# 5 Lexikalische Analyse mit `lex`

## 5.1 Einführung

**Lexer:**
Liest Eingabestrom und zerlegt ihn in Token.

**Token:**

- Endprodukt
- Weiter verarbeitet, z.B. von Parser (Yacc)

**Lexer** heißen auch **„Scanner“** oder **„lexikalische Analysatoren“**.

`lex` ist ein **Generator** für Lexer.

Wir benutzen hier `flex`, den Lexer des **Gnu-Projektes**.

Das folgende **Beispiel sucht** nach dem **String** `<username>` und ersetzt ihn durch den aktuellen Benutzernamen. **Alle andere Eingabe** wird direkt in die Ausgabe **kopiert**:
`username.l`:

```
%{
#include <stdio.h>
#include <stdlib.h>
%}
%option main
%%
"<username>"    printf("%s", getenv("USER"));
```

**Interessant** ist hier zunächst **nur** die **letzte Zeile**:

**Vorne** steht ein **Muster**, **hinten** eine **Aktion**.

Es gibt eine **eingebaute Default-Aktion**. Sie **druckt** den **aktuellen Buchstaben** einfach aus.

►► `make username`

Anmerkung: Dies ist ein schönes Beispiel, wie man **ohne Makefile**, nur mit `make`s eingebauten Regeln, alles **übersetzen** kann.

Lex **generiert** ein **C-Programm**.

`username.txt`:

```
Hallo <username>, dies ist ein Beispieltext.
```

►► `./username < username.txt`

## 5.2 Grundaufbau einer Lex-Datei

**Lex-Datei:**
Definitionen
`%%`
Regeln
`%%`
evtl. Unterprogramme

### 5.2.1 Definitionen

In den Definitionen kann **einiges für Lex** definiert werden, wie oben z.B. eine **Option** `main`.

Außerdem kann man **Header-Files** für den generierten C-Code angeben, in „`%{`" und „`%}`".

### 5.2.2 Regeln

**Regel:**
Muster Aktion

Jede **Regel** besteht aus einem **Muster** und einer **Aktion**, durch **Whitespace** getrennt.

### 5.2.3 Muster

Muster sind **reguläre Ausdrücke**, mit einigen **Erweiterungen**.

Text in **doppelten Anführungszeichen** ist **gequotet**. Ich **empfehle** das für alle **festen Token**. Im obigen username-Beispiel war es z.B. schon notwendig wg. der spitzen Klammern.

### 5.2.4 Aktionen

Eine Aktion ist ein **Stück C-Code**.

Entweder ist sie eine **einzelne C-Anweisung**, mit **Semikolon** abgeschlossen, wie oben gesehen.

Oder sie ist ein **C-Block** in **geschweiften Klammern**. Dann darf sie auch über **mehrere Zeilen** gehen.

### 5.2.5 Unterprogramm-Abschnitt

Im **optionalen** Unterprogramm-Abschnitt kann man **Hilfsfunktionen** definieren.

Im obigen **Beispiel** war er **leer** und **fehlte**.

Hierhin kommt auch die **main()-Funktion**. Im **Beispiel** wurde eine einfache main()-Funktion allerdings auch **automatisch generiert**, durch die Option `main`.

### 5.2.6 Beispiel: Wort-Zähl-Programm

`wcount.l`:

```
%{
#include <stdio.h>
int num_lines = 0, num_chars = 0;
%}
%option noyywrap
%%
"\n"            { num_lines++; num_chars++; }
.               { num_chars++; }
%%
int main() {
    yylex();
    printf("%d Zeilen, %d Buchstaben\n", num_lines, num_chars);
}
```

Der **Punkt** ist ein regulärer Ausdruck, der genau auf einen beliebigen Buchstaben (außer Newline) paßt.

Hier haben wir eine **selbstgeschriebene main()-Funktion**. Sie ruft **yylex()** auf, die generierte Lexer-Funktion.

▶▶ `make wcount`

▶▶ `./wcount < wcount.l`

Zum Vergleich:

▶▶ `wc wcount.l`

### 5.2.7 Beispiel: Scanner für Pascal-artige Sprache, mit weiteren Lex-Konstrukten

`pcount.l`:

```
%{
#include <math.h>
#include <stdio.h>
%}
%option noyywrap
DIGIT   [0-9]
ID      [a-zA-Z][a-zA-Z0-9_]*
%%
{DIGIT}+         { printf("Eine Ganz-Zahl: '%s' (%d)\n",
                   yytext, atoi(yytext));
                 }
{DIGIT}+"."{DIGIT}*     {
                   printf("Eine Fliesskomma-Zahl: '%s' (%g)\n",
                          yytext, atof(yytext));
                 }
"if"|"then"|"begin"|"end"|"procedure"|"function"|"="|";"        {
                   printf("Ein Schluesselwort: '%s'\n", yytext);
                 }
{ID}             { printf("Ein Bezeichner: '%s'\n", yytext); }
"+"              |
"-"              |
"*"              |
"/"              { printf("Ein Operator: '%s'\n", yytext); }
"{"[^}\n]*"}"    /* ueberspringe einzeilige Kommentare */
[ \t\n]+         /* ueberspringe White-Space */
.                { printf("Unbekanntes Zeichen: '%s'\n", yytext);}
%%
int main(int argc, char **argv) {
    if(argc > 1)
        yyin = fopen(argv[1], "r");
    yylex();
}
```

Dies Programm **liest** ein **Pascal-artiges Programm** ein und **druckt,** was es an **Token** findet.

Es wird auch **etwas bearbeitet**. So werden die **Zahlen** aus den Strings **konvertiert**.

Hier gibt es **einiges neues**:

Die Variable `yytext` enthält immer den String des **aktuellen Tokens**.

Im Definitions-Teil kann man **Abkürzungen** für Teile regulärer Ausdrücke **definieren**.

Zur **Benutzung** muß man die definierten Namen **in geschweifte Klammern** einschließen.

Die **Schlüsselworte** werden mit „`|`" getrennt, der **Alternative der regulären Ausdrücke**.

Die Operatoren haben „`|`" **als Aktion**. Das heißt, daß die Aktion für diese Regel dieselbe ist wie für die folgende Regel. Der **Effekt** ist ungefähr wie eine reguläre-Ausdruck-Alternative auf oberster Ebene.

In der main()-Funktion wird der Datei-Pointer **yyin** verwendet. Lex liest die **Eingabe** von diesem Datei-Pointer. Per **Default** ist er gleich **stdin**. Man kann ihn aber auch wie hier umsetzen.

Eine **Eingabedatei zum Scanner**:

`pcount.txt`:

```
procedure foo
    function bar
    begin
        bar = 700000000.1
    end
begin
    tmp = bar
end
{ Beginn des Hauptprogramms: }
begin
    foo;
    zahl = 000042 + 8
end
```

►► `make pcount`

►► `./pcount < pcount.txt`

## 5.3 Online-Aufgabe: Einfacher Taschenrechner

Schreibe einen einfachen Taschenrechner. Man soll **Fließkommazahlen und Operatoren eingeben** können. Operatoren sind „+", „−", „∗", „/" und „C".

Der Rechner soll einen **Akkumulator** für die aktuelle Zahl haben. Wenn eine **neue Zahl** eingegeben wurde, dann soll sie über dem **aktuellen Operator** mit dem Akkumulator **verknüpft** werden. Anschließend soll der neue Akkumulator **ausgegeben** werden.

Der **Operator „C"** übertragt die neue Zahl einfach in den Akkumulator. Bei seiner Eingabe wird außerdem der Akkumulator auf 0 gesetzt.