Compiler

Die unsichtbaren Diener – Der Compiler und seine Artgenossen

Wozu Compiler?

- Eine der ältesten und grundlegendsten Technologien
- Compilertechnologie hat viele Anwendungsgebiete
  - Übersetzung von Programmiersprachen
  - Syntaxgetriebene Editoren, Kommandointerpreter
  - Halbformale Datenbeschreibungen (z.B. XML)
- Wenn wir Compiler nicht verstehen, verstehen wir kaum, wie Programme am Rechner funktionieren
  - Was geschieht aufgrund folgender Programmzeilen?

```
for (i = 0; i < 10; i++) { a[i] = a[i+1]*x.y.z; print(a[i]); }
o.addSalary(float sum);
if (o instance of Student) student = o else person = o;
```

  - Wie werden solche Anweisungen ausgeführt?

Was ist ein Compiler?

- Transformiert Texte einer „höheren“ formalen Sprache auf eine “niedrigere” formale Sprache

- Der Quellcode ist üblicherweise lesbar für Menschen; der Zielcode für Maschinen
  - C → Intel Assembler; Pascal → Motorola Maschinencode
- Ausführbarer Zielcode

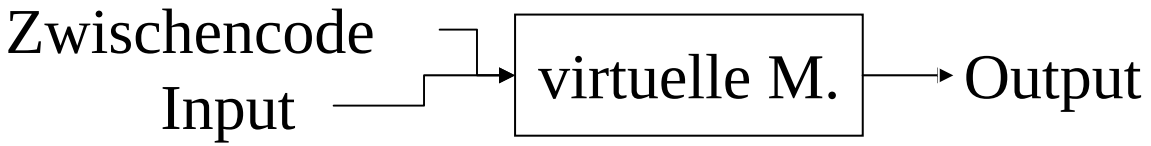
Interpretation, Front-end, Back-end

- Der Quellcode kann direkt *interpretiert* werden
  - Kommandointerpreter, Syntaxgetriebene Editoren, Scriptsprachen

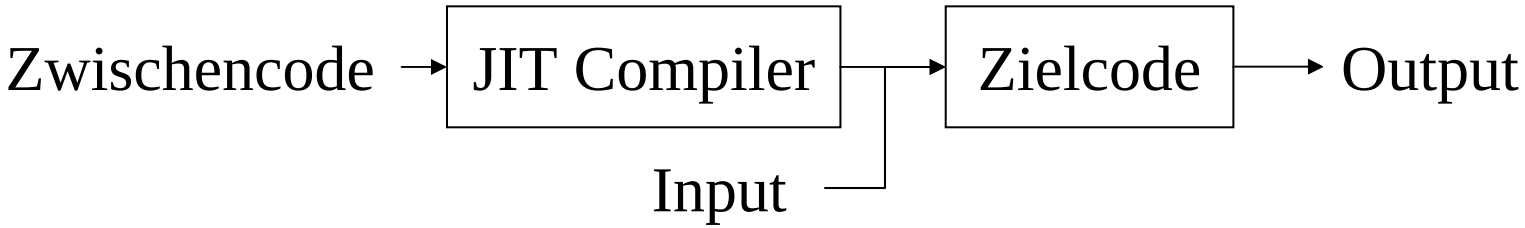
- Sprachabhängiges *front-end* → Zwischensprache
  - Z.B. Common Language Runtime (CLR in .NET)
- Zielabhängiges *back-end*

Interpretierte Ausführung, JIT Compilation

- Der Zwischencode kann von einer virtuellen Maschine (z.B. Java VM) interpretiert werden
  - Z.B. Java Byte Code – JVM läuft etwa im Browser



- Zwischencode wird “just-in-time” (JIT) übersetzt
  - Zur Laufzeit (z.B. Java); beim Laden (z.B. im .NET)



