Übersetzerbau / Compiler Construction

- Überblick
- Lexikalische Analyse

Teil der Folien mit freundlicher Genehmigung von
Prof. Dr. Hanspeter Mössenböck
http://www.ssw.uni-linz.ac.at/General/Staff/HM

Überblick - Motivation

Kurze Geschichte des Übersetzerbaus

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

Wir betrachten hier nur den Übersetzerbau von imperativen Sprachen

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

Wozu lernen wir Übersetzerbau?

Gehört zur Allgemeinbildung jedes Informatikers

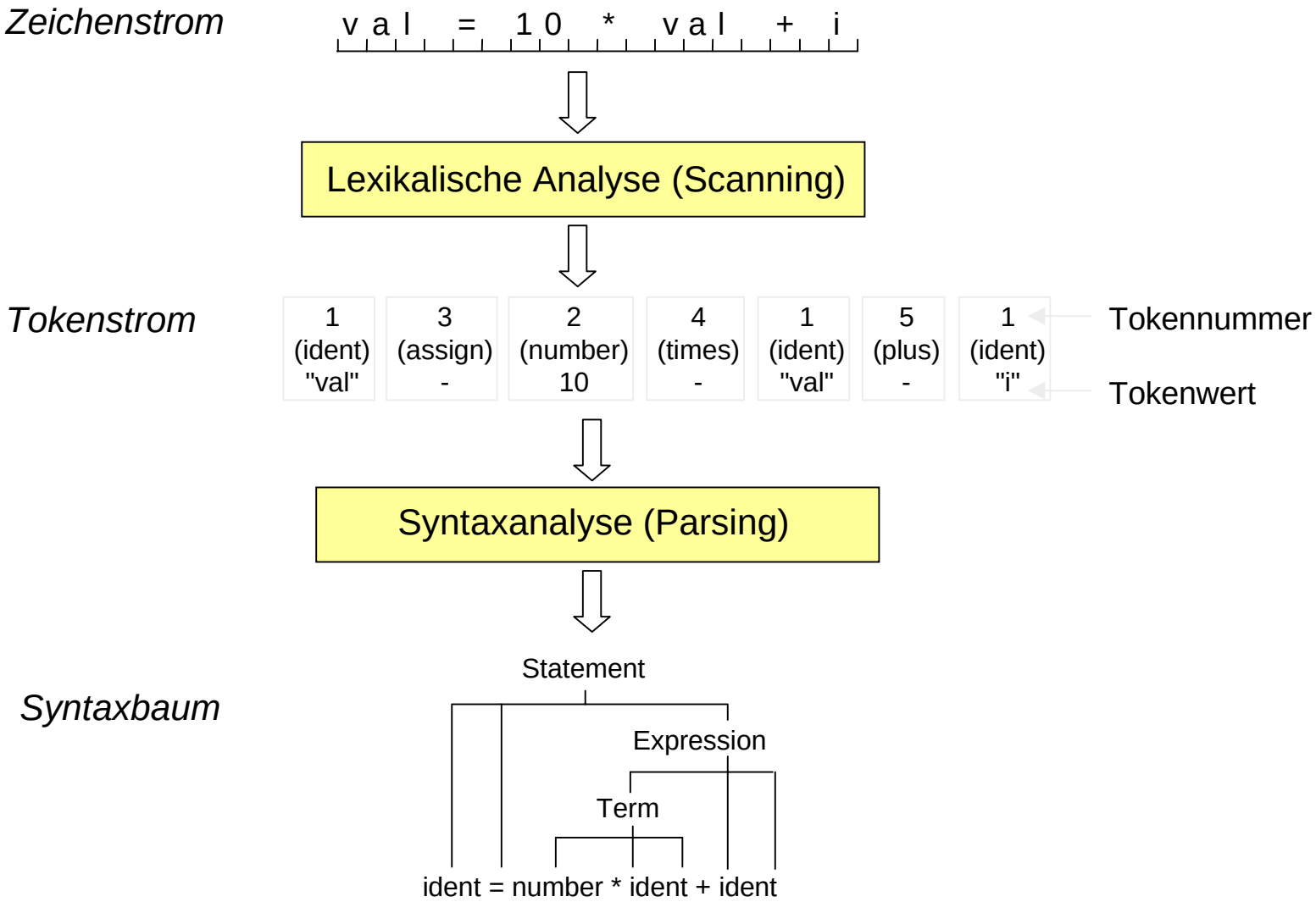
- Wie funktioniert ein Compiler?
- In welchen Code werden Sprachkonstrukte übersetzt? (Gefühl für Effizienz)
- Gefühl für gutes Sprachdesign
- Wie funktioniert ein Computer auf Maschinenebene?
  (Instruktionssatz, Register, Adressierungsarten, Laufzeitdatenstrukturen, ...)

Übersetzerbau-Kenntnisse sind auch im Software-Engineering nützlich

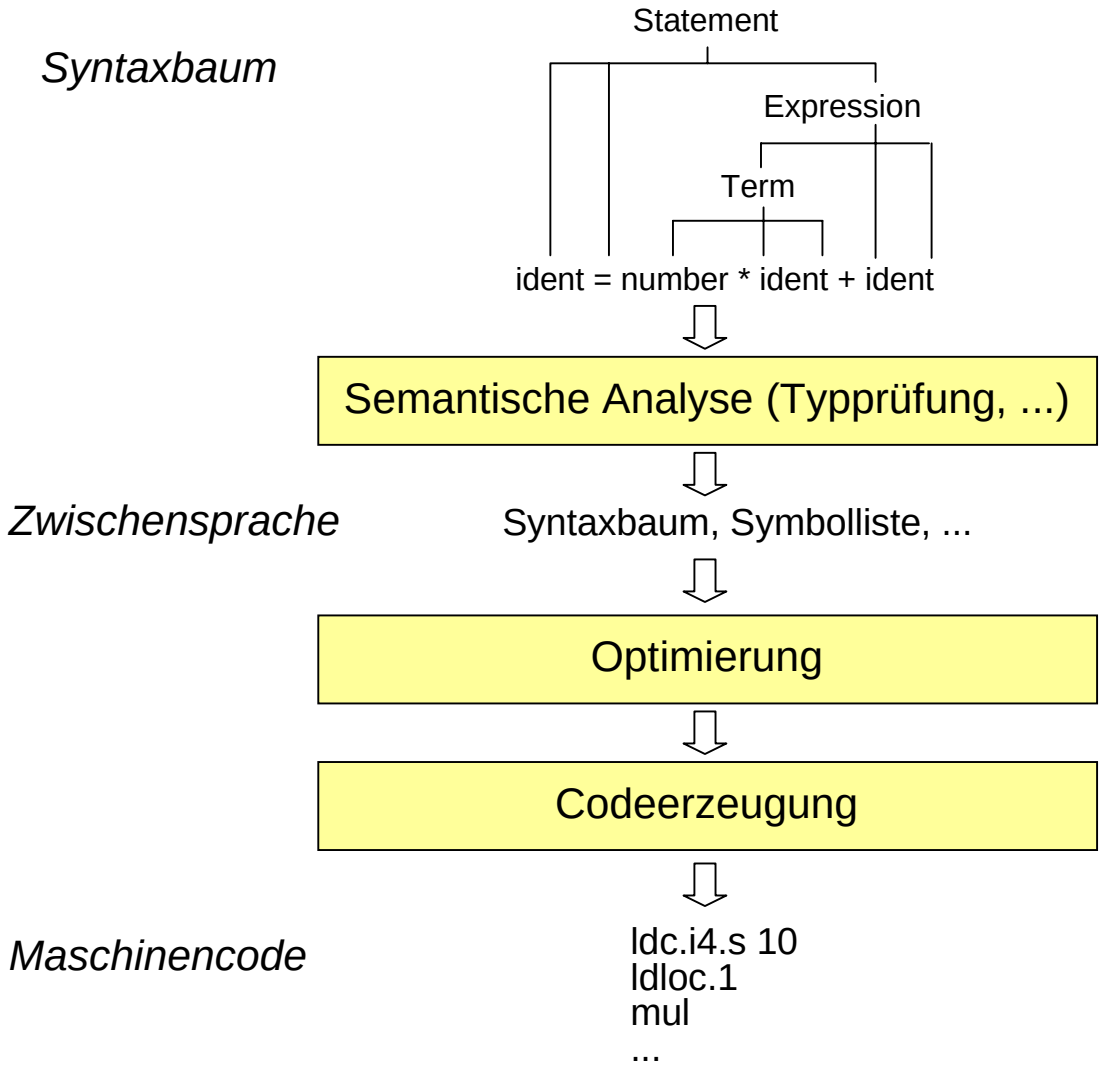
- Lesen syntaktisch strukturierter Kommandoparameter
- Lesen strukturierter Dateien (z.B. XML-Dateien, Stücklisten, Bild-Dateien, ...)
- Suchen in hierarchischen Namensräumen
- Interpretative Abarbeitung von Codes
- ...

Überblick - Struktur eines Compilers

Dynamische Struktur eines Compilers

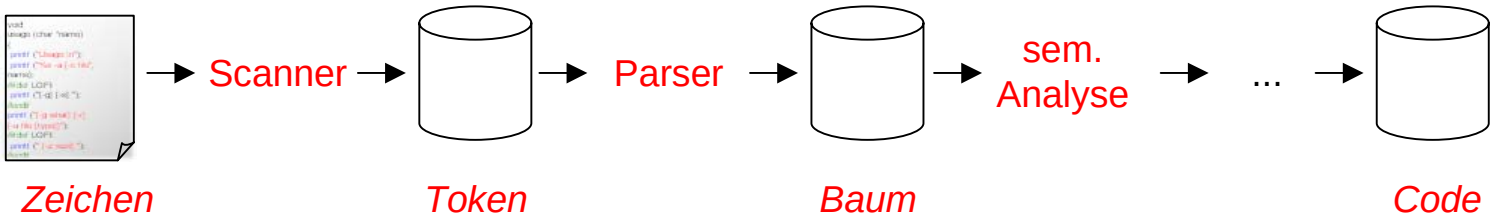


Dynamische Struktur eines Compilers



Mehrpass-Compiler

Phasen sind eigene Programme, die nacheinander ablaufen



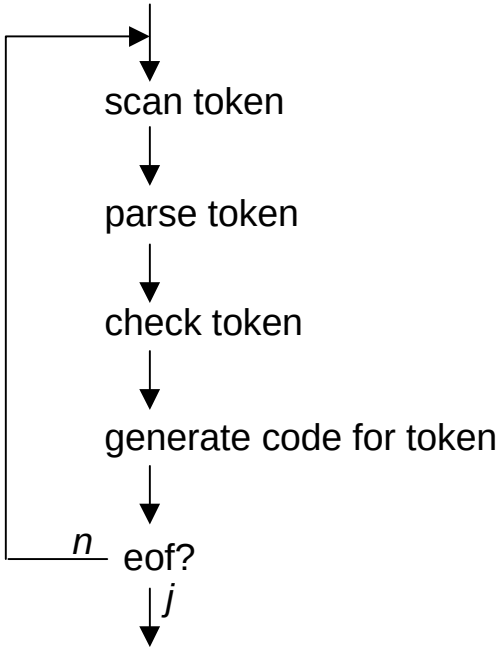
Jede Phase liest von einer Datei und schreibt ihre Ausgabe auf eine neue Datei

Wann ist das notwendig?

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

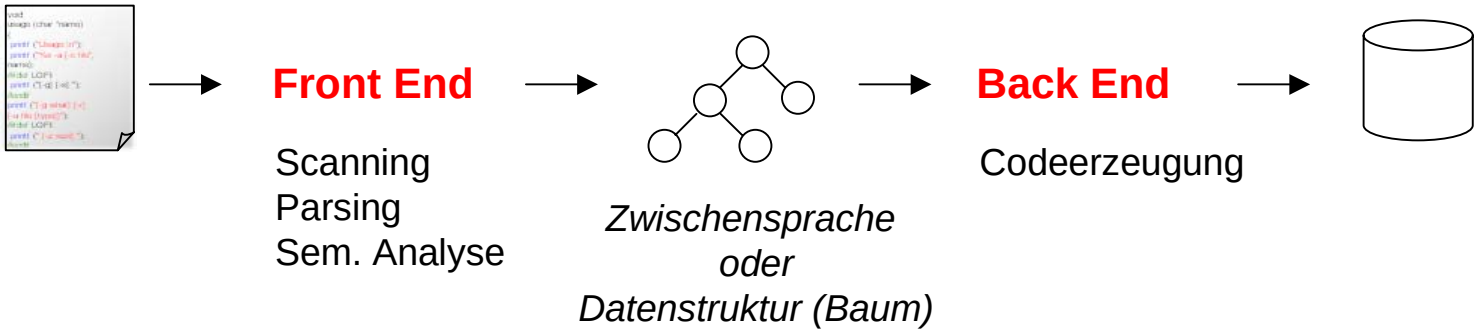
Einpass-Compiler

Die einzelnen Phasen arbeiten verzahnt



Während das Quellprogramm gelesen wird, wird bereits das Zielprogramm (Code) erzeugt.

Heute oft Zweipass-Compiler



sprachabhängig

Java
C#
Pascal

maschinenabhängig

Pentium
PowerPC
SPARC

beliebig kombinierbar

n*m versus n+m

Vorteile

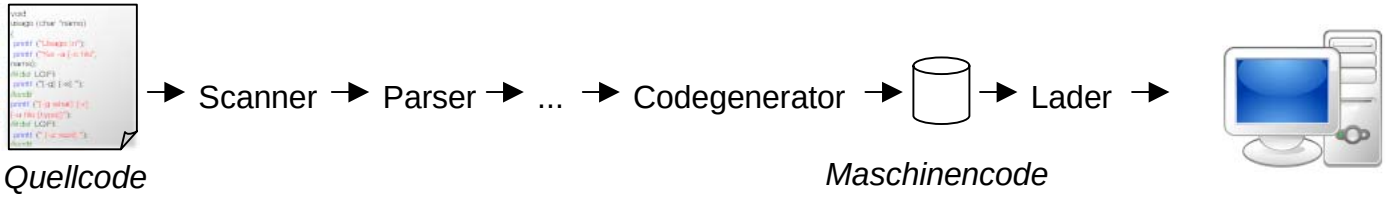
- bessere Portierbarkeit
- Kombination beliebiger Front Ends mit beliebigen Back Ends möglich
- Zwischensprache ist einfacher optimierbar als Quellsprache

Nachteile

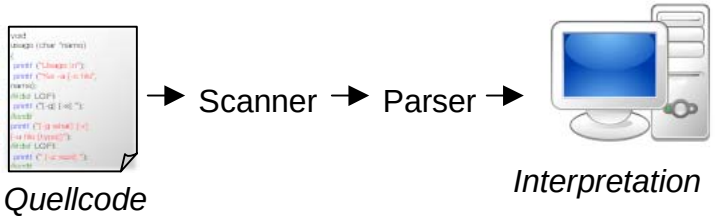
- etwas langsamer
- mehr Speicherverbrauch

Compiler versus Interpreter

Compiler übersetzt in Maschinencode

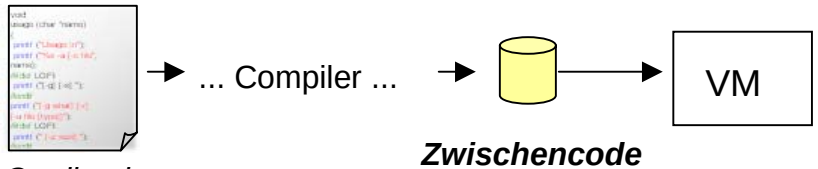


Interpreter führt Quellprogramm "direkt" aus



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

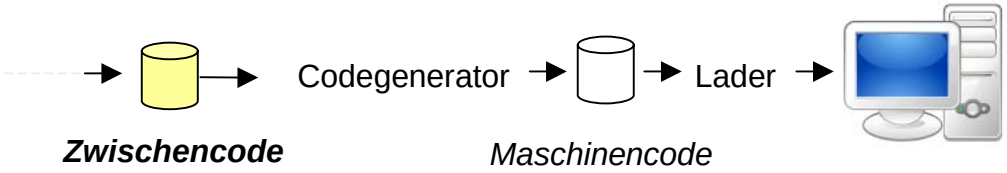
Auch Interpretation von Zwischencode möglich



Zwischencode
(z.B. Common Intermediate Language (CIL, ByteCode))

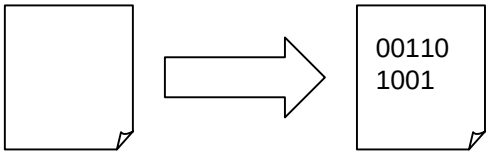
- Quellcode wird in den Code einer *virtuellen Maschine* (VM) übersetzt
- VM interpretiert den Code; simuliert physische Maschine: langsam (~*10)

Just In Time Compilation (JIT)



- Code für virtuelle Maschine wird zur Ladezeit in den Code für physische Maschine übersetzt.
- -> Ausgeführt wird Code für physische Maschine: schnell

all in one

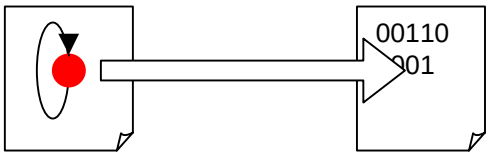


- Gesammter Code wird beim Laden übersetzt. Nachteil: Verzögerung von Programmstart

incremental

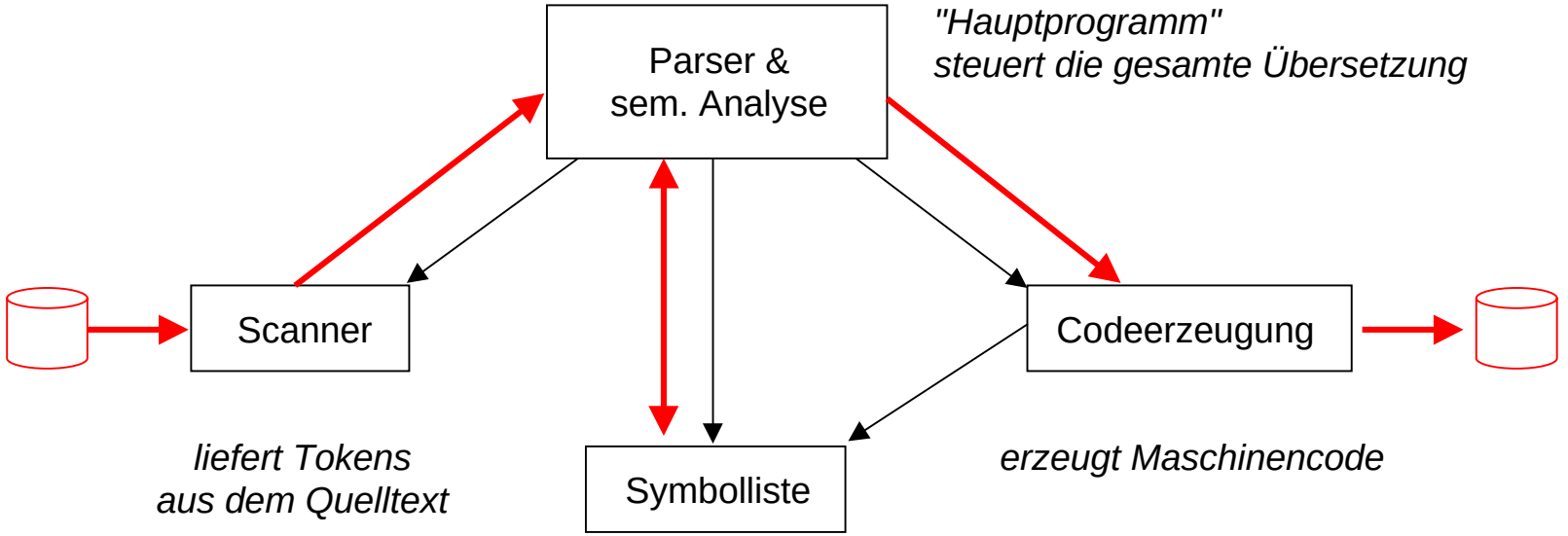
- Code wird vor der ersten Ausführung übersetzt (meist auf Granulariät von Methoden)

hot-spot



- Code wird interpretiert. Die Teile, die besonders häufig durchlaufen werden, werden in Maschinencode übersetzt.
- Programm startet sofort und läuft sich "warm"

Statische Struktur eines Compilers



Aufrufbeziehung

Datenfluss

Überblick - Grammatiken

Woraus besteht eine Grammatik?

Beispiel

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

Vier Bestandteile

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

EBNF-Schreibweise

Extended Backus-Naur Form

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

Symbol	*Bedeutung*	*Beispiele*	
String	bedeutet sich selbst	"=", "while"	
Name	bezeichnet T- oder NT-Symbol	ident, Statement	
=	trennt Regelseiten	A = b c d .	
.	schliesst eine Regel ab		
|	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 | ...

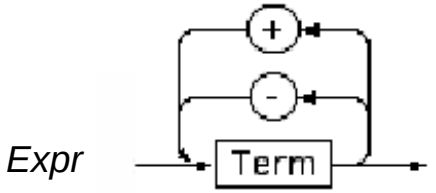
Konvention

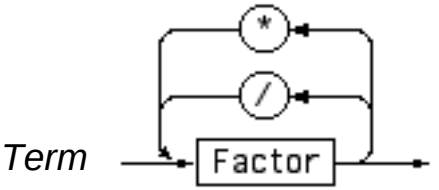
- Terminalsymbole beginnen mit Kleinbuchstaben (z.B. ident)
- Nonterminalsymbole beginnen mit Grossbuchstaben (z.B. Statement)

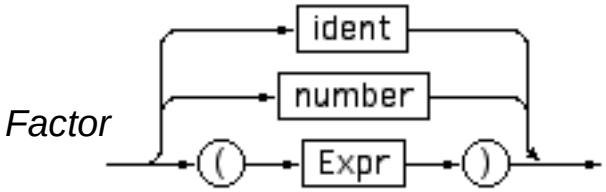
Beispiel: Grammatik der arithmetischen Ausdrücke

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

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, "("

Rekursion

Eine Produktion is rekursiv, wenn $A \rightarrow w_1\ A\ w_2$

Verwendet zur Darstellung von Wiederholungen und Schachtelungen

Direkte Rekursion $A \rightarrow w_1\ A\ w_2$

Linksrekursion	A = b | A a.	A → A a → A a a → A a a a → b a a a a a ...
Rechtsrekursion	A = b | a A.	A → a A → a a A → a a a A → ... a a a a a b
Zentralrekursion	A = b | "(" A ")".	A → (A) → ((A)) → (((A))) → (((... (b)...)))

Indirekte Rekursion $A \rightarrow w_1\ A\ w_2$

Beispiel

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

Expr → Term → Factor → "(" Expr ")"

Beseitigung von Linksrekursion

Linksrekursion stört bei der Topdown-Syntaxanalyse

A = b | A a.

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

Umwandlung von Rekursion in Iteration

E = T | E "+" T.

Überlegen, welche Phrasen erzeugt werden können

T
T + T
T + T + T
...

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

E = T { "+" T }.

Grammatikklassen

Hierarchie von Noam Chomsky (1956)

Grammatiken sind prinzipiell Mengen von Produktionen der Form a = b.

Klasse 0 **Unbeschränkte Grammatiken** (a und b beliebig)

Beispiel: A = a A b | B c B.
aBc = d.
dB = bb.

A aAb → aBcBb → dBb → bbb

Erkennbar durch Turingmaschinen

Klasse 1 **Kontextsensitive Grammatiken**

Beispiel: a A = a b c.

Erkennbar durch linear beschränkte Automaten

Klasse 2 **Kontextfreie Grammatiken**)

Beispiel: A = a A c.

Erkennbar durch Kellerautomaten

Klasse 3 **Reguläre Grammatiken** (a = NT, b = T | T NT)

Beispiel: A = b | b B.

Erkennbar durch endliche Automaten

Im Übersetzerbau sind nur diese beiden Grammatikklassen relevant

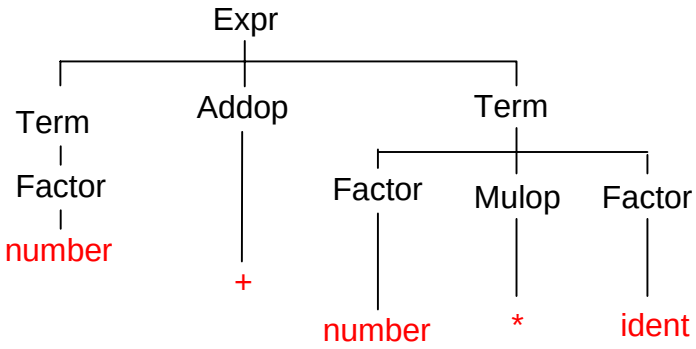
Überblick - Abstrakter Syntaxbaum und Mehrdeutigkeit

Abstrakter Syntaxbaum

Zeigt die Ableitungsstruktur für einen Satz einer Grammatik

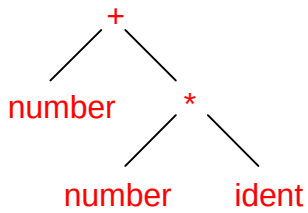
z.B. für 10 + 3 * i

Abstrakter Syntaxbaum (Abstract Syntax Tree)



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

(Expression) Baum (Blätter = Operanden, innere Knoten = Operatoren)



oft als interne Programmdarstellung
für Optimierungen verwendet

Mehrdeutigkeit

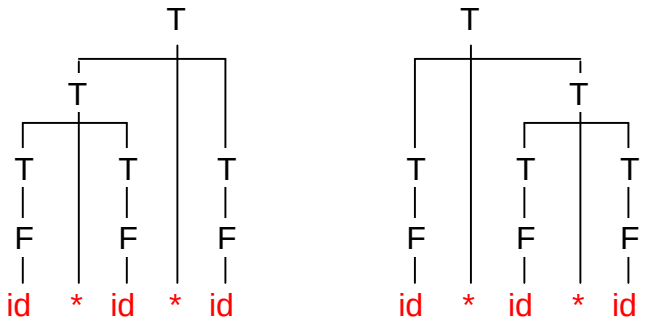
Eine Grammatik ist mehrdeutig, wenn man für einen Satz mehrere Syntaxbäume angeben kann.

Beispiel

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

Beispielsatz: id * id * id

Über diesen Satz können zwei verschiedene Syntaxbäume gebaut werden



Mehrdeutige Grammatiken sind zur Syntaxanalyse ungeeignet!

Beseitigung von Mehrdeutigkeit

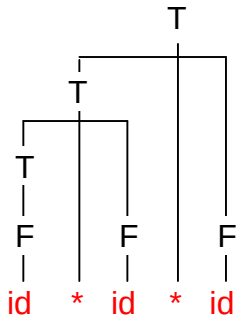
Im Beispiel

```
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. T hat Priorität vor F

nur dieser Syntaxbaum ist möglich

Noch besser: Umformung in EBNF (Entfernung der Rekursion)

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

Aufgabe

- Was ergibt folgende Java Programmsequenz

```
int k = 3; // a = 0, 1, -1
if (a >= 0) if (a==0)  k = 1; else k = 2;
```

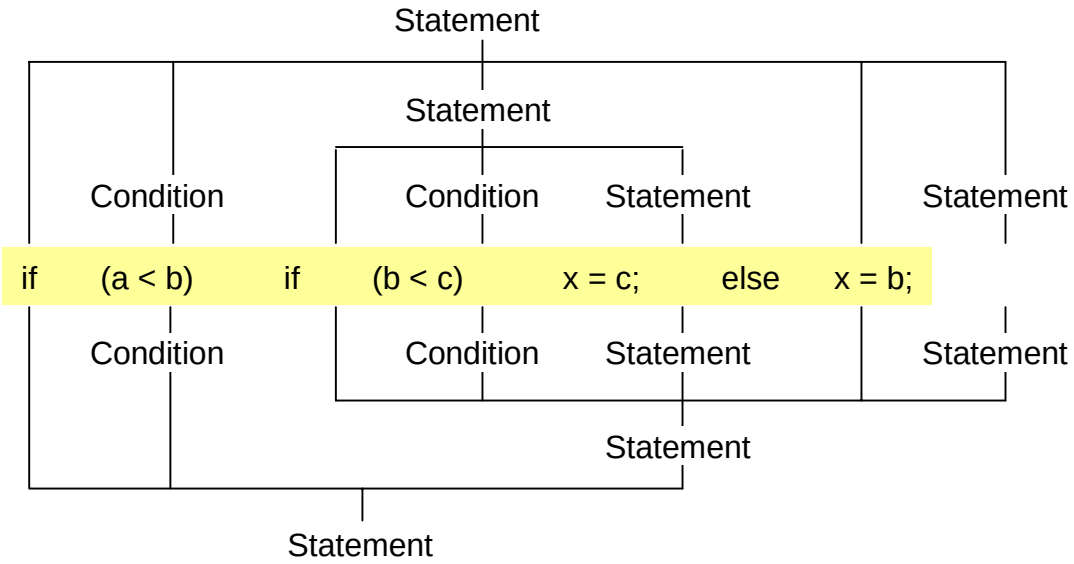
a	k
0	1
1	2
-1	3

Inhärente Mehrdeutigkeit

Es gibt Sprachen, die inhärent mehrdeutig sind

Beispiel: Dangling Else

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



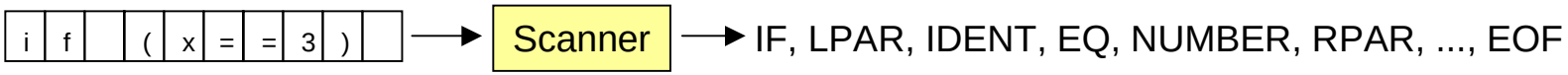
Es lässt sich keine eindeutige Grammatik finden!

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

Lexikalische Analyse - Aufgaben

Aufgaben der Lexikalischen Analyse

1. **Liefert Terminalsymbole** (Tokens)



Zeichenfolge

Tokenfolge
(muss mit EOF enden)

2. **Überliest bedeutungslose Zeichen**
   - Leerzeichen
   - Tabulatoren
   - Zeilenenden (CR, LF)
   - Kommentare

Token haben eine syntaktische Struktur, z.B.

```
ident =     letter { letter | digit }.
number =    digit { digit }.
if =        "i" "f".
eq =        "=" "=".
...
```

Warum ist die Erkennung der Token nicht Teil der Syntaxanalyse?

Warum nicht Teil der Syntaxanalyse?

Würde die Syntaxanalyse verkomplizieren
(z.B. schwierige Unterscheidung zwischen Schlüsselwörtern und Namen)

```
Statement   =  ident "=" Expr ";"
            |  "if" "(" Expr ")" ... .
```

müsste geschrieben werden als

```
Statement   =  "i" (   "f" "(" Expr ")" ...
                   |   not_f { letter | digit } "=" Expr ";"
                   )
            |  not_i { letter | digit } "=" Expr ";".
```

Scanner muss auch Leerzeichen, Tabs, Zeilenenden, Kommentare entfernen
(können überall vorkommen => würde komplizierte Grammatik ergeben)

```
Statement   =  "if" {Blank} "(" {Blank} Expr {Blank} ")" {Blank} ... .
Blank = " " | "\r" | "\n" | "\t" | Comment.
```

Für Token reichen reguläre Grammatiken
(einfacher und schneller analysierbar als kontextfreie Grammatiken)

Lexikalische Analyse - Reguläre Grammatiken und endliche Automaten

Reguläre Grammatiken

Definition
Eine Grammatik heisst regulär, wenn sie sich durch Regeln der folgenden Art ausdrücken lässt:

```
A = a.
A = b B.
```

a, b ∈ TS

A, B ∈ NTS

Beispiel Grammatik für Namen

```
Ident   =  letter
        |  letter Rest.
Rest    =  letter
        |  digit
        |  letter Rest
        |  digit Rest.
```

z.B. Ableitung des Namens xy3

Ident → letter Rest → letter letter Rest → letter letter digit

Andere Definition
Eine Grammatik heisst regulär, wenn sie sich durch eine einzige EBNF-Regel ohne Rekursion ausdrücken lässt.

Beispiel Grammatik für Namen

```
Ident   =  letter { letter | digit }.
```

Beispiele

Lässt sich folgende Grammatik in eine reguläre Grammatik umformen?

```
E = T { "+" T }.
T = F { "*" F }.
F = id.
```

Lässt sich folgende Grammatik in eine reguläre Grammatik umformen?

```
E = F { "*" F }.
F = id | "(" E ")".
```

Beschränkungen regulärer Grammatiken

Reguläre Grammatiken können *keine Klammerstrukturen* ausdrücken.
D.h. keine Zentralrekursion möglich!

Zentralrekursion wird aber in Programmiersprache oft gebraucht.

- geschachtelte Ausdrücke — Expr → ... "(" Expr ")" ...
- geschachtelte Anweisungen — Statement → "do" Statement "while" "(" Expr ")"
- innere Klassen — Class → "class" "{" ... Class ... "}"

Daher benötigt man für die Syntaxanalyse solcher Sprachen kontextfreie Grammatiken.

Lexikalische Strukturen sind meist regulär

Namen	letter { letter | digit }
Zahlen	digit { digit }
Zeichenketten	"\"" { noQuote } "\""
Schlüsselwörter	letter { letter }
Sonderzeichen	">" "="

Ausnahme: geschachtelte Kommentare

/* ..... /* ... */ ..... */

Müssen im Scanner sonderbehandelt werden

Reguläre Ausdrücke

Andere Schreibweise für reguläre Grammatiken

Definition

1. ε (leere Kette) ist ein regulärer Ausdruck
2. Ein Terminalsymbol ist ein regulärer Ausdruck
3. Wenn a und b reguläre Ausdrücke sind, sind auch folgende Ausdrücke regulär

a b	
(a | b)	
(a)?	ε | a
(a)*	ε | a | aa | aaa | ...
(a)+	a | aa | aaa | ...

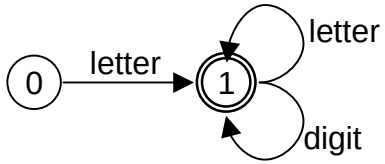
Beispiele

"w" "h" "i" "l" "e"	while
letter (letter | digit)*	Namen
digit+	Zahlen

Endlicher Automat

Automat zur Erkennung einer regulären Sprache
(engl. **DFA** = deterministic finite automaton)

Beispiel



Endzustand

Startzustand per Konvention immer 0

Zustandsübergangsfunktion als Tabelle

d	letter	digit
z0	z1	error
z1	z1	z1

"endlich", weil d explizit angeschrieben werden kann

Definition

Ein deterministischer endlicher Automat ist ein 5-Tupel (Z, S, d, z0, F)

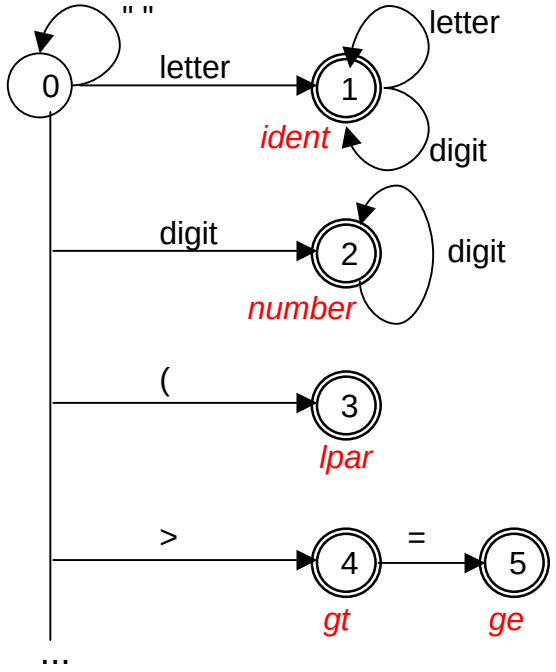
- Z — Menge von Zuständen
- S — Menge von Eingabesymbolen
- d: Z x S — Zustandsübergangsfunktion
- z0 — Anfangszustand
- F — Menge von Endzuständen

Die durch einen DFA erkannte **Sprache** ist die Menge aller Symbolfolgen, die vom Startzustand in einen Endzustand führen

- Ein DFA hat einen Satz erkannt
  - wenn sich der DFA in einem Endzustand befindet
  - ***und*** wenn die Eingabe zu Ende ist ***oder*** kein Übergang mit dem nächsten Symbol möglich ist

Scanner als DFA

Man kann sich den Scanner als grossen DFA vorstellen



Beispiel

Eingabe: *max >= 30*

z0 —m a x→ z1

- kein Übergang mehr mit " " in z1
- *ident* erkannt

z0 —> =→ z5

- überliest Blanks am Anfang
- bleibt nicht in z4 stehen
- kein Übergang mehr mit " " in z5
- *ge* erkannt

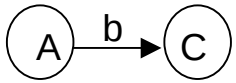
z0 —3 0→ z2

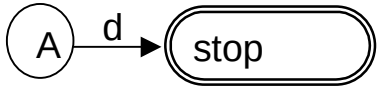
- überliest Blanks am Anfang
- kein Übergang mehr mit " " in z2
- *number* erkannt

Scanner beginnt nach jedem erkannten Symbol wieder in Zustand 0.

Umwandlung reg. Grammatik => DFA

Kann nach folgendem Schema erfolgen

A = b C. → 

A = d. → 

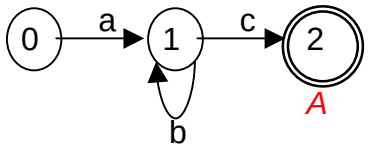
Beispiel

Grammatik

A = a B | b C | c.
B = b B | c.
C = a C | c.

Automat

Implementierung eines DFA (Variante 1)




```
int state = 0;
char ch = read();
 for (;;) {
    switch (state) {
       case 0:    if (ch == 'a') { state = 1; ch = read(); break; }
                  else {state = -1; break;}
       case 1:    if (ch == 'b') { state = 1;  ch = read(); break; }
                  else if (ch == 'c') { state = 2;  ch = read(); break; }
                  else {state = -1; break;}
       case 2: return A;
       default : return errorToken;
    }
```

Implementierung eines DFA (Variante 2)

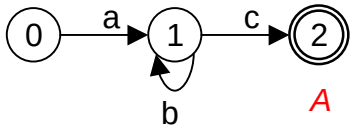
Speicherung von d als Matrix

```
int[,] delta = new int[maxStates, maxSymbols];
int lastState, state = 0;  // DFA starts in state 0
do {
    int sym = next symbol;
    lastState = state;
    state = delta[state][sym];
} while (state != undefined);
assert(lastState ∈ F);  // F is set of final states
return recognizedToken[lastState];
```

Das ist ein Beispiel für einen universellen *tabellengesteuerten Algorithmus*

Beispiel für d

A = a { b } c.



d	a	b	c	
0	1	-	-	
1	-	1	2	
2	-	-	-	E

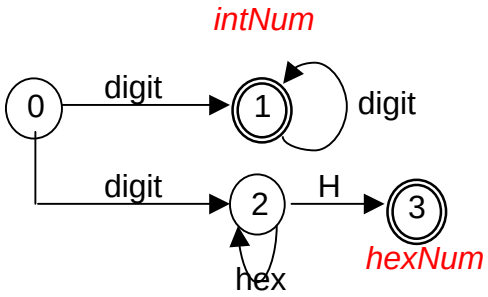
```
int[][] delta = {{1,-1,-1}, {-1,1,2}, {-1,-1,-1}};
```

Diese Implementierung ist allerdings nicht besonders effizient.

Nichtdeterministischer DFA (NDFA)

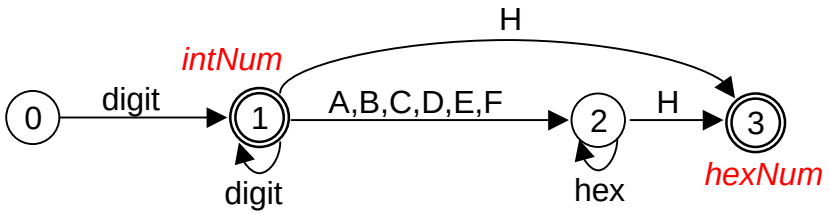
Beispiel

intNum =	digit { digit }.
hexNum =	digit { hex } "H".
digit =	"0" | "1" | ... | "9".
hex =	digit | "A" | ... | "F".



nicht deterministisch,
weil 2 digit-Übergänge
in z0 möglich sind

Jeder NDFA kann in einen äquivalenten DFA umgewandelt werden
(Algorithmus siehe z.B. Aho, Sethi, Ullman: Compilerbau)



Lexikalische Analyse - Scanner-Implementierung

Schnittstelle des Scanners

```
class Scanner {
    static void     init (String s) {...}
    static Token    next () {...}
}
```

Methoden sind aus Effizienzgründen static.
Es gibt nur einen einzigen Scanner pro Compiler.

Initialisierung des Scanners

```
Scanner.init("4 * 3 + 43");
```

Lesen des Tokenstroms

```
Token t;
for (;;) {
    t = Scanner.next();
    ...
}
```

Tokens

```
class Token {
    int kind;        // token code
    int pos;         // token pos (for error messages)
    int val;         // token value (for number and charCon)
    string str;      // token string (for numbers and identifiers)
}
```

Token-Codes

Fehlertoken

final static int

```
NONE = 0,
```

Tokenklassen

```
IDENT = 1,
NUMBER = 2,
CHARCONST = 3,
```

Operatoren und Sonderzeichen

```
PLUS    = 4,   /* + */
MINUS   = 5,   /* - */
TIMES   = 6,   /* * */
SLASH   = 7,   /* / */
REM     = 8,   /* % */
EQ      = 9,   /* == */
GE      = 10,  /* >= */
GT      = 11,  /* > */
LE      = 12,  /* <= */
LT      = 13,  /* < */
NE      = 14,  /* != */
AND     = 15,  /* && */
OR      = 16,  /* || */
ASSIGN  = 17,  /* = */
PPLUS   = 18,  /* ++ */
MMINUS  = 19,  /* -- */
SEMICOLON = 20, /* ; */
COMMA   = 21,  /* , */
PERIOD  = 22,  /* . */
LPAR    = 23,  /* ( */
RPAR    = 24,  /* ) */
LBRACK  = 25,  /* [ */
RBRACK  = 26,  /* ] */
LBRACE  = 27,  /* { */
RBRACE  = 28,  /* } */
```

Schlüsselwörter

```
BREAK  = 29,
CLASS  = 30,
CONST  = 31,
ELSE   = 32,
IF     = 33,
NEW    = 34,
READ   = 35,
RETURN = 36,
VOID   = 37,
WHILE  = 38,
WRITE  = 39,
```

End of file

```
EOF    = 40;
```

Scanner-Implementierung

Statische Variablen im Scanner

im Compiler meist mit Stream Klassen realisiert

```
static String input;        // Eingabestrom
static char ch;             // nächstes noch unverarbeitetes Zeichen
static int pos;             // Position
```

Methode init()

```
public static void init (String r) {
    input = r;
    pos = 0;
    nextCh();  // liest erstes Zeichen nach ch und erhöht pos auf 1
}
```

Methode nextCh()

```
static void nextCh () {
   if (pos < input.length()
       ch = input.charAt(pos++);
   else
        ch = EOF;
}
```

- *ch* = nächstes Eingabezeichen
- liefert bei Ende des Eingabestroms *EOF*
- führt *pos* mit

Methode next()

```
public static Token next () {
   while (ch <= ' ') nextCh();  // skip blanks, tabs, eols
   Token t = new Token(); t.pos = pos;
   switch (ch) {
      case 'a': ... case 'z': case 'A': ... case 'Z': ReadName(t); break;
      case '0': case '1': ... case '9': readNumber(t); break;
      case ';': nextCh(); t.kind = Token.SEMICOLON; break;
      case '.': nextCh(); t.kind = Token.PERIOD; break;
      case EOF: t.kind = Token.EOF; break;  // no nextCh() any more
      ...
      case '=':      nextCh();
                     if (ch == '=') { nextCh(); t.kind = Token.EQ; }
                     else t.kind = Token.ASSIGN;
                     break;
      case '&':      nextCh();
                     if (ch == '&') { nextCh(); t.kind = Token.AND; }
                     else t.kind = NONE;
                     break;
      ...
      case '/':      nextCh();
                     if (ch == '/') {
                        do nextCh(); while (ch != '\n' && ch != EOF);
                        t = Next();  // call scanner recursively
                     } else t.kind = Token.SLASH;
                     break;
      default:       nextCh(); t.kind = Token.NONE; break;
   }
   return t;
} // ch holds the next character that is still unprocessed
```

Namen, Schlüsselwörter

Zahlen

einfache Tokens

zusammengesetzte Tokens

Kommentare

fehlerhaftes Zeichen

Weitere Methoden

`static void readName (Token t)`

- *ch* enthält zu Beginn den ersten Buchstaben des Names
- *readName* liest weitere Buchstaben, Ziffern und '_' und speichert sie in *t.str*
- sucht den Namen in einer Schlüsselworttabelle (Hashing oder binäres Suchen)
  wenn gefunden: t.kind = *Schlüsselwortcode*;
  sonst: t.kind = Token.IDENT;
- *ch* enthält am Ende das erste Zeichen nach dem Namen

`static void readNumber (Token t)`

- *ch* enthält zu Beginn die erste Ziffer der Zahl
- *readNumber* liest weitere Ziffern und speichert sie in *t.str*;
  konvertiert am Ende die Ziffernkette in eine Zahl und speichert diese in *t.val.*
  wenn Überlauf: Fehler melden
- t.kind = Token.NUMBER;
- *ch* enthält am Ende das erste Zeichen nach der Zahl

One Symbol Look Ahead

- Es wird ein Symbol vorausgeschaut (Look ahead)
- reicht für viele Grammatiken aus
- Sonst zwei oder mehr Symbole vorauslesen

```
static int la; // kind des lookahead tokens
static Token token; // zuletzt erkanntes Token
static Token laToken; // lookahead token (noch nicht erkannt

public static void scan() {
    token = laToken;
    laToken = Scanner.next();
    la = laToken.kind;
}
```

- Die Check Methode

```
public static void check (int expected) throws Exception {
  if (la == expected) scan();  // erkannt, daher weiterlesen
  else error(Token.names[expected]+" expected");
}
```

Effizienzüberlegungen

Typische Programmgrösse

ca. 1000 Anweisungen
→ ca. 6000 Symbole
→ ca. 60000 Zeichen

Lexikalische Analyse ist die zeitaufwendigste Phase in einem Compiler
(ca. 20-30% der Übersetzungszeit)

Jedes Zeichen so selten wie möglich umspeichern

deshalb ist *ch* global und kein Parameter von *nextCh()*

Gepuffert lesen

```
StringBuffer b = new StringBuffer();
BufferedReader buf = new BufferedReader(new FileInputStream("MyProg.zs"));
String line;
while ((line = buf.readLine()) != null) buf.append(line+'\n');
Scanner.Init(buf.toString());
```

ist aber nur bei grossen Eingabedateien spürbar

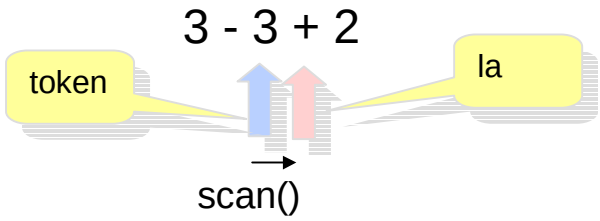
Parser Einführung

Der Parser

- Beliebige einfache mathematischer Ausdrücke
  - 3 + 3 oder 3 + 2 - 3 oder 3 - 3 + 2
- obige mathematische Ausdrücke werden durch folgende Grammatik beschrieben:
  - Expression = Term {("+" | "-") Term}
  - Term = Number
- für jede Produktion wird nun eine entsprechende Methode geschrieben:

```
static void expr() throws Exception {
 term();
 while (Scanner.la == Token.PLUS
       || Scanner.la == Token.MINUS) {
   Scanner.scan();
   int op = Scanner.token.kind;
   term();
}

 static void term() throws Exception {
  if Scanner.la == Token.NUMBER) {
   Scanner.scan();
 }
```



NUMBER "-" NUMBER "+" NUMBER

3 3 2

Die vollständige Grammatik

- Beispiele mathematischer Ausdrücke
  - 3 + 3 + 3 oder 3 + 3*2 + 2 oder 3 * (2 + 3)
- Grammatik dazu (nicht mehr regulär, aber für Parser kein Problem)
  - Expression = Term {("+" | "-") Term}
  - Term = Factor {("*" | "/") Factor}
  - Factor = Zahl | "(" Expression ")"

Nicht mehr regulär wegen Rekursion

```
static void expr() throws Exception {
  term();
  while (Scanner.la == Token.PLUS
         || Scanner.la == Token.MINUS) {
    Scanner.scan();
    int op = Scanner.token.kind;
    term();
}

static void term() throws Exception {
  factor();
  while (Scanner.la == Token.TIMES
|| Scanner.la == Token.SLASH) {
    Scanner.scan();
    int op = Scanner.token.kind;
    factor();
  }
}
```

```
static void factor() throws Exception {
  if (Scanner.la == Token.LBRACK) {
    Scanner.scan();
    expr();
    Scanner.check(Token.RBRACK);
  } else if (Scanner.la == Token.NUMBER) {
    Scanner.scan();
  }
}


public static void main(String[] s) {
 Scanner.init("3+2-4");
 Scanner.scan();
 expr();
}
```

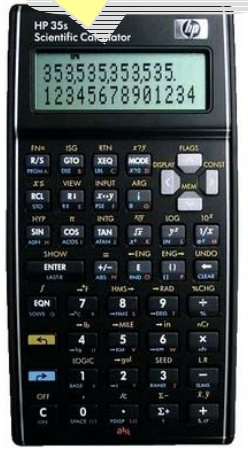
Erster Taschenrechner von HP

Die Auswertung eines Ausdrucks in UPN-Notation

- **läuft folgendermassen ab:**
- Zahlen werden auf den Stack gelegt
- Operatoren werden sofort ausgewertet, dazu werden 2 Elemente vom Stack geholt
- Das Resultat wird wieder auf den Stack gelegt
- Am Ende enthält der Stack das Schlussresultat
- Der Ausdruck: 6* (5 + (2 + 3) * 8 + 3)
  wird wie folgt ausgewertet

HP 35 1972

HP 35s 2007



Ausdruck	Aktion	Stack
6 5 2 3 + 8 * + 3 + *	push von 6,5,2 und 3	3 2 5 6
6 5 2 3 **+** 8 * + 3 + *	+ auswerten, 3 + 2 ergibt 5	5 5 6
6 5 2 3 + ***8*** * + 3 + *	push von 8	8 5 5 6
6 5 2 3 + 8 ******* + 3 + *	* auswerten	40 5 6
6 5 2 3 + 8 * **+** 3 + *	+ auswerten	45 6
6 5 2 3 + 8 * + *3* + *	push von 3	3 45 6
6 5 2 3 + 8 * + 3 **+** *	+ auswerten	48 6
6 5 2 3 + 8 * + 3 + *******	* auswerten	288

Berechnung des Ausdruckes - Stack Variante

Logik kann direkt in Parser hineinkodiert werden

```
static int stack[] = new int[10];
static int sp = 0;

static void push(int val) {
  stack[sp++]=val;
}

static int pop() {
  return stack[--sp];
}
```

+ einfach, effizient

+ Stackrechner sehr einfach

- Vermischung von Syntax und Semantik

bei Compiler: statt auszuwerten wird Code erzeugt

```
void factor()  throws Exception{
   if (Scanner.la  == Token.LBRACK) {
      Scanner.scan();
      expr();
      Scanner.check(Token.RBRACK);
   }
   else if (Scanner.la == Token.NUMBER) {
      Scanner.scan();
      push(Scanner.token.val);;
    }
   else Scanner.error("illegal Symbol");
 }
```

```
static void expr() throws Exception {
 term();
 while (Scanner.la == Token.PLUS
        || Scanner.la == Token.MINUS) {
  Scanner.scan();
  int op = Scanner.token.kind;
  term();
  if (op == Token.PLUS) {
        push(pop()+pop());
  }
  else {
        push(-pop()+pop());
  }
 }
}
```

Zusammenfassung

- Übersetzungsvorgang
- Sprachen, Automaten, Übergangstabellen
- Parser
- Arithmetische Ausdrücke

Noch Fragen ?