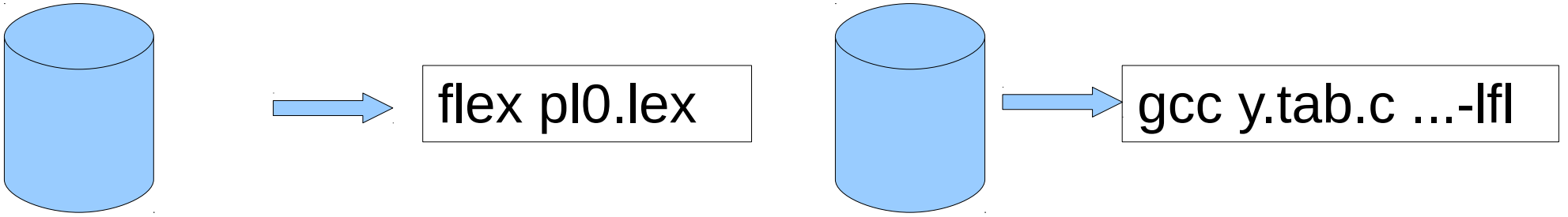
lex / flex

http://flex.sourceforge.net/manual/index.html#Top

- Klassische Unix Werkzeuge
- Lexergenerator für Compiler/Interpreter
- Konzipiert für das Zusammenwirken mit den Parsergeneratoren yacc und bison
- Scannergenerator für Commandlinearguments
- Morpheme/Token werden mit Hilfe regulärer Ausdrücke beschrieben
- Lexer kann als Unterprogramm jedes erkannte Token liefern oder als Pass die gesamte Quelle scannen.
- Erzeugt ein c-Programm

Verwendung



- Option erlauben Variationen flex -o t1.c t1.lex
- Aufpassen: Optionen vor lex-Datei angeben!!!
  - -o outputfile
  - Generierung einer C++-Scannerklasse

    (--yyclass=NAME –c++)
  - Option -i generiert einen nicht casesensitiven Scanner

  - Weitere Optionen unter man flex
  - Generierung eines c-headerfiles

    flex --header-file=lex.h tz5.lex

Reguläre Ausdrücke

- Beschreiben Zeichenfolgen eines Alphabetes
- Operationen zur Beschreibung sind dabei:
  - Die Aneinanderreihung (Konkatenation)
  - Die Unterscheidung (Alternative)
  - Die Wiederholung (Iteration)
  - Die Verneinung (Negation)
- Häufig gelten dabei Vorrangregeln, wobei die Operatorpriorität von Konkatenation zu Iteration steigt.
- Im Bedarfsfall kann geklammert werden

Reguläre Ausdrücke

- Neben den Operationen müssen auch die vorkommenden Zeichen beschrieben werden.
- Zeichenfolgen (abc, a1, while, 123)
- Klassen von Zeichen ([0-9], [A-Z], [1,3,5,7,9])
- Beliebiges Zeichen außer newline (.)
- Escape Sequenz \... (\n, \t, \0x41)

Besondere Zeichen

- . Jedes Zeichen außer \n wird akzeptiert
- ^ als erstes Zeichen: Anfang einer neuen Zeile
- [^...] alles außer … wird akzeptiert
- $ Als letztes Zeichen eines Ausdrucks wird das Zeilenende akzeptiert
- <...> Markiert am Regeleanfang (1. Zeichen) einen speziellen Status, der mit BEGIN eingestellt wird. Die Regel ist nur gültig, wenn der angegebene Status eingeschaltet ist.

Wiederholungen...

