# 4. Der Scanner-Generator lex

## Inhalte der Vorlesung

1. Einführung
2. Lexikalische Analyse
3. Der Textstrom-Editor sed
4. → Der Scanner-Generator lex (2 Termine)
5. Syntaxanalyse und der Parser-Generator yacc (3 T.)
6. Syntaxgesteuerte Übersetzung
7. Kontextanalyse
8. Transformation und Code-Erzeugung (?)
9. Übersetzungssteuerung mit make



## 4. Der Scanner-Generator lex

4.1 Grundlagen
4.2 Fortgeschrittenes



## 4.1 lex: Grundlagen

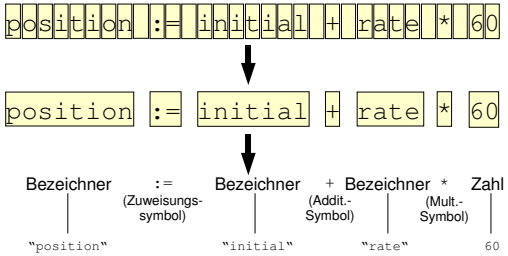
4.1.1 Einführung
4.1.2 Aufbau einer lex-Datei
4.1.3 Einfacher Aufruf von flex
4.1.4 Basiskonstrukte
4.1.5 Weitere nützliche Konstrukte
4.1.6 Übung: Einfacher Taschenrechner
4.1.7 Der Match-Algorithmus
4.1.8 Kommunikation mit einem Parser
4.1.9 Übung: Konversion römischer Zahlen

## 4.2 lex: Fortgeschrittenes


4.2.1 Wiederholung und Festigung
4.2.2 Scanner-Zustände, Grundlagen
4.2.3 Umlenken der Ein- und Ausgabe
4.2.4 Reguläre Ausdrücke, Fortgeschrittenes
4.2.5 Scanner-Zustände, Fortgeschrittenes
4.2.6 Mehrere Lexer in einem Programm
4.2.7 Aufruf und Datei-Optionen von flex
4.2.8 flex und andere Lexer




## Lexikalische Analyse




- Gruppierung der Eingabezeichen in Lexeme
  - Leerzeichen werden entfernt
- Zuordnung Lexem → Symbol
  - Symbole (Token): Bezeichner, `:=`, `+`, `*`, Zahl, ...
  - Symbol hat z.T. Wert als Attribut



## Varianten von lex

- lex von AT&T (das Original)
- diverse kommerzielle Versionen
- POSIX-Standard für lex
- flex des Gnu-Projekts
  - fast vollständig POSIX-kompatibel
  - Obermenge von lex/AT&T
  - für viele Plattformen



## 4.1 lex: Grundlagen


4.1.1 Einführung
→ 4.1.2 Aufbau einer lex-Datei
4.1.3 Einfacher Aufruf von flex
4.1.4 Basiskonstrukte
4.1.5 Weitere nützliche Konstrukte
4.1.6 Übung: Einfacher Taschenrechner
4.1.7 Der Match-Algorithmus
4.1.8 Kommunikation mit einem Parser
4.1.9 Übung: Konversion römischer Zahlen

## Aufbau einer lex-Datei

*Definitionen*
%%
*Regeln*
%%
*Unterprogramme* } optional



## Einsetzen des Benutzernamens

- Eingabe soll zur Ausgabe kopiert werden
- aber: „`<username>`“ → aktueller Benutzername

Demo



## Einsetzen des Benutzernamens: Lösung

- username.l:

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

  - Regel: vorne Muster, hinten Aktion
  - Default-Regel: druckt aktuelles Zeichen
  - Definitionen:
    - C-Definitionen in `%{`, `%}`
    - lex-Option hinter `%option`



## Einfacher Aufruf von flex

- `flex username.l`
  - Default-Ausgabedatei: lex.yy.c
  - Option „`-t`“: Ausgabe nach StdOut
- Tipp: „`make <progname>`“ ohne Makefile erzeugt Executable

## Kommentare im Regelteil

- wie in C:

```
/* bla bla
   bla bla */
```

- aber: *muß* eingerückt sein!
  - lex-Regeln:
    - niemals eingerückt
    - Aktion muß auf gleicher Zeile beginnen
  - im Definitions- und Unterprogrammteil nicht nötig



## 4.1 lex: Grundlagen


4.1.1 Einführung
4.1.2 Aufbau einer lex-Datei
4.1.3 Einfacher Aufruf von flex
→ 4.1.4 Basiskonstrukte
4.1.5 Weitere nützliche Konstrukte
4.1.6 Übung: Einfacher Taschenrechner
4.1.7 Der Match-Algorithmus
4.1.8 Kommunikation mit einem Parser
4.1.9 Übung: Konversion römischer Zahlen




## Reguläre Ausdrücke von lex

- wichtigste Unterschiede zu sed/grep:
  - `"while"`: Literale können/sollten in `""` stehen
  - `\n` – Newline: (fast) normales Zeichen
    - `[^abc]` paßt auch auf Newline
      - ggf. Abhilfe: `[^abc\n]`
    - `.` (Punkt) paßt nicht auf Newline
  - kein Backslash vor `+`, `?`, `{`, `}`, `(`, `)`
  - kein `\<`, `\>`
  - kein `\1`, `\2`, ...



