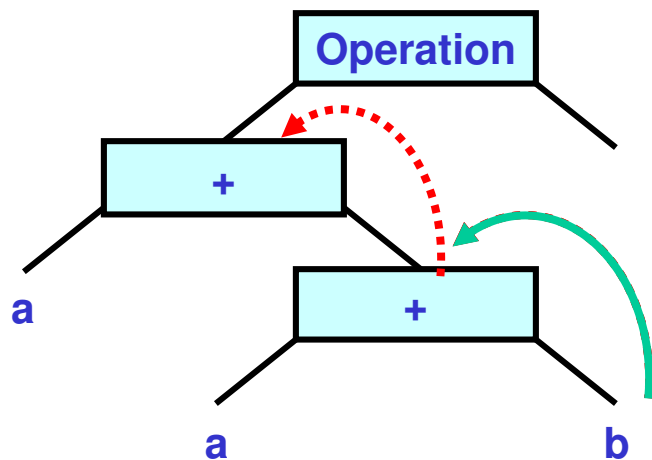
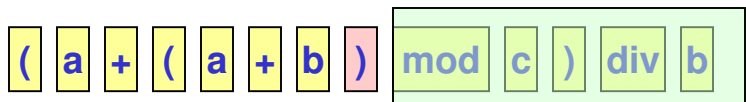
Beispiel: Bestimmung der syntaktischen Struktur (3)



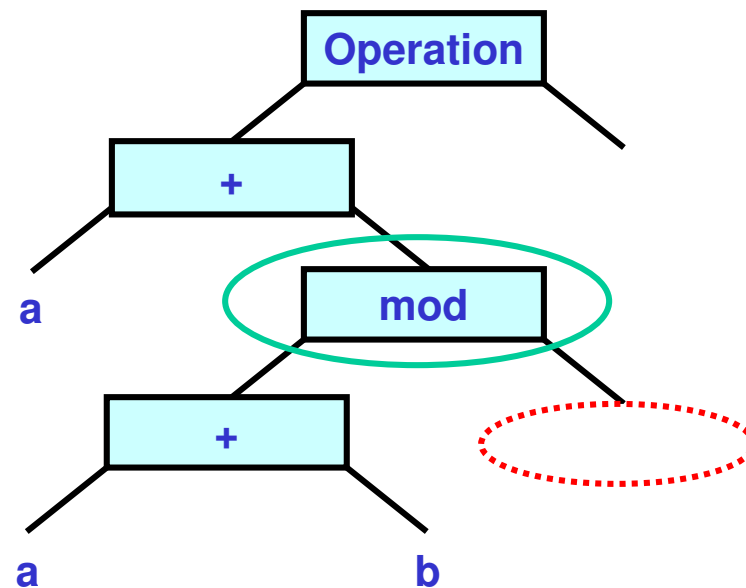
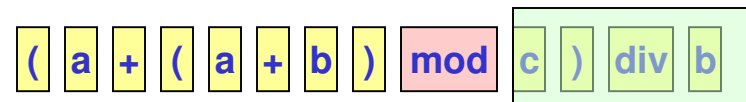
Beispiel: Bestimmung der syntaktischen Struktur (4)



Beispiel: Bestimmung der syntaktischen Struktur (5)