Formale Sprachen

- Eine formale Sprache wird definiert durch
  - Ein endliches Alphabet, z.B. {0, 1} oder {a, … z, A, … Z}
  - Eine Grammatik als eine Menge von Regeln (Syntax)
- Ein Wort (oder Satz) über das Alphabet Σ
  - $w_j = x_{j1}\ x_{j2} \ldots x_{jm}$, wo $\Sigma = \{x_i \mid 1 \le i \le n\}$
  - $|w_j| = m$ ist die Länge von w_j
  - Σ^+ : Menge der Wörter über Σ – transitive Hülle
  - Σ^* : $\Sigma^+ + \{\varepsilon\}$ – transitive und reflexive Hülle ($|\varepsilon|=0$)
  - Z.B. $\Sigma = \{0, 1\}$; $\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 11, 000, 001, \ldots\}$
- KEINE menschlichen Sprachen
  - Ziel ist Eindeutigkeit – schlecht für Humor oder Dichtung
  - Reife Theorie, relevant für viele Gebiete der Informatik

Grammatiken

- Grammatik *G* ist definiert als: $G = (V_N, V_T, P, S)$
  - V_N: Menge der *Variablen* (*non-terminal* Symbole)
  - V_T : Menge der *Terminale* (sind Teil der Sprache)
  - P: Menge der *Produktionen* (Regel der Satzbildung, der Syntax)
  - S: Start Symbol ($S \in V_N$)
  - $V_N \cap V_T = \emptyset$ (die zwei Mengen sind disjunkt)
  - $V_N \cup V_T = V$ (die Vereinigung beider Mengen ist V)
- P enthält Ausdrücke der folgenden Form
  - $\alpha \rightarrow \beta$ ($\alpha \in V^+$ and $\beta \in V^*$)
  - Wenn $\alpha \rightarrow \beta \in P$ und $\gamma, \delta \in V^*$:
    $\gamma\ \alpha\ \delta \Rightarrow \gamma\ \beta\ \delta$: $\gamma\ \beta\ \delta$ ist eine *direkte Ableitung* von $\gamma\ \alpha\ \delta$

Sprache, erzeugt durch eine Grammatik

- Ableitung
  - Wenn $\alpha_1 \Rightarrow \alpha_2$, $\alpha_2 \Rightarrow \alpha_3$, … $\alpha_{n-1} \Rightarrow \alpha_n$ dann
  - $\alpha_1 \Rightarrow^* \alpha_n$ d.h. α_n ist eine *Ableitung* von α_1
  - $\Rightarrow^*$ ist die reflexive und transitive Hülle von $\Rightarrow$
    - Ableitung in 0 oder mehr Schritten
  - $\Rightarrow^n$ Ableitung in n Schritten
- Die Sprache L, generiert durch Grammatik G ist
  - $\{w \mid w \in V_T^* \wedge S \Rightarrow^* w\}$
  - Die Menge aller Wörter, die aus Terminalen bestehen und aus dem Startsymbol abgeleitet werden können
  - Z.B.: G: $V_N = \{S\}$, $V_T = \{0, 1\}$, $P = \{S \rightarrow 0S1, S \rightarrow 01\}$
  - $L(G) = \{0^n1^n \mid n \geq 1\}$

Grammatik Typen (Noah Chomsky)

- Typ-3 (regulär)
  - Wenn $A \in V_N$, $B \in \{V_N, \varepsilon\}$, $w \in V_T^*$; Form der Produktionen
    $A \rightarrow w\,B$ or $A \rightarrow B\,w$ (rechtslinear bzw. linkslinear)
- Typ-2 (kontextfrei)
  - $\forall\ A \rightarrow \beta \in P \quad A \in V_N$ (Einzelvariable) und $\beta \# \varepsilon$ ($\varepsilon \notin L(G)$)
  - A kann in jedem Kontext durch β ersetzt werden
- Typ-1 (kontextsensitiv)
  - A kann nur in einem bestimmten Kontext
    (wie zwischen α_1 und α_2) durch β ersetzt werden
    $\alpha_1 A\,\alpha_2 \rightarrow \alpha_1 \beta \alpha_2$ $\quad (\alpha_1, \alpha_2, \beta \in V^*;\ \beta \# \varepsilon$ und $A \in V_N)$
  - $\forall\ \alpha \rightarrow \beta \in P$ gilt: $\quad |\beta| \geq |\alpha|$.
- Typ-0

Mächtigkeit der Grammatiken

- Regulär ⊂ kontextfrei ⊂ kontextsensitiv ⊂ Typ-0
  - Typ-0 Grammatiken umfassen auch kontextsensitive Grammatiken, kontextsensitive umfassen kontextfreie und kontextfreie umfassen reguläre Grammatiken
- Reguläre Grammatiken erzeugen die gleiche Klasse von Sprachen wie *endliche* Automaten
- Kontextfreie Grammatiken erzeugen die gleiche Klasse von Sprachen wie *Kellerautomaten*
- Zwei Grammatiken sind äquivalent, gdw. (genau dann, wenn) sie die gleiche Sprache erzeugen
  - $G_1 \equiv G_2$, gdw. $L(G_1) = L(G_2)$

Regulär vs. kontextfreie Grammatiken

	Regulär	**Kontextfrei**
Anwendung	Lexikale Analyse	Syntaktische Analyse
Erkannt durch	Endliche Automaten (Zustände + Übergänge)	Kellerautomaten (Zustände + Übergänge + Keller)
Produktionen	$A \rightarrow a \mid aB$	$A \rightarrow \alpha$
Grenzen (was geht NICHT)	Z.B. Konstruktpaare (Verschachtelung, Klammerung) ($a^n b^n$ [$n \geq 1$])	Z.B. Prüfung von Übereinstimmung von Typdeklaration und Verwendung von Typen) ($a^n b^m c^n d^m$ [$n \geq 1$, $m \geq 1$])

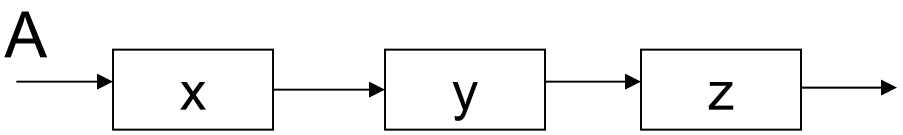
Metasprachen

- Definieren andere – komplexere – formale Sprachen
- EBNF (Extended Backus-Naur Form)
  - John Backus, Peter Naur, Niklaus Wirth
  - Variablen: Namen für syntaktische Konstrukte
  - Terminale: Elemente der Sprache, zwischen “ “

A = x y z	Sequenz: A besteht aus x, gefolgt durch y und z
A = x_1 | x_2 | ... x_k	Alternative: A ist x_1 oder x_2 oder ... x_k
A = x [y] z	Option: A ist xz oder xyz
A = x {y} z	Wiederholung: A ist xz oder xyz oder xy...yz
A = x $\{y\}^+$ z	Wiederholung: y zumindest einmal (xz ist illegal)

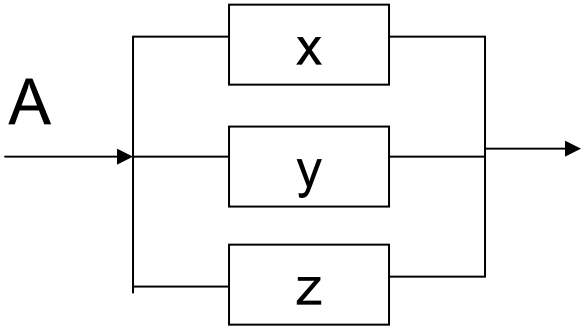
Syntaxdiagramme – EBNF graphisch

Seq. (A = x y z)



