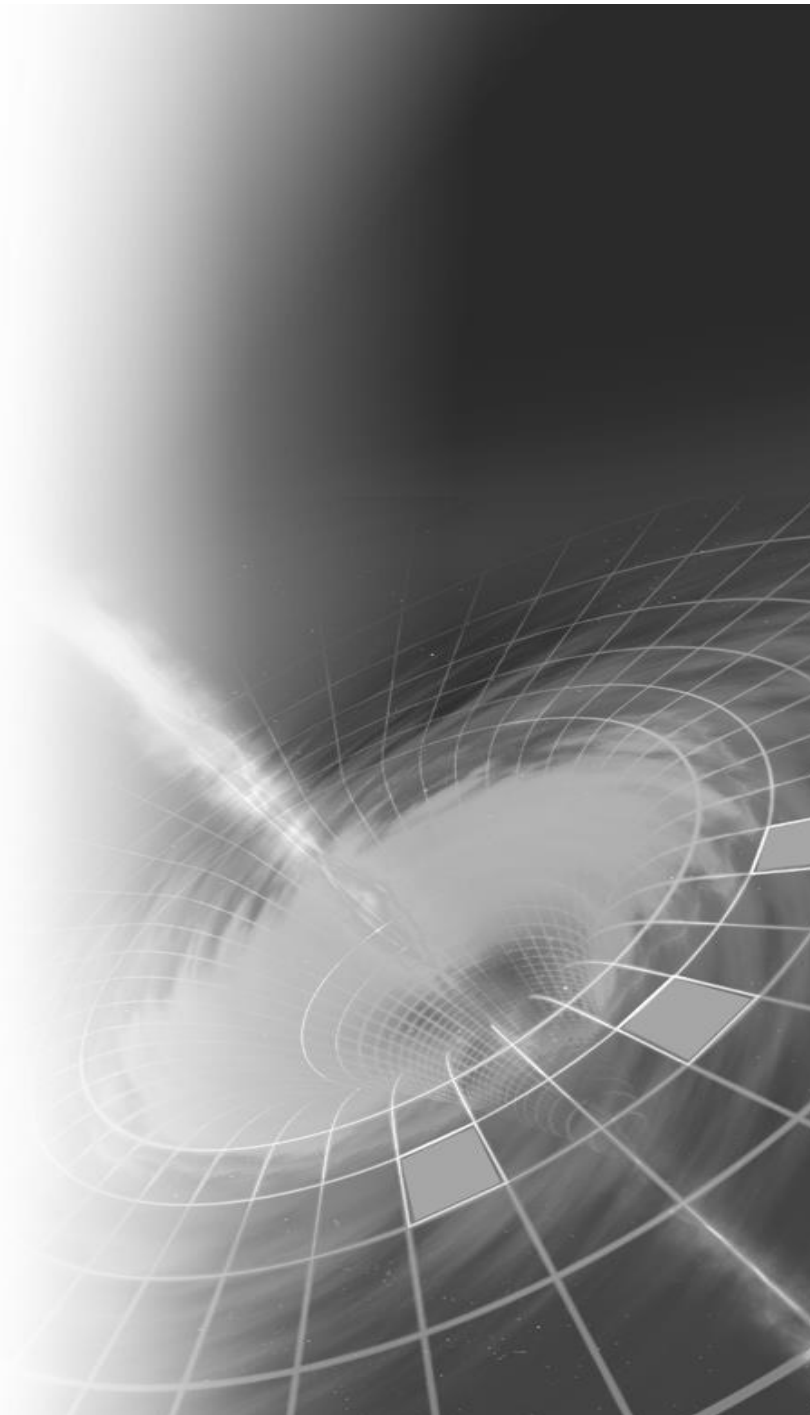
Bachelor-Programm

Compilerbau

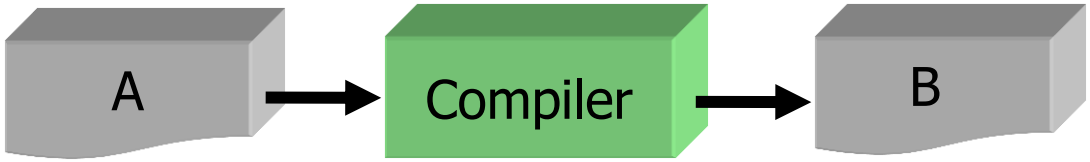
im SoSe 2014



Prof. Dr. Joachim Fischer
Dr. Klaus Ahrens
Dr. Andreas Kunert
Dipl.-Inf. Ingmar Eveslage

fischer@informatik.hu-berlin.de

Was ist ein Compiler ?



Ein **Compiler**

auch *Kompiler*;
englisch für *zusammentragen* bzw. lateinisch *compilare* ‚aufhäufen'

ist ...

ein **Computerprogramm**,

das ein (anderes) Programm,

das in einer bestimmten **Programmiersprache** geschrieben ist,

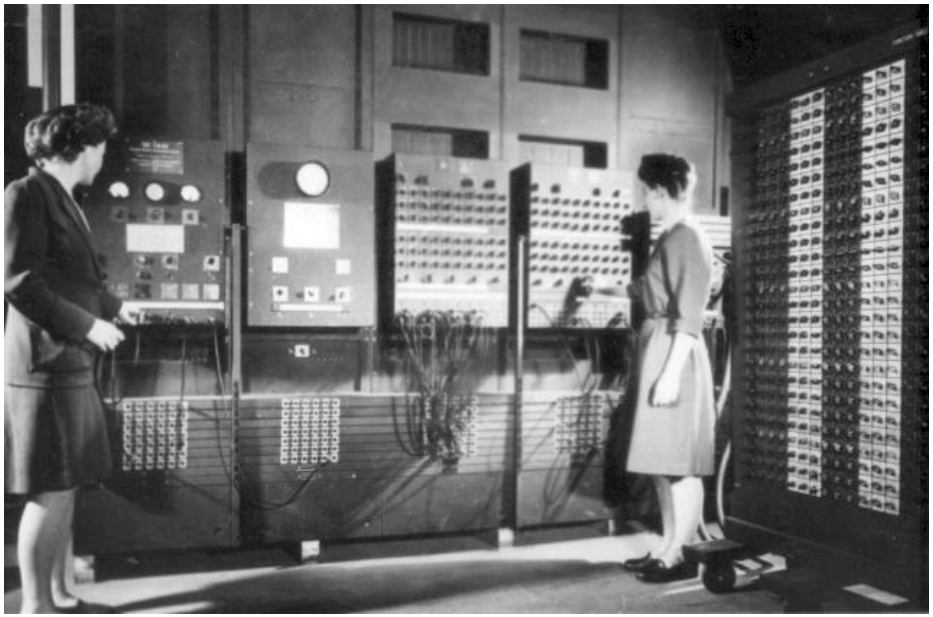
in eine bestimmte **Form** so überführt wird,

die von einem **Computer** ausgeführt werden kann.

nach welchen Prinzipien erfolgt die Ausführung ?