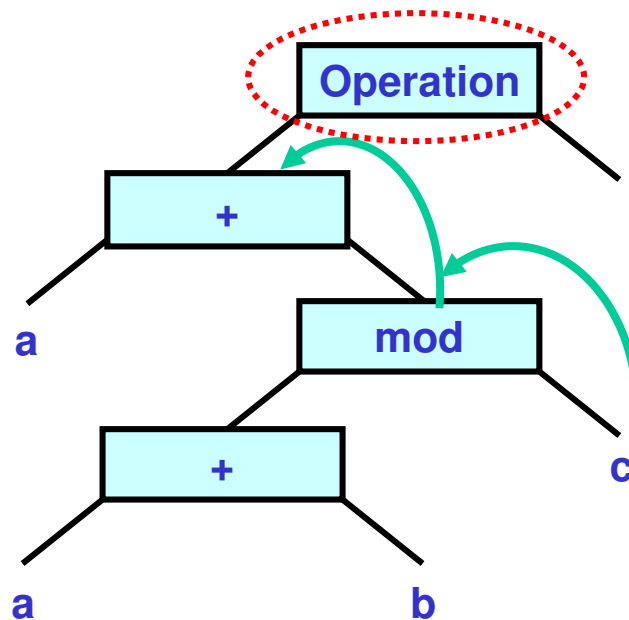
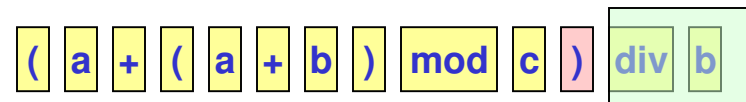
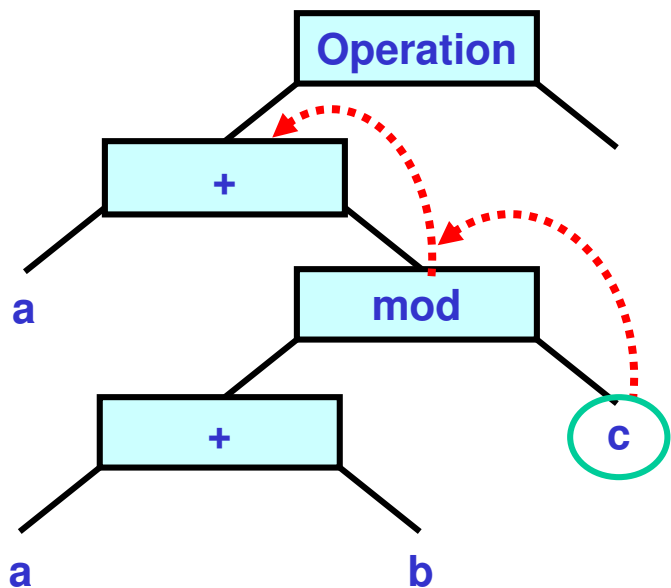
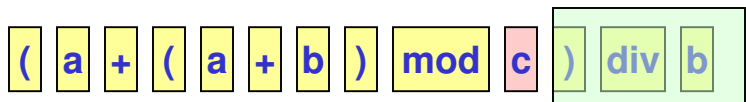


Wir erwarten: „)“



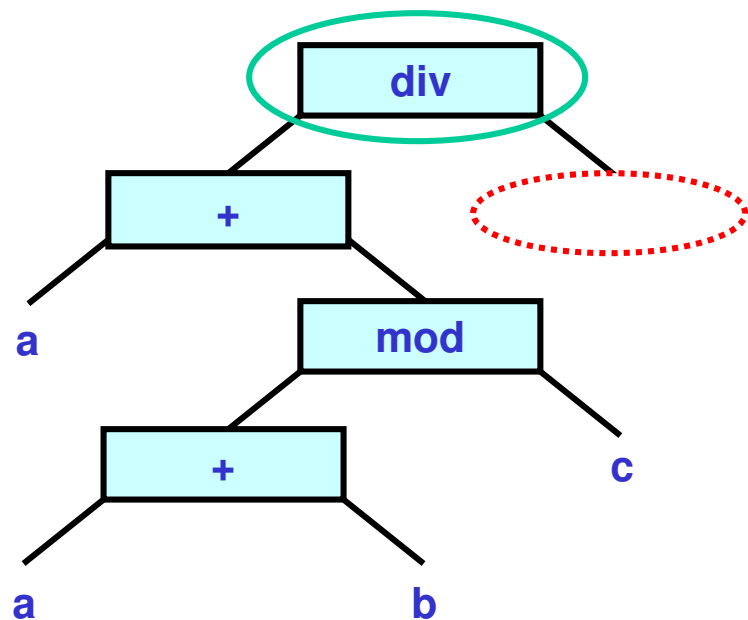
Der Baum wurde umstrukturiert, da „mod“ stärker bindet als „+“

Beispiel: Bestimmung der syntaktischen Struktur (6)

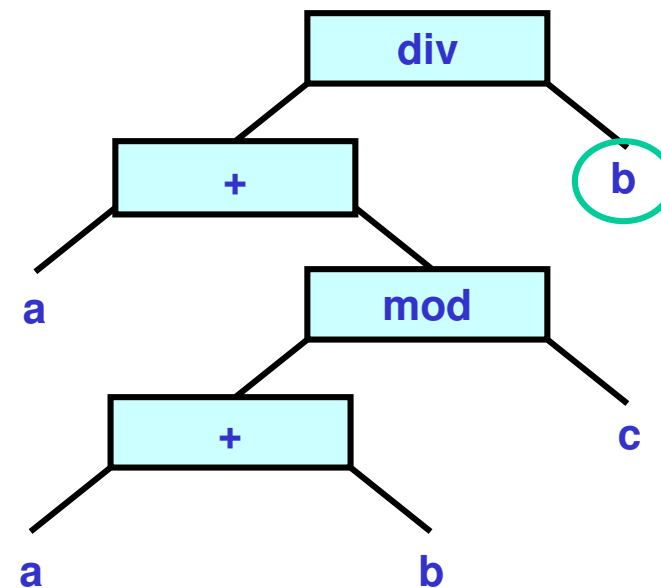


Beispiel: Bestimmung der syntaktischen Struktur (7)

(a + (a + b) mod c) div b



(a + (a + b) mod c) div b



2.7. Semantische Analyse

Mit **kontextfreien Grammatiken** können wir nur **Obermengen der höheren Programmiersprachen** beschreiben:

Korrekte syntaktische Struktur \nRightarrow **gültiges Programm**

Der Grund liegt darin, dass den Sprachelementen Eigenschaften zugeordnet werden, die abhängig vom Kontext sind, in dem diese Sprachelemente auftreten.

