



Reguläre Ausdrücke

Aufbau von Lex-Programmen

Arbeiten mit Lex



Ein regulärer Ausdruck beschreibt eine nicht leere Menge von Zeichenfolgen (Textmuster).

Bei der lexikalischen Analyse mit Lex wird zu einer Zeichenfolge geprüft, ob sie zu einem vorgegebenen Textmuster passt.

Wenn dafür mehrere Alternativen bestehen, dann wird Immer die längste mögliche verwendet.



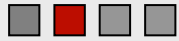
Reguläre Ausdrücke

Ein regulärer Ausdruck ist eine Folge "terminaler" Textzeichen und Metazeichen.

- Terminale Zeichen sind Zeichen, die in den Textmustern direkt auftreten dürfen.
- Metazeichen sind Operatoren, mit denen komplexe Textmuster beschrieben werden können.

Literale

Wenn ein regulärer Ausdruck aus einer Folge terminaler Zeichen besteht, dann sprechen wir von einem Literal. Im einfachsten Fall besteht er aus genau einem Zeichen.



Zeichenklassen

Die folgenden Operatoren stellen reguläre Ausdrücke dar, die jeweils eine Klasse von Zeichen beschreiben:

.

[]

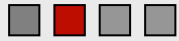
[abc]

[a-z]

[a-zAEIOU]

[-abc]

[^0-9]



Wiederholungsoperatoren

Es sei a ein regulärer Ausdruck. Die folgenden Operatoren machen Angaben über das mehrfache Vorkommen des regulären Ausdrucks a .

a^*

$a?$

a^+



Kontextoperatoren

Zeichenmanipulationssprachen erlauben meist die Angabe, in welchem Kontext der gesuchte Ausdruck vorkommen soll. Diese Möglichkeit besteht in beschränktem Maße auch für reguläre Ausdrücke. Hier kann spezifiziert werden, ob das gesuchte Textmuster am Anfang oder am Ende einer Zeile vorkommen soll. Der so gebildete Ausdruck ist wieder regulär.

$\wedge a$

$a\$$



Komplexe Ausdrücke

Aus regulären Ausdrücken a und b werden komplexere gebildet wie folgt:

ab

$a \mid b$



Operator-Prioritäten

Die Reichweite der Operatoren ist auf die unmittelbare linke oder rechte Umgebung beschränkt. Die Operatorfolge '[..]', '?', '+', '*', Verkettung und '|' ist nach fallenden Prioritäten geordnet. Operatoren gleicher Priorität werden von links nach rechts bearbeitet. Abweichungen von der durch diese Regeln gegebenen Verarbeitungsreihenfolge können durch Klammerung mit runden Klammern erreicht werden.



Maskierung von Metazeichen

Wenn das Zeichen '\' vor einem Metazeichen steht, dann wird es als Terminalzeichen behandelt.

`^\.` steht für einen Punkt am Satzanfang

`^.` steht für ein beliebiges Zeichen am Satzanfang.

Wenn das '\'-Zeichen vor einem Terminalzeichen steht, so hat dies Keinen Effekt. 'a' und '\a' sind also äquivalent.



Lex-Kurzbeschreibung

LEX ist ein Generator zur Erstellung von Programmen, deren Ablauf durch das Vorkommen von Textmustern im Eingabedatenstrom gesteuert wird. Die Textmuster werden durch reguläre Ausdrücke beschrieben. Als Hauptanwendungsgebiete lassen sich anführen:



Lexikalische Analyse

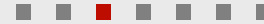
Zerlegung von Quellprogrammen oder Kommandosequenzen in Befehle, Operatoren, Argumente usw., also in Token. Der Name Lex ist von dieser Aufgabe abgeleitet.

Datentransformation

Umwandlung von Zeichenfolgen des Eingabedatenstroms in korrespondierende Ausgabezeichenfolgen, ähnlich wie bei Editoren.

Datenauswertung

Erstellung von Statistiken auf der Basis von einzelnen Zeichen oder Tokens, z.B. Zählen von Zeichen- und Worthäufigkeiten, Feststellen der Übergangswahrscheinlichkeiten zwischen Zeichen usw.



Ein LEX Quellprogramm besteht aus den folgenden Teilen:

Definitionen

%%

Regeln

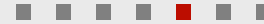
%%

Funktionen



Der Definitionsteil

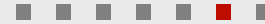
Definitionen sind optional. Der Definitionsteil enthält Vereinbarungen, die vom Programmgenerator Lex für die Programmerzeugung benutzt werden. Außerdem können hier speziell gekennzeichnete Anweisungen der Wirtssprache stehen, die von LEX in das Zielprogramm übernommen werden.



