Compilerbau

Prof. Dr. Hanspeter Mössenböck

http://ssw.jku.at/Teaching/Lectures/CB/VL/

1. Überblick

Hanspeter Mössenböck, Institut für Systemsoftware, JKU


Kurze Geschichte des Compilerbaus

Früher Geheimwissenschaft, heute eines der am besten erforschten Informatikgebiete

1957	Fortran	Erste Compiler (arithmetische Ausdrücke, Anweisungen, Prozeduren)
1960	Algol	Erste saubere Sprachdefinition (Grammatiken in Backus-Naur-Form, Blockstruktur, Rekursion)
1970	Pascal	Benutzerdefinierte Typen, virtuelle Maschinen (P-Code)
1985	C++	Objektorientierung, Exceptions, Templates
1995	Java	Just-in-time-Compilation, mobiler Code

Wir betrachten hier nur den imperativen Compilerbau

Für funktionale Sprachen (z.B. Lisp) oder logische Sprachen (z.B. Prolog) sind andere Techniken nötig.

Wozu lernen wir Compilerbau?

Gehört zur Allgemeinbildung jedes Informatikers

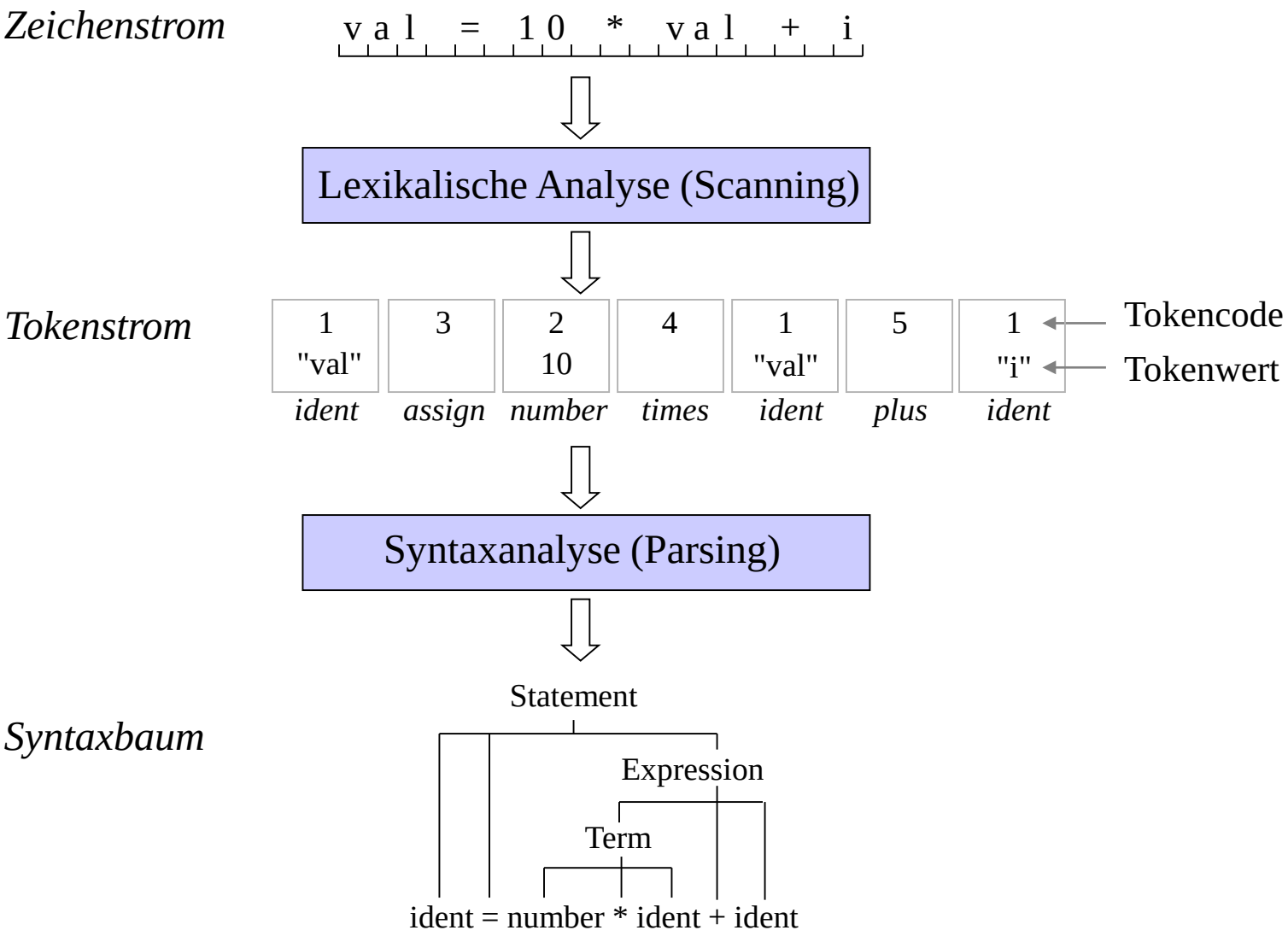
- Wie funktioniert ein Compiler?
- Wie funktioniert ein Computer auf Maschinenebene?
  (Instruktionssatz, Register, Adressierungsarten, Laufzeitdatenstrukturen, ...)
- In welchen Code werden Sprachkonstrukte übersetzt? (Gefühl für Effizienz)
- Gefühl für gutes Sprachdesign; Umgang mit Grammatiken
- Gibt Gelegenheit für ein nichttriviales Programmierprojekt im Studium