## Aufbau einer Aktion

- Stück C-Code
  1. einzelne C-Anweisung (mit Semikolon)
  2. C-Block (in geschweiften Klammern)
     - dann auch über mehrere Zeilen

## Die main()-Funktion

- generiert
  - Beispiel oben
  - `%option main` im Definitionsteil
- eigene
  - im Unterprogrammteil

Demo



## Wort-Zählprogramm

- `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);
}
```

- `make wcount`
- `./wcount < wcount.l`
- zum Vergleich: `wc < wcount.l`



## 4.1 lex: Grundlagen


4.1.1 Einführung
4.1.2 Aufbau einer lex-Datei
4.1.3 Einfacher Aufruf von flex
4.1.4 Basiskonstrukte
→ 4.1.5 Weitere nützliche Konstrukte
4.1.6 Übung: Einfacher Taschenrechner
4.1.7 Der Match-Algorithmus
4.1.8 Kommunikation mit einem Parser
4.1.9 Übung: Konversion römischer Zahlen




## Scanner für eine Pascal-artige Sprache

- liest Pascal-artiges Programm
- druckt, was es an Token findet
- bearbeitet:
  - Strings → Zahlen
- meldet lexikalische Fehler

Demo

## Scanner für eine Pascal-artige Sprache: Eingabedatei

- pcount.txt:

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



## Scanner für eine Pascal-artige Sprache: Lösung

- pcount.l:

```
%{
#include <math.h>
#include <stdio.h>
%}
%option main
%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);}
```



## Weitere nützliche Konstrukte

- Abkürzungen für reguläre Ausdrücke
  - im Definitionsteil definieren
  - Benutzung: in geschweiften Klammern
- mehrere Ausdrücke für eine Aktion
  - „|“ anstelle einer Aktion
- Variable yytext
  - aktuelles Token
- Makro ECHO
  - druckt yytext



## Übung: Einfacher Taschenrechner

- Eingaben:
  - Fließkommazahl *nnnn.nnnn* oder *nnnn*
  - **+**, **−**, *****, **/**, **c**
- ein Akkumulator
- Eingabe in Infix, ohne Prioritäten
  - Operator: merken
  - Zahl: Akku Op Zahl → Akku
  - Operator **c**
    - bei Ausführung: Zahl → Akku
- Ausgabe: neuer Akku

Demo

## 4.1 lex: Grundlagen


4.1.1 Einführung
4.1.2 Aufbau einer lex-Datei
4.1.3 Einfacher Aufruf von flex
4.1.4 Basiskonstrukte
4.1.5 Weitere nützliche Konstrukte
4.1.6 Übung: Einfacher Taschenrechner
→ 4.1.7 Der Match-Algorithmus
4.1.8 Kommunikation mit einem Parser
4.1.9 Übung: Konversion römischer Zahlen




## Der Match-Algorithmus

1. Suche Muster, das auf Präfix der Eingabe paßt
   - mehrere passen:
     1) längstes Muster (longest match)
     2) erstes Muster (first match)
2. setze `yytext` auf Text, `yyleng` auf Länge
3. führe Aktion aus
4. gehe zu 1.



## Der Match-Algorithmus (2)

- kein Muster paßt: Default-Regel
  - ein Zeichen paßt
  - das Zeichen wird gedruckt
- einfachstes flex-Programm:

```
%option main
%%
```

Demo



## Einfachstes flex-Programm

- `make trivial`
- `./trivial < pcount.txt`

## Kommunikation mit einem Parser

- Unterbrechung der Scan-Schleife:
  `return n;`
  - Nummer des Symbols in `n`
  - weitere Informationen in globalen Variablen (`yytext`, ...)
- Fortsetzung der Scan-Schleife:
  `yylex();`
- Symbolnummern in gemeinsamer Header-Datei

Demo



## Kommunikation mit einem Parser: Lösung

- „Pascal“-Scanner → „Pascal“-Übersetzergerüst
  - Scanner und Parser(-Gerüst)
    - Parser akzeptiert *jede* Folge von Token
- `pcomp-scan.l`
- `pcomp-parse.c`
- `pcomp.tab.h`
- `make -f pcomp.mk`



## Kommunikation mit einem Parser: Lösung (2)

- `pcomp-scan.l:`