Spektrum von Rechnern / Prozessoren

Electronic Numerical Integrator and Computer (ENIAC) 1946



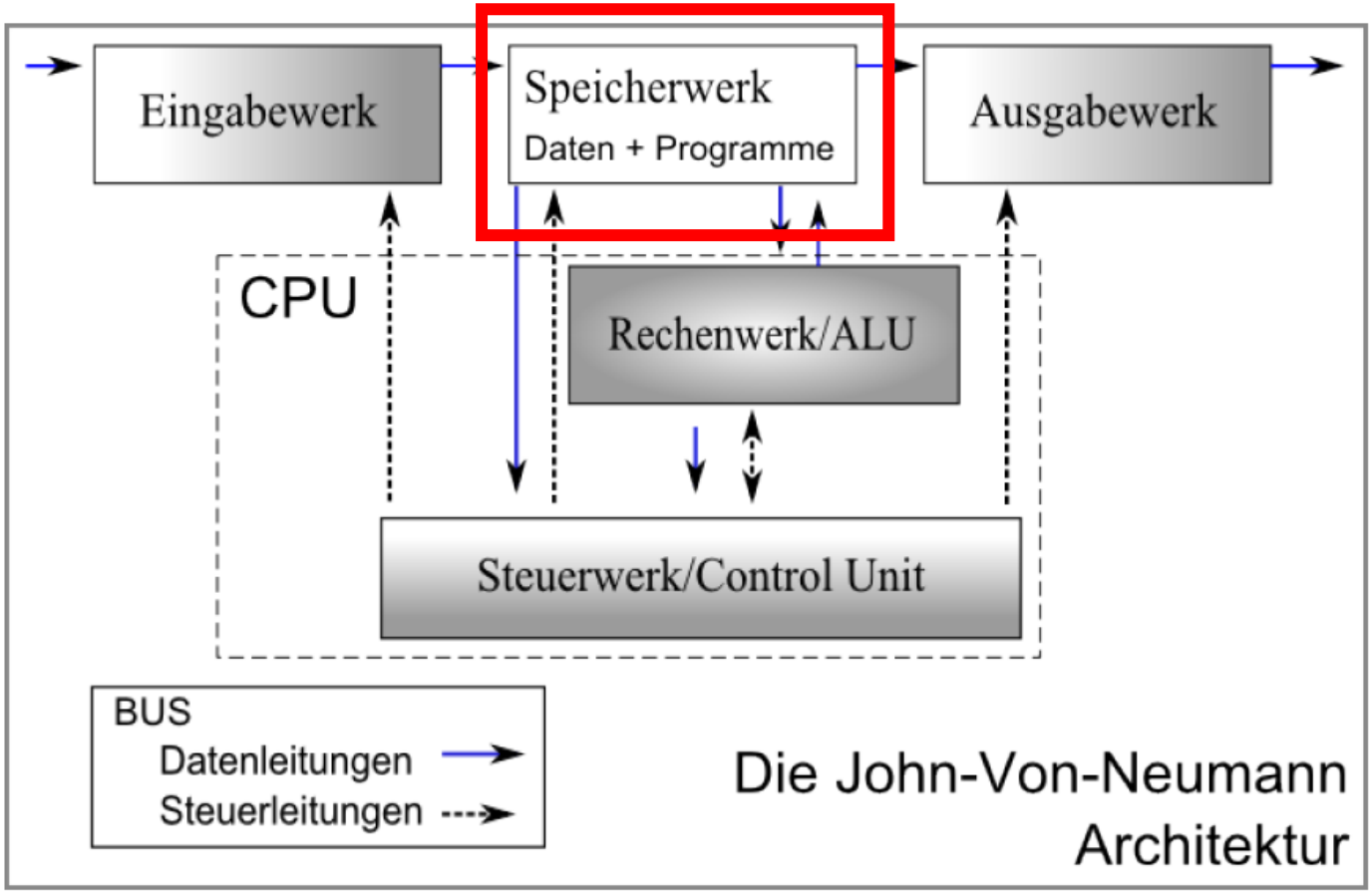
Berechnung ballistischer Tabellen

~ 32-Bit-Chipsatz 1983;
Advanced RISC Machines (ARM)

Mobile-Phone,
PDAs, Router
Tablet-Computersysteme und
Spielkonsolen

Stabile konzeptionelle Randbedingungen für den Compilerbau

(1) die Rechnerarchitektur als …



Stabile konzeptionelle Randbedingungen *für den Compilerbau*

(2) Prinzipien, die durch die Architektur bestimmt sind

- Struktur des Rechners ist unabhängig vom zu lösenden Problem.
  Programme müssen im Speicher abgelegt werden.
- Für Daten, Programme und Resultate wird derselbe Speicher genutzt.
  → Dadurch können alle Daten im Speicher sowohl als Programme als auch als Eingabedaten interpretiert werden.
- Der Speicher ist eindimensional in fortlaufenden Zellen adressiert.
  → Aufeinanderfolgende Programmbefehle folgen auch im Speicher aufeinander.
- Von der eigentlichen Programmfolge kann durch Sprungbefehle abgewichen werden.
- Daten werden grundsätzlich binär kodiert.
- Es existiert eine Reihe von Standardfunktionen für arithmetische, logische und Transport-Befehle sowie bedingte Sprünge

Stabile konzeptionelle Randbedingungen *für den Compilerbau*

(3) Prinzipien, bestimmt durch die Theorie der formalen Sprachen

- Algol 60 Report führte bereits die grammatikalische Definition der Syntax der Sprache als formale Definition unter Verwendung der Backus-Naur-Form als Notationstechnik ein,

  die bald allgemein üblich wurde

Compiler ← → Übersetzer (aber nicht allgemein benutzt)

Ein **Übersetzer** übersetzt

ein Programm aus einer formalen **Quellsprache**

in

ein semantisches Äquivalent in einer formalen **Zielsprache**

Compiler sind spezielle Übersetzer,
die Programmcode aus problemorientierten Programmiersprachen,
(sogenannten Hochsprachen)

in

- eine Assemblersprache,
- maschinell ausführbare Maschinensprache oder
- Bytecode

überführen.

Unsere Definition

Ein **Compiler** ist

ein **Computerprogramm**, das ein (anderes) Programm,

das in einer bestimmten **Programmiersprache** geschrieben ist,

in eine bestimmte **Form** so überführt wird,

die von einem **Computer** ausgeführt werden kann:

- eine Assemblersprache,
- maschinell ausführbare Maschinensprache oder
- Bytecode überführen.

Ein Compiler, der

den Quellcode einer Hochsprache in den Quellcode einer anderen Hochsprache übersetzt,

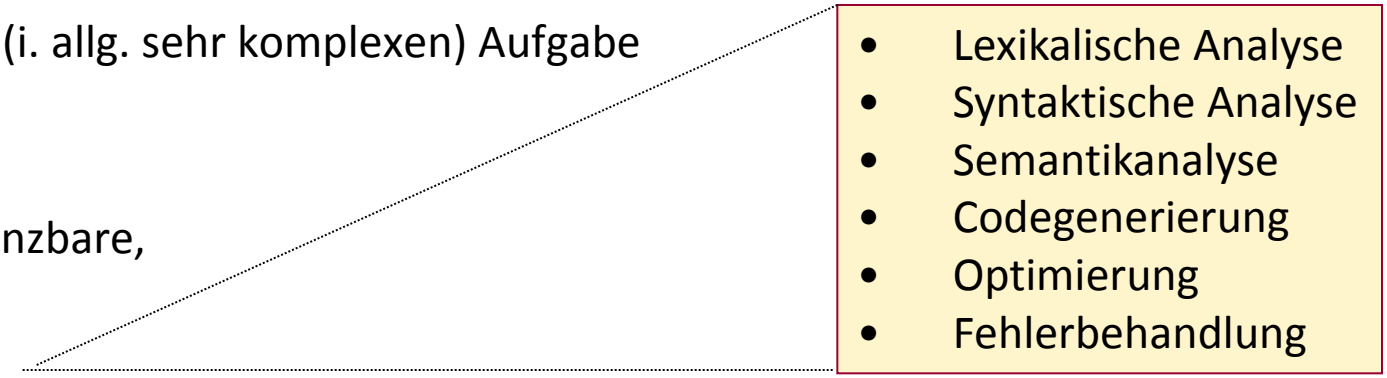
wird als **Transcompiler** bezeichnet (Beispiel: Pascal → C)

Compilationsprozess

- **Ziel**:
  Überführung eines Programms einer Sprache in ein Programm einer anderen Sprache

- zur Bewältigung dieser (i. allg. sehr komplexen) Aufgabe wird sie in
  - einzelne,
  - voneinander abgrenzbare,
  - leichtere

  Aufgaben gegliedert



- Aufgaben werden in einem wohl strukturierten Organisationsprozess (**Compilationsprozess**) behandelt

Struktur der LV

Dr. Ahrens
3 Vorlesungen
17.4., 24.4., 29.4.