Beispiel 1: Typ eines Ausdrucks

Aufgrund des Einsatzes überladener Operatoren kann der Typ von Ausdrücken und Operanden häufig nur als kontextabhängiges Attribut bestimmt werden („**Typprüfung**“).

Beispiel 2: Identifikation eines Bezeichners

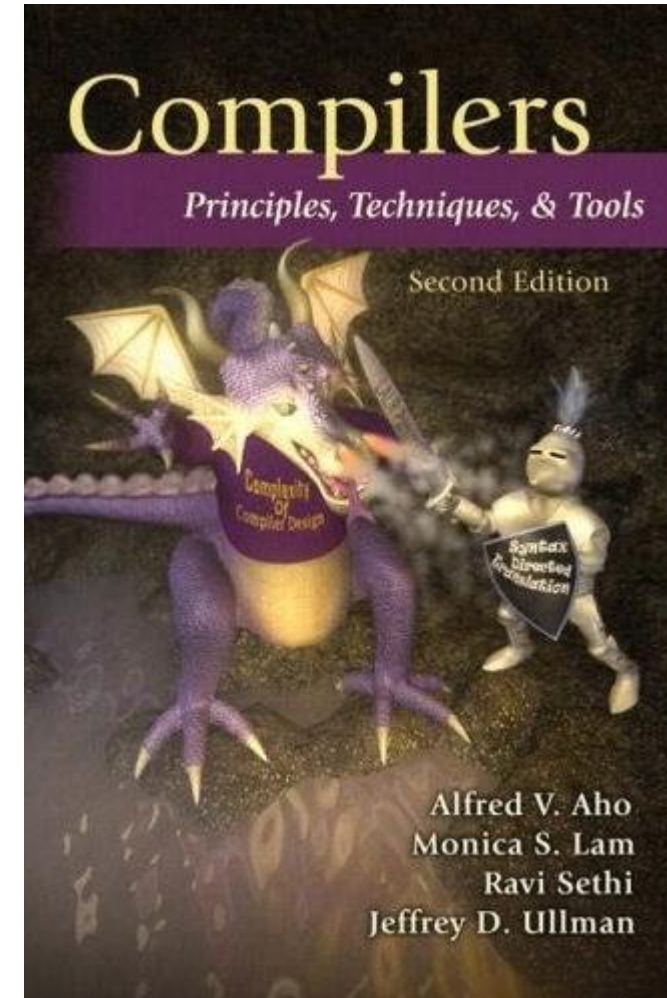
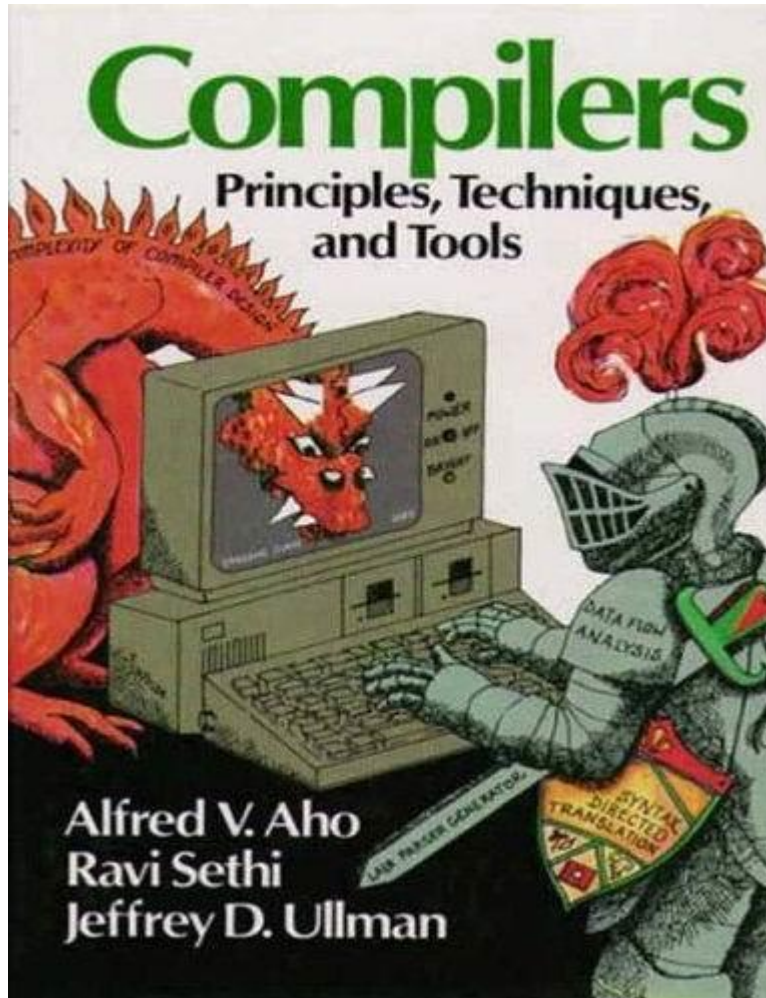
Die Gültigkeitsregeln einer Programmiersprache legen fest, in welchem Kontext die Definition eines Bezeichners gültig ist („**Bezeichneridentifikation**“).