Compilerbau-Kenntnisse sind auch im Software Engineering nützlich

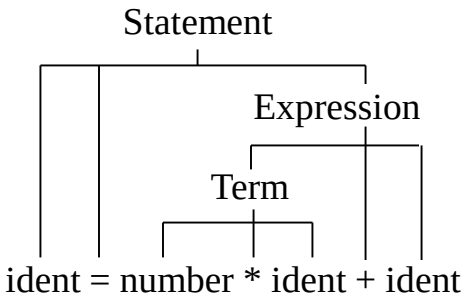
- Lesen syntaktisch strukturierter Kommandozeilenparameter
- Lesen strukturierter Dateien (z.B. XML-Dateien, Stücklisten, Bild-Dateien, ...)
- Suchen in hierarchischen Namensräumen
- Interpretative Abarbeitung von Codes
- ...

1. Überblick

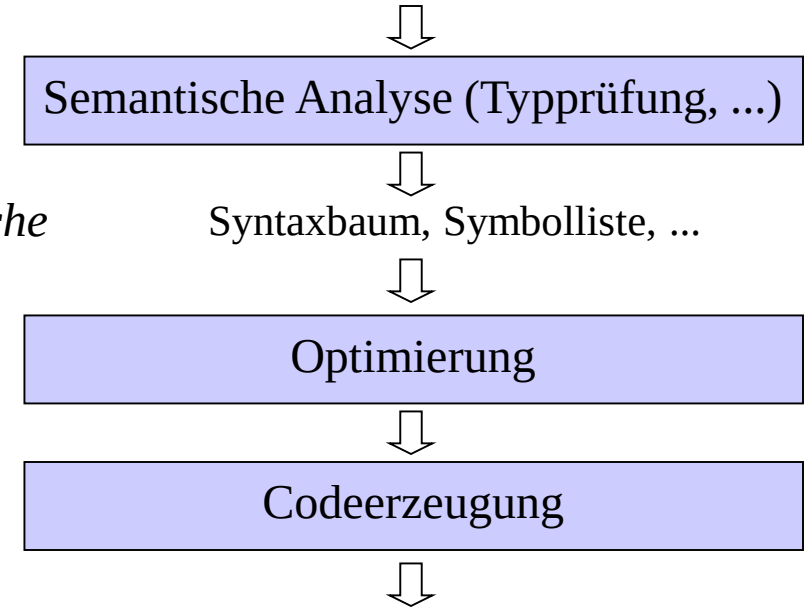
Dynamische Struktur eines Compilers



Dynamische Struktur eines Compilers



Syntaxbaum



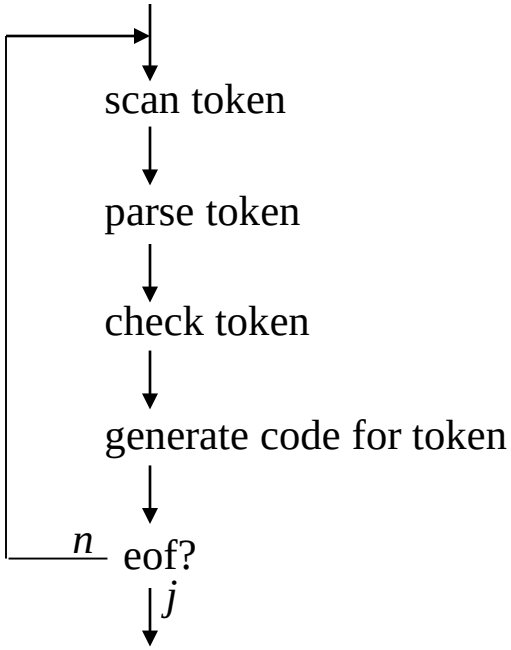
Zwischensprache Syntaxbaum, Symbolliste, ...

Maschinencode

```
const 10
load  1
mul
...
```

Einpass-Compiler

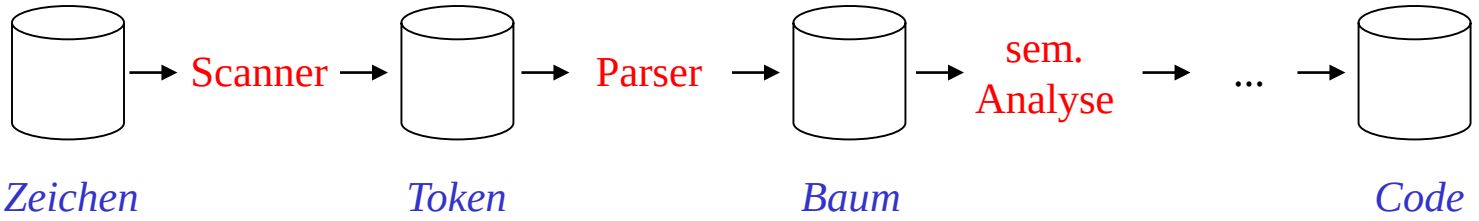
Die einzelnen Phasen arbeiten verzahnt



Während das Quellprogramm gelesen wird, wird bereits das Zielprogramm erzeugt

Mehrpass-Compiler

Phasen sind eigene Programme, die nacheinander ablaufen

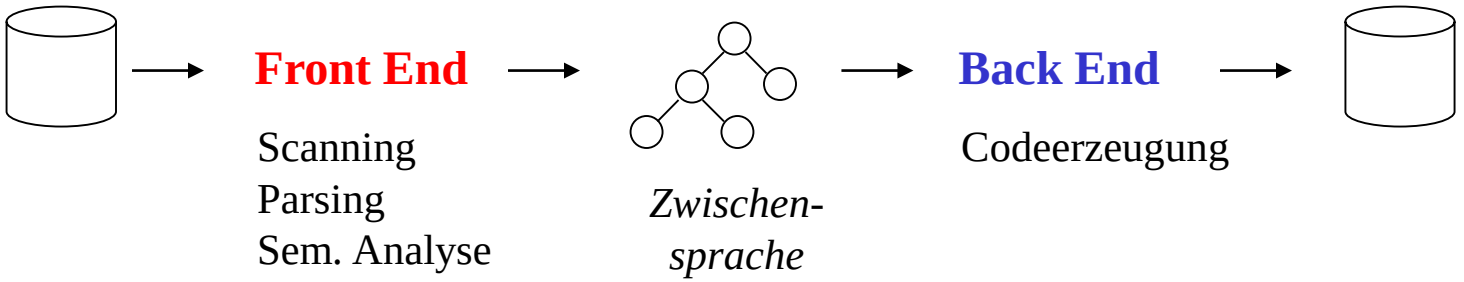


Jede Phase liest von einer Datei und schreibt ihre Ausgabe auf eine neue Datei

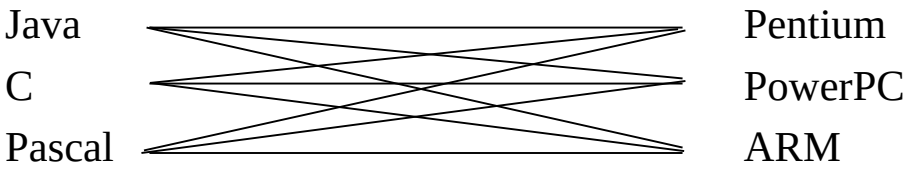
Wann ist das notwendig?

- wenn der Hauptspeicher zu klein ist (heute irrelevant)
- wenn die Sprache sehr komplex ist (auch eher irrelevant)
- wenn einfache Portierbarkeit gewünscht ist

Heute oft Zweipass-Compiler



sprachabhängig **maschinenabhängig**



beliebig kombinierbar

Vorteile

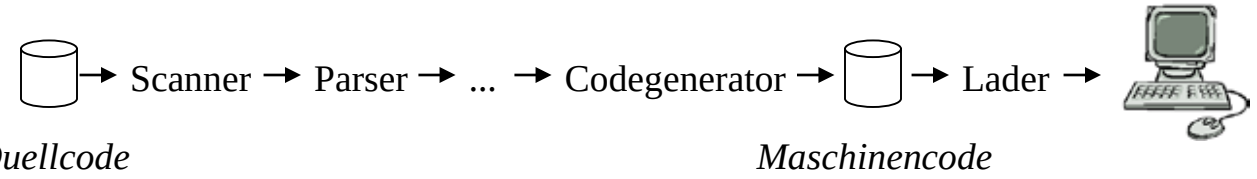
- bessere Portierbarkeit
- Kombination beliebiger Front Ends mit beliebigen Back Ends möglich
- Zwischensprache ist einfacher optimierbar als Quellsprache

Nachteile

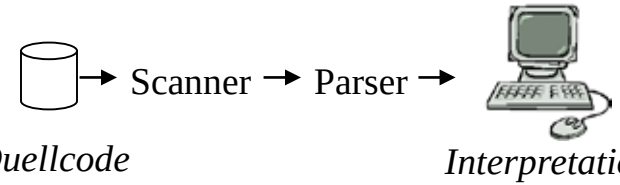
- langsamer
- mehr Speicherverbrauch

Compiler versus Interpreter

Compiler übersetzt in Maschinencode

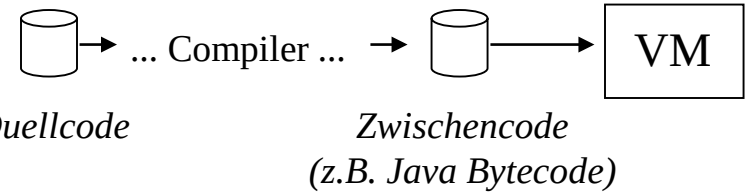


Interpreter führt Quellprogramm "direkt" aus



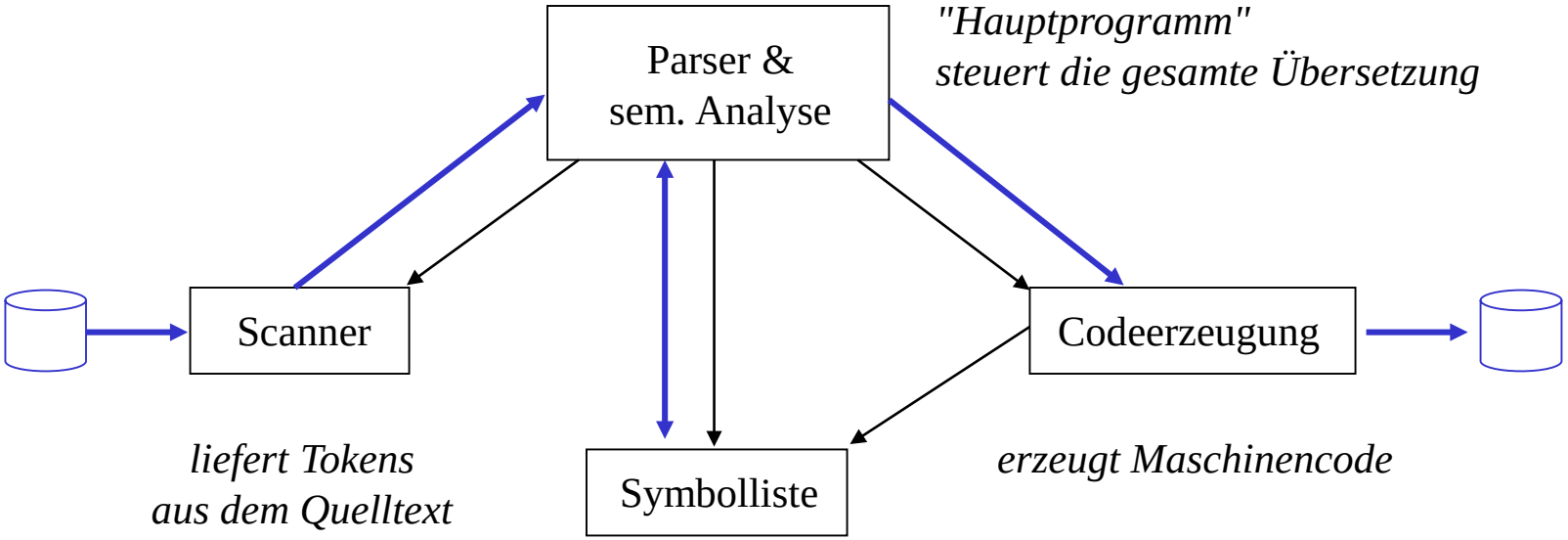
- Anweisungen in einer Schleife laufen jedesmal erneut durch Scanner und Parser

Auch Interpretation von Zwischencode möglich



- Quellcode wird in den Code einer *virtuellen Maschine* (VM) übersetzt
- VM interpretiert den Code; simuliert physische Maschine

Statische Struktur eines Compilers



Aufrufbeziehung

Datenfluss

1. Überblick

Woraus besteht eine Grammatik?

Beispiel

```
Statement = "if" "(" Condition ")" Statement ["else" Statement].
```

Vier Bestandteile

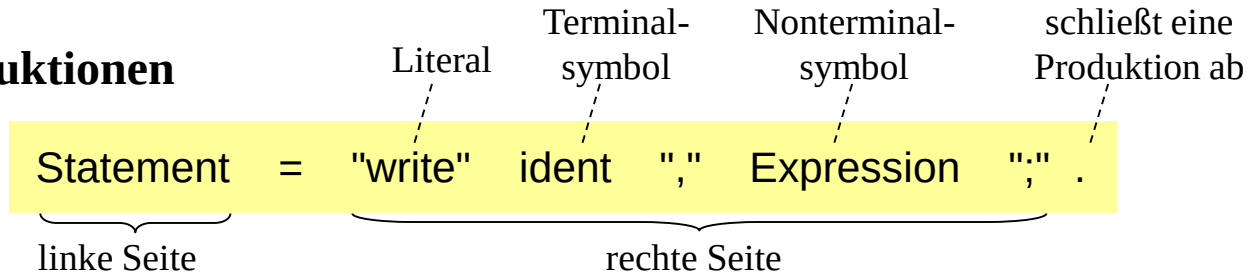
Terminalsymbole	werden nicht mehr weiter zerlegt	"if", ">=", ident, number, ...
Nonterminalsymbole	werden weiter zerlegt	Statement, Condition, Type, ...
Produktionen	Ableitungsregeln	Statement = Designator "=" Expr ";". Designator = ident ["." ident]. ...
Startsymbol	oberstes Nonterminal-symbol	Java

EBNF-Schreibweise

Extended Backus-Naur Form

John Backus: entwickelte ersten Fortran-Compiler
Peter Naur: gab den Algol60-Report heraus

Produktionen



Konvention:

- Terminalsymbole beginnen mit Kleinbuchstaben
- Nonterminalsymbole beginnen mit Großbuchstaben

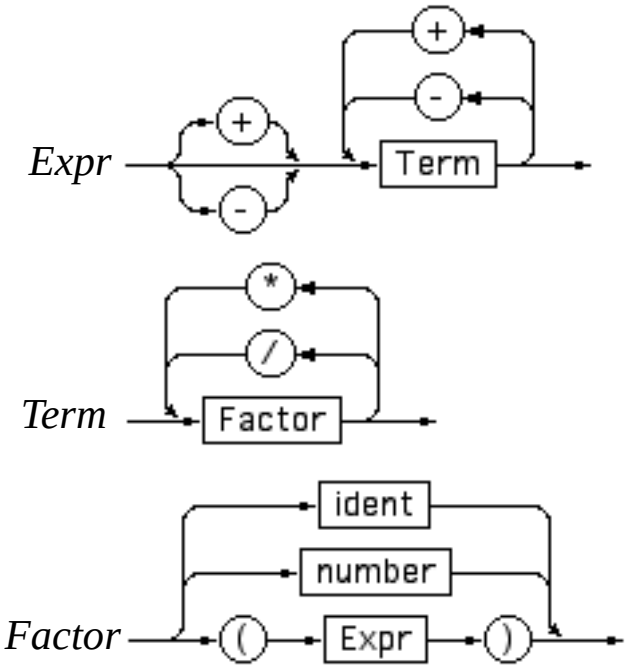
Metasymbole

|	trennt Alternativen	a | b | c	≡ a oder b oder c
(...)	fasst Alternativen zusammen	a (b | c)	≡ ab | ac
[...]	Option	[a] b	≡ ab | b
{...}	Wiederholung	{a}b	≡ b | ab | aab | aaab | ...

Beispiel: Grammatik der arithmetischen Ausdrücke

Produktionen

```
Expr   = ["+" | "-"] Term {("+" | "-") Term}.
Term   = Factor {("*" | "/") Factor}.
Factor = ident | number | "(" Expr ")".
```



Terminalsymbole

Einfache TS: "+", "-", "*", "/", "(", ")"
(nur 1 Ausprägung)

Terminalklassen: ident, number
(mehrere Ausprägungen)

Nonterminalsymbole

Expr, Term, Factor

Startsymbol

Expr

Vorrangregeln

Mit Grammatiken lassen sich auch Vorrangregeln von Operatoren ausdrücken

```
Expr   = ["+" | "-"] Term {("+" | "-") Term}.
Term   = Factor {("*" | "/") Factor}.
Factor = ident | number | "(" Expr ")".
```

Eingabe: - a * 3 + b / 4 - c

⇒ - ident * number + ident / number - ident

⇒ - Factor * Factor + Factor / Factor - Factor

⇒ - Term + Term - Term

⇒ Expr

* und / binden stärker als + und -

\- bezieht sich nicht auf *a*, sondern auf *a**3

Wie müsste man die Grammatik umformen, so dass sich - auf *a* bezieht?

Terminale Anfänge von NTS

Mit welchen TS kann ein NTS beginnen?

```
Expr   = ["+" | "-"] Term {("+" | "-") Term}.
Term   = Factor {("*" | "/") Factor}.
Factor = ident | number | "(" Expr ")".
```

First(Factor) = ident, number, "("

First(Term) = First(Factor)
= ident, number, "("

First(Expr) = "+", "-", First(Term)
= "+", "-", ident, number, "("

Terminale Nachfolger von NTS

Welche TS können auf ein NTS folgen?

```
Expr   = ["+" | "-"] Term {("+" | "-") Term}.
Term   = Factor {("*" | "/") Factor}.
Factor = ident | number | "(" Expr ")".
```

Follow(Expr) = ")", eof

Follow(Term) = "+", "-", Follow(Expr)
= "+", "-", ")", eof

Follow(Factor) = "*", "/", Follow(Term)
= "*", "/", "+", "-", ")", eof

Wo kommt *Expr* auf der rechten Seite einer Produktion vor? Welche TS können dort folgen?

Begriffe der Formalen Sprachen

Alphabet

Die Menge der Terminal- und Nonterminalsymbole einer Grammatik

Kette

Eine endliche Folge von Symbolen aus einem Alphabet

Ketten werden mit griechischen Symbolen bezeichnet (α, β, γ, ...)
z.B: α = ident + number
β = - Term + Factor * number

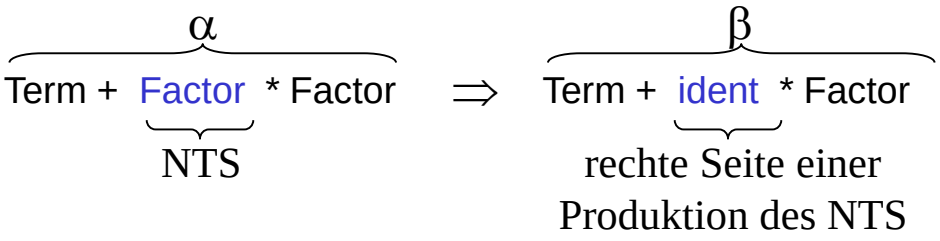
Leere Kette

Die Kette, die kein Symbol enthält.
Wird mit ε bezeichnet.

Ableitungen und Reduktionen

Ableitung

$\alpha \Rightarrow \beta$ (direkte Ableitung)



$\alpha \Rightarrow^{*} \beta$ (indirekte Ableitung) $\alpha \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow ... \Rightarrow \gamma_n \Rightarrow \beta$

$\alpha \Rightarrow^{L} \beta$ (linkskanonische Ableitung) das <u>linkeste</u> NTS in α wird durch seine rechte Seite ersetzt

$\alpha \Rightarrow^{R} \beta$ (rechtskanonische Ableitung) das <u>rechteste</u> NTS in α wird durch seine rechte Seite ersetzt

Reduktion

Gegenteil einer Ableitung:
Die rechte Seite einer Produktion in β wird durch das entsprechende NTS ersetzt

Löschbarkeit

Eine Kette α heißt löschbar, wenn sie in die leere Kette abgeleitet werden kann ($\alpha \Rightarrow^* \varepsilon$)

Beispiel

```
X = Y Z.
Y = [b].
Z = c | d | .
```

Y ist löschbar: $Y \Rightarrow \varepsilon$

Z ist löschbar: $Z \Rightarrow \varepsilon$

X ist löschbar: $X \Rightarrow Y\ Z \Rightarrow Z \Rightarrow \varepsilon$

Weitere Begriffe

Phrase

Eine aus einem Nonterminalsymbol ableitbare Kette.

Term-Phrasen sind z.B.: Factor
Factor * Factor
ident * Factor
...

```
Expr   = ["+" | "-"] Term {("+" | "-") Term}.
Term   = Factor {("*" | "/") Factor}.
Factor = ident | number | "(" Expr ")".
```

Satzform

Eine aus dem Startsymbol ableitbare Phrase.

Z.B.: Expr
Term + Term + Term
Term + Factor * ident + Term
...

Satz

Eine Satzform, die nur aus Terminalsymbolen besteht.

Z.B.: ident * number + ident

Sprache (Formale Sprache)

Die Menge aller Sätze einer Grammatik (meist unendlich groß).
Z.B.: Die Sprache Java ist die Menge aller gültigen Java-Programme

Rekursion

Eine Produktion ist rekursiv, wenn $X \Rightarrow^* \omega_1\ X\ \omega_2$

Verwendet zur Darstellung von Wiederholungen und Schachtelungen

Direkte Rekursion $X \Rightarrow \omega_1\ X\ \omega_2$

Linksrekursion	X = b | X a.	X ⇒ X a ⇒ X a a ⇒ X a a a ⇒ ... ⇒ b a a a a a
Rechtsrekursion	X = b | a X.	X ⇒ a X ⇒ a a X ⇒ a a a X ⇒ ... ⇒ a a a a a b
Zentralrekursion	X = b | "(" X ")".	X ⇒ (X) ⇒ ((X)) ⇒ (((X))) ⇒ (((... (b)...)))

Indirekte Rekursion $X \Rightarrow^* \omega_1\ X\ \omega_2$

Beispiel

```
Expr   = Term {"+" Term}.
Term   = Factor {"*" Factor}.
Factor = id | "(" Expr ")".
```

Expr ⇒ Term ⇒ Factor ⇒ "(" Expr ")"

Beseitigung von Linksrekursion

Linksrekursion stört bei der Topdown-Syntaxanalyse

X = b | X a.

Beide Alternativen fangen mit *b* an.
Der Parser kann sich nicht entscheiden, welche er wählen soll

Linksrekursion kann immer in Iteration umgewandelt werden

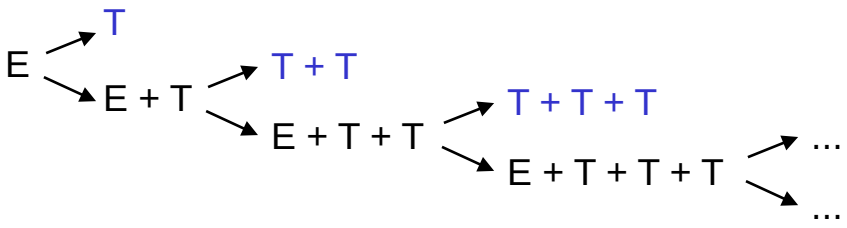
X ⇒ baaa....a

X = b {a} .

Anderes Beispiel

E = T | E "+" T.

Überlegen, welche Phrasen erzeugt werden können



Daraus sieht man, wie die iterative EBNF-Regel auszusehen hat

E = T {"+" T}.

1. Überblick

Einfache BNF-Schreibweise

Terminalsymbole	ohne Hochkommas (ident, +, -)
Nonterminalsymbole	in spitzen Klammern (<Expr>, <Term>)
Regelseiten	durch ::= getrennt

BNF-Grammatik der arithmentischen Ausdrücke

```
<Expr>    ::=  <Sign> <Term>
<Expr>    ::=  <Expr> <Addop> <Term>

<Sign>    ::=  +
<Sign>    ::=  -
<Sign>    ::=

<Addop>   ::=  +
<Addop>   ::=  -

<Term>    ::=  <Factor>
<Term>    ::=  <Term> <Mulop> <Factor>

<Mulop>   ::=  *
<Mulop>   ::=  /

<Factor>  ::=  ident
<Factor>  ::=  number
<Factor>  ::=  ( <Expr> )
```

- Alternativen werden zu mehreren Regeln
- Wiederholung muss durch Rekursion ausgedrückt werden

Vorteile

- weniger Metasymbole (kein |, (), [], {})
- Syntaxbaum lässt sich einfacher konstruieren

Nachteile

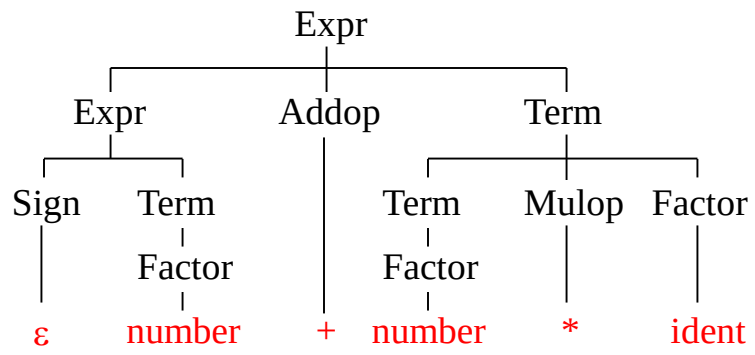
- länger
- schwerer lesbar

Syntaxbaum

Zeigt die Ableitungsstruktur für einen Satz einer Grammatik

z.B. für 10 + 3 * i

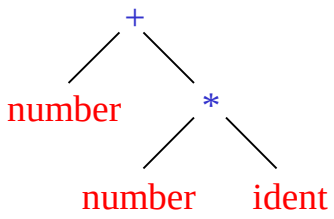
Konkreter Syntaxbaum (Parse Tree)



wäre mit EBNF nicht so einfach
wegen [...] und {...}, z.B.:
Expr = [Sign] Term {Addop Term}.

Vorrangregeln beachtet:
NTS weiter unten im Baum haben
Vorrang vor NTS weiter oben im Baum.

Abstrakter Syntaxbaum (Blätter = Operanden, innere Knoten = Operatoren)



oft als interne Programmdarstellung
für Optimierungen verwendet

Mehrdeutigkeit

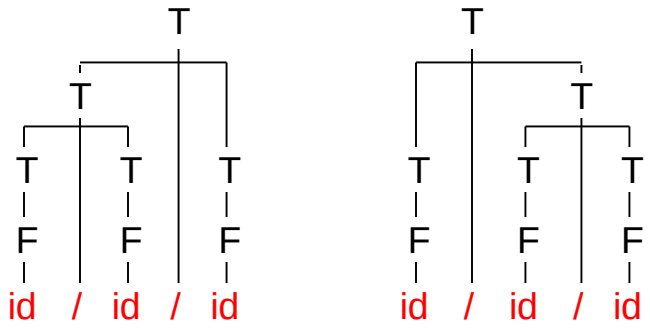
Eine Grammatik ist mehrdeutig, wenn man für einen Satz mehrere Syntaxbäume angeben kann.

Beispiel

```
T = F | T "/" T.
F = id.
```

Beispielsatz: T ⇒ T / T ⇒ T / T / T ⇒ F / F / F ⇒ id / id / id

Über diesen Satz können zwei verschiedene Syntaxbäume gebaut werden



Mehrdeutige Grammatiken sind zur Syntaxanalyse ungeeignet!

Beseitigung von Mehrdeutigkeit

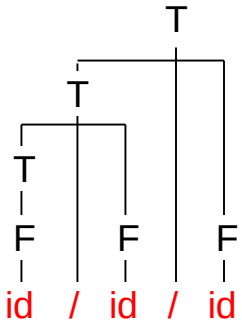
Im Beispiel

```
T = F | T "/" T.
F = id.
```

ist nicht die Sprache mehrdeutig, sondern nur die Grammatik.

Die Grammatik kann umgeformt werden zu

```
T = F | T "/" F.
F = id.
```



d.h. Ausdruck wird von links nach rechts ausgewertet

nur dieser Syntaxbaum ist möglich

Noch besser: Umformung in EBNF

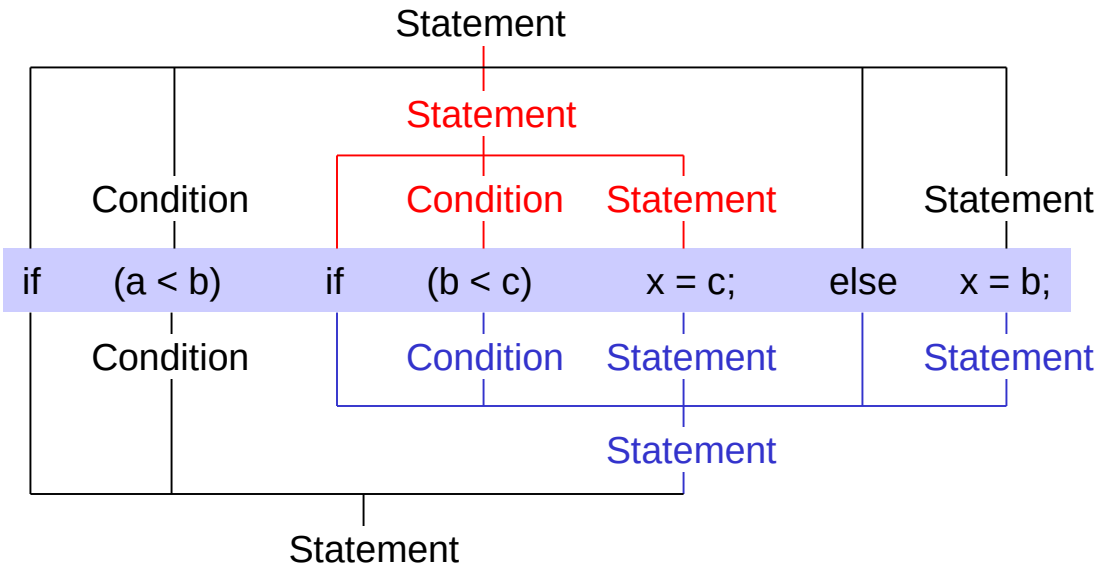
```
T = F {"/" F}.
F = id.
```

Inhärente Mehrdeutigkeit

Es gibt Sprachen, die inhärent mehrdeutig sind

Beispiel: Dangling Else

```
Statement =  Assignment
           | "if" Condition Statement
           | "if" Condition Statement "else" Statement
           | ... .
```



Es lässt sich keine eindeutige Grammatik finden!

Ausweg in Java
Man erkennt die längst-mögliche rechte Seite der Statement-Produktion
⇒ führt zum unteren Syntaxbaum

1. Überblick

Grammatikklassen

Hierarchie von Noam Chomsky (1956)

Grammatiken sind prinzipiell Mengen von Produktionen der Form $\alpha = \beta$.

Klasse 0 **Unbeschränkte Grammatiken** (α und β beliebig)

Beispiel:
```
X       = a X b | Y c Y.
a Y c   = d.
d Y     = b b.
```
$X \Rightarrow aXb \Rightarrow aYcYb \Rightarrow dYb \Rightarrow bbb$

Erkennbar durch Turingmaschinen

Klasse 1 **Kontextsensitive Grammatiken** ($|\alpha| \leq |\beta|$)

Beispiel: `a X = a b c.`

Erkennbar durch linear beschränkte Automaten

Klasse 2 **Kontextfreie Grammatiken** (α = NT, β beliebig)

Beispiel: `X = a b c.`

Erkennbar durch Kellerautomaten

Klasse 3 **Reguläre Grammatiken** (α = NT, β = T | T NT)

Beispiel: `X = b | b Y.`

Erkennbar durch endliche Automaten

Im Compilerbau sind nur diese beiden Grammatikklassen relevant

1. Überblick

Beispiel-Programm


```
program P
   final int size = 10;
   class Table {
      int[] pos;
      int[] neg;
   }
   Table val;
{
   void main()
      int x, i;
   {  //---------- initialize val ----------
      val = new Table;
      val.pos = new int[size];
      val.neg = new int[size];
      i = 0;
      while (i < size) {
         val.pos[i] = 0; val.neg[i] = 0; i++;
      }
      //---------- read values ----------
      read(x);
      while (x != 0) {
         if (0 < x && x < size) val.pos[x]++;
         else if (-size < x && x < 0) val.neg[-x]++;
         read(x);
      }
   }
}
```

Hauptprogramm; keine getrennte Übersetzung

Klassen (ohne Methoden)

globale Variablen

lokale Variablen

Lexikalische Struktur von MicroJava

Namen ident = letter {letter | digit | '_'}.

Zahlen number = digit {digit}. alle Zahlen sind vom Typ *int*

Zeichenkonstanten charConst = " ' " char " ' ". alle Zeichenkonstanten sind vom Typ *char* (dürfen \r, \n, \' und \\ enthalten)

keine Strings

Schlüsselwörter

program	class					
if	else	while	read	print	return	break
void	final	new				

Operatoren

+	-	*	/	%	++	--
==	!=	>	>=	<	<=	
&&	||					
(	)	[	]	{	}	
=	+=	-=	*=	/=	%=	
;	,	.				

Kommentare // ... eol

Typen *int* *char* Arrays Klassen

Syntaktische Struktur von MicroJava

Programmstruktur

```
Program  =  "program" ident
            {ConstDecl | VarDecl | ClassDecl}
            "{" {MethodDecl} "}".
```

```
program P
   ... declarations ...
{  ... methods ...
}
```

Deklarationen

```
ConstDecl   =  "final" Type ident "=" (number | charConst) ";".
VarDecl     =  Type ident {"," ident} ";".
MethodDecl  =  (Type | "void") ident "(" [FormPars] ")"
               {VarDecl} Block.

Type        =  ident [ "[" "]" ].
FormPars    =  Type ident {"," Type ident}.
```

nur eindimensionale Arrays

Syntaktische Struktur von MicroJava

Anweisungen

```
Block       =  "{" {Statement} "}".
Statement   =  Designator  (   AssignOp Expr ";"
                           |   ActPars ";"
                           |   "++" ";"
                           |   "--" ";"
                           )
            |  "if" "(" Condition ")" Statement ["else" Statement]
            |  "while" "(" Condition ")" Statement
            |  "break" ";"
            |  "return" [Expr] ";"
            |  "read" "(" Designator ")" ";"
            |  "print" "(" Expr ["," number] ")" ";"
            |  Block
            |  ";".
AssignOp    =  "=" | "+=" | "-=" | "*=" | "/=" | "%=".
ActPars     =  "(" [ Expr {"," Expr} ] ")".
```

- Eingabe von *System.in*
- Ausgabe auf *System.out*

Syntaktische Struktur von MicroJava

Ausdrücke

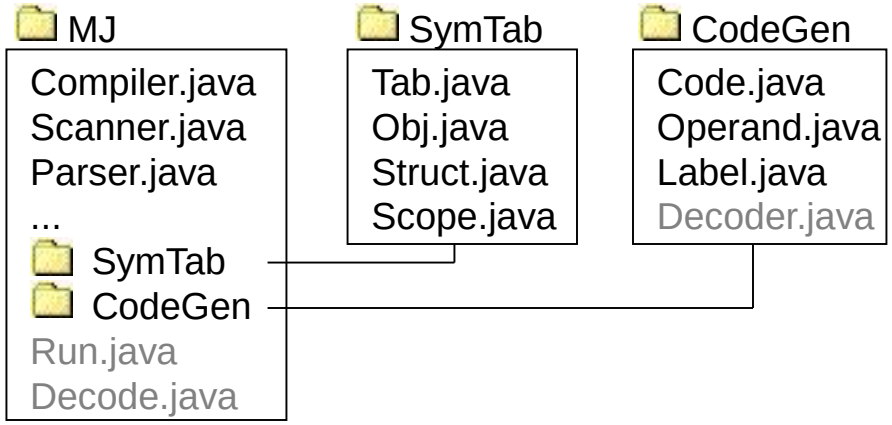
```
Condition  =  CondTerm {"||" CondTerm}.
CondTerm   =  CondFact {"&&" CondFact}.
CondFact   =  Expr Relop Expr.
Relop      =  "==" | "!=" | ">" | ">=" | "<" | "<=".
```

```
Expr       =  ["-"] Term {Addop Term}.
Term       =  Factor {Mulop Factor}.
Factor     =  Designator [ ActPars ]
           |  number
           |  charConst
           |  "new" ident [ "[" Expr "]" ]
           |  "(" Expr ")".
Designator =  ident { "." ident | "[" Expr "]" }.
Addop      =  "+" | "-".
Mulop      =  "*" | "/" | "%".
```

keine Konstruktoren

Aufruf

Paketstruktur



Compiler-Aufruf

```
java MJ.Compiler  myProg.mj
```

myProg.mj → Compiler → myProg.obj

Ausführung

```
java MJ.Run  myProg.obj  -debug
```

myProg.obj → Interpreter

Decodierung

```
java MJ.Decode  myProg.obj
```

myProg.obj → Decoder → bytecode on console