- * der vor * stehende (Teil-)Ausdruck kommt 0x, 1x oder mehrfach vor
- + der vor + stehende (Teil-)Ausdruck kommt mindestens ein mal vor
- ? der vor ? stehende (Teil-)Ausdruck kann vorkommen
- {n} der vor {n} kommt n mal vor
- {n,m} der vor {n,m} kommt mindestens n, höchstens m mal vor
- "..." markiert eine Zeichenkette, die in dieser Form zu akzeptieren ist

Reihenfolge der Pattern

- Reihenfolge der Patternlines ist relevant
- Patternlines werden von oben nach unten getestet.
- Ist ein Token erkannt, so werden die dazugehörigen Zeichen aus dem Eingabestrom entfernt.
- Daraus folgt:
- Patternlines für Schlüsselwörter am Anfang
- Patternlines für Identifier später

Aufbau einer lex/flex-Quelldatei

definition division
%%
rules division
%%
functions division

flex erlaubt Kommentierung im C-Stil mit /* … */ in allen drei Sektionen.
Die Kommentarzeilen müssen mit einem Leerzeichen beginnen!
C-Code wird geklammert in
%{
%}
Oder beginnt mit wenigstens einem Leerzeichen

Definition devision

- C-Code, meist Includes, Typvereinbarungen

```
%{
    #include <stdio.h>
    #include <stdlib.h>
%}
```

```
%top{
    #include <stdio.h>
    #include <stdlib.h>
%}
```

- Tokendefinitionen

```
%token T_Ident      268
%token T_Num        269
%token T_ERG        270
```

- Macrodefinitionen

```
DIGIT    [0-9]
```

Bei Verwendung des Symbols muss dieses in { } eingeschlossen werden

Rules devision

- Besteht aus Patternlines
- Patternlines beginnen mit einem regulären Ausdruck oder einer Startcondition
- C-Code kann sich nach mind. einem Leerzeichen anschließen, bei mehr als einer Zeile als Block

Vordefinierte Symbole

Name	Function
int yylex(void)	call to invoke lexer, returns token
char *yytext	pointer to matched string
yyleng	length of matched string
yylval	value associated with token
int yywrap(void)	wrapup, return 1 if done, 0 if not done
FILE *yyout	output file
FILE *yyin	input file
INITIAL	initial start condition
BEGIN	condition switch start condition
ECHO	write matched string

flex:	
yy_scan_string(const char* pstr)	
yy_scan_bytes(const char *bytes, int len)	

Erstes Beispiel

ersetzen mehrerer white spaces durch ein Leerzeichen

```
%%
[ \t]+   printf(" ");
%%
main()
{
  yylex();
}
int yywrap()
{ return 1;}
```

Aufruf des Lexers

Funktion zum Umschalten der Eingabedatei, kann entfallen, -lfl stellt dann eine defaultfunktion bereit

```
T1.dat:das  ist    ein Text        mit viel  Freiraum.

beck@Examples> lex t1.lex
beck@Examples> gcc lex.yy.c -lfl
beck@Examples> ./a.out <t1.dat
das ist ein Text mit viel Freiraum.
```

Erstes Beispiel

ersetzen mehrerer white spaces durch ein Leerzeichen

```
%%
[ \t]+   printf(" ");
%%
main()
{
  yylex();
}
int yywrap()
{ return 1;}
```

Aufruf des Lexers

Funktion zum Umschalten der Eingabedatei, kann entfallen, -lfl stellt dann eine defaultfunktion bereit

```
T1.dat:das  ist    ein Text        mit viel  Freiraum.

beck@Examples> lex t1.lex
beck@Examples> gcc lex.yy.c -lfl
beck@Examples> ./a.out <t1.dat
das ist ein Text mit viel Freiraum.
echo 'das  ist   ein    Text   mit  Freiraum' | ./t1
```

2. Beispiel

Zeichen/Zeilen zählen

```
%{
int num_lines = 0, num_chars = 0;
%}

%%
\n      ++num_lines; ++num_chars;
.       ++num_chars;

%%
main()
{
  yylex();
  printf( "# of lines = %d, # of chars = %d\n",
          num_lines, num_chars );
}
```

Leider

Ein Mensch sieht schon seit Jahren klar:
Die Lage ist ganz unhaