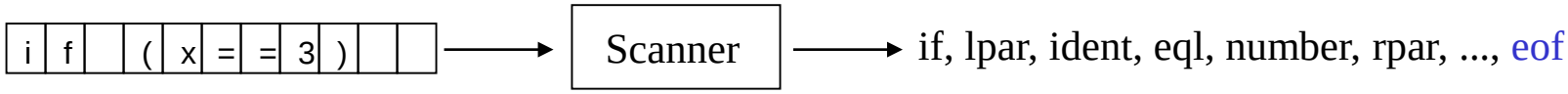
2. Lexikalische Analyse

© 2023. This work is licensed under a CC BY-NC-SA 4.0 license
Hanspeter Mössenböck, Institut für Systemsoftware, JKU

Aufgaben der Lexikalischen Analyse

1. **Liefert Terminalsymbole** (Tokens)



Zeichenfolge

Tokenfolge
(muss mit eof enden)

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

Warum nicht Teil der Syntaxanalyse?

Token haben eine syntaktische Struktur, z.B.

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

Warum ist die Erkennung der Token nicht Teil der Syntaxanalyse?

Warum ist z.B. *ident* ein Terminalsymbol und kein Nonterminalsymbol?

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

Man müsste auch Leerzeichen, Tabs, etc. in der Grammatik berücksichtigen
(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)

2. Lexikalische Analyse

Reguläre Grammatiken

Definition

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

```
X = a.
X = b Y.
```

a, b ∈ TS
X, Y ∈ 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.

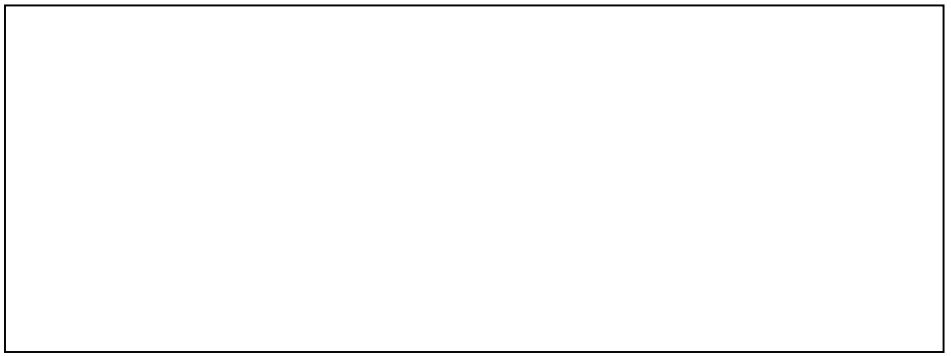
Beispiel Grammatik für Namen

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

Beispiele

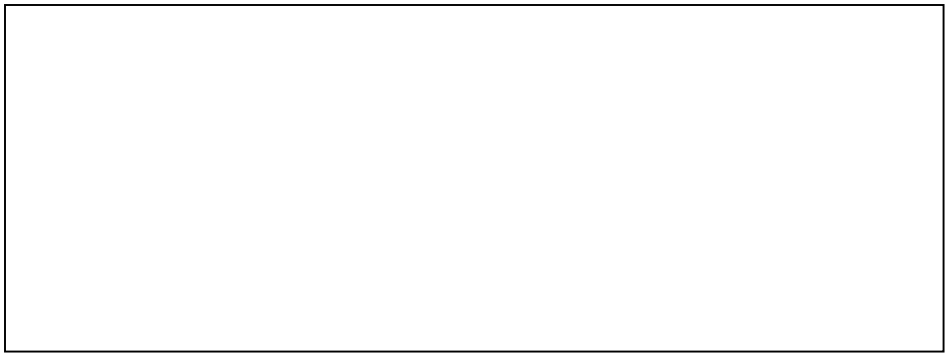
Lässt sich folgende Grammatik in eine reguläre Grammatik umformen?

```
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

- geklammerte Ausdrücke — Expr $\Rightarrow^*$ ... "(" Expr ")" ...
- geschachtelte Anweisungen — Statement $\Rightarrow$ "do" Statement "while" "(" Expr ")"
- innere Klassen — Class $\Rightarrow$ "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}
Operatoren	">" "="

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 α und β reguläre Ausdrücke sind, sind auch folgende Ausdrücke regulär

		EBNF
α β		
α | β		
(α)		
(α)?	α | ε	[α]
(α)+	α | αα | ααα | ...	α{α}
(α)*	ε | α | αα | ααα | ...	{α}

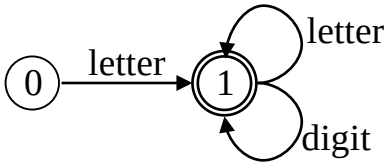
Beispiele

"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

δ	letter	digit
z0	z1	error
z1	z1	z1

"endlich", weil δ explizit angeschrieben werden kann

Definition

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

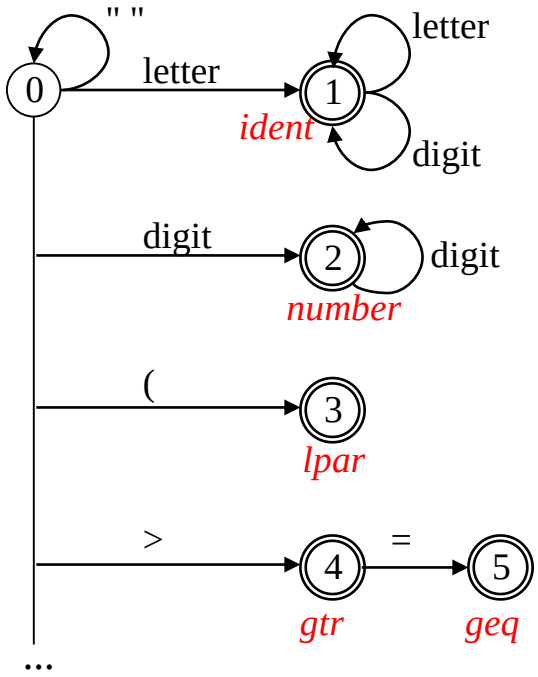
- Z — Menge von Zuständen
- S — Menge von Eingabesymbolen
- $\delta: Z \times S \rightarrow Z$ — Zustandsübergangsfunktion
- z0 — Anfangszustand
- F — Menge von Endzuständen

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

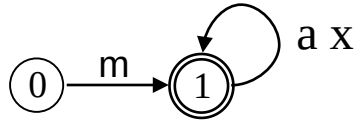
Scanner als DFA

Man kann sich den Scanner als großen DFA vorstellen

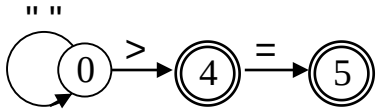


Beispiel

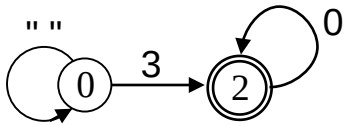
Eingabe: max >= 30



- kein Übergang mehr mit " " in (1)
- *ident* erkannt



- überliest Blanks am Anfang
- bleibt nicht in (4) stehen
- kein Übergang mehr mit " " in (5)
- *geq* erkannt

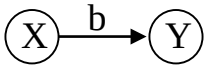


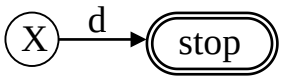
- überliest Blanks am Anfang
- kein Übergang mehr mit " " in (2)
- *number* erkannt

Scanner beginnt nach jedem erkannten Symbol wieder in (0)

Umwandlung reg. Grammatik ↔ DFA

Kann nach folgendem Schema erfolgen

X = b Y. ⇔ 

X = d. ⇔ 

Beispiel

Grammatik

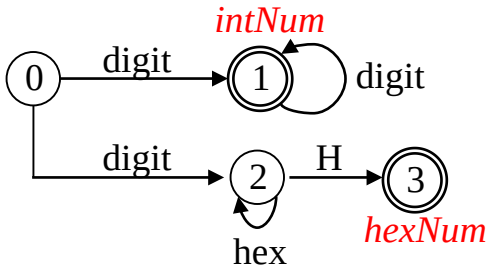
X = a Y | b Z | c.
Y = b Y | c.
Z = a Z | c.

Automat

Nichtdeterministischer Automat (NDFA)

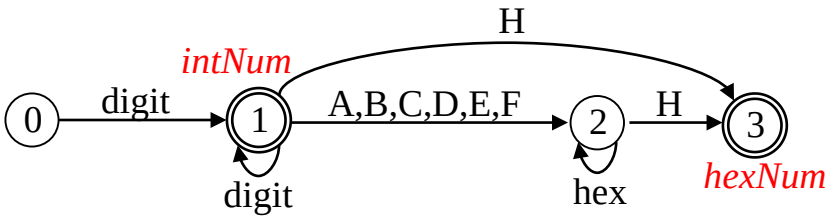
Beispiel

```
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)



Implementierung eines DFA (Variante 1)

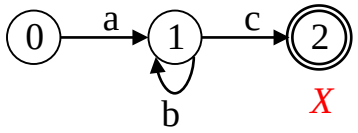
Speicherung von δ als Matrix

```
int[][] delta = new int[maxStates][maxSymbols];
int state = 0, lastState;   // DFA starts in state 0
do {
    int sym = next char;
    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 δ

X = a {b} c.

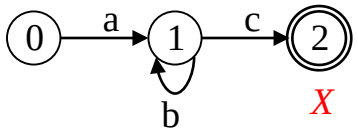


δ	a	b	c	
0	1	-	-	
1	-	1	2	
2	-	-	-	E

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

Diese Implementierung ist allerdings nicht besonders effizient

Implementierung eines DFA (Variante 2)



Auscodieren der Zustände

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

In C# geht es einfacher und effizienter

```
char ch = read();
s0:  if (ch == 'a') { ch = read(); goto s1; }
     else goto err;
s1:  if (ch == 'b') { ch = read(); goto s1; }
     else if (ch == 'c') { ch = read(); goto s2; }
     else goto err;
s2:  return X;
err: return errorToken;
```

2. Lexikalische Analyse

Schnittstelle des Scanners


```
class Scanner {
   static void    init (Reader r) {...}
   static Token   next () {...}
}
```

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

Initialisierung des Scanners (im Hauptprogramm)

```
InputStream s = new FileInputStream("myfile.mj");
Reader r = new InputStreamReader(s);
Scanner.init(r);
```

Lesen des Tokenstroms (im Parser)

```
do {
   Token t = Scanner.next();
   ...
} while (t != endOfFileToken);
```

Tokens

```
class Token {
   int kind;       // token code
   String val;     // token value
   int numVal;     // numeric token value (for number and charCon)
   int line;       // token line (for error messages)
   int col;        // token column (for error messages)
}
```

Token-Codes für MicroJava

Fehlertoken

static final int

```
none = 0,
```

Tokenklassen

```
ident = 1,
number = 2,
charCon = 3,
```

Operatoren und Sonderzeichen

```
plus     = 4,   /* + */
minus    = 5,   /* - */
times    = 6,   /* * */
slash    = 7,   /* / */
rem      = 8,   /* % */
eql      = 9,   /* == */
neq      = 10,  /* != */
lss      = 11,  /* < */
leq      = 12,  /* <= */
gtr      = 13,  /* > */
geq      = 14,  /* >= */
and      = 15,  /* && */
or       = 16,  /* || */
assign   = 17,  /* = */
plusas   = 18,  /* += */
minusas  = 19,  /* -= */
timesas  = 20,  /* *= */
slashas  = 21,  /* /= */
remas    = 22,  /* %= */
pplus    = 23,  /* ++ */
mminus   = 24,  /* -- */
semicolon = 25, /* ; */
comma    = 26,  /* , */
period   = 27,  /* . */
lpar     = 28,  /* ( */
rpar     = 29,  /* ) */
lbrack   = 30,  /* [ */
rbrack   = 31,  /* ] */
lbrace   = 32,  /* { */
rbrace   = 33,  /* } */
```

Schlüsselwörter

```
break_   = 34,
class_   = 35,
else_    = 36,
final_   = 37,
if_      = 38,
new_     = 39,
print_   = 40,
program_ = 41,
read_    = 42,
return_  = 43,
void_    = 44,
while_   = 45,
```

End of file

```
eof      = 46;
```

Scanner-Implementierung

Statische Variablen im Scanner

```
static Reader in;                       // Eingabestrom
static char ch;                         // nächstes noch unverarbeitetes Zeichen
static int line, col;                   // Zeile und Spalte des Zeichens ch
static final char eofCh = '\uffff';     // Zeichen, das bei eof geliefert wird
```

Methode init()

```
public static void init (Reader r) {
   in = r;
   line = 1; col = 0;
   nextCh();  // liest erstes Zeichen, speichert es in ch und erhöht col auf 1
}
```

Methode nextCh()

```
private static void nextCh() {
   try {
      ch = (char) in.read(); col++;
      if (ch == '\n') { line++; col = 0; }
   } catch (IOException e) { ch = eofCh; }
}
```

- *ch* = nächstes Eingabezeichen
- führt *line* und *col* mit

Methode next()

```
public static Token next() {
   while (ch <= ' ') nextCh();  // skip blanks, tabs, eols
   Token t = new Token(); t.line = line; t.col = col;
   switch (ch) {
      case 'a': case 'b': ... case 'z': case 'A': case 'B': ... case 'Z':
                readName(t); break;
      case '0': case '1': ... case '9':
                readNumber(t); break;
      case ';':   nextCh(); t.kind = semicolon; break;
      case '.':   nextCh(); t.kind = period; break;
      case eofCh: t.kind = eof; break;  // no nextCh() any more
      ...
      case '=':   nextCh();
                  if (ch == '=') { nextCh(); t.kind = eql; } else t.kind = assign;
                  break;
      case '&':   nextCh();
                  if (ch == '&') { nextCh(); t.kind = and; } else t.kind = none;
                  break;
      ...
      case '/':   nextCh();
                  if (ch == '/') {
                     do nextCh(); while (ch != '\n' && ch != eofCh);
                     t = next();  // call scanner recursively
                  } else t.kind = slash;
                  break;
      default:    nextCh(); t.kind = 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

private static void readName(Token t)

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

private static void readNumber(Token t)

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

Weitere Methoden

private static void readCharCon(Token t)

- *ch* enthält zu Beginn ein einfaches Hochkomma
- *readCharCon* liest weitere Zeichen bis zum schließenden Hochkomma und speichert sie in *t.val*
- *ch* enthält am Ende das erste Zeichen nach dem schließenden Hochkomma
- Stellt folgende Werte ein:
  t.kind = charCon;
  t.numVal = *numerischer Zeichenwert*;

gültige Zeichenkonstanten

'x'
'\r'
'\n'
'\\'

ungültige Zeichenkonstanten

'xy'
''
'x

Scanner meldet Fehler,
liefert aber *charCon*

Effizienzüberlegungen

Typische Programmgröße

ca. 1000 Anweisungen
⇒ ca. 6000 Token
⇒ ca. 60000 Zeichen

Lexikalische Analyse ist die zeitaufwändigste 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()*

Eventuell gepuffert lesen

```
InputStream s = new FileInputStream("myfile.mj");
Reader r = new InputStreamReader(s);
r = new BufferedReader(r);
Scanner.init(r);
```

ist aber nur bei großen Eingabedateien spürbar