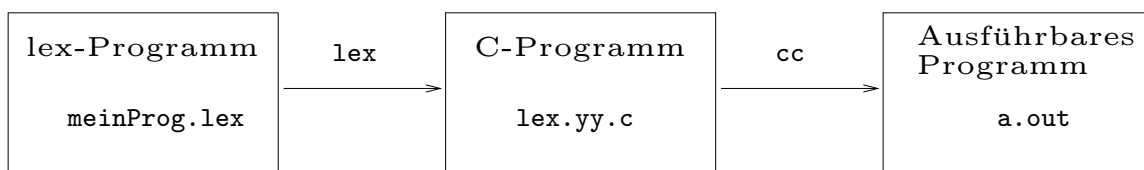


# 1 Einleitung, Motivation

- `lex` ist ein universelles Werkzeug für die lexikalische Analyse
- Entwicklung in den 70er Jahren als Ergänzung zum Parsergenerator `yacc`
- Erhebliche Vereinfachung bei der Erstellung eines Scanners
  - ⊕ kürzere Entwicklungszeit
  - ⊕ bessere Lesbarkeit
  - ⊕ leichtere Änderbarkeit
  - ⊖ ggf. schlechtere Effizienz
- GNU Variante heißt `flex`  
(<http://www.gnu.org/software/flex/>)

- Vorgehensweise:
  1. Spezifikation regulärer Ausdrücke und dazugehöriger Aktionen in Zielsprache  
→ `lex`-Programm
  2. Übersetzung des `lex`-Programms in Zielsprache
  3. ggf. Einbinden in eigenes Programm, Übersetzung in Maschinencode
- Zielsprache kann C oder C++ sein



## Einsatzgebiet:

- Compilerbau
- Textverarbeitung
- Chiffrierung

## 2 Lex-Programme

### 2.1 Struktur

*Definitionsteil*

%%

*Regelteil*

%%

*Benutzerdefinierte Routinen*

Der Regelteil ist der Kern eines lex-Programms

### 2.2 Der Regelteil

- Liste regulärer Ausdrücke  $r_i$  und dazugehöriger Aktionen  $aktion_i$

$r_1$	$aktion_1$
$r_2$	$aktion_2$
$\vdots$	$\vdots$
$r_n$	$aktion_n$

- eine Aktion besteht aus C-Anweisungen

## Beispiel 1 (Das erste Lex-Programm)

```
%%  
a    printf("i");  
e    printf("i");  
o    printf("i");  
u    printf("i");
```

## 2.2.1 Reguläre Ausdrücke

### Einfache Zeichen und Metazeichen

- Alphabet  $\Sigma$  ist der Zeichensatz des Systems
- Einige Zeichen haben besondere Bedeutung in regulären Ausdrücken, die sog. *Metazeichen*

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

- Behandlung als normales Zeichen durch Voranstellen von “\” oder Einbinden in “ ”

### Beispiel 2 (Erkennung von Metazeichen)

`%%`

`\\/\\/ printf("Doppelslash");`

`"?" printf("Fragezeichen");`

## Escape-Sequenzen und Spezialzeichen

Für einige Bestandteile eines Textes gibt es keine Zeichen.

Hierfür gibt es Ersatzdarstellungen:

<code>\b</code>	Backspace (Zurücksetzzeichen)
<code>\f</code>	Formfeed (Seitenvorschub)
<code>\n</code>	Newline (Zeilenvorschub)
<code>\t</code>	Tabulator
<code>\ddd</code>	Zeichen, das dem Oktalwert <i>ddd</i> entspricht
<code>^</code>	Anfang einer Zeile
<code>\$</code>	Ende einer Zeile

### Beispiel 3 (Erkennung von Leerzeilen)

```
%%
```

```
^\n      ;
```

## Klassen von Zeichen

- Menge oder Bereich von Zeichen mit “[” und “]”, z.B.

[xyz]	Eines der Zeichen x, y oder z
[A-Z]	Ein Großbuchstabe

- Beliebiges Zeichen mit “.”

.	beliebiges Zeichen aus $\Sigma$ (außer $\backslash n$ )
---	---

- Ausschluß von Zeichen mit “^”, z.B.

[^0-9]	alle Zeichen außer Zahlen, $\Sigma \setminus \{0, \dots, 9\}$
--------	--

Operator “^” ist nur bei Mengen anwendbar.  
Es wird *kein* Komplement Operator für reguläre Ausdrücke zur Verfügung gestellt!

## Zusammengesetzte reguläre Ausdrücke

Seien  $r, s$  reguläre Ausdrücke.

$r s$	$r$ oder $s$ (Vereinigung)
$rs$	$r$ gefolgt von $s$ (Konkatenation)
$r^*$	beliebig oft $r$ , auch keinmal
$(r)$	deckt $r$ ab
$r^+$	beliebig oft $r$ , mindestens einmal
$r\{n,m\}$	zwischen $n$ und $m$ Vorkommen von $r$ , $0 \leq n \leq m$
$r?$	ein- oder keinmal $r$

Priorität der Operatoren (aufsteigend):

|

Konkatenation

$*$   $+$   $?$

()

## Beispiel 4 (C-Identifier)

```
%%  
([a-zA-Z_])([a-z]| [A-Z]|_|[0-9])*  
printf("Identifier\n");
```

## Beispiel 5 (Ganze Zahlen)

```
%%  
0| [+ -]? [1-9] [0-9]* printf("Ganze Zahl\n");
```

## Kontextsensitivität

r/s	Es wird eine Zeichenkette erkannt, die von $r$ akzeptiert wird, <i>und</i> zusätzlich eine Zeichenkette folgt, die von $s$ akzeptiert wird.
-----	---

## Beispiel 6 (Klassifikation von C-Identifiern)

```
%%
([a-zA-Z_][a-zA-Z_0-9]*)/[ \t]*\[
    printf("Array\n");
([a-zA-Z_][a-zA-Z_0-9]*)/[ \t]*\(
    printf("Funktion/Makro\n");
([a-zA-Z_][a-zA-Z_0-9]*)
    printf("anderer\n");
.
\n
```

## 2.2.2 Aktionen

- Zur Erinnerung:

$r_1$	$aktion_1$
$r_2$	$aktion_2$
$\vdots$	$\vdots$
$r_n$	$aktion_n$

- Eine Aktion besteht aus  $m$  C-Anweisungen,  
 $m \geq 0$

$m = 0$ : *leere Aktion* ; bzw. `{ }`

$m = 1$ : keine Klammerung nötig

$m \geq 2$ : Klammerung mit `{ ... }`

- Paßt kein regulärer Ausdruck, wird eine *default-Aktion* ausgeführt; diese schreibt die gelesenen Zeichen unverändert auf die Standardausgabe.

- Ausführung *derselben* Aktion für mehrere reguläre Ausdrücke ist möglich durch Aktionszeichen “|”  
→ Aktion des nächsten regulären Ausdrucks wird ausgeführt

### Beispiel 7 (Entfernen von Whitespaces)

```
%%  
" "      |  
\t       |  
\n       ;
```

### 2.2.3 Lex-interne Variablen, Makros und Funktionen

`yytext`, `yylen`

char-Array `yytext` (mit Länge `yylen`) enthält den aktuell gelesenen Eingabestring

#### Beispiel 8 (`toLowerCase`)

`%%`

```
[A-Z]      printf("%c", yytext[0] + 32);
```

`yylineno`

enthält aktuelle Zeilennummer des Eingabetextes

`ECHO`

Makro, welches den Inhalt von `yytext` ausgibt

Die folgenden Funktionen bieten *Lookahead*-Möglichkeiten:

`yylless()`

Zurückschreiben von gelesenen Zeichen des Eingabetextes:

`yylless(n)`

es verbleiben  $n$  gelesene Zeichen in `yytext` ( $\leftrightarrow$   $yyleng - n$  Zeichen werden zurückgeschrieben).

`input()`, `output()`, `unput()`

Verarbeitung einzelner Zeichen des Eingabetextes:

- `input()`  
liefert nächstes Zeichen des Eingabetextes
- `output(c)`  
gibt Zeichen  $c$  auf `stdout` aus
- `unput(c)`  
schiebt Zeichen  $c$  wieder zurück in den Eingabetext

`yywrap()`

Funktion, die beim Erreichen des Dateiendes der Eingabe aufgerufen wird.

- Kann vom Benutzer überschrieben werden
- Rückgabewert  $\neq 0 \rightarrow$  Programm wird verlassen
- Rückgabewert  $= 0 \rightarrow$  Programm arbeitet auf neuer Eingabe weiter

## 2.2.4 Zweideutige Regeln

### **Problem:**

Mehrere reguläre Ausdrücke decken denselben Eingabestring bzw. Präfixe desselben Eingabestrings ab.

### **Lösung:**

1. Es wird der reguläre Ausdruck ausgewählt, der den längsten Eingabestring abdeckt.
2. Gibt es mehrere reguläre Ausdrücke, die einen gleichlangen Eingabestring abdecken, so wird der zuerst definierte ausgewählt.

## 2.3 Der Definitionsteil

Zur Erinnerung:

*Definitionsteil*

%%

*Regelteil*

%%

*Benutzerdefinierte Routinen*

Funktion des Definitionsteils:

- Einbinden von C-Code
- Spezifikation sog. *regulärer Definitionen* zur Vereinfachung regulärer Ausdrücke
- Definition von Startbedingungen

## Einbinden von C-Code

- C-Code wird folgendermaßen eingebunden:

```
%{
```

```
    C-Anweisungen
```

```
%}
```

oder

```
□C-Anweisung_1;
```

```
□C-Anweisung_n;
```

- C-Anweisungen sind hier in der Regel Präprozessor-Anweisungen und Variablen-Deklarationen
- Hilfsfunktionen werden im Teil *Benutzerdefinierte Routinen* eingebunden

## Reguläre Definitionen

Reguläre Definitionen dienen der Vereinfachung von regulären Ausdrücken. Eine reguläre Definition ist eine Folge von Regeln der Form

$$\begin{array}{ll} Name_1 & r_1 \\ Name_2 & r_2 \\ \vdots & \vdots \\ Name_n & r_n \end{array}$$

wobei  $Name_1, \dots, Name_n$  paarweise verschiedene Bezeichner sind und jedes  $r_i$  ein regulärer Ausdruck über  $\Sigma \cup \{\underline{\{Name_1\}}, \dots, \underline{\{Name_{i-1}\}}\}$ .

## Beispiel 9 (C-Identifizier, vereinfacht)

L [a-z] | [A-Z] | \_

D [0-9]

%%

{L}({L}|{D})\*

```
printf("%s%s", "Identifizier: ", yytext);
```

## Startbedingungen

- Startbedingungen erlauben zustandsabhängige Aktivierung von regulären Ausdrücken
- Definition von Startbedingungen  
`%s name_1 name_2 ... name_n`
- Aktivierung eines Zustands im Aktionsteil eines regulären Ausdrucks mit  
`BEGIN name_i;`
- Der Normalzustand, in dem alle regulären Ausdrücke ohne Startbedingung aktiv sind, ist 0, d.h. mit  
`BEGIN 0;`  
wechselt man in den Normalzustand
- Kennzeichnung von zustandsabhängigen regulären Ausdrücken  
`<name_i>regAusdruck`                      Aktion  
bzw.  
`<name_i>,<name_j>regAusdruck`      Aktion

## Beispiel 10 (Entfernung von C-Kommentaren)

%s KOMMENTAR

%%

<KOMMENTAR>"\*/"      BEGIN 0;

<KOMMENTAR>.      |

<KOMMENTAR>\n      ;

"/\*"      BEGIN KOMMENTAR;

## 2.4 Benutzerdefinierte Routinen

Zur Erinnerung:

*Definitionsteil*

*%%*

*Regelteil*

*%%*

*Benutzerdefinierte Routinen*

Funktion des dritten Teils:

- Code wird unverändert nach `lex.yy.c` kopiert
- Einbinden von benutzerdefinierten Funktionen, die in den Aktionen benutzt werden können
- Einbinden des `main`-Programms oder der Funktion `yywrap()`

## Beispiel 11 (LineCounter)

```
%{  
int lineCounter = 0;  
%}  
  
%%  
\n      lineCounter++;  
.  
;  
  
%%  
yywrap(){  
    printf("Number of Lines: %d\n",  
           lineCounter);  
    return 1;  
}
```

### 3 Bedienung von `lex`

1. Erstellen des `lex`-Programms, z.B.

```
meinProg.lex
```

2. Übersetzen des `lex`-Programms in C-File `lex.yy.c` mit

```
lex meinProg.lex
```

3. Übersetzen von `lex.yy.c` (sowie Hinzulink-  
ken der `lex`-Bibliothek `libl.a` bzw. `libfl.a`  
für `flex`)

- (a) in eigenständiges Programm mit

```
cc lex.yy.c -ll (bzw. -lfl
```

- (b) zusammen mit eigenem Programm  
`meinProg.c`, welches die Funktion  
`yylex()` aufruft, mit

```
cc meinProg.c lex.yy.c -ll
```

## 4 Aufbau von `lex.yy.c`

1. Lex-interne Variablen-Deklarationen und Makro-Definitionen
2. C-Programmfragmente, die im Definitionsteil angegeben werden
3. Die Funktion `yylex()`
  - (a) Lesen des Eingabetextes
  - (b) Erkennen des passenden regulären Ausdrucks und Verzweigung zum zugehörigen Aktionsteil
  - (c) Ausführung des Aktionsteils
4. Benutzerdefinierte Routinen
5. Zustandstabellen, die von `lex` aus den regulären Ausdrücken generiert worden sind
6. Inhalt der Datei `ncform`; enthält im wesentlichen die Funktion `yylook()`, welche die Zustandstabellen durchläuft