- **Teil I**
  Die Programmiersprache C
- **Teil II**
  Methodische Grundlagen des Compilerbaus
- **Teil III (Praktikum)**
  Entwicklung von Komponenten eines Compilers

- zweite Programmiersprache in der BA-Ausbildung (neben Java)
  - universell, hardware-nah
  - keine OO-Sprache
  - Präprozessor, Pointer-Konzept
- Unix (1973), Sprache für eingebettete Systeme
- lange Compiler-Tradition (effiziente Codegeneratoren für verschiedenste Plattformen)

ESZ, Raum 0.310
9:15 Uhr

Struktur der LV

Prof. Dr. Fischer
Vorlesungen
Di, Do

- **Teil I**
  Die Programmiersprache C
- **Teil II**
  Methodische Grundlagen des Compilerbaus
- **Teil III (Praktikum)**
  Entwicklung von Komponenten eines Compilers

ESZ, Raum 0.310
9:15 Uhr

- Compilationsprozess
- Formalismen zur Sprachbeschreibung
- Lexikalische Analyse:
  - der Scanner
  - der Parser
- Parsergeneratoren: Yacc, Bison
- Statische Semantikanalyse
- Ausblick
  - Laufzeitsystem,
  - Codegenerierung

Struktur der LV

14-tägige Termine
8 Praktikumsgruppen, 3 Praktikumsleiter

- **Teil I**
  Die Programmiersprache C
- **Teil II**
  Methodische Grundlagen des Compilerbaus
- **Teil III (Praktikum)**
  Entwicklung von Komponenten eines Compilers

- einfache Quellsprache (an C angelehnt)
- Analysekomponenten in C
- Compilergenerierungswerkzeuge zur Erzeugung von Komponenten für die lexikalische, syntaktische und semantische Analyse

Dr. K. Ahrens
Dr. A. Kunert
Dipl.-Inf. I. Eveslage

Di 11-13 Uhr, ESZ Raum 0.310 (nach der Vorlesung)

Mi 13-15 Uhr, ESZ Raum 0.313

Do 11-13 Uhr, ESZ Raum 1.305 (nach der Vorlesung)

Fr 9-11 Uhr, ESZ Raum 1.303

Position

- **Kapitel 1**
  Compilationsprozess
- Kapitel 2
  Formalismen zur Sprachbeschreibung
- Kapitel 3
  Lexikalische Analyse: der Scanner
- Kapitel 4
  Syntaktische Analyse: der Parser
- Kapitel 5
  Parsergeneratoren: Yacc, Bison
- Kapitel 6
  Statische Semantikanalyse
- Kapitel 7
  Ausblick: Laufzeitsystem,
  Codegenerierung

- **Einführung**

  Compiler,

  Phasen,

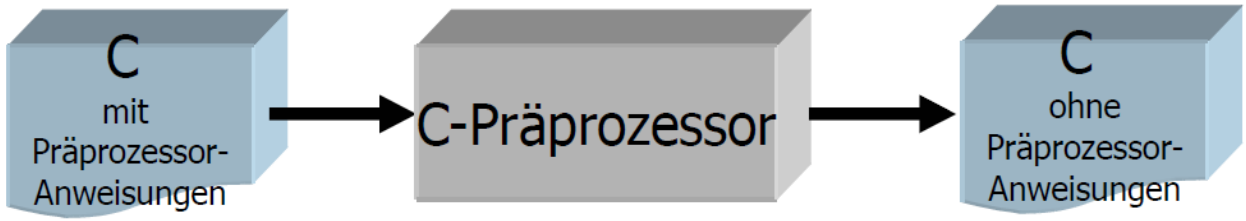
  Pässe

Compilationsprozess (Forts.)

- einem Compilationsprozess können
  weitere Verarbeitungsprozesse vor- und nachgeschaltet sein
  (die nicht zur Compilation gerechnet werden)

- typische Beispiele
  - **vorher**:
    Präprozessorverarbeitung (s. C)

  - **nacher**:
    Assemblierung, Programmverbindung, Laden des Programms



- Einfügen von Dateien

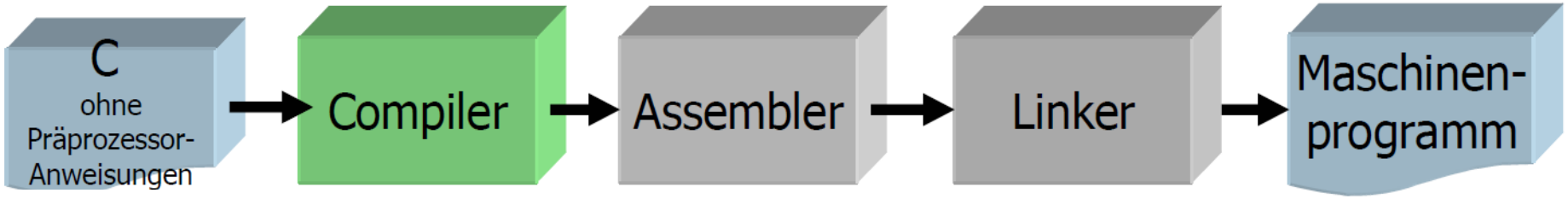
```
#include <stdio.h>      (Systemdatei)
#include "stack.h"      (nutzerdefiniert)
```

- Ersetzen von Text(Makros)

```
#define L 5                          (Konstantendefinition)
#define add100(x) ((x)+100)          (parametrisiert)
```

- bedingte Übersetzung

```
#ifdef TEST
        printf("Testversion");
#else
        printf("Produktionsversion");
#endif
```



Compiler-Phase, Compiler-Pass, Zwischencode

- **Compilerphase**
  eigenständige abgegrenzte Aufgabe des Compilationsprozesses

- **Compilerpass**
  in sich abgeschlossene separate oder auch z.T. kombinierte Realisierung einzelner Phasen durch einen Compiler

- **Zwischencode**
  temporäre Datenstrukturen (u.U. externe Dateien)
  über die die Compilerpässe untereinander kommunizieren

Vor- und Nachbereitung einer Compilation

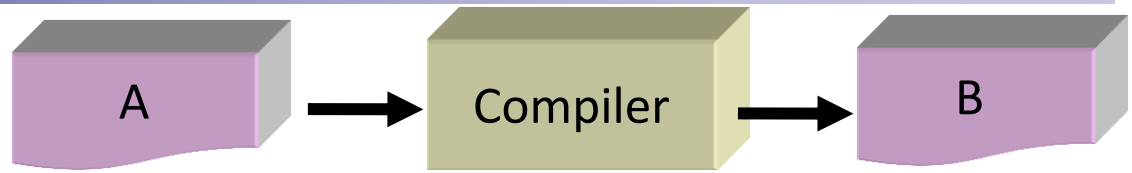
- einem Compilationsprozess können
  weitere Verarbeitungsprozesse vor- und nachgeschaltet sein
  (*die nicht zur eigentlichen Übersetzung gerechnet werden*)