```
%{
#include <math.h>
#include <stdio.h>
#include "pcomp.tab.h"

int yylval_i;
double yylval_d;
char *yylval_s;
%}
%option noyywrap
DIGIT    [0-9]
ID       [a-zA-Z][a-zA-Z0-9_]*
%%
(DIGIT)+        { yylval_i = atoi(yytext);
                  return TOK_INT;
                }
(DIGIT)+"."(DIGIT)*     {
                  yylval_d = atof(yytext);
                  return TOK_DOUBLE;
                }
"if"            return TOK_IF;
"then"          return TOK_THEN;
"begin"         return TOK_BEGIN;
"end"           return TOK_END;
"procedure"     return TOK_PROCEDURE;
"function"      return TOK_FUNCTION;
"="             return TOK_EQUAL;
";"             return TOK_SEMICOLON;
{ID}            {
                  yylval_s = yytext;
                  return TOK_ID;
                }
"+"             return TOK_PLUS;
"-"             return TOK_MINUS;
"*"             return TOK_TIMES;
"/"             return TOK_DIV;
"{"[^\n]*"}"    /* ueberspringe einzeilige Kommentare */
[ \t\n]+        /* ueberspringe White-Space */
.               { printf("Unbekanntes Zeichen: '%s'\n", yytext);}
```



## Kommunikation mit einem Parser: Lösung (3)

- `pcomp-parse.c:`

```
#include <stdio.h>
#include "pcomp.tab.h"

extern int yylval_i;
extern double yylval_d;
extern char *yylval_s;

int main() {
    int tok;
    /* Dieser (ziemlich sinnlose) Parser akzeptiert "jede" Folge von Token. */
    while (tok=yylex()) {
        switch (tok) {
        case TOK_INT:
            printf("Eine Ganz-Zahl: %d\n", yylval_i);
            break;
        case TOK_DOUBLE:
            printf("Eine Fliesskomma-Zahl: %g\n", yylval_d);
            break;
        case TOK_IF:
        case TOK_THEN:
        case TOK_BEGIN:
        case TOK_END:
        case TOK_PROCEDURE:
        case TOK_FUNCTION:
        case TOK_EQUAL:
        case TOK_SEMICOLON:
            printf("Ein Schluesselwort.\n");
            break;
        case TOK_ID:
            printf("Ein Bezeichner: '%s'\n", yylval_s);
            break;
        case TOK_PLUS:
        case TOK_MINUS:
        case TOK_TIMES:
        case TOK_DIV:
            printf("Ein Operator.\n");
            break;
        default:
            printf("Interner Fehler: Unbekanntes Token %d!\n", tok);
            exit(1);
        }
    }
    return 0;
}
```

## Kommunikation mit einem Parser: Lösung (4)

- `pcomp.tab.h`:

```
#define TOK_INT          1
#define TOK_DOUBLE       2
#define TOK_IF           3
#define TOK_THEN         4
#define TOK_BEGIN        5
#define TOK_END          6
#define TOK_PROCEDURE    7
#define TOK_FUNCTION     8
#define TOK_EQUAL        9
#define TOK_SEMICOLON    10
#define TOK_ID           11
#define TOK_PLUS         12
#define TOK_MINUS        13
#define TOK_TIMES        14
#define TOK_DIV          15
```



## Übung: Konversion römischer Zahlen

- römische Zahlen → arabische Zahlen
- andere Zeichen unverändert
- römische Zahlen:
  - I, II, III → 1, 2, 3
  - IV → 4, V → 5
  - IX → 9, X → 10, XX → 20, XXX → 30
  - XL → 40, L → 50
  - XC → 90, C → 100, CC → 200, CCC → 300
  - CD → 400, D → 500
  - CM → 900, M → 1000, MM → 2000, ...



## Übung: Konversion römischer Zahlen (2)

- Annahmen:
  - nur Großbuchstaben sind römische Zahlen
  - anderer Text nur in Kleinbuchstaben
  - alle römischen Ziffern sind korrekt angeordnet
- Vorgehen:
  - addiere römische Ziffern zu Akku
  - anderes Zeichen: drucke (ggf.) Akku, 0 → Akku
  - „CCC“ soll als *ein* Lexem erkannt werden, Wert ist yyleng * 100
    - analog „III“, „XXX“ und „MMMMM“

Demo



## Übung: Konversion römischer Zahlen (3)

- Zusatzaufgabe:
  - alle illegalen Folgen von römischen Ziffern sollen zu einer Fehlermeldung führen
    - Ziffern in absteigender Reihenfolge sortiert
    - nie mehr als 3 gleiche Ziffern hintereinander, außer M
  - Hinweise:
    - Fehlerbehandlungsregeln nach normalen Regeln: „first match“ macht das Leben einfacher
    - nach X, XX, XXX oder XL darf keine Ziffer folgen, die den Wert von X oder höher hat
    - nach L darf keine Ziffer folgen, die den Wert von L oder höher hat, und auch nicht XL oder XC

Demo

# 4. Der Scanner-Generator lex

4.1 Grundlagen
4.2 Fortgeschrittenes