Der Regelteil

Der Regelteil muss vorhanden sein. Die Regeln enthalten Benutzerangaben in Form einer Tabelle:

- die linke Spalte der Tabelle enthält reguläre Ausdrücke und
- die rechte Spalte der Tabelle enthält die zugeordneten Aktionen.



Der Funktionsteil

Im Funktionsteil werden in erster Linie Routinen der Wirtssprache untergebracht, die vom Regelteil aufgerufen werden. Wenn vor oder nach dem Scannen des Eingabe-Stroms Aktionen durchgeführt werden sollen, dann kann dies z.B. mit einem main()-Programm folgender Art geschehen:

```
int main()
{
    /* Aktionen vor dem Scannen */
    yylex(); /* Scanner starten */
    /* Aktionen nach dem Scannen */
}
```



Benutzerdefinierte Aktionen im Regelteil

Der Benutzer kann im Regelteil beliebige Aktionen des Analyseprogramms spezifizieren. Dafür sind alle Anweisungen der Host-Sprache zugelassen. Es können eigene Variablen verwendet werden, aber auch implizit von Lex generierte Datenobjekte.



Beispiel:

```
%{
/* Vereinbarungen */
int i=0;
}%
/* Definitionen */
/* hier keine Definitionen noetig ... */
%%
/* Regeln */
Colour          { printf ("color"); i++; }
%%
/* Funktionen */
yywrap (void)
{
printf ("'colour' wurde %3d-mal in 'color, umgewandelt\n", i);
}
```



Funktionen:

- yywrap()** Bei EOF wird yywrap() überprüft, bei true = Ende
bei false = nächste Datei
- yylex()** Funktion zum Starten/Fortsetzen der Analyse
- *yytext** Pointer oder Array auf letzten Token
- yylen** Länge des aktuellen Token
- yyless(n)** Alle Zeichen außer den ersten n wieder zurück in den Eingabestream
- yyterminate()** Beendet den Scanner und liefert eine 0 zurück
- input()** Lädt nächstes Zeichen aus dem Eingabestream



Aufrufoperationen:

- b** erzeugt eine Backup-File lex.backup
- d** Läßt Scanner in Debug-Mode laufen
- h/-?** Hilfe wird auf stdin angezeigt
- i** case-insensitive, also Groß/Kleinschreibung wird ignoriert
- p** Performancereport
- ooutput** Ausgabe-Dateiname: output (Statt lex.yy.c)
- ++** C++-Generator, aber mit einigen Risiken verbunden (Kompilierungsproblemen)



Häufigkeitsanalyse:

```
%{  
#define MAXANZ 36  
long int anzahl[MAXANZ]={0};  
long int gesamt=0;  
%}  
/* Definitionen */  
LCHAR [a-z]           /*Alle Kleinbuchstaben*/  
UCHAR [A-Z]          /*Alle Großbuchstaben */  
DIGIT [0-9]          /*Alle Ziffern      */  
IGNOR [^a-zA-Z0-9]  /*Alle anderen Zeichen*/  
%%
```



*/*Regeln*/*

```
{LCHAR} {anzahl[(*yytext)-'a']++; gesamt++;}
```

```
{UCHAR} {anzahl[*yytext-'A']++; gesamt++;}
```

```
{DIGIT} {anzahl[*yytext-'0'+26]++; gesamt++;}
```

```
{IGNOR}
```

```
%%
```

```
/* ein Kleinbuchstabe*/
```

```
/* ein Großbuchstabe */
```

```
/* eine Ziffer*/
```

```
/* andere Zeichen ignorieren*/
```



```
/* Funkton-Teil, Platz für Funktionen der Host language*/
int main()
{
  int i=0;
  char c='a';
  printf("\nHaeufigkeitsanalyse");           /*Überschrift*/
  printf("\n-----\n");
  yylex();                                   /* Analyse durchführen*/
  while(i<MAXANZ-10)      /*Ausgabe der Buchstaben und ihrer Häufigkeiten */
  {
    printf("\nAnzahl von \'%c\' = %-4d",c++,anzahl[i++]);
  }
  while(i< MAXANZ)      /*Ausgabe der Ziffern und ihrer Häufigkeiten*/
  {
    printf("\nAnzahl von \'%d\' = %-4d",i-26,anzahl[i++]);
  }
  printf("\nGesamtanzahl Zeichen: %d\n\n",gesamt);
}
```



Kurz-Einführung in Lex



Ende



Danke