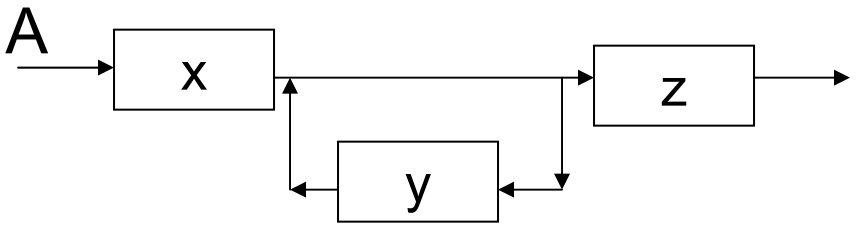
Terminale

Alternative (A = x | y | z)

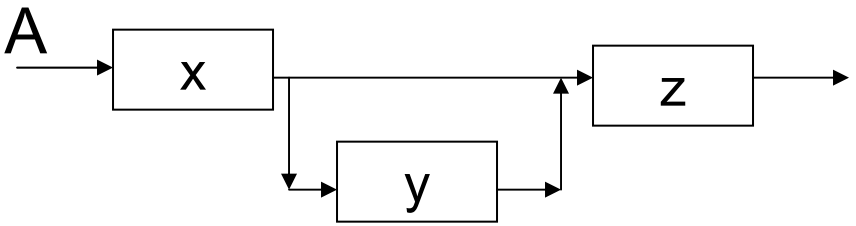


Variablen

Wiederholung (A = x {y} z)

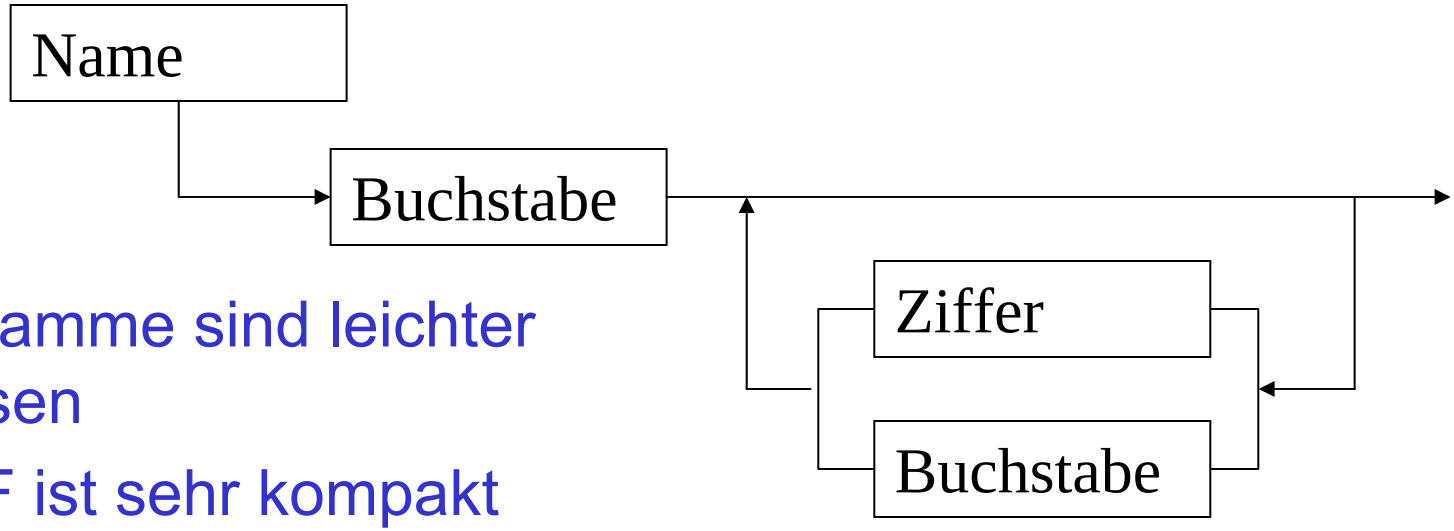


Option (A = x [y] z)



EBNF Beispiel

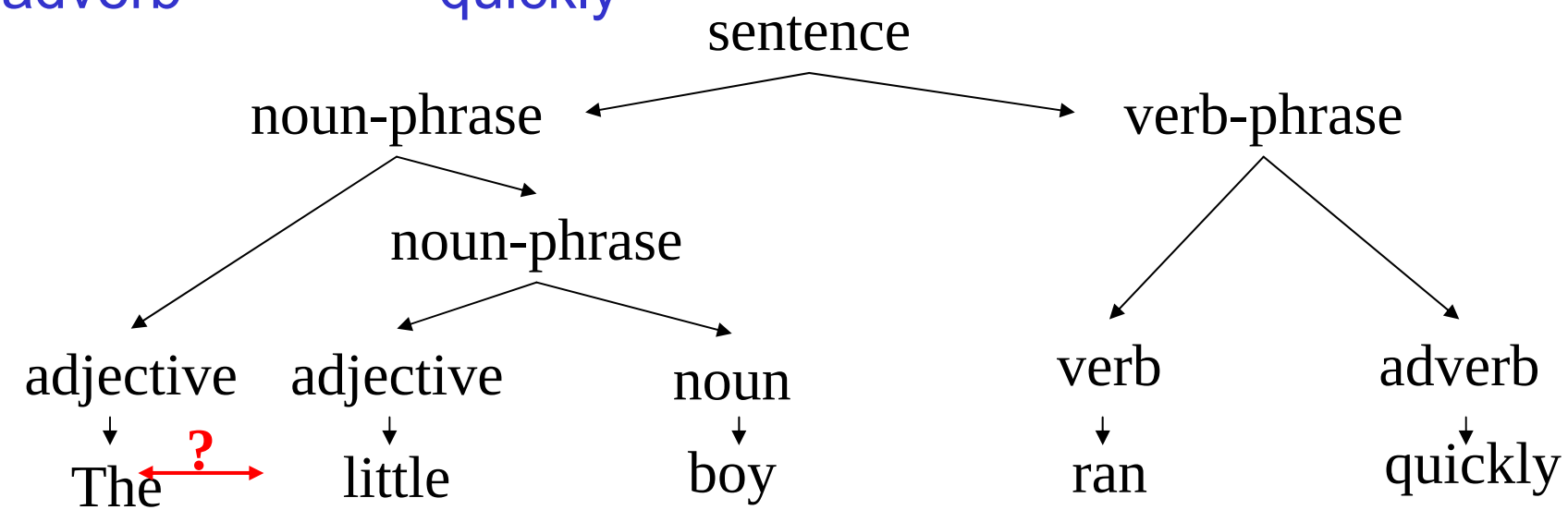
- Ziffer = “0“|“1“|“2“|“3“|“4“|“5“|“6“|“7“|“8“|“9“
- Zahl = Ziffer { Ziffer }
- Buchstabe = “a“ | “b“ | ... | “z“ | “A“ | “B“ | ... | “Z“
- Name = Buchstabe { Buchstabe | Ziffer }
- Zahl = Ziffer | Ziffer Zahl (rekursive Definition von Zahl)



- Diagramme sind leichter zu lesen
- EBNF ist sehr kompakt

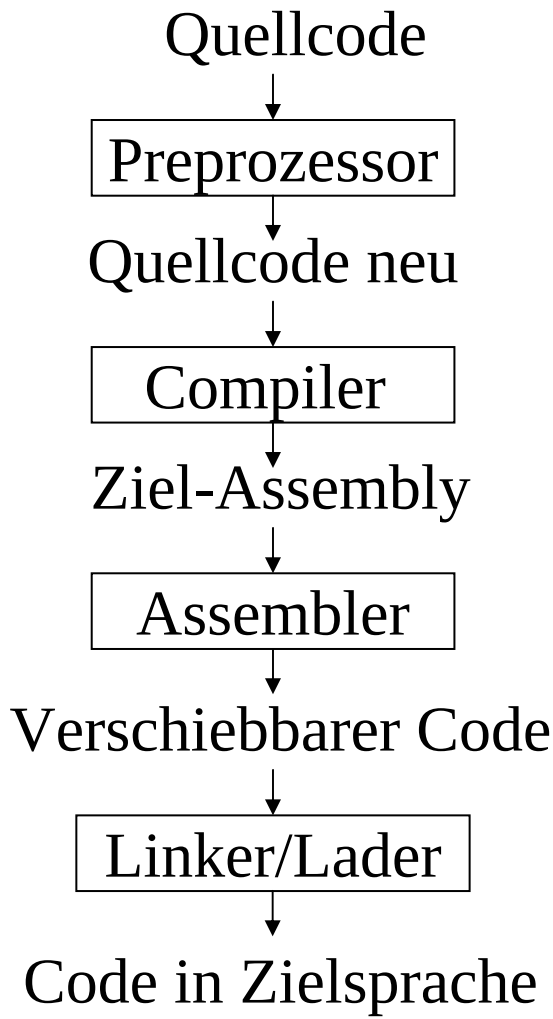
Grammatik Beispiel (in EBNF)

1. sentence = noun-phrase verb-phrase
2. noun-phrase = adjective noun-phrase | adjective noun
3. verb-phrase = verb adverb
4. adjective = “The” | “little”
5. noun = “boy”
6. verb = “ran”
7. adverb = “quickly”



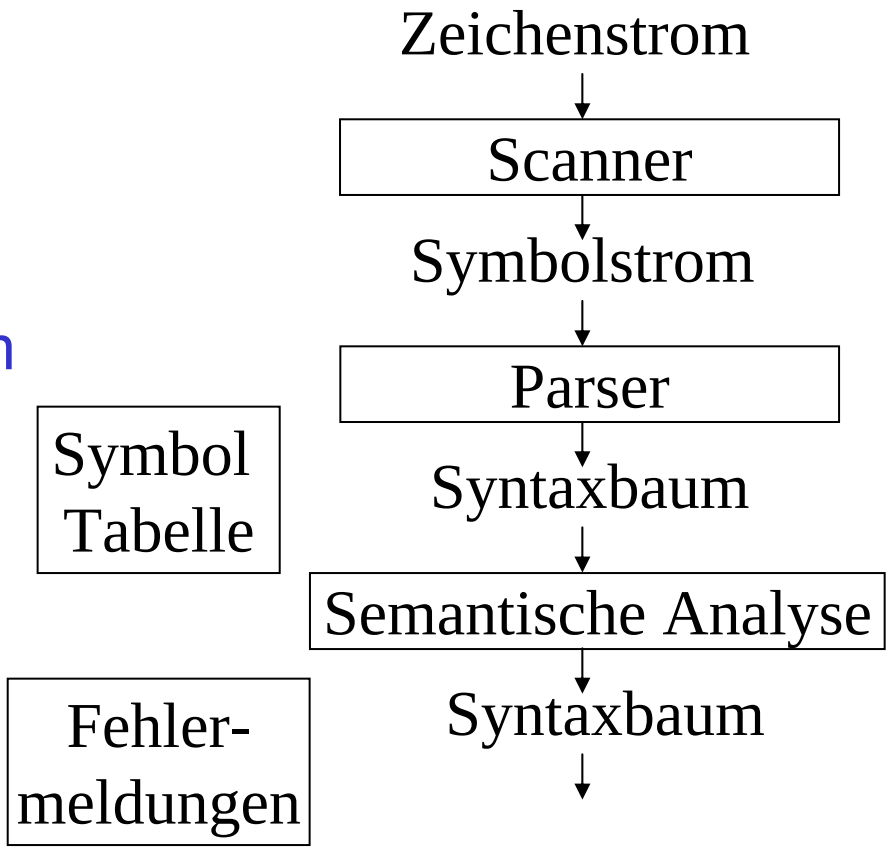
Phasen der Übersetzung / Ausführung

1. Quelle-zu-Quelle Preprozessor
   - Z.B. Markos, wie in C++
2. Compiler erzeugt Assembly Code für die Zielmaschine
3. Assembler erzeugt verschiebbaren Binärcode
4. Linker und/oder Lader berechnen die endgültigen Adressen. Die sind entweder
   - Absolut oder
   - Relativ zum Wert eines Registers



Struktur eines Compilers – Analyse

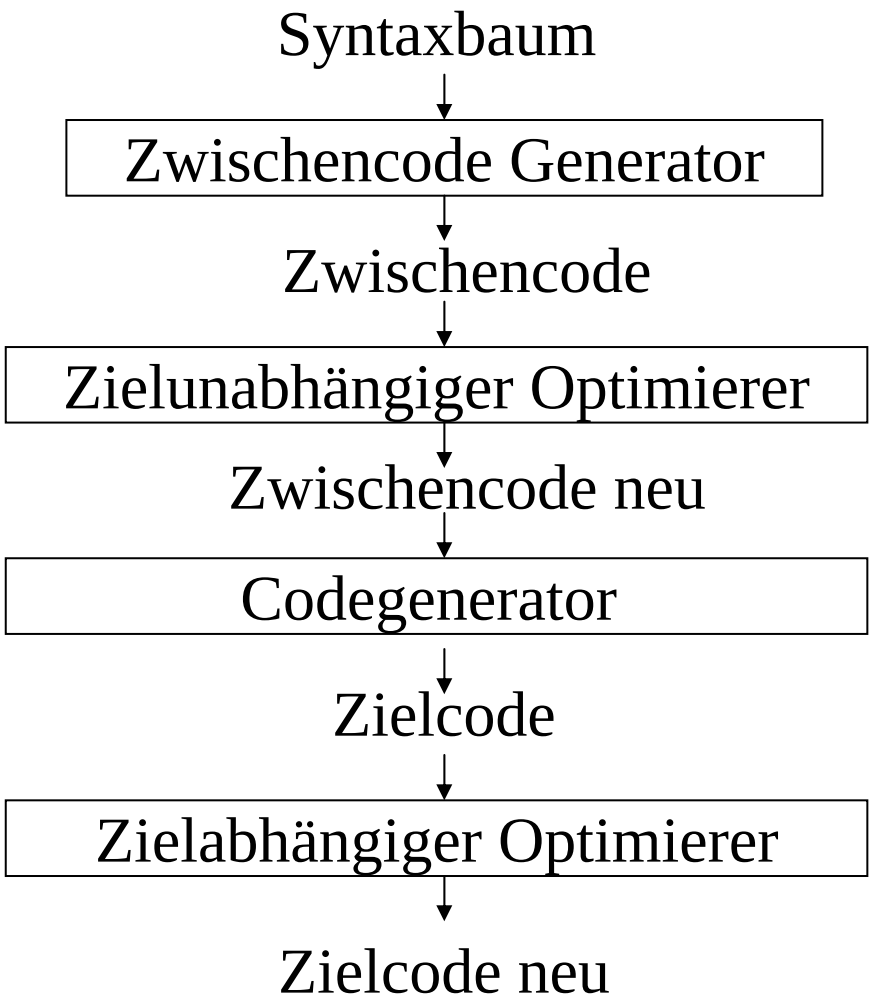
1. Lexikalische Analyse (der *Scanner)* wandelt den Zeichenstrom in einen Strom von Symbolen (Token) um
2. Der *Parser* prüft die Syntax und erzeugt den Syntaxbaum
3. Die *semantische Analyse* prüft kontextsensitive Eigenschaften, wie Typkompatibilität
4. Symboltabelle speichert die syntaktischen Elemente
5. Fehler können in jedem Schritt erzeugt werden



Struktur eines Compilers – Synthese

1. Zwischencode wird erzeugt
   - Fehler können auch noch hier gefunden werden
2. Zielunabhängige Optimierung
3. Zielcode wird erzeugt
4. Zielabhängige Optimierung

"Optimierung" nicht im Sinn von Op. Research: Bedeutet Verbesserung



“Paradigmas” von Programmiersprachen

- Prozedural (Fortran, Algol-60, C, Pascal)
  - Parametrisierte Prozeduren, wie P(p1, p2) und globaler Zustand
  - Im Mittelpunkt steht das „Verb“ (die Prozedur)
- Objekt-orientiert (C++, Java, C#, Modula-3)
  - Objekte definieren Zustandsraum (Instanz-Variablen) und Verhalten (Methoden)
  - Methoden werden *dynamisch* (zur Laufzeit) *gebunden*: o.m(p1, p2)
- Funktional (Lisp, ML)
  - Idempotente Funktionsaufrufe, ohne Seiteneffekte (im Idealfall)
- Logisch (Prolog)
  - Deklarative Fakten und Einschränkungen definieren das Programm
- Mengen basiert (SET-L, SQL)
- Parallel (High-performance Fortran, Vienna Fortran)
- Verteilt (Orca, Java RMI)

Eine kleine Sprachgeschichte (1)

- Fünfziger Jahre
  - Maschinencode, Assembly-Code
  - Bedarf an *höheren,* “abstrakten” Sprachen
  - Fortran (für Wissenschaft), Cobol (für Wirtschaft)
    - Syntax + Semantik “definiert” durch den (IBM) Compiler
    - 1958: Übersetzter Code nur zweimal langsamer als händischer
- Sechziger Jahre
  - Bedarf an *wissenschaftlich fundierte* Sprachen
  - Algol-60 – „the birth of computer science“ (N. Wirth)
    - Formale Syntax; Allgemeine Ausdrücke; Blockstruktur; Gültigkeitsbereiche; Parameterübergabe, Rekursion; Boolean
  - Simula-67 (O.-J. Dahl, K. Nygaard)
    - Erste objekt-orientierte Programmiersprache
    - “Wiederentdeckt” 15 Jahre später (Smalltalk-80)

Eine kleine Sprachgeschichte (2)

- Siebziger Jahre
  - Bedarf an sicheren (*safe)* Sprachen
- Pascal (N. Wirth)
  - Strikte, statische Typprüfung + benutzerdefinierte Typen
  - Unterstützt „strukturierte Programmierung“ durch gründlich gewählte Anweisungen und Typkonstruktoren
- C (D. Ritchie)
  - Entstanden als Implementierungssprache für Unix
  - Eine Art „höhere“ Assembly-Sprache
  - Ist nicht sicher (z.B. Pointers ohne Typ)
  - Merkwürdige Karriere: C als Implementierungssprache für große Anwendungen – eine grobe Fehlentwicklung

Eine kleine Sprachgeschichte (3)

- Achtziger Jahre - Modularität, Objekt-Orientierung
  - Mesa (Xerox PARC)
    - Interface- und Implementierungsmodule, Multi-threading
  - Modula-2 (N. Wirth) – wie Mesa, nur einfacher
  - Smalltalk-80 (A. Goldmann, Xerox PARC)
  - Eiffel (B. Meyer), Ada (J. Ichbiah), Modula-3 (DEC SRC)
  - C++ (B. Stroustrup)
    - Sehr mächtig, Mehrfachvererbung, nicht typsicher (C-Kompatib.)
- Neunziger Jahre
  - Java (J. Gosling, Sun)
    - Typsicher (fast), Multi-threading, Garbage Collection
  - C# (Microsoft)
    - Mächtiger, effizienter und sicherer als Java – zu spät?