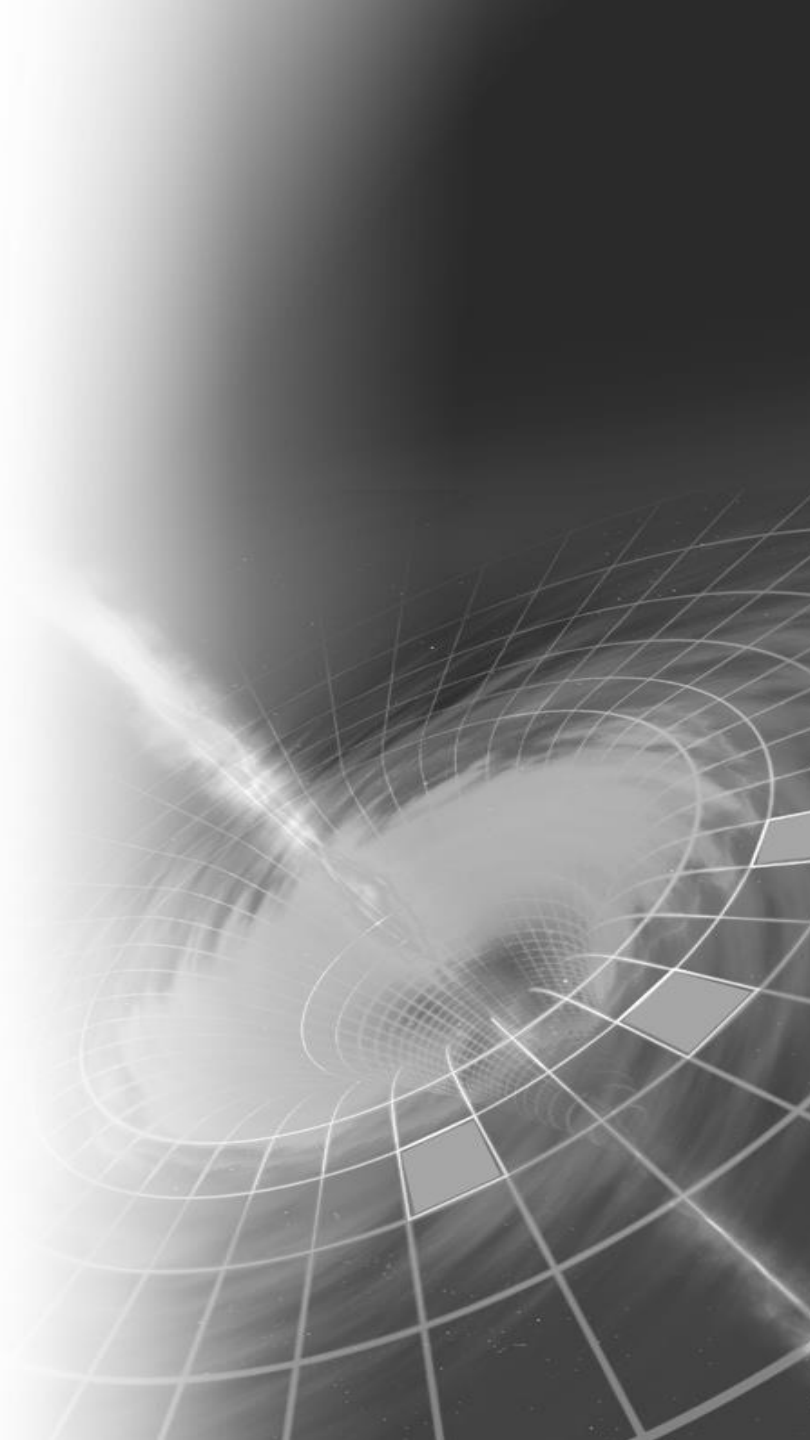
Bachelor-Programm

Compilerbau

im SoSe 2013

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

fischer@informatik.hu-berlin.de



3. Lexikalische Analyse: der Scanner

- Aufgaben eines Scanners als Compilerkomponente (Ein Ad-hoc-Scanner)
- Reguläre Ausdrücke
- Endliche Automaten zur Erkennung regulärer Ausdrücke
- Tabellengesteuerter Scanner und Symboltabelle
- Konzepte für einen generischen Scanner
- Scanner-Generatoren: Vorgehensweise, Implementation (lex, flex)

Implementierungsvarianten eines Scanners

… unter Nutzung

1. einer konventionellen Programmiersprache
   bei Einsatz von Standard-Bibliotheken zur E/A

2. einer Assembler-Sprache
   bei eigener Organisation der Eingabe

3. eines Scanner-Generators
   bei Spezifikation von RAs und
   Bereitstellung von Routinen zum Lesen und zur Pufferung der Eingabe

Generatoren für die lexikalische Analyse

- … werden eingesetzt um die lexikalische Analyse eines Compilers möglichst kompakt und einfach zu spezifizieren
  Eingabesprache: ist eine Domain Specific Language (DSL)

- Eingabe: Ein Zeichenstrom (ASCII, UTF-8, ...)

- mehrere Möglichkeiten zur Anbindung eines Parsers:
  - Push-Schnittstelle:
    bei erkanntem Token wird eine Methode des Parsers aufgerufen
  - Pull-Schnittstelle:
    der Parser ruft eine Methode im Lexer auf,
    um das nächste Token zu bekommen.

  (meist am einfachsten zu programmieren)

- typischerweise erfolgt keine automatische Unterstützung für die Behandlung von Symboltabellen und Fehlern (da oft sehr sprachspezifisch)

Vor- und Nachteile von Generatoren

Vorteile

- Kompaktheit der Spezifikation von Scannern
- *bedeutend:* verständlich und gut wartbarer Code, damit weniger fehleranfällig
- automatische Konsistenzprüfung
- kurze Entwicklungszyklen, schnelles Prototyping möglich
- Scanner-Generatoren erzeugen sehr performanten Code

Nachteile

- neue Sprache/Werkzeug muss erlernt werden
- Integration in Build-System notwendig
- Fehlersuche oft sehr mühsam da generierter Code schlecht menschenlesbar
- fehlende Flexibilität

  in der Praxis gibt es häufig kleinere Ausnahmen die man nur umständlich oder gar nicht mit Generatoren umsetzen kann

Verbreitete Generatoren

- ***lex***: Unix-Werkzeug zur Generierung von Scannern
  - akzeptiert eine Menge von Mustern,
    die erweiterte reguläre Ausdrücke und Definitionen darstellen
  - Hieraus produziert lex ein durch Tabellen gesteuertes **C**-Programm, das in der Lage ist, Zeichenfolgen zu erkennen, die den Mustern entsprechen
  - Zusätzlich können Aktionen angegeben werden, die bei Erkennung eines Musters ausgeführt werden sollen

- ***flex*** ist eine Open-Source Implementierung von *lex*
  - zahlreiche Portierungen auf andere Programmiersprachen
  - Beispiele: JLex (Java), CsLex (C#), Alex (Haskell), . . .

- Es gibt auch Scanner, die **nicht** mit regulären Ausdrücken/endlichen Automaten arbeiten

  Vertreter ist **antlr**
  hier kommen LL(k)-Grammatiken zum Einsatz

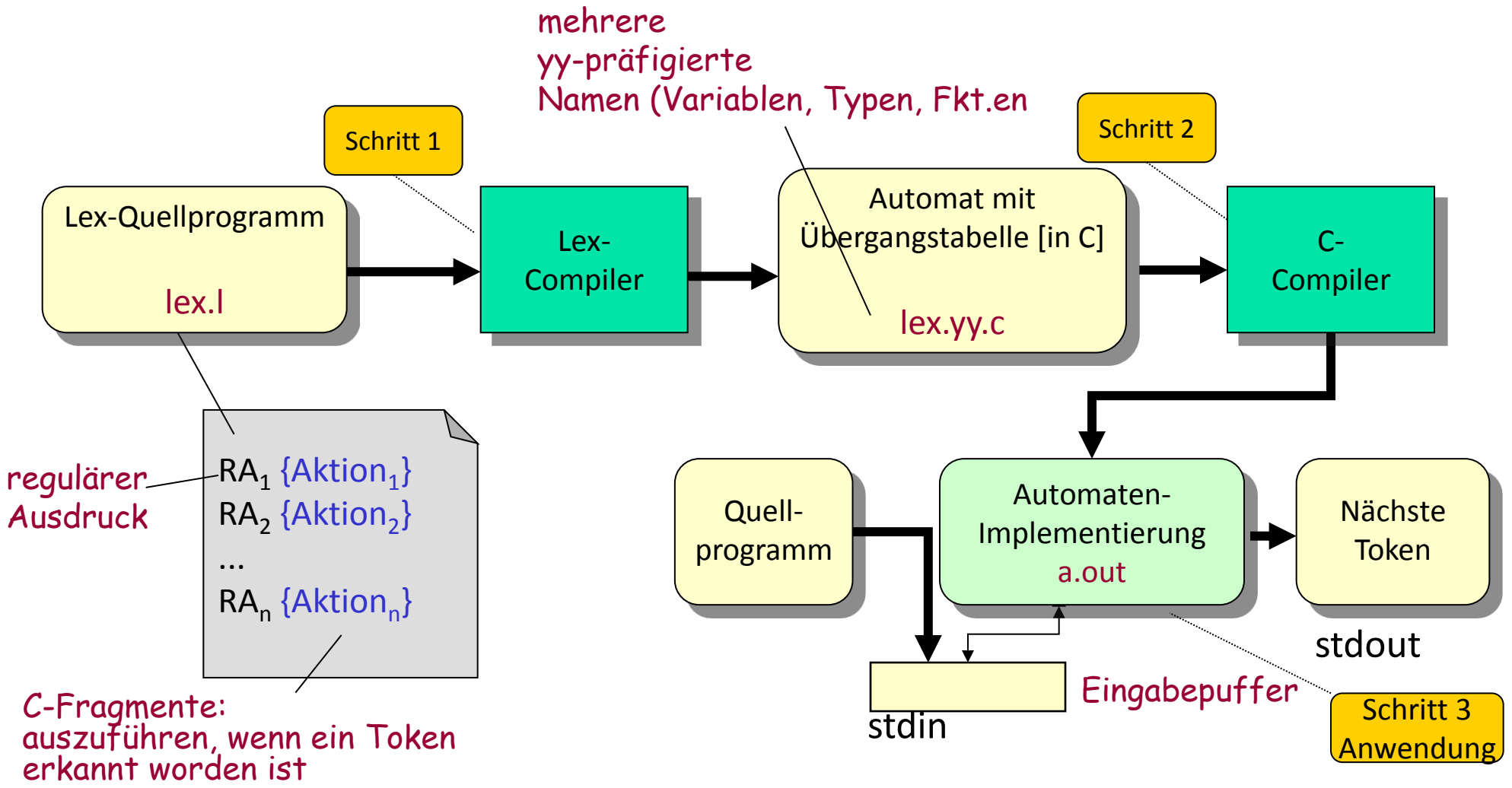
Flex (Werkzeug im Praktikum)

- … ist ein schneller Scanner-Generator
  (d.h., der generierte Scanner ist schneller als der von lex)

- es gibt einen Modus, der Scanner als C++ -Programm liefert

- weitere Informationen
  im Unix: man flex

Funktionsweise des Scanner-Generators lex



Allgemeine Struktur eines Lex-Quellprogramms

drei Teile:

<Deklarationen>
%%
<Übersetzungsregeln>
%%
<Hilfsfunktionen>

C-Fragment
%{
- Deklaration von Variablen und
  symbolischer Konstanten
  (auch per Einfügung von C-Header-Dateien)
- reguläre Definitionen
%}

kann benutzen

<RegulärerAusdruck$_1$> { Aktion$_1$}
<RegulärerAusdruck$_2$> { Aktion$_2$}
…
<RegulärerAusdruck$_n$> { Aktion$_n$}

können alternativ separat kompiliert werden

in Aktionen benötigte Funktionen
(evtl. auch die main()-Funktion, die den lexer als Funktion aufruft)

wird Lexer mit Parser verzahnt,
wird der Parser die main-Funktion bereitstellen

… oder separates Lex-Anwendungsprogramm,
stellt main-Funktion bereit

Beispiel: Ausgabe aller Bezeichner mit Zeilennummer

<Deklarationen>
%%
<Übersetzungsregeln>
%%
<Hilfsfunktionen>

```
%option noyywrap
letter          [_A-Za-z]
digit           0-9]
letterOrDigit   [_A-Za-z0-9]
whitespace      [ \t\n]
other           .
%%
{letter}{letterOrDigit}*    printf( "%d: %s\n", yylineno, yytext );
{whitespace}                ;
{other}                     ;
```

zur Verfügung stehende Variablen

- **extern char** yytext[]
  Zeichenfolge,
  die auf das erkannte Muster passt
- **extern int** yyleng
  Länge von yytext
- **extern int** yylineno;
  Nummer der aktuellen Eingabezeile
- … und weitere

angegebene Muster können (wie hier) mehrdeutig sein

Lex benutzt zur Auflösung bekannte Regeln

Lexer-Verhaltensweisen (1)

- jeweils verbliebende Eingabe wird zeilenweise eingelesen
  bis das **längste Präfix** gefunden ist, das mit dem Muster RA_i übereinstimmt
- repräsentieren dabei immer noch zwei Muster die gleiche Eingabe,
  so wird das erste Muster innerhalb der Übersetzungsregeln verwendet.

- dann erfolgt die Ausführung der Aktion A_i
  - i. allg. gibt eine Aktion die Steuerung an den Parser zurück
  - wenn nicht (z.B. bei Leerzeichen-Bearbeitung) sucht Lexer nach weiteren Token

häufig:
- int
- float
- char*

Tokenkennung ist zugleich der Diskriminator für den union-Typ von yylval

- der Lexer liefert
  - per Return die Tokenkennung
  - zusätzlich werden über die globale YYSTYPE-Variable yylval weitere Informationen über das gefundene Lexem geliefert

≙ tokenval (Hand-Scanner)

Vordefinierte Bezeichner von Lex (Auswahl)

Lex- Bezeichner	Bedeutung
int yylex (**void**)	Ruf des Lexers, liefert Token zurück
int yyerror (**void**)	Nutzercode, der im Fehlerfall ausgeführt wird
YYSTYPE yylval	universeller Wert einer numerischen Lexems (Wert eines union-Typs)
char *yytext	Zeiger zum String, das dem RA entspricht
int yyleng	Länge des Strings, das dem RA entspricht
int yywrap (**void**)	Lex-Anwendung auf mehrere Eingabe-Files (1 falls fertig, 0 sonst)
FILE *yyout	Ausgabefile (default: stdout)
FILE *yyin	Inputfile (default: stdin)
INITIAL	initiale Startbedingung
BEGIN condition	Einschalten der Startbedingung
ECHO	Aktionsmakro: Ausgabe des akzeptierten Strings

Notwendige Angabe im Deklarationsteil:

```
%option noyywrap
/* falls nur ein Lex-File als Eingabe */
```

Zeichen mit Sonderrolle

- Alphabet Σ ist der Zeichensatz des Systems

- einige Zeichen spielen eine Sonderrolle bei der Spezifikation regulärer Ausdrücke, die sog. Metazeichen:

  \ ^ $. [ ] | ( ) * + ? { } ” % < > /

- Behandlung der Metazeichen als normale Zeichen durch
  - Voranstellung des Zeichens \ oder
  - Einbindung in ” ”

Beispiel: *Zählung von Zeichen, Wörtern und Zeilen einer Datei*

```
%{
int nchar, nword, nline;
%}
%option noyywrap
%%
\n                  { ++nchar; ++nline; }
[^ \t\n]+           { ++nword; nchar += yyleng; }
.                   { ++nchar; }
%%
int main() {
        yylex();
        printf( "%d %d %d\n", nchar, nword, nline );
}
```

[^ \t\n]
jedes beliebige Zeichen, das verschieden ist von
- Leerzeichen
- Tabulator
- Newline

Reguläre Ausdrücke in Lex

einige Zeichen aus Σ
werden auch als Meta-Zeichen benutzt

Ausdruck	passt auf ...	Beispiel
c	Zeichen c, außer Operatoren	a
\c	Zeichen c (Abschaltung bzw. Zuschaltung der Sonderrolle)	* bzw. \n
"s"	String s (beliebige Zeichen)	"a*"
.	beliebiges Zeichen (*Catch all*), außer Zeilenwechsel(!!!)	a.b
^	Zeilenanfang (Dateianfang oder nach \n)	^abc
$	Zeilenende	abc$
[s]	beliebiges Zeichen von s	[abc]
[^s]	beliebiges Zeichen, das nicht in s enthalten ist	[^abc]
r*	null oder mehr r's	a*
r+	ein oder mehr r's	a+
r?	null oder ein r	a?
r{m,n}	m- bis n-faches Vorkommen von r	a{1,5}
$r_1 r_2$	r_1 gefolgt von r_2	ab
$r_1 \mid r_2$	r_1 oder r_2	a|b
(r)	r	(a|b)
r_1 / r_2	r_1 , wenn dahinter r_2 folgt	abc/123

verschärftes look-ahead,
123 wird rückgeschrieben

Lexer-Verhaltensweisen

Lookahead-Operator

- Lex liest automatisch **ein** Zeichen über das letzte hinaus, das zum Lexem gehört und setzt die Eingabe danach wieder zurück

- Spezialfall: das Lexem soll nur dann akzeptiert werden, wenn danach ganz bestimmte Zeichen folgen (die aber nicht zum Lexem gehören)

  z.B. IF / \(.* \) {letter} hier muss die Eingabe um mehrere Zeichen zurückgesetzt werden

Start des Folgemusters

Muster: öffnende Klammer \(
beliebige ZK .*
schließende Klammer \)
Buchstabe

Möglicher Mix von Zuständen und RAs in Lex

- man kann den Zustandsautomaten in eine Menge von Teilautomaten strukturieren:
  jeder hat dabei seinen individuellen Startzustand

- jeder RA kann mit einer Menge von Startzuständen präfigiert sein,
  für die er gültig ist (d.h. zu welchem Teilautomaten zugehörig)

- im Aktionsteil kann der Startzustand explizit geändert werden
  damit lassen sich Übergänge mit RA markieren
  (und nicht nur mit Symbolen)

Effekt: abstraktere Automaten mit Ausführungswechsel
von einem zum anderen

Möglicher Mix von Zuständen und RAs in Lex (Forts.)

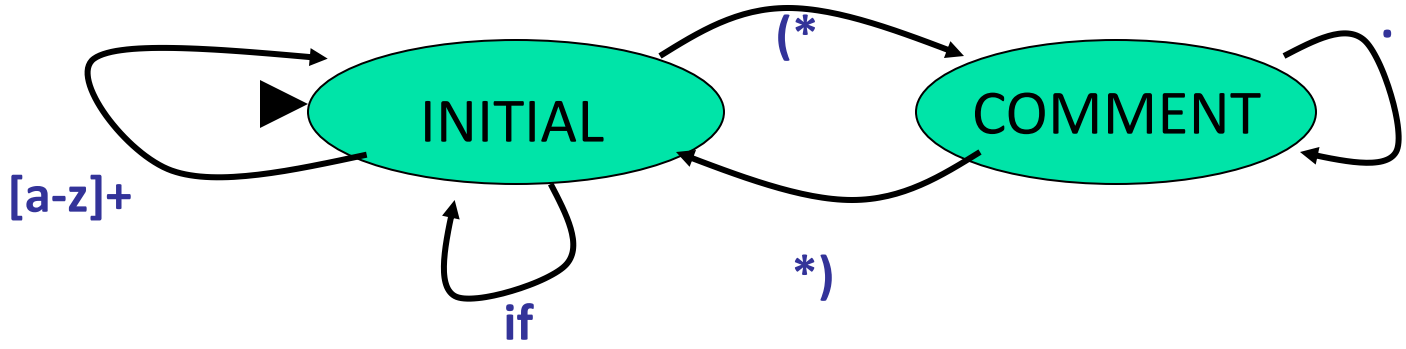
Startkonvention:

- Der Scanner befindet sich zu Beginn im Default-Startzustand:
  INITIAL

  (muss nicht explizit vereinbart werden)

Behandlung von Kommentarklammern: "Hierarchische" Automaten

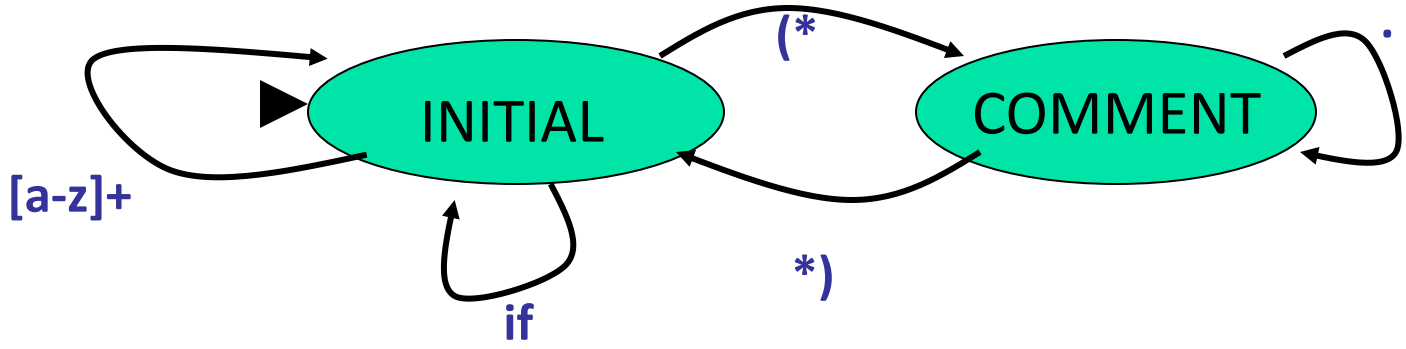
- eigentlich Automaten auf gleicher Ebene




```
...
%x INITIAL COMMENT/* Startzustände mit exklusiver Regelzuordnung */
%%
<INITIAL>if                    {...;  return IF;}
<INITIAL>[a-z]+                {...;  return ID;}
<INITIAL>"(*"                  {...;  BEGIN COMMENT;}    /* Automatenumschaltung */
<INITIAL>.                     {...;  EM_error("illegal character");}
<COMMENT>"*)"                  {...;  BEGIN INITIAL;}      /* Automatenrückschaltung */
<COMMENT>.                     {...; } /* Kommentar wird überlesen */
```

Behandlung von Kommentarklammern: "Hierarchische" Automaten

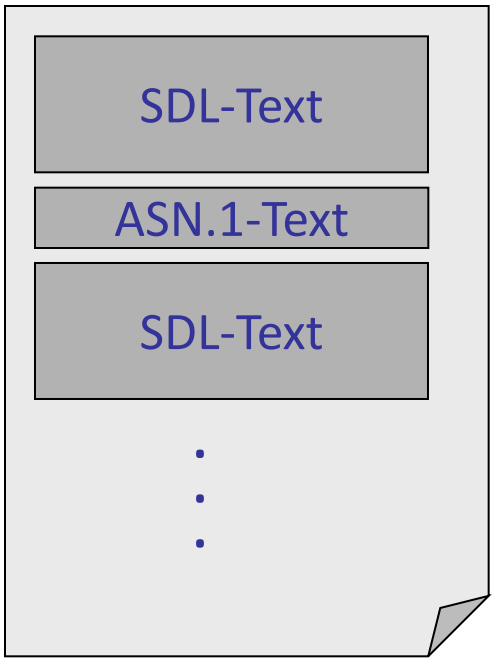
- eigentlich Automaten auf gleicher Ebene




```
...
%x INITIAL COMMENT/* Startzustände mit exklusiver Regelzuordnung */
%%
<INITIAL>if                    {...;  return IF;}
<INITIAL>[a-z]+                {...;  yyval.sval= strdup (yytext);  return ID;}
<INITIAL>"(*"                  {...;  BEGIN COMMENT;}    /* Automatenumschaltung */
<INITIAL>.                     {...;  EM_error("illegal character");}
<COMMENT>"*)"                  {...;  BEGIN INITIAL;}      /* Automatenrückschaltung */
<COMMENT>.                     {...; } /* Kommentar wird überlesen */
```

Weiterer Anwendungsfall "Hierarchische" Automaten

- Integration von Programmfragmenten einer anderen Sprache (Mix von Quellsprachen)



Lokale ASN.1-Datentypdefintion
in einer SDL-Funktionseinheit

Die lexikalischen Regeln beider Sprachen unterscheiden sich