Detaillierte Betrachtungen der semantischen Analyse setzen Kenntnisse aus dem Gebiet der **attributierten Grammatiken** voraus.

Wir verzichten hier auf eine Behandlung und verweisen auf die Literatur, insbesondere auf den „Klassiker“:

A.V. Aho, R. Sethi, J.D. Ullmann, „Compilers: Principles, Techniques and Tools“, Addison-Wesley, 1986

„Rotes Drachenbuch“ und „Lila Drachenbuch“



2.8. Code-Erzeugung

Die Code-Erzeugung ist der **zentrale Teil der Synthese**. Sie umfasst:

a) Die Abbildung auf die Speicherstrukturen der Zielmaschine

Die „Objekte“ (im allgemeinen Sinne) der Quellsprache müssen auf die Speicherstrukturen der Zielmaschine abgebildet werden.

b) Die Abbildung auf den Befehlssatz der Zielmaschine

Die in der Quellsprache zulässigen Anweisungsfolgen müssen auf Instruktionsfolgen der Zielmaschine abgebildet werden. Offenbar ist die Befehlsauswahl nicht unabhängig von der Speicherabbildung.

Eine detaillierte Behandlung aller Aspekte der Code-Erzeugung würde den Rahmen der Vorlesung sprengen.

Exemplarisch betrachten wir in diesem Kapitel die **Zuteilung von Registern**, weil

- hiermit **Design-Entscheidungen** für eine effiziente Implementierung gut **veranschaulicht** werden können und
- die Registerverwaltung starke **Querbezüge zur Speicherverwaltung durch das Betriebssystem** hat.

[2.8.1. Registerzuteilung in einem Durchgang \(„on-the-fly“\)](#)

[2.8.2. Registerzuteilung in mehreren Durchgängen](#)

2.8.1. Registerzuteilung in einem Durchgang („on-the-fly“)

Bei der Zuteilung „on-the-fly“ erfolgt die Auswahl von Registern beim Erzeugen der jeweiligen Befehle, d.h. **ohne Berücksichtigung der Art der späteren Verwendung**. Eine „**geschickte**“ **Verwaltung** der Register ist somit **kaum möglich**.

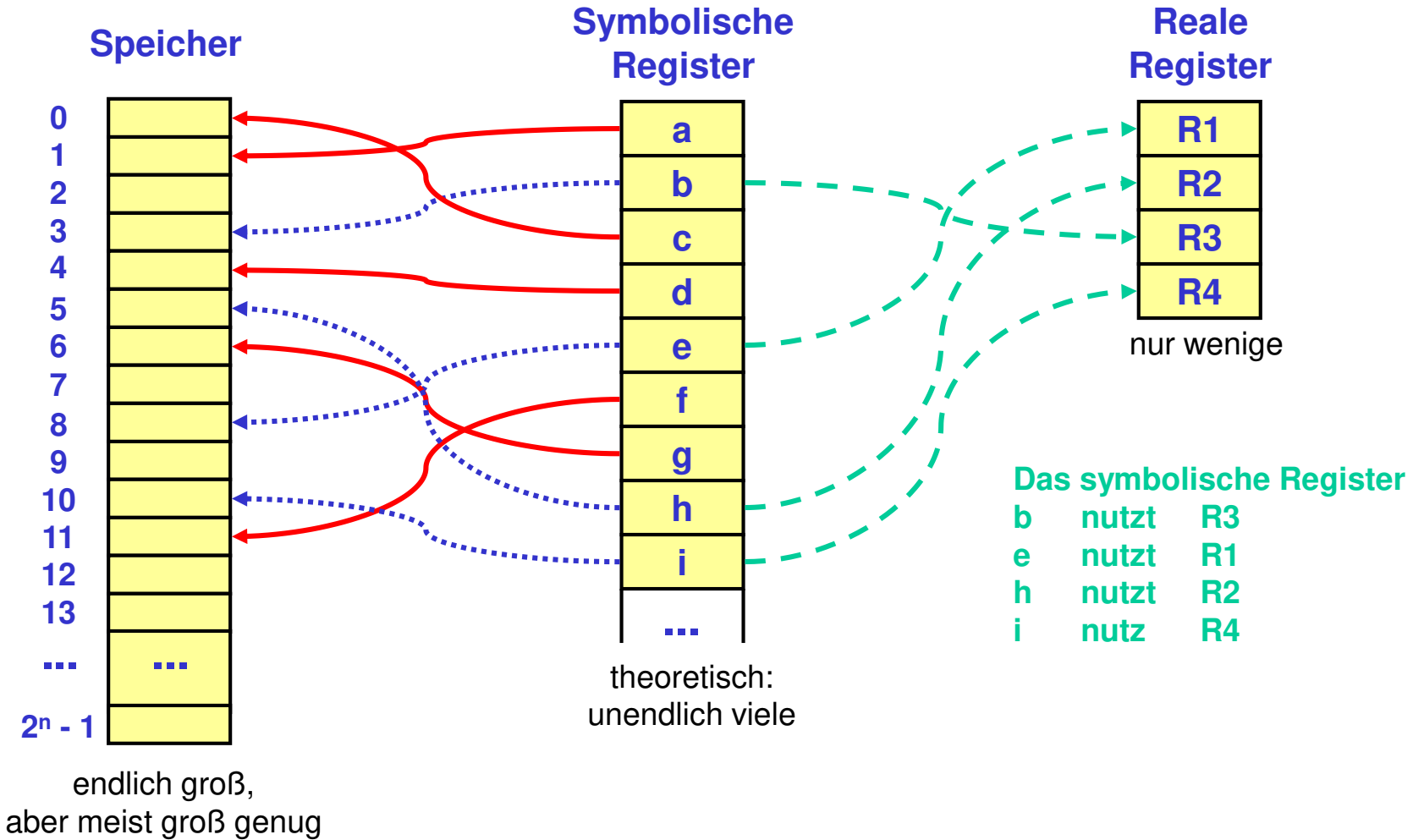
Die Registerzuteilung vereinfacht sich bei dieser Strategie zu einer leicht realisierbaren **Verwaltung freier bzw. verfügbarer Register**:

get_reg Liefert ein freies Register und kennzeichnet es als belegt.

free_reg Gibt ein zugeteiltes Register wieder frei.

Sind nicht genügend reale Register vorhanden, dann kann mit „**symbolischen Registern**“ gearbeitet werden. Jedem belegten symbolischen Register wird dabei ein reales Register oder eine Speicherzelle („**Hilfsspeicher**“) mit dem jeweils aktuellen Inhalt zugeordnet.

Zur Verwendung symbolischer Register

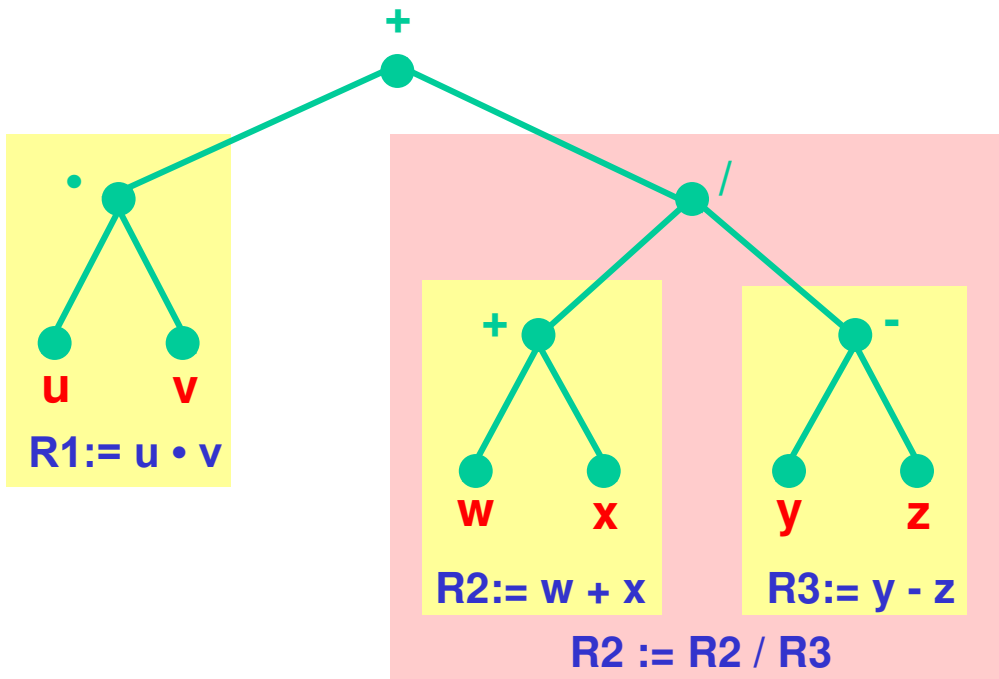


- Dem symbolischen Register entspricht aktuell kein reales Register, sein Inhalt ist „gesichert“.
- Der aktuelle Inhalt des symbolischen Registers ist in einem realen Register enthalten. Aus früherer Sicherung existiert zusätzlich noch sein ggf. veralteter Inhalt im Speicher.
- Dem symbolischen Register ist aktuell ein reales Register zugeordnet.

Beispiel: Registerzuteilung bei Ausdrucksbäumen

Seien **u, ..., z:** Symbolische Adressen (von Werten, die nicht in Registern vorliegen)
R1, ..., R4: Register

Dann kann bei einer 2-Adress-Maschine die Registerzuteilung „**aufwärts von links nach rechts**“ wie folgt vorgenommen werden:



Anmerkung:

Nach Bearbeitung eines Teilbaums werden alle intern verwendeten Register freigegeben. Nur das jeweilige Ergebnis bleibt zunächst noch erhalten.

mov u, R1

mov v, R2

}

Den Variablen u,v
Register zuordnen.

mul R2, R1

}

Den Variablen w, x
Register zuordnen.

add R3, R2

mov y, R3

mov z, R4

}

Den Variablen y, z
Register zuordnen.

sub R4, R3

div R3, R2

add R2, R1

Welches Register sichern und neu verwenden ?

Stehen nicht genügend reale Register zur Verfügung, dann stellt sich die Frage, **welches Register einer neuen Nutzung zugeführt werden soll, nachdem sein aktueller Inhalt gesichert wurde**. Bekannte Strategien sind:

a) FIFO (First-In-First-Out):

- Wähle das reale Register, das vor der längsten Zeit einem symbolischen Register zugeteilt wurde.
- Möglicherweise wird dieses aber immer noch intensiv genutzt.

b) LRU (Least Recently Used):

- Wähle das reale Register, auf dessen Inhalt am längsten nicht mehr zugegriffen wurde.

c) Zukunftsorientierung:

- Wähle das reale Register, das am längsten nicht mehr (oder gar nicht mehr) benötigt wird.
- Diese Strategie ist nicht einsetzbar bei Registerzuteilung on-the-fly.

2.8.2. Registerzuteilung in mehreren Durchgängen

Zukunftsorientierung ist möglich bei Registerzuteilung in mehreren Durchgängen:

- Beim **ersten Durchgang** wird die **Verwendung der Werte** festgestellt („lernen“).
- Beim **zweiten Durchgang** erfolgt die **Registerzuteilung**,
- Müssen Registerinhalte **verdrängt** werden, dann wählen wir solche, die erst **möglichst spät wieder benötigt** werden.

Anmerkung:

Auf die **Sicherung** von zu verdrängenden Registerinhalten **kann verzichtet werden**, **wenn** der Inhalt aus dem Speicher geholt und im **Register nicht modifiziert** wurde.

Abschließend betrachten wir noch eine mögliche Realisierung der vorausschauenden Registerzuteilung, die **Belady** für „**Seitenaustauschverfahren**“ (vgl. „virtueller Speicher“) vorgeschlagen hat.

Das Verfahren von Belady basiert auf einem **Graphen der Lebensdauer der Werte innerhalb von „Grundblöcken“**:

Ein Grundblock ist ein Code-Abschnitt ohne Marken und Sprünge.

(Kein Sprung nach Außen, kein Sprung hinein).

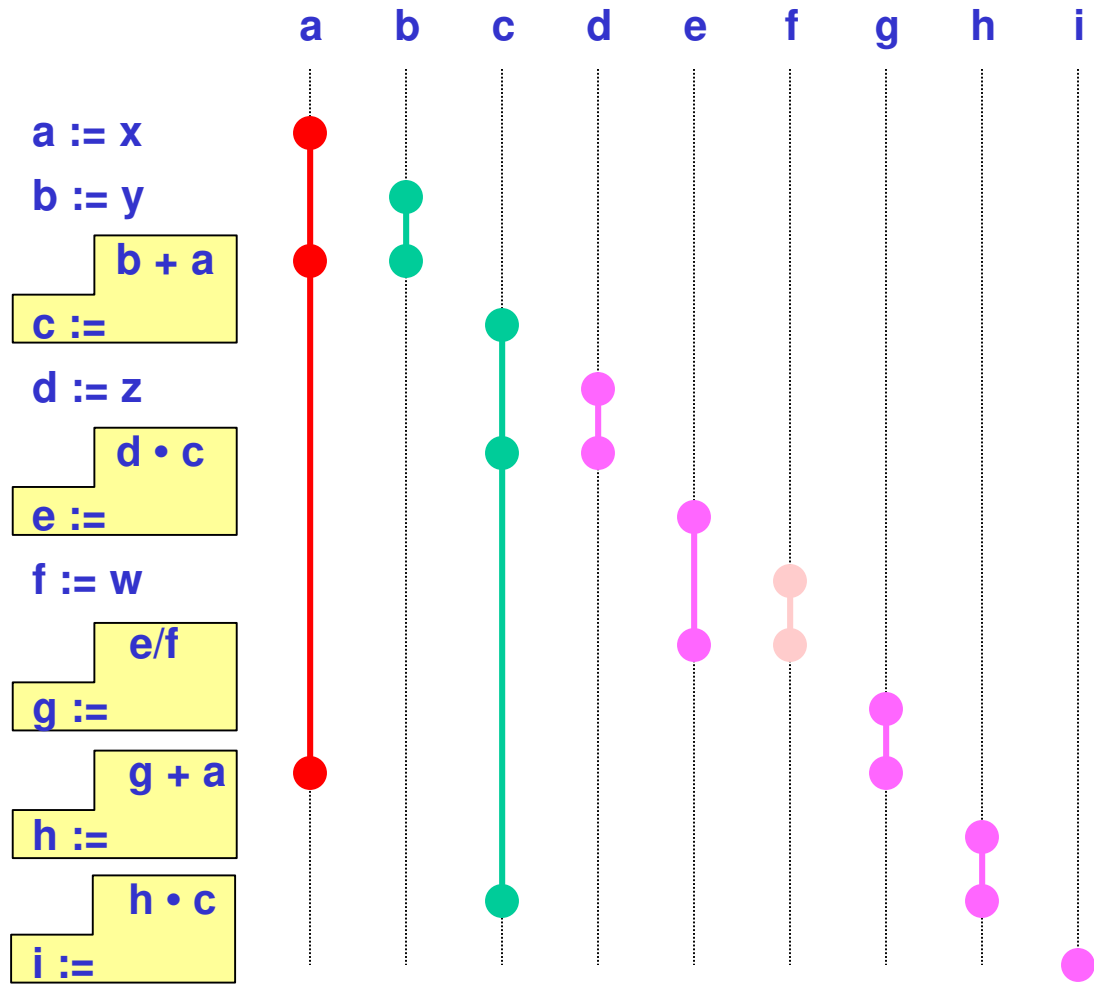
Beispiel: Lebensdauern der Werte eines Grundblocks

a, ..., i:
Deskriptoren für
Zwischen-
ergebnisse

w, x, y, z:
Symbolische
Adressen

R1, ..., R4
Register

Wir schreiben
Zuweisungen und
arithmetische
Operationen in
getrennte Zeilen.



4 Register reichen aus:	R1	R2	R2	R3	R3	R4	R3	R3	R3
3 Register, c werde verdrängt:	*					R2			
3 Register, a werde verdrängt:	*	R1							

2.9. Zusammenfassung (Kapitel 2)

- Die Funktionalität eines Laders umfasst Allocation, Relocation, Linking und Loading.
- Dynamisches Binden und Laden kann die Effizienz verteilter Systeme massiv verbessern.
- Ein Übersetzer beinhaltet Komponenten für die Analyse und für die Synthese. Die Analyse gliedert sich in die aufeinanderfolgenden Phasen „lexikalische Analyse“, „syntaktische Analyse“ und „semantische Analyse“.
- Kontextfreie Grammatiken nehmen bei der formalen Spezifikation von Programmiersprachen eine zentrale Stellung ein.
- Die lexikalische Analyse zergliedert den Quelltext in die Grundsymbole (Token) der jeweiligen Programmiersprache. Sie basiert auf dem formalen Konzept der endlichen Automaten.
- Syntaxdiagramme können automatisch in deterministische endliche Automaten umgesetzt werden, die genau die im Syntaxdiagramm spezifizierte Sprache akzeptieren.
- Nicht-deterministische endliche Automaten sind für die praktische Informatik unbrauchbar. Es kann aber stets ein äquivalenter deterministischer endlicher Automat konstruiert werden.
- Die syntaktische Struktur einer höheren Programmiersprache kann mit einer kontextfreien Grammatik formal spezifiziert werden. Eine reguläre Grammatik ist hierfür aber nicht ausreichend. Daher sind endliche Automaten nicht für die Bestimmung der syntaktischen Struktur einsetzbar. Statt dessen können Kellerautomaten verwendet werden.
- Höhere Programmiersprachen beinhalten kontextsensitive Festlegungen, die nicht mittels einer kontextfreien Grammatik vorgenommen werden können.
- Bei der Code-Erzeugung können symbolische Register eingesetzt werden. Die Abbildung auf die realen Register kann sich nur dann an der Lebensdauer von Werten orientieren, wenn diese aufgrund eines früheren Code-Durchlaufs bekannt ist.

