Einführung in den Scannergenerator `lex`

Dino Ahr

20. November 2001

Inhaltsverzeichnis

1 Einleitung

Der Scannergenerator `lex`[1] ist ein universelles Werkzeug für die lexikalische Analyse und wurde in den 70er Jahren von M. E. Lesk und E. Schmidt als Ergänzung zu dem Parsergenerator `yacc` entwickelt.

Mittels `lex` wird die Erstellung von Programmen zur lexikalischen Analyse erheblich vereinfacht. Dies wird dadurch erreicht, daß die gewünschten zu verarbeitenden lexikalischen Konstrukte auf einer höheren Abstraktionsebene spezifiziert werden, nämlich mit Hilfe *regulärer Ausdrücke*. Ein sog. *lex-Programm* besteht dann im wesentlichen aus einer Folge von regulären Ausdrücken, sowie dazugehörigen Aktionen, die ausgeführt werden, wenn der jeweilige Ausdruck erkannt wurde.

Das lex-Programm wird mittels `lex` in ein C-Programm[2] `lex.yy.c` übersetzt, welches einen Eingabetext von der Standardeingabe liest und diesen in Strings zerlegt, die durch die regulären Ausdrücke abgedeckt werden. Das C-Programm kann schließlich in eigenen Anwendungen eingebunden werden (s. Abbildung 1).

Obwohl `lex` zunächst nur für den Einsatz im Compilerbau gedacht war, entpuppte es sich als brauchbares Werkzeug in vielen anderen Anwendungsbereichen, wie z.B. in der Textverarbeitung, bei Kommandoprozessoren und bei Chiffrierungen.

Natürlich können alle diese Aufgaben auch ohne Zuhilfenahme von `lex` bewerkstelligt werden. Die Vorteile von `lex` liegen jedoch klar in der kürzeren Entwicklungszeit, in der besseren Lesbarkeit und der leichteren Änderbarkeit der Programme. Ein Nachteil von Scannern, die mittels `lex` erzeugt werden, könnten schlechtere Effizienz im Vergleich zu handcodierten Scannern sein.

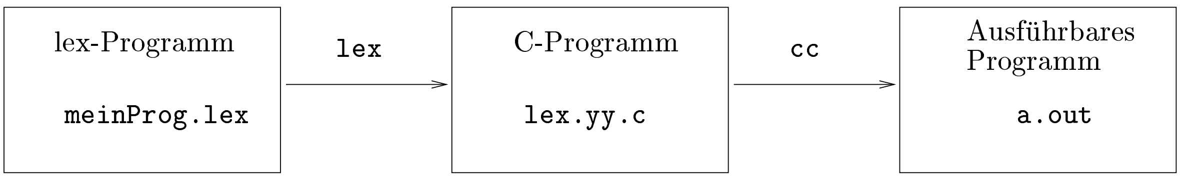

Abbildung 1: Übersetzung eines lex-Programmes

2 Lex-Programme

In diesem Abschnitt werden die Struktur und die einzelnen Bestandteile eines lex-Programms beschrieben.

2.1 Struktur eines lex-Programms

Ein lex-Programm besteht aus drei Teilen, welche durch jeweils zwei aufeinanderfolgende Prozentzeichen in einer Zeile voneinander getrennt werden:

Definitionsteil
`%%`
Regelteil

[1] Es gibt eine frei erhältliche GNU Version von `lex`, die `flex` heißt.

[2] `flex` ist in der Lage, C und C++ Code zu erzeugen.

```
%%
```
Benutzerdefinierte Routinen

Die Abschnitte *Definitionsteil* und *Benutzerdefinierte Routinen* können auch weggelassen werden. Werden die benutzerdefinierten Regeln weggelassen, so kann das zweite Paar von Prozentzeichen entfallen.

Da der Abschnitt mit den lex-Regeln der wichtigste Teil eines lex-Programms darstellt, wird dieser als nächstes behandelt.

2.2 Der Regelteil

Der Regelteil eines lex-Programms enthält eine Liste von regulären Ausdrücken r_i und dazugehörigen Aktionen $aktion_i$, die in Form einer Tabelle angeordnet sind:

$$\begin{array}{ll} r_1 & aktion_1 \\ r_2 & aktion_2 \\ \vdots & \quad\vdots \\ r_n & aktion_n \end{array}$$

Die Übersetzung eines solchen lex-Programms resultiert in einem C-Programm, welches sich folgendermaßen verhält: Der Eingabetext wird Zeichen für Zeichen gelesen und es wird versucht, einen passenden regulären Ausdruck r_i zu identifizieren. Wird dieser gefunden, so wird die Aktion $aktion_i$ ausgeführt und fortgefahren. Eine Aktion besteht aus C-Befehlen. Wird für eine Zeichenkette kein passender regulärer Ausdruck gefunden, so wird diese unverändert auf die Standardausgabe geschrieben.

Beispiel 1 (Das erste Lex-Programm) *Das folgende lex-Programm ersetzt in einem Eingabetext jeden der Vokale a, e, o, u mit dem Vokal i.*

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

Die regulären Ausdrücke bestehen hier einfach aus den Buchstaben a, e, o, u; die Aktionen sind printf-Befehle, die jeweils ein Zeichen "i" auf die Standardausgabe schreiben.

Im folgenden wird nun genauer beschrieben, wie reguläre Ausdrücke in einem lex-Programm spezifiziert werden.

2.2.1 Reguläre Ausdrücke

Das Alphabet Σ, das den regulären Ausdrücken zugrunde liegt, ist der Zeichensatz des Systems. Einige dieser Zeichen – die sog. *Metazeichen* – haben eine besondere Bedeutung in einem regulären Ausdruck. Dies sind:

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

Die Bedeutung dieser Metazeichen wird in den folgenden Abschnitten erläutert.

Einfache Zeichen und Metazeichen. Zur Erkennung einzelner Zeichen oder Zeichenketten werden diese einfach als regulärer Ausdruck angegeben. Soll ein Zeichen im Text erkannt werden, welches ein Metazeichen ist, so muß die Sonderbedeutung des Metazeichens mittels eines vorangestellten Backslashs "\" ausgeschaltet werden. Alternativ kann das Metazeichen auch in Anführungsstriche gesetzt werden.

Soll ein Leerzeichen oder eine Zeichenkette, welche Leerzeichen enthält, erkannt werden, so müssen diese in Anführungsstriche gesetzt werden.

Beispiel 2 (Einfache Zeichen) *Das folgende lex-Programm erkennt einige Schlüsselwörter in einem C-Programm und gibt entsprechende symbolische Konstanten, die im Definitionsteil definiert werden, als Token an das aufrufende Programm zurück:*

```
%{
#define IF    1
#define THEN  2
#define ELSE  3
#define DO    4
#define WHILE 5
%}

%%
if      return(IF);
then    return(THEN);
else    return(ELSE);
do      return(DO);
while   return(WHILE);
```

Beispiel 3 (Erkennung von Metazeichen) *Das folgende lex-Programm erkennt zwei aufeinanderfolgende Slashes "//" (z.B. Beginn einer Kommentarzeile in C++ oder Java) sowie ein Fragezeichen.*

```
%%
\/\/    printf("Doppelslash");
"?"     printf("Fragezeichen");
```

Beispiel 4 (Leerzeichen) *Das folgende Lex-Programm erkennt eine Zeichenkette, die Leerzeichen enthält.*

```
%%
"Interdisziplinaeres Zentrum fuer Wissenschaftliches Rechnen"
                                        printf("IWR");
```

Escape-Sequenzen und Spezialzeichen. Für einige Bestandteile eines Textes, wie z.B. ein Zeilentrenner oder ein Tabulator, gibt es keine Zeichen. Es gibt jedoch die folgenden Ersatzdarstellungen:

`\b`	Backspace (Zurücksetzzeichen)
`\f`	Formfeed (Seitenvorschub)
`\n`	Newline (Zeilenvorschub)
`\r`	Return (Wagenrücklauf)
`\t`	Tabulator
`\ddd`	Zeichen, das dem Oktalwert *ddd* entspricht

Bei der Erkennung von lexikalischen Einheiten ist es auch manchmal von Interesse, ob diese am Anfang oder am Ende einer Zeile stehen. Auch hierfür gibt es spezielle Zeichen:

`^`	Anfang einer Zeile
`$`	Ende einer Zeile

Beispiel 5 (Erkennung von Leerzeilen) *Mit dem folgenden lex-Programm lassen sich Leerzeilen aus einem Text entfernen. Gesucht wird nach einem Zeilenanfang gefolgt von einem Zeilenvorschub.*

```
%%
^\n      ;
```

Klassen von Zeichen. Um ein beliebiges Zeichen oder eines aus einer Menge von Zeichen zu erkennen, werden die Metazeichen "`.`" bzw. "`[`" und "`]`" verwendet.

`.`	Ein beliebiges Zeichen (außer `\n`)
`[xyz]`	Eines der Zeichen x, y oder z

Soll eine größere Menge von Zeichen abgedeckt werden, z.B. Klein- oder Großbuchstaben, dann können mittels dem Metazeichen "`-`" auch Bereiche für die Zeichen angegeben werden. So kann man beispielsweise für die Klasse der Groß- bzw. Kleinbuchstaben schreiben

`[a-z]`	Ein Kleinbuchstabe
`[A-Z]`	Ein Großbuchstabe

Beispiel 6 (Erkennung von Trennungen) *Mit dem folgenden lex-Programm lassen sich z.B. Trennungen in einem Text erkennen.*

```
%%
[a-zA-Z]-$        printf("Trennung");
```

Möchte man gewisse Zeichen einer Klasse ausschließen, so wird dies mit dem Metazeichen "`^`" bewerkstelligt. Sollen z.B. alle Zeichen außer den Zahlen abgedeckt werden, so wird dies folgendermaßen spezifiziert:

`[^0-9]`	alle Zeichen außer Zahlen

Der Operator "`^`" ist nur bei einer Klasse von Zeichen anwendbar und kann nicht auf reguläre Ausdrücke angewendet werden. Es gibt also in `lex` keinen Komplement Operator für reguläre Ausdrücke!

Zusammengesetzte reguläre Ausdrücke. Reguläre Ausdrücke lassen sich zu neuen regulären Ausdrücken kombinieren. Für die regulären Ausdrücke r und s ergeben sich:

`r|s`	r oder s (Vereinigung)
`rs`	r gefolgt von s (Konkatenation)
`r*`	beliebig oft r, auch keinmal
`r+`	beliebig oft r, mindestens einmal
`r?`	ein- oder keinmal r
`(r)`	deckt r ab
`r{n,m}`	zwischen n und m Vorkommen von r, $0 \leq n \leq m$
`{name}`	benamte reguläre Ausdrücke (s. Abschnitt 2.3.2)

Die *Priorität* der Operatoren ist in aufsteigender Reihenfolge

```
|
{}
Konkatenation
* + ?
()
```

Beispiel 7 (C-Identifier, ganze Zahlen) *Die regulären Ausdrücke des folgenden lex-Programms erkennen Bezeichner eines C-Programms (engl. Identifier) und ganze Zahlen.*

```
%%
([a-zA-Z_])([a-z]|[A-Z]|_|[0-9])*  printf("Identifier\n");
0|[+-]?[1-9][0-9]*                 printf("Ganze Zahl\n");
```

Beispiel 8 (ifdef) *Die drei regulären Ausrücke*

```
if|ifdef
if(def){0,1}
if(def)?
```

erkennen alle die Zeichenkette "if" oder "ifdef".

Kontextsensitivität. Es lassen sich auch reguläre Ausdrücke erkennen, die in einem speziellen Kontext stehen müssen. Dieser Sachverhalt wird mit dem Metazeichen "/" gekennzeichnet:

`r/s`	Es wird eine Zeichenkette erkannt, die von r akzeptiert wird, *und* zusätzlich eine Zeichenkette folgt, die von s akzeptiert wird.

Beispiel 9 (Klassifikation von Identifiern) *Das folgende lex-Programm erkennt wiederum Identifier in einem C-Programm. Zusätzlich wird noch der Kontext erkannt, in dem der Identifier benutzt wird.*

```
%%
([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

Für jeden regulären Ausdruck des Regelteils gibt es eine dazugehörige Aktion, welche ausgeführt wird, wenn eine Zeichenkette durch den regulären Ausdruck abgedeckt wird.

Für jedes lex-Programm existiert implizit eine *default-Aktion*, welche aktiviert wird, wenn keiner der angegebenen regulären Ausdrücke paßt. Diese schreibt die gelesene Zeichenkette unverändert auf die Standardausgabe.

Eine Aktion besteht aus C-Anweisungen. Soll mehr als eine C-Anweisung in einer Aktion ausgeführt werden, so müssen diese in geschweiften Klammern "`{`" und "`}`" gesetzt werden. Eine *leere Aktion* wird durch eine leere C-Anweisung `;` bzw. einen leeren Block "`{}`" realisiert.

Soll für mehrere reguläre Ausdrücke dieselbe Aktion ausgeführt werden, so wird dies durch das Metazeichen "`|`" kenntlich gemacht. Wird für einen regulären Ausdruck anstelle einer Aktion das Zeichen "`|`" angegeben, so gilt für diesen die Aktion des nächsten regulären Ausdrucks.

Beispiel 10 (Entfernen von Whitespaces) *Das folgende lex-Programm entfernt whitespaces, d.h. Leerzeichen, Tabulatorzeichen und Zeilentrenner, aus dem Eingabetext. Dies wird einfach dadurch bewerkstelligt, daß bei Erkennung eines solchen Zeichens eine leere Aktion ausgeführt wird.*

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

2.2.3 Lex-interne Variablen, Makros und Funktionen

Der von `lex` generierte C-Code stellt Variablen, Makros und Funktionen bereit, die in den Aktionen genutzt werden können.

Das char-Array `yytext` und dessen Länge `yyleng`. Beim Ausführen einer Aktion wird sehr häufig die Zeichenkette benötigt, die durch den zugehörigen regulären Ausdruck erkannt wurde. Diese befindet sich in dem char-Array `yytext`[3]. Die Länge der Zeichenkette läßt sich über die Integer Variable `yyleng` abfragen.

Beispiel 11 (toLower) *Das folgende Lex-Programm wandelt in einem gegebenen Text alle Großbuchstaben in Kleinbuchstaben um. Die Kleinbuchstaben bleiben unverändert.*

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

Die Variable `yylineno`. Die Integer Variable `yylineno` enthält die Zeilennummer des Eingabetextes, in welcher sich der Leseprozeß aktuell befindet.

[3] Alle Variablen und Funktionen im Zusammenhang mit `lex` haben den Präfix `yy`, um Namenskonflikte zu vermeiden.

Das Makro `ECHO`. Das Makro `ECHO` gibt den Inhalt von `yytext` aus. Es ist in der von `lex` generierten Datei `lex.yy.c` folgendermaßen definiert:

```
#define ECHO fprintf(yyout, "%s",yytext)
```

Der Dateizeiger `yyout` ist mit der Standardausgabe verbunden.

Die Funktion `yymore`. Es können Situationen auftreten, in denen der reguläre Ausdruck nur einen Teil des gewünschten Eingabestrings abdecken kann. In solchen Fällen ist es wünschenswert, daß derselbe reguläre Ausdruck auf den weiteren Eingabetext angwendet wird und der Inhalt des Arrays `yytext` nicht überschrieben wird. Hierfür bietet `lex` die Funktion `yymore()` an.

Beispiel 12 (Extraktion von String-Konstanten) *Es soll ein lex-Programm erstellt werden, das die String-Konstanten eines C-Programms extrahiert. Die Schwierigkeit besteht hier darin, daß auch der Fall berücksichtigt werden muß, wenn die String-Konstante Anführungsstriche enthält. So ist z.B.*

```
"Das Programm gibt \"Hello World!\" aus."
```

eine gültige String-Konstante.
Das folgende lex-Programm erkennt auch solche String-Konstanten.

```
%%
\"[^"]*      { if (yytext[yyleng-1] == '\\')
                 yymore();
               else
                 ECHO;
             }
.            |
\n           ;
```

Für die obige String-Konstante würde zunächst der Teilstring

```
"Das Programm gibt \
```

erkannt und in `yytext` *abgespeichert. Da* `yytext` *mit \ endet, wird* `yymore()` *aufgerufen, was zur Folge hat, daß die Regel erneut angewendet wird und die Zeichenkette* `"Hello World!\` *an* `yytext` *angehängt wird. Da* `yytext` *wieder mit \ endet, wiederholt sich der Prozeß, so daß letztendlich die String-Konstante vollständig eingelesen ist und mit* `ECHO` *ausgegeben wird. Für alle anderen Zeichen wird eine leere Aktion ausgeführt.*

Die Funktion `yyless`. Mittels der Funktion `yyless(`n`)` lassen sich gelesene Zeichen des Eingabtextes wieder zurückschreiben. Dabei besagt der Parameter n, daß n Zeichen in dem Array `yytext` verbleiben und $yyleng - n$ gelesene Zeichen wieder in die Eingabe zurückgeschoben werden.
Diese Funktion ist nützlich, wenn eine Vorausschau (engl. Lookahead) auf den Eingabetext erforderlich ist.

Die Funktionen `input`, `output` und `unput`. Die Funktion `input()` liefert das nächste Zeichen des Eingabetextes. Dieses Zeichen wird also dann *nicht* mehr durch die Anwendung eines regulären Ausdrucks verarbeitet.

Soll ein gelesenes Zeichen c doch durch einen regulären Ausdruck verarbeitet werden, so läßt sich dieses mit der Funktion `unput(c)` wieder in den Eingabetext zurückschieben. Die gemeinsame Anwendung von `input` und `unput(c)` bietet also wiederum eine Lookahead Möglichkeit.

Die Funktion `output(c)` gibt das Zeichen c auf die Standardausgabe aus.

Die Funktion `yywrap`. Die Routine `yywrap()` wird von `lex` beim Erreichen des Dateiendes aufgerufen. Durch Überschreiben dieser Funktion kann der Benutzer festlegen, was nach Lesen des Eingabetextes weiter zu tun ist.

Ein Rückgabewert ungleich 0 bewirkt, daß die Funktion `yylex` verlassen wird. Die Funktion `yylex` steuert den Scanprozeß (s. Abschnitt 4). Dies ist die Voreinstellung, wenn `yywrap` nicht überschrieben wird.

Setzt man den Rückgabewert gleich 0, so wird `yylex` veranlaßt weiterzuarbeiten. Dazu ist es natürlich erforderlich, daß ein weiterer Eingabetext auf der Standardeingabe zur Verfügung gestellt wird.

Das Makro `REJECT`. Manchmal kann es wünschenswert sein, daß ein Eingabestring bzw. ein Teilstring des Eingabestrings durch mehrere Regeln abgedeckt wird, damit die zugehörigen Aktionen abgearbeitet werden. Für diese Zwecke stellt `lex` das Makro `REJECT` zur Verfügung.

`REJECT` muß immer die letzte Anweisung einer Aktion sein und kann nicht in den benutzerdefinierten Routinen verwendet werden.

2.2.4 Zweideutige Regeln

Bei der Spezifizierung regulärer Ausdrücke kann es vorkommen, daß mehrere reguläre Ausdrücke denselben Eingabestring bzw. Präfixe desselben Eingabestrings abdecken. Hier benutzt `lex` die folgenden Regeln, um zu entscheiden, welcher reguläre Ausdruck benutzt wird:

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 erste in der Liste ausgewählt.

2.2.5 Benutzerdefinierte Variablen im Regelteil

Im Regelteil lassen sich Variablen deklarieren werden, welche in den Aktionen benutzt werden können. Solche C-Anweisungen müssen der Konvention genügen, daß zwischen Zeilenanfang und Anweisung mindestens ein Leer- bzw. Tabulatorzeichen ist (s. auch Abschnitt 2.3.1).

2.3 Der Definitionsteil

Im Definitionsteil eines lex-Programmes lassen sich C-Programmteile einbinden, Reguläre Definitionen festlegen, Startbedingungen festlegen, Zeichensatztabellen definieren und lex-interne Tabellengrößen einstellen.

2.3.1 Einbinden von C-Code

Im Definitionsteil eines lex-Programms können C-Programmfragmente angegeben werden, die unverändert in das generierte C-Programm `lex.yy.c` eingefügt werden. Dabei handelt es sich in der Regel um Präprozessor-Anweisungen und Variablen-Deklarationen. Das Einbinden geschieht mittels der Sonderzeichen "`%{`" und "`%}`" in der folgenden Form:

```
%{
C-Programmfragmente
%}
```

Dabei müssen die beiden Begrenzer "`%{`" und "`%}`" jeweils allein in einer Zeile stehen.

Es ist auch möglich, C-Anweisungen ohne Angabe der Begrenzer in `lex.yy.c` einbinden zu lassen. In diesem Fall muß jede Zeile, die eine C-Anweisung enthält, mit mindestens einem Leerzeichen oder Tabulatorzeichen beginnen.

Die erstgenannte Vorgehensweise wird empfohlen, da diese besser lesbar und weniger fehleranfällig ist.

Alle in diesem Teil eingebundenen Variablen sind *global* im Gegensatz zu den im Regelteil eingebundenen Variablen, die nur in den Aktionen des Regelteils benutzt werden dürfen (s. Abschnitte 2.2.5 und 4).

2.3.2 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 $Name_1, ..., Name_n$ paarweise verschiedene Bezeichner sind und jedes r_i ein regulärer Ausdruck über $\Sigma \cup \{\{Name_1\}, \ldots, \{Name_{i-1}\}\}$. Mittels des Ausdrucks $\{Name_i\}$ läßt sich der zugeordnete reguläre Ausdruck r_i in anderen Ausdrücken referenzieren.

Häufig vorkommende Teile eines regulären Ausdrucks lassen sich somit durch einen kurzen prägnanten Bezeichner abkürzen und in weiter unten stehenden regulären Ausdrücken bzw. in den regulären Ausdrücken des Regelteils verwenden.

Beispiel 13 (C-Identifier mit regulärer Definition) *Im folgenden einfachen lex-Programm wird veranschaulicht, wie reguläre Definitionen den regulären Ausdruck zur Erkennung eines Identifiers vereinfachen können.*

```
L   [a-z]|[A-Z]|_
D   [0-9]
%%
{L}({L}|{D})*    printf("%s%s", "Identifier: ", yytext);
```

Beispiel 14 (Gültige C Gleitpunkt-Konstanten) *Das folgende lex-Programm liest eine Liste von Gleitpunkt-Konstanten und entscheidet für jede, ob diese in C gültig ist.*

Eine Gleitpunkt-Konstante in C besteht aus einem ganzzahligen Teil, einem Dezimalpunkt, einem Dezimalbruch, den Zeichen `e` *oder* `E`*, einem ganzzahligen Exponenten mit optionalem Vorzeichen und einem optionalen Typ Suffix, nämlich einem der Buchstaben* `f`*,* `F`*,* `l` *oder* `L`*. Entweder der ganzzahlige Teil oder der Dezimalbruch kann fehlen (aber nicht beide); entweder der Dezimalpunkt oder der Exponent beginnend mit* `e` *oder* `E` *kann fehlen (aber nicht beide). Der Typ der Konstanten wird durch den Suffix bestimmt. Suffixe* `f` *oder* `F` *machen sie zu* `float`*,* `l` *oder* `L` *zu* `long double` *und ohne Suffix zu* `double`*.*

```
Z       [0-9]
ZF      [0-9]+
ZOPT    {Z}*
S       [flFL]?
%%

^({ZOPT}\.{ZF}|{ZF}\.){S}/\n                |
^({ZOPT}\.{ZF}|{ZF}[.]?)[eE][+-]?{ZF}{S}/\n  printf("%s (erlaubt)", yytext);
[ \t]+          ;
```

2.3.3 Startbedingungen

Es können auch Problemstellungen auftreten, bei denen es wünschenswert ist, daß zu einem bestimmten Zeitpunkt nur ein Teil der regulären Ausdrücke angewendet wird und die restlichen inaktiv sind. Dies läßt sich in einem lex-Programm mittels sog. Startbedingungen realisieren. Mit der Anweisung `%s` oder `%S` wird eine Menge von Startbedingungen wie folgt definiert:

```
%s name_1 name_2        name_n
```

Die einzelnen Namen der Startbedingungen müssen durch Leerzeichen, Tabulatorzeichen oder Kommata getrennt werden. Eine griffigere Bezeichnung für eine Startbedingung wäre einfach *Zustand.*

Soll nun eine Startbedingung bzw. ein Zustand `name_i` aktiviert werden, so geschieht dies im Aktionsteil eines regulären Ausdrucks mit der Anweisung

```
BEGIN name_i;
```

Der Normalzustand, in dem alle regulären Ausdrücke aktiv sind, ist `0`, d.h. mit der Anweisung

```
BEGIN 0;
```

wechselt man in den Normalzustand.

Denjenigen regulären Ausdrücken, welche nur bei einem gewissen Zustand bzw. bei gewissen Zuständen aktiv sein sollen, wird der Zustandsname bzw. die Zustandsnamen jeweils eingefaßt in spitze Klammern “<” und “>” und ggf. getrennt durch Kommata vorangestellt:

```
<name_i>regAusdruck            Aktion
```

bzw.

```
<name_i>,<name_j>regAusdruck    Aktion
```

Alle anderen regulären Ausdrücke, denen kein Zustandsname vorangestellt ist, sind *immer* aktiv.

Beispiel 15 (Entfernung von C-Kommentaren) *Das folgende lex-Programm entfernt alle Kommentare aus einem C-Programm.*

```
%s KOMMENTAR

%%
<KOMMENTAR>"*/"    BEGIN 0;
<KOMMENTAR>.       |
<KOMMENTAR>\n      ;
"/*"               BEGIN KOMMENTAR;
```

Nach Lesen der Zeichenkette "`/*`*" wird der Zustand* `KOMMENTAR` *aktiviert. Somit werden also nun auch die regulären Ausdrücke beachtet, die* `<KOMMMENTAR>` *vorangestellt haben. Wird das Ende des Kommentars "*`*/`*" gelesen, wird durch* `BEGIN 0` *wieder der Normalzustand eingeschaltet.*

2.3.4 Zeichensatztabellen

Sollen Eingabetexte verarbeitet werden, die nicht im ASCII-Code vorliegen, so muß in den entsprechenden lex-Programme eine Zeichensatztabelle definiert werden. Die Definition einer Zeichensatztabelle wird mit den Begrenzern `%T` kenntlich gemacht. In ihr wird in jeder Zeile der numerische Wert seinem zugehörigen Zeichen zugeordnet:

```
%T
Zahl     Zeichen
Zahl     Zeichen

%T
```

Ein Beispiel für die Notwendigkeit von Zeichensatztabellen ergibt sich, wenn Eingabetexte im EBCDI-Format oder im Unicode-Format vorliegen.

2.3.5 Lex-interne Tabellengrößen

Bei der Umwandlung eines lex-Programms in das C-Programm `lex.yy.c` werden die regulären Ausdrücke in einen deterministischen endlichen Automaten (EA) umgewandelt, dessen Zustandstabelle verwendet wird, um diesen zu simulieren.

Für die Größe der intern verwendeten Tabellen (Zahl der Zustände, Zahl der Übergänge, etc.) gibt es Standardeinstellungen. Falls man ein lex-Programm mit einer Menge von komplexen regulären Ausdrücken hat, kann es sein daß die Standardeinstellungen vergrößert werden müssen. Dies geschieht auch im Definitionsteil mit den folgenden Anweisungen:

`%n` *zahl*	Zahl der Zustände (Voreinstellung: 4000)
`%a` *zahl*	Zahl der Übergänge (Voreinstellung: 10000)

2.3.6 Debugging-Mode einschalten

In dem generierten Programm `lex.yy.c` sind Debugging-Operationen vorhanden. Diese geben beim Verarbeiten des Eingabetextes Informationen über den aktuellen Zustand des endlichen Automaten aus.

Zu diesem Zweck muß im Definitionsteil die symbolische Konstante `LEXDEBUG` definiert werden:

```
%{
#define LEXDEBUG
%}
```

Ferner muß beim Übersetzen des lex-Programms eine Option angeggeben werden, welche eine interne Variable `debug` auf 1 setzt (s. Abschnitt 3). Wird diese Option durch die vorliegende lex-Implementierung nicht unterstützt, so muß diese Änderung per Hand in `lex.yy.c` vorgenommen werden.

2.4 Benutzerdefinierte Routinen

Alle Zeilen, die im dritten Teil eines lex-Programmes, also nach dem zweiten Paar `%%`, auftreten, werden unverändert in das generierte C-Programmes `lex.yy.c` kopiert.

Dieser Abschnitt ist dafür vorgesehen, Funktionen abzulegen, die in den Aktionen des Regelteils benötigt werden. Z.B. würde man hier eine überschriebene Version der Funktion `yywrap` ablegen.

Beispiel 16 (LineCounter) *Das folgende einfache lex-Programm zählt die Anzahl der Zeilen des Eingabtextes und gibt diese am Ende mittels der Funktion* `yywrap` *aus (s. Abschnitt 2.2.3).*

```
%{
int lineCounter = 0;
%}

%%
\n      lineCounter++;
.       ;

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

3 Bedienung von `lex`

Die Bedienung von `lex` ist sehr einfach. Wie in Abbildung 1 skizziert, wird ein lex-Programm, z.B. `meinProg.lex`, zunächst in ein C-Programm `lex.yy.c` übersetzt. Dies geschieht einfach mit dem Aufruf:

```
lex meinProg.lex
```

U.a. sind folgende Optionen für `lex` möglich:

`-t`	Ausgabe des generierten Programms auf die Standardausgabe
`-v`	liefert eine Statistikzeile über die Benutzung der lex-internen Tabellen
`-d`[4]	setzt die Variable `debug` auf 1; zusätzlich muß die Konstante `LEXDEBUG` definiert werden (s. Abschnitt 2.3.6)

Anschließend muß das C-Programm `lex.yy.c` noch übersetzt werden und die lex-Bibliothek `libl.a` hinzugelinkt werden. Dies läßt sich mit dem Aufruf

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

bewerkstelligen. Das folgende `Makefile` faßt diese Schritte zusammen:

```
LEX_PROGRAM = lineCounter.lex
LEX         = lex
LEX_FLAGS   = -v
LEXLIB      = -ll
CC          = gcc

all: lex.yy.o
        $(CC) lex.yy.o $(LEXLIB)

lex.yy.c: $(LEX_PROGRAM)
        $(LEX) $(LEX_FLAGS) $(LEX_PROGRAM)

clean:
        rm -rf *.o lex.yy.c *~
```

4 Aufbau von `lex.yy.c`

In diesem Abschnitt wird der Aufbau des von `lex` generierten C-Programms `lex.yy.c` skizziert. Dieses enthält folgende Abschnitte:

1. Lex-interne Variablen-Deklarationen und Makro-Definitionen
2. C-Programmfragmente, die im Definitionsteil angegeben werden (s. Abschnitt 2.3.1)
3. Die Funktion `yylex`, welche wie folgt gegliedert ist
   (a) Benutzerdefinierte Variablen des Regelteils (s. Abschnitt 2.2.5)
   (b) Switch-Anweisung, welche die Aktionen des Regelteils ausführt
4. Benutzerdefinierte Funktionen, die im dritten Abschnitt des lex-Programms angegeben worden sind (s. Abschnitt 2.4).

[4] Diese Option ist nicht in allen lex-Implementierungen verfügbar.

5. Zustandstabellen, die von `lex` aus den regulären Ausdrücken generiert worden sind.
6. Inhalt der Datei `ncform`, die sich im lex-Installationspfad befindet. Diese enthält im wesentlichen die Funktion `yylook()`, welche die Zustandstabellen durchläuft.

A Ausführliches Beispiel

In diesem ausführlichen Beispiel wird nochmal ein Großteil der besprochenen Bestandteile eines lex-Programms demonstriert.

Das hier diskutierte lex-Programm überprüft C-Programme auf Klammerungsfehler und fehlerhafte Kommentare. Dies wird durch Zählung von öffnenden und schließenden Klammern und Kommentarzeichen bewerkstelligt. Es wird *nicht* die syntaktische Korrektheit der Klammerung überprüft.

Am Anfang einer jeden Zeile wird die Schachtelungstiefen der einzelnen Konstrukte, die unmittelbar vor dieser Zeile vorlagen, ausgegeben. Danach wird die gelesene Zeile ausgegeben. Die Zeichen `(`, `)`, `{` und `}` gehören nur dann zu einer Klammerung, wenn sie nicht in einem Kommentar, in einer Zeichenkette oder als char-Konstante auftreten.

Beispiel 17 (C-Programm Schachtelung)

```
%{
int geschweift=0,
    klammer   =0,
    kommentar =0;
char zeichen;
%}

%s KOMMENTAR

%%
^\n             { printf("%5d: {%d} (%d) /*%d*/ |\n",
                        yylineno, geschweift, klammer, kommentar);
                }
^[^\n]+         { printf("%5d: {%d} (%d) /*%d*/ |",
                        yylineno, geschweift, klammer, kommentar);
                  yyless(0);
                }

"/*"            { kommentar++; ECHO; BEGIN KOMMENTAR; }
<KOMMENTAR>"*/" { kommentar--; ECHO; BEGIN 0; }
<KOMMENTAR>.    |
<KOMMENTAR>\n   ECHO;

\"              { ECHO;
                  while ((zeichen=input()) != '"'){
                     putchar(zeichen);
                     if (zeichen == '\\')
                        putchar(input());
                  }
                  putchar(zeichen);
                }

\{              { geschweift++; ECHO; }
```

```
\}              { geschweift--; ECHO; }
\(              { klammer++; ECHO; }
\)              { klammer--; ECHO; }

'\"'            |
"'{'"           |
"'}'"           |
"'('"           |
"')'"           |
.               |
\n              ECHO;

%%
yywrap()
{
  if (klammer)
    printf("Klammerung () nicht ausgeglichen\n");
  if (geschweift)
    printf("geschweifte Klammerung { } nicht ausgeglichen\n");
  if (kommentar)
    printf("Kommentar nicht abgeschlossen\n");
}

int main(int argc, char *argv[])
{
  if (freopen(argv[1], "r", stdin) == NULL){
    fprintf(stderr, "Datei %s kann nicht geoeffnet werden\n", argv[1]);
    exit(1);
  }
  yylex();
  return(0);
}
```

B Referenzen

Der Inhalt dieser Einführung orientiert sich eng an [Her99]. In diesem Buch befinden sich viele ausführliche Beispiele, die die Anwendung von `lex` illustrieren.

`lex` ist unter den GNU Nutzungsbedingungen unter dem Namen `flex` verfügbar [Fle97]. Es existiert ein ausführliches Handbuch dazu unter [Fle95].

Eine guten Überblick über die verfügbaren Versionen von `lex` auf verschiedenen Plattformen sowie einen FAQ enthält die Seite [Lex].

Literatur

[Fle95] Flex, A fast scanner generator, Edition 2.5. WWW: `http://www.gnu.org/manual/flex-2.5.4/`, March 1995.

[Fle97] Flex, A fast scanner generator, Code. WWW: `ftp://ftp.gnu.org/gnu/flex/`, 1997.

[Her99] Helmut Herold. *LINUX-UNIX-Profitools, awk, sed, lex, yacc und make.* Linux/Unix und seine Werkzeuge. Addison Wesley, 3rd edition, 1999.

[Lex] Lex and Yacc overview, FAQ. WWW: `http://www.uman.com/lexyacc.shtml`.