- typische Beispiele
  - **vorher**:
    Präprozessorverarbeitung (s. Sprache C)

  - **nacher**:
    Assemblierung, Programmverbindung, Laden des Programms

Compilerpass

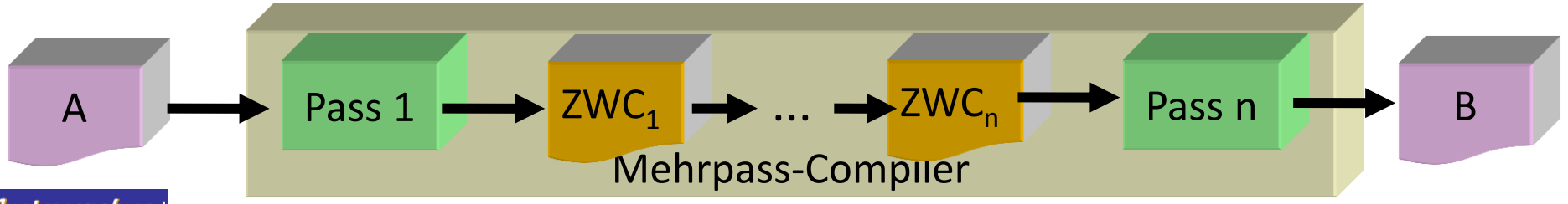
Compilerpässe
als Transformationsschritte eines Compilers



Unterscheidung zwischen

Einpasscompiler:
Programm wird nur einmal durchmustert

Mehrpasscompiler:
besteht aus verschiedenen
Übersetzungsprogrammen/Pässen
(ZWC = Zwischencode)



Kriterien für Passeinteilung

- **Ziel-Rechnerarchitektur** (Hauptspeicherausstattung):
  Einpass-Compiler benötigen mehr Hauptspeicher
- **Eigenschaften der Quellsprache**:
  einige Sprachen lassen sich nicht in einem Pass übersetzen
- **Effizienz des Compilers:**
  Einpass-Compiler sind schneller
- **Effizienz des erzeugten Programms:**
  Mehrpass-Compiler generieren i.d.R. effizienteren Code
- **Lesbarkeit/Wartbarkeit:**
  Mehrpass-Compiler günstiger
- **Portabilität:**
  Mehrpass-Compiler günstiger

benötigter Zw-Code kann heute meist im HS abgelegt werden

Pass-Strukturen ausgewählter Compiler

Pascal: **Pascal-360-Compiler**

Einpass-Compiler: Objektcode für IBM 360/370

Modula-2: **M-Compiler (Wirth)**

Vierpass-Compiler

1. lexikalische und syntaktische Analyse,
   semantische Analyse (Aufbau der Symboltabelle)
2. semantische Analyse (Deklaration)
3. semantische Analyse (Prozedurkörper)
4. M-Code-Generierung

Pass-Strukturen ausgewählter Compiler

C: **GNU C-Compiler**

Mehrpass-Compiler

1. C→ attributierter Syntaxbaum (Darstellung des Quelltextes)
   lexikalische, syntaktische, semantische Analyse

2. attributierter Syntaxbaum → virtueller Code (RTL-Code)
   Zwischencodegenerierung

Register-**T**ransfer-**L**ogic-Code
einheitlicher Zielcode in der GNU-Sprachfamilie

3. RTL-Code → RTL-Code-Optimierung

   ...

20. RTL-Code → RTL-Code-Optimierung

plattformunabhängig

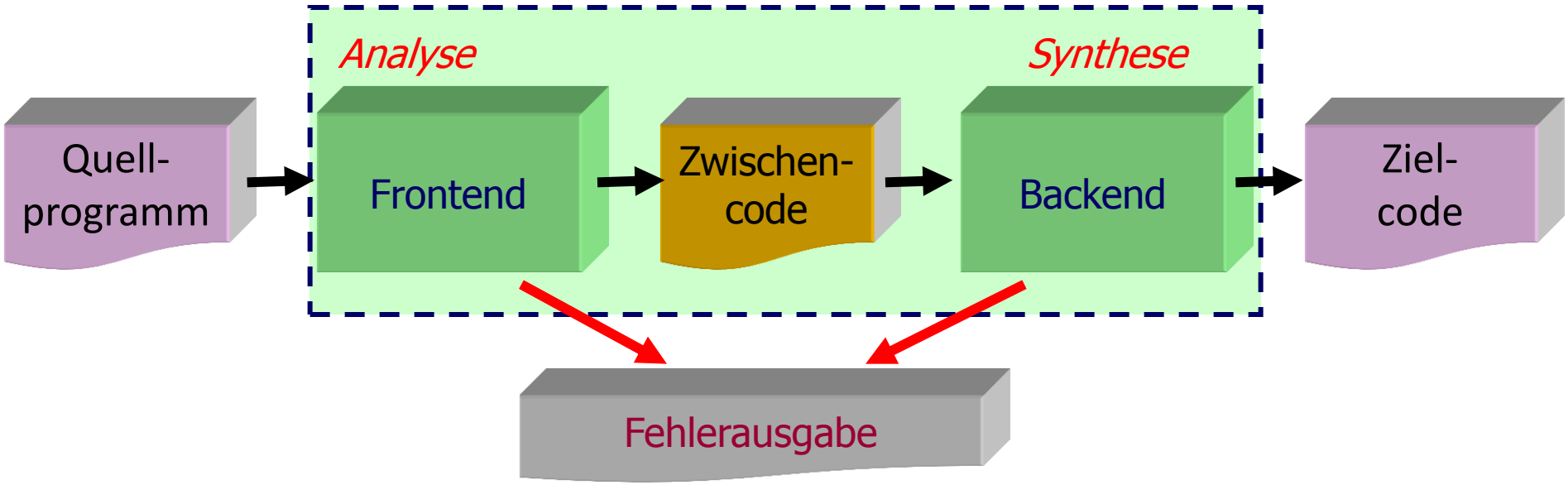
21. RTL-Code → Assemblercode (plattformabhängig)

danach: betriebssystemabhängige Assembler, Objektlinker → **Maschinencode**

Qualitäten eines Compilers

- Erzeugung von korrektem Code
- Effizienz des erzeugtes Programms
- Effizienz des Compilers
- Übersetzungszeit ~ Programmgröße (Unterstützung separater Übersetzungen)
- gute Fehlerdiagnostik
- gute Debugger-Integration
- Möglichkeit zum Aufruf von Programmen in anderen Sprachen
- konsistente, vorhersehbare Optimierung

Traditioneller Compiler-Ansatz



Lösen des sogenannten Wortproblems:
Erkennung eines Programm als gültiges Wort einer Sprache

→ Problem der Theoretischen Informatik

- Frontend erzeugt aus Quellprogramm korrekten Zwischencode
- Backend erzeugt vom Zwischencode korrekten Zielcode
- Zwischencode zur Unterstützung mehrerer Quellsprachen/Zielsprachen

Position

- Kapitel 1
  Compilationsprozess
- **Kapitel 2**
  Formalismen zur Sprachbeschreibung
- Kapitel 3
  Lexikalische Analyse: der Scanner
- Kapitel 4
  Syntaktische Analyse: der Parser
- Kapitel 5
  Parsergeneratoren: Yacc, Bison
- Kapitel 6
  Statische Semantikanalyse
- Kapitel 7
  Ausblick: Laufzeitsystem,
  Codegenerierung

- Einführung
  Compiler,
  Phasen,
  Pässe
- **Alphabete, Sprachen**
  Grammatiken,
  Automatenmodelle

Formalismen im Compiler-Frontend (Big Picture)

Reguläre Sprache (1b)

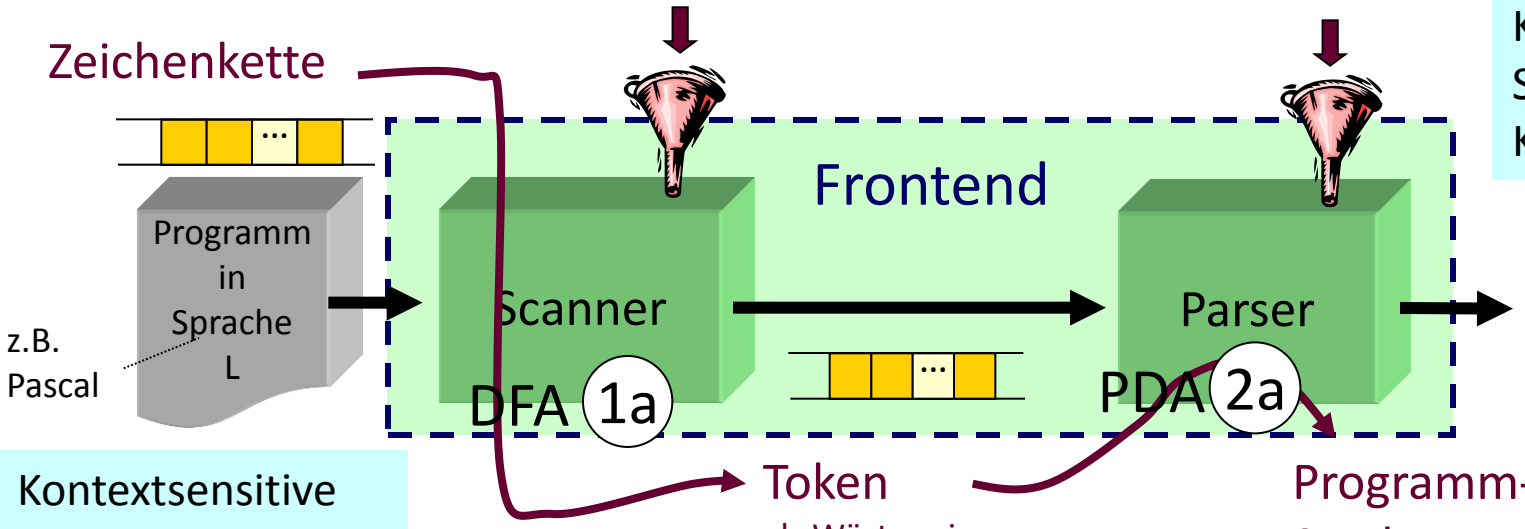
Sprache zur Festlegung erlaubter Pascal-Token

Bildungsvorschrift: *Reguläre Ausdrücke*

Kontextfreie Grammatik (2b)

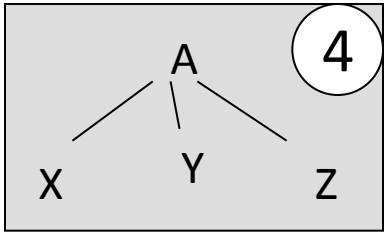
Bildungsvorschrift: *Syntaxregeln*

G= (V, S, P, S)



Kontextsensitive Sprache

3 Konsequenz: Sonderbehandlung von Kontextabhängigkeiten



Syntaxbaum als Zwischencode

Alphabet

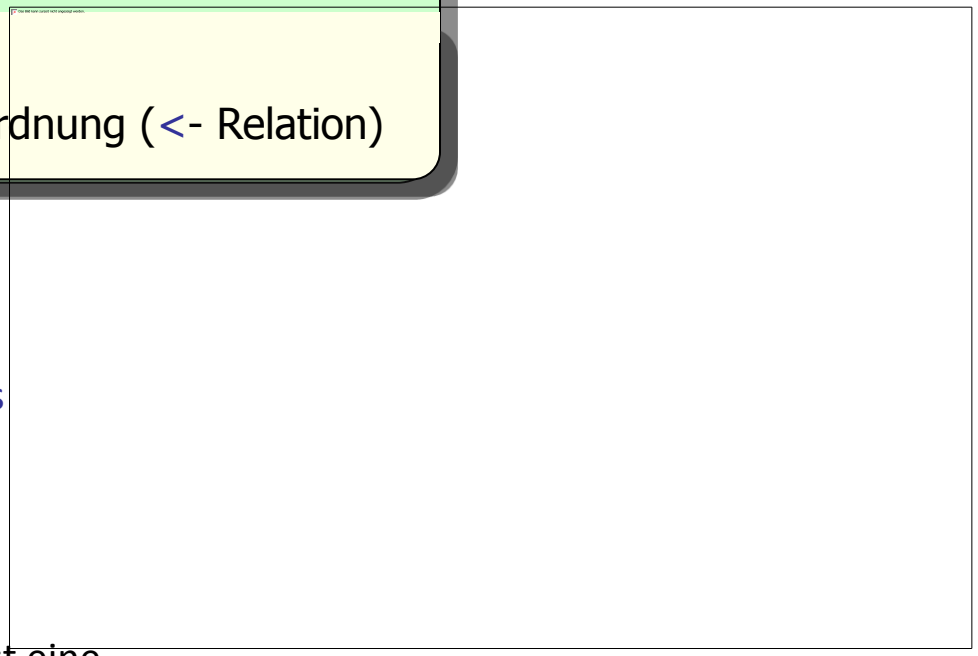
- zentrales Konzept für Sprachen, Zustandsautomaten und Grammatiken

Definition: Alphabet

endliche Menge Σ von Zeichen
mit einer darauf erklärten vollständigen Ordnung (<- Relation)

Beispiele:

Σ = {0, 1} binäres Alphabet,
Σ = ascii-Zeichensatz,
Σ = akzeptierte Token-Menge eines Scanners



- jedes (textuelle) **Computerprogramm** ist eine
  wohlgeordnete Aneinanderreihung von **Symbolen**
  aus einem Alphabet (als Menge zulässiger Token)
- jedes **Symbol** (Token) eines Computerprogramms ist eine
  wohlgeordnete Aneinanderreihung von Zeichen
  aus einem Alphabet (als Menge zulässiger Zeichen einer Sprache)

Zeichenkette (Wort)

sei Σ beliebiges Alphabet

Eine Zeichenkette (Wort) ist eine endliche Folge von beliebigen Zeichen a_i aus Σ

Beispiel:

$\Sigma = \{0, 1\}$ Zeichenkette: 01101

Weitere Begriffe

leere Zeichenkette ist eine Zeichenkette (Wort), das keine Zeichen enthält

- wird mit ε bezeichnet
- kann aus jedem Alphabet stammen

Länge einer Zeichenkette

- als Anzahl der Positionen

Beispiel

$|011| = 3, \quad |\varepsilon| = 0$

Potenzen eines Alphabets

Σ^k ist die Menge aller Zeichenketten der Länge k von Zeichen/Symbolen aus Σ

Zeichenkette (Wort): formal

Induktive Definition: Zeichenkette (ZK) über Σ

(1) die leere Zk ε ist eine ZK über Σ

(2) wenn α eine beliebige ZK über Σ ist,
dann ist auch $x\alpha$ eine ZK über Σ , falls $x \in \Sigma$)

➔ ß ist g.d. eine ZK über Σ,
wenn sie laut (1) oder (2) gebildet wurde

Bem.: eine ZK über Σ wird auch als **Wort** bezeichnet

Spezielle Wortmengen

leeres Wort	$\varepsilon: \emptyset \rightarrow \Sigma$	Wort, bestehend aus keinem Zeichen