Die Kunst der Übersetzung ... für Sie gefunden

Die kleine **Web site / Web page** Clips schließen die GIF Spinnwebe Kunst ein. Diese Bilder werden im raster eingefahren, formatiert bei einer äußerst kleinen Größe, damit sie auf einer **Spinnwebe-Seite** für **font** orte wie Knöpfe und Stangen benutzt werden können.

MasterClips **Taufbecken** Browser läßt Sie ansehen, druckt, und schließt keine der 2,000 **wahre Art-Taufbecken** in der Sammlung an. Die 2000 Taufbecken, die eingeschlossen wurden, die den Taufbecken Nutzen benutzen, mögen angeschlossen werden, oder durch Windows. Die Taufbecken und die **gesunden Wirkungen** sind nicht zugänglich durch den Browser.

Sound effects

Wenn Sie irgendwelche Schwierigkeiten haben, die diesen Nutzen benutzen, bitte benutzen Sie die üblichen Taufbecken-Installation-Nutzen, die mit Windows kommen. Das Warnen: jener Windows kann eine begrenzt Anzahl von Taufbecken überhäufen, und Ihr Computer wird sich drastisch verlangsamen, wenn Sie die maximale Zahl von Taufbecken anschließen. Wir empfehlen, daß Sie bleiben, schloß Ihre Gesamtsumme Taufbecken zu weniger an der 500 für optimale Nützlichkeit.

Video Clips

Wir haben die Welt gesucht, um all Ihren Arbeitsfläche-Verlag Bedürfnisse zu treffen. MasterClips Browser ist ein mächtiger Nutzen, der Sie läßt, greifen Sie zu, sehen Sie an, schreiben Sie ab, und manipulieren Sie clipart-Bilder mit **Video-Klammern**.

Quelle: <http://einklich.net/rec/imsi.htm>

Syntax und Semantik von Programmiersprachen

Eine automatische Übersetzung kann nur gelingen, wenn **Quellsprache und Zielsprache hinreichend formal definiert** sind.

Die **Syntax** (griech. syntaxis = Zusammenordnung, Lehre vom Satzbau) einer Programmiersprache legt fest, welche Zeichenreihen **korrekt formulierte Programme der Sprache** sind und welche nicht.

Die **Semantik** (Semantik = Bedeutungslehre) einer Programmiersprache legt fest, welche **Bedeutung korrekt formulierte Programme** der Sprache haben.

Syntax und Semantik von Programmiersprachen sind durch **strikte, formale Regeln** festgelegt. Daher ist es sogar möglich, **Übersetzungsalgorithmen** aus diesen Regeln **systematisch abzuleiten**:

 **Automatischer Übersetzerbau**

Achtung:

Zielprogramme müssen nicht nur korrekt, sondern auch **schnell und speicher-effizient ausführbar** sein. Die **Optimierungsaufgaben** umfassen insbesondere:

- **geschickten Einsatz der Register,**
- **geschickte Nutzung von Zwischenergebnissen.**

Syntax and semantics of programming languages

An automatic translation can succeed only if source language and target language are sufficiently formally defined.

The **syntax** (griech. syntaxis = Zusammenordnung, theory of the satzbau) of a programming language specifies, which indication rows are **correctly formulated programs of the language** and which not.

The **semantics** (semantics = meaning teachings) of a programming language specifies, which meaning correctly formulated programs of the language have.

Syntax and semantics of programming languages are fixed by **strict, formal rules**. Therefore it is even possible to **derive algorithms** from these rules **systematically**:

 **Automatic building of translators**

Note:

Goal programs must be not only correctly, but also **fast and memory-efficiently executable**.

The **optimization tasks** cover in particular:

- **skillful employment of registers,**
- **skillful use of intermediate results.**

Übersetzung: <http://world.altavista.com/babelfish/tr>