Σ^0	$\{\alpha \mid \alpha: \emptyset \rightarrow \Sigma\} = \{\varepsilon\}$	Menge der Worte der Länge 0
Σ^n	$\{\alpha \mid \alpha: \{1,\ldots,n\} \rightarrow \Sigma\}$	Menge der Worte der Länge n
Σ^*	$\bigcup_{n\geq 0} \Sigma^n$	Menge aller Worte über Σ
Σ^+	$\bigcup_{n\geq 1} \Sigma^n$	Menge aller nichtleeren Worte über Σ

Beispiel:

$\Sigma = \{0, 1\}$, dann $\Sigma^2 = \{00, 01, 10, 11\}$

$\Sigma^* = \{0, 1\}^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$

Sprachen *(erste, noch unbefriedigende Definition)*

Formale Sprachen über ein Alphabet Σ sind

Teilmengen von Σ^*

$L \subseteq \Sigma^*$ (L ist eine Sprache über das Alphabet Σ)

Eine Sprache ist die (möglicherweise unendliche) Menge zulässiger Wörter.

Beispiel: {w | w ist ein syntaktisch fehlerfreies C-Programm}

Aufgabe: Zulässigkeit formal fassen

Bem: Es ist nicht verlangt, dass dabei jedes Zeichen des Alphabets benutzt wird:

sei $L \subseteq \Sigma^*$ eine Sprache, dann ist auch $L \subseteq \Sigma 1^*$ mit $\Sigma 1 \subseteq \Sigma$ eine Sprache

Operationen auf Sprachen

- Seien L und M beliebige Sprachen

Operation	Definition
Vereinigung von L und M: $L \cup M$	
Verkettung von L und M: $L \circ M$	
Hülle von L: L^*	
positive Hülle von L: L^+	

Operationen auf Sprachen

- Seien L und M beliebige Sprachen

Operation	Definition
Vereinigung von L und M: $L \cup M$	$L \cup M = \{s \mid s \in L \text{ oder } s \in M\}$
Verkettung von L und M: $L \circ M$	$L \circ M = \{xy: x \in L \text{ und } y \in M\}$
Hülle von L: L^*	$L^* = \bigcup_{i=0}^{\infty} L^i$ /* i als Wortlänge */
positive Hülle von L: L^+	$L^+ = \bigcup_{i=1}^{\infty} L^i$

$L^0 = \{\varepsilon\}$, Wort der Länge 0

Wort-Problem *als praktisches Problem des Compilerbaus*

Frage:

Gehört ein gegebenes Wort zu einer bestimmten Sprache?

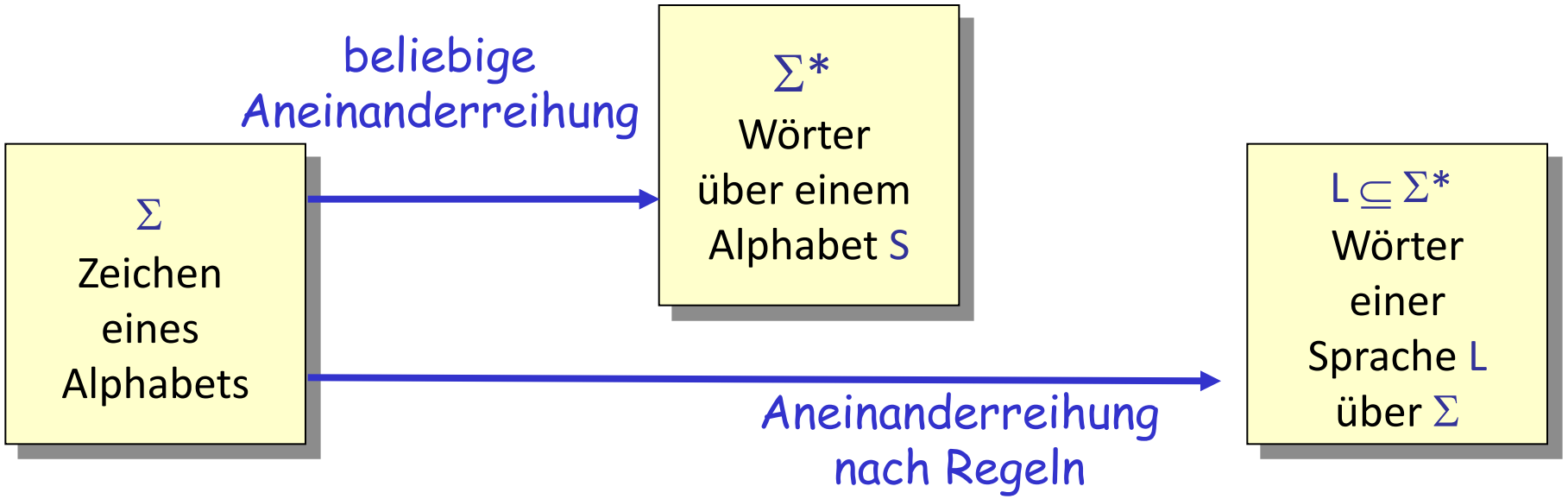
Definition: **Allgemeines Wortproblem**

Geg. sei eine Zeichenkette w in Σ^*.
ES ist zu entscheiden, ob w in L ($L \subseteq \Sigma^*$) enthalten ist oder nicht.

ein Problem der Theoretischen Informatik

… aber auch zentrales Problem des Compilerbaus:
Ist ein gegebenes Programm (als Zeichenkette w)
ein zulässiges Programm der Sprache L ?

Zusammenhang: Alphabet, Wort, Sprache



Regeln legen fest,

- welche Teilzeichenketten zulässige Token der Sprache (Reguläre Ausdrücke) sind
- welche Worte aus Σ^* zu L gehören (Syntaxregeln)
- welche Bedeutung die Wörter haben (Semantikregeln)

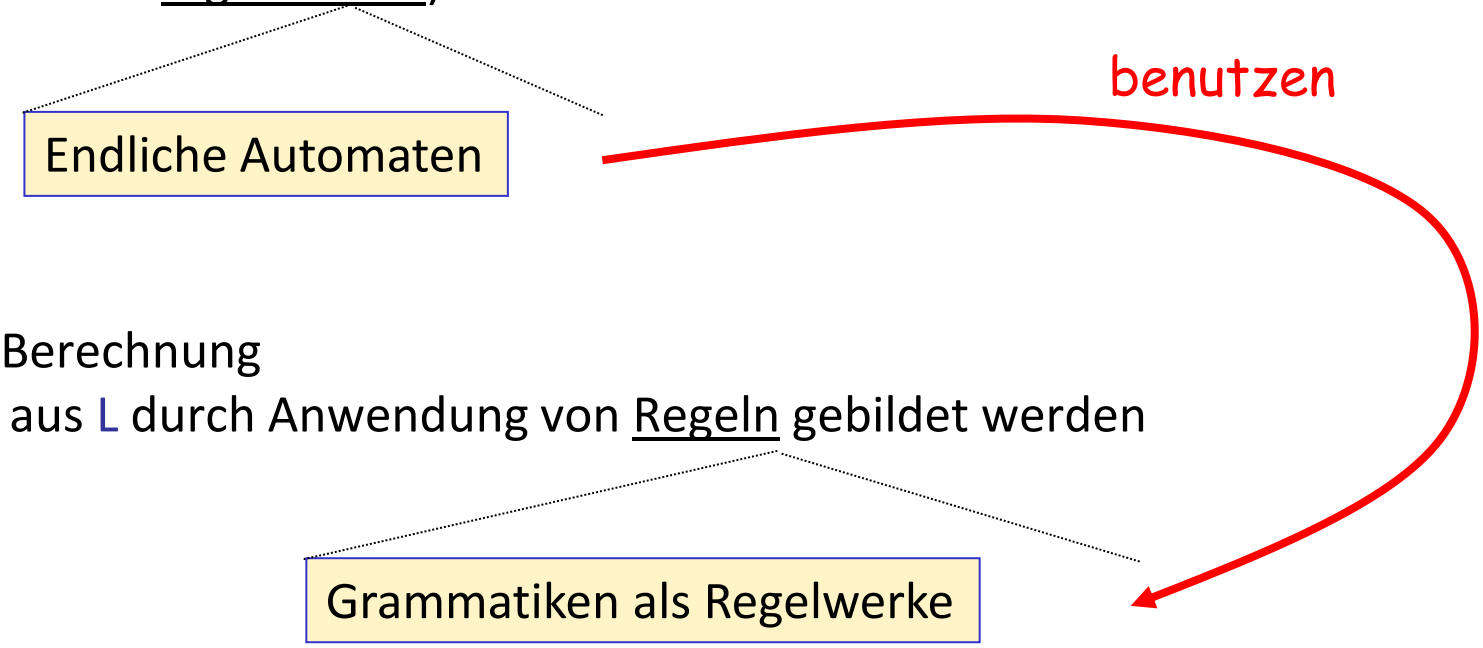
Lexikalische Analyse

Syntaxanalyse
Semantikanalyse

Prinzipien zur Definition einer Sprache

1. **Aufzählung** aller gültigen Wörter
   (nur, wenn es nicht zu viele sind)

2. **Erkennung**
   sei $\alpha \in \Sigma^*$, so ist zu prüfen, ob $\alpha \in L$
   (Prüfung durch einen Algorithmus)



3. **Generierung**/ Berechnung
   indem Wörter aus L durch Anwendung von Regeln gebildet werden

Position

- **Kapitel 1**
  Compilationsprozess
- Kapitel 2
  Formalismen zur Sprachbeschreibung
- Kapitel 3
  Lexikalische Analyse: der Scanner
- Kapitel 4
  Syntaktische Analyse: der Parser
- Kapitel 5
  Parsergeneratoren: Yacc, Bison
- Kapitel 6
  Statische Semantikanalyse
- Kapitel 7
  Ausblick: Laufzeitsystem,
  Codegenerierung

- Einführung
  - Compiler,
  - Phasen,
  - Pässe
- **Formalismen zur Sprachbeschreibung**
  - Alphabete, Sprachen
  - **Grammatiken**
  - Automatenmodelle

Formalismus: Grammatik

- **Grammatik**:
  übliche Methode zur formalen Beschreibung/ Definition einer Computersprache
  - eingeführt von: N. Chomsky (MIT)
    hoffte, natürliche Sprachen beschreiben zu können
  - erstmal eingesetzt:
    J.W. Backus beim ersten FORTRAN-Compiler

- **Backus-Naur-Form (BNF):**
  einfache Notation zur Beschreibung von Grammatikregeln

BNF

Notationsform von Produktionsregeln:

▪ Ableitungszeichen	linke Seite ::= rechte Seite oder linke Seite → rechte Seite
▪ Nichtterminalsymbole (Variablen)	< ...> oder Kursiv-Schreibweise
▪ Terminalsymbole	Symbole ohne <> -Klammerung oder '...'
▪ Alternativen-Trennung	|

Grundidee von Grammatiken am Beispiel

- Beispiel-Regel (informal):
  jeder Bezeichner (Identifier) besteht aus
  - einem Buchstaben,
  - gefolgt von beliebig vielen Buchstaben oder Ziffern

- Formalisierungsvarianten

Kurzschreibweise für drei Regeln

Charakterisierung durch Sprachoperatoren

$$L_{idf} = L_{letter} \circ \left(L_{letter} \cup L_{digit}\right)^*$$

$$L_{letter} = \{'A','B',...,'Z','a','b',...,'z'\}$$

$$L_{digit} = \{'0','1',...,'9'\}$$

Charakterisierung durch Grammatik-Regeln

$$Idf \rightarrow letter\ X$$

$$X \rightarrow \varepsilon \mid letter\ X \mid digit\ X$$

$$letter \rightarrow 'A'|'B'|...|'Z'|'a'|'b'|...|'z'$$

$$digit \rightarrow '0'|'1'|...|'9'$$

Idf als Variable für jeden beliebigen Identifier
X Variable für beliebige ZK des Alphabets

Ableitung eines Wortes *entsprechend der Grammatik*

- Als Ableitung wird
  der Vorgang bezeichnet,
  ein Wort nach den Regeln einer
  formalen Grammatik zu erzeugen.

$$\alpha\ Idf\ \beta \Rightarrow \alpha\ letter\ X\ \beta$$

Charakterisierung durch Grammatik-Regeln

$$Idf \rightarrow letter\ X$$

$$X \rightarrow \varepsilon \mid letter\ X \mid digit\ X$$

$$letter \rightarrow 'A' \mid 'B' \mid ... \mid 'Z' \mid 'a' \mid 'b' \mid ... \mid 'z'$$

$$digit \rightarrow '0' \mid '1' \mid ... \mid '9'$$

- Anwendung einer Regel bedeutet,
  dass in dem bislang erzeugten Wort ein **Teilwort**, das einer linken Regelseite entspricht, durch die rechte Seite der Regel ersetzt wird
  (auch als Ableitung bezeichnet)

- Ableitungsschritte werden ausgehend von einem **Startsymbol** solange ausgeführt bis das entstandene Wort nur noch aus Terminal-Symbolen besteht
  - ACHTUNG: Reihenfolge der Schritte ist i.allg. nicht eindeutig

- jedes so erzeugte Wort gehört dann zu der von der Grammatik erzeugten
  oder definierten Sprache

Ableitung eines Wortes *am Beispiel*

unter den Variablen einer Grammatik
muss es ein festgelegtes Startsymbol
geben.
sei hier: *Idf*

angewendete Regel

Ableitung eines Wortes als Bezeichner

$$
\begin{aligned}
Idf &\Rightarrow letter\ X \quad (1)\\
&\Rightarrow letter\ digit X \quad (4)\\
&\Rightarrow letter\ digit digit X \quad (4)\\
&\Rightarrow letter\ digit digit \quad (2)\\
&\Rightarrow B\ digit digit \quad (6)\\
&\Rightarrow B\ 3\ digit \quad (\ldots)\\
&\Rightarrow B\ 31 \quad (\ldots)
\end{aligned}
$$

B31 ist ein gültiger Bezeichner
als Wort der Sprache, die durch die Grammatik definiert ist

Charakterisierung durch Grammatik-Regeln

$$
\begin{aligned}
(1)\quad & Idf \rightarrow letter\ X\\
(2)\quad & X \rightarrow \varepsilon\\
(3)\quad & X \rightarrow letter\ X\\
(4)\quad & X \rightarrow digit\ X\\
(5)\quad & letter \rightarrow \text{'A'}\\
(6)\quad & letter \rightarrow \text{'B'}\\
& \ldots\\
(\ldots)\quad & letter \rightarrow \text{'z'}\\
(\ldots)\quad & digit \rightarrow \text{'0'}\\
& \ldots\\
(\ldots)\quad & digit \rightarrow \text{'9'}
\end{aligned}
$$

Allgemeine Grammatikdefinition

Definition: Grammatik (Typ 0: allgemein, ohne zusätzliche Einschränkungen)

... ist ein 4-Tupel $G = (V, \Sigma, P, S)$ mit

- V
  ist die (endliche) Menge der Variablen (Nichtterminalsymbole/ Metasymbole) der Grammatik
  sie wird benutzt, um die Grammatik zu strukturieren
- Σ
  ist die (endliche) Menge der Terminalsymbole mit $\Sigma \cap V = \emptyset$
  in unserem Falle ist sie identisch mit der gültigen Token-Menge, die der Scanner liefert
- $P \subseteq (V \cup \Sigma)^+ \times (V \cup \Sigma)^*$
  ist die (endliche) Menge der Produktionsregeln,
  sie legt fest, wie Terminal- und Nicht-Terminalsymbole kombiniert werden können, um ein Wort der Sprache zu bilden
- $S \in V$ ist als ausgezeichnetes Nicht-Terminalsymbol das Startsymbol

Produktionsregel und Ableitung

$P \subseteq (V \cup \Sigma)^+ \times (V \cup \Sigma)^*$

Produktionsregel: statt (α, β) häufig intuitive Notation $\alpha \rightarrow \beta$

$Idf \rightarrow letter\ X$

Definition: Ableitung (als Relation $\Rightarrow$)

$$x, y, \alpha, \beta, \gamma \in (V \cup \Sigma)^*,\ A \in V$$

$$x \Rightarrow y :\Leftrightarrow \exists \alpha, \beta, \gamma, A : x = \alpha A \beta \wedge y = \alpha \gamma \beta \wedge (A, \gamma) \in P$$

beschreibt Ableitung einer ZK aus einer anderen:

eine Variable wird ersetzt durch rechte Seite der Regel, die für diese Variable festgelegt ist

Grammatik-basierte Sprachen

Definition: reflexiv-transitive Hülle $\Rightarrow^*$

$$x, y, w_i \in (V \cup \Sigma)^*$$

$$x \Rightarrow^* y :\Leftrightarrow x = y \vee (x \Rightarrow y) \vee (\exists w_1, \ldots, w_n : x \Rightarrow w_1 \Rightarrow \ldots \Rightarrow w_n \Rightarrow y)$$

Eine ZK y ist genau dann mittels der Hülle aus einer ZK x ableitbar, wenn
- die beiden entweder identisch sind oder aber sich y aus
- einem oder
- mehreren Zwischenschritten aus x

ableiten lässt.

Definition: Grammatik-basierte Sprache

$$L(G) := \{ x \in \Sigma^* \mid S \Rightarrow^* x \}$$

eine Grammatik G generiert eine Sprache L(G)

Menge aller Worte x, die sich aus dem Startsymbol S ableiten lassen

Erzeugung einer Sprache

Menge aller ZK,
die mit einem Buchstaben beginnen und dann mit einem beliebig langen (evtl. leeren) Wort
- gebildet aus Buchstaben und Ziffern -
enden

Sprache: $L(G_{Idf})$

$$L(G) := \{x \in \Sigma^* \mid S \Rightarrow^* x\}$$

Grammatik: $G_{Idf} = (V, \Sigma, P, S)$

$$V = \{Idf, X\}$$
$$\Sigma = \{A, B, ..., Z, a, b, ... z, 0, 1, ..., 9\}$$
$$S = Idf$$
$$P \subseteq (V \cup \Sigma)^+ \times (V \cup \Sigma)^*$$

$$Idf \rightarrow letter\ X$$
$$X \rightarrow \varepsilon \mid letter\ X \mid digit\ X$$
$$letter \rightarrow 'A' \mid 'B' \mid ... \mid 'Z' \mid 'a' \mid 'b' \mid ... \mid 'z'$$
$$digit \rightarrow '0' \mid '1' \mid ... \mid '9'$$

Alle Regeln von G_{Idf} sind sehr einfach

Grammatiken werden nach Regeltyp klassifiziert: Typ-0, Typ-1, Typ-2, Typ-3

... und Sprachen nach ihren erzeugenden Grammatiken

Zusammenhang: Formale Sprache und Grammatik

- eine (allgemeine) Grammatik G bestimmt eine Sprache L(G) mit

  $L(G) = \{w \in \Sigma^*, \; S \Rightarrow^* w\}$

- $w \in L(G)$ wird als Wort der Sprache L(G) bezeichnet

Grammatiktypen
- Typ-0: ohne Einschränkung
- Typ-1: kontextsensitiv
- Typ-2: kontextfrei
- Typ-3: regulär

Bem.: Sprachen, die für den Compilerbau spannend sein wollen, müssen eine **effiziente** Wort-Ableitung erlauben

→ praktische Bedeutung der Klassifikation von Grammatiken, hinsichtlich dieser Eigenschaft:
(1) reguläre Grammatiken
(2) kontextfreie Grammatiken

G_{Idf} ist eine reguläre Grammatik

$L(G_{Idf})$ ist eine reguläre Sprache

Äquivalenz von Grammatiken

Grammatik: $G_1 = (V_1, \Sigma, P_1, S_1)$

$$V_1 = \{Idf, X\}$$
$$\Sigma = \{A, B, ..., Z, a, b, ... z, 0, 1, ..., 9\}$$
$$S_1 = Idf$$
$$P_1 \subseteq (V_1 \cup \Sigma)^+ \times (V_1 \cup \Sigma)^*$$

$$Idf \rightarrow letter\ X$$
$$X \rightarrow \varepsilon \mid letter\ X \mid digit\ X$$
$$letter \rightarrow 'A' \mid 'B' \mid ... \mid 'Z' \mid 'a' \mid 'b' \mid ... \mid 'z'$$
$$digit \rightarrow '0' \mid '1' \mid ... \mid '9'$$

Grammatik: $G_2 = (V_2, \Sigma, P_2, S_2)$

$$V_2 = \{Idf', X\}$$
$$\Sigma = \{A, B, ..., Z, a, b, ... z, 0, 1, ..., 9\}$$
$$S_2 = Idf'$$
$$P_2 \subseteq (V_2 \cup \Sigma)^+ \times (V_2 \cup \Sigma)^*$$

$$Idf' \rightarrow letter\ X \mid letter$$
$$X \rightarrow letter\ X \mid digit\ X \mid letter \mid digit$$
$$letter \rightarrow 'A' \mid 'B' \mid ... \mid 'Z' \mid 'a' \mid 'b' \mid ... \mid 'z'$$
$$digit \rightarrow '0' \mid '1' \mid ... \mid '9'$$

Definition: Äquivalenz von Grammatiken

Seien $G_1 = (V_1, \Sigma, P_1, S_1)$, $G_2 = (V_2, \Sigma, P_2, S_2)$ zwei Grammatiken mit dem selben Alphabet Σ.

G_1 und G_2 heißen äquivalent, falls $L(G_1) = L(G_2)$.

Klasseneinteilung von Grammatiken

Definition: Grammatiktypen der Chomsky-Hierarchie

Sei $G = (V, \Sigma, P, S)$ eine Grammatik. G heißt

kontextsensitiv oder vom Typ 1, wenn für alle Regeln $u \to v$ gilt: $|u| \leq |v|$,

kontextfrei oder vom Typ 2, wenn $P \subseteq V \times (V \cup \Sigma)^*$,

regulär oder vom Typ 3, wenn $P \subseteq V \times (\{\varepsilon\} \cup \Sigma \cup \Sigma V)$.

$A \to c$

$A \to \varepsilon$

$A \to \alpha B c \beta$

$A \to \varepsilon$

$A \to a$

$A \to aB$

Grammatik ohne Einschränkung nennt man Grammatik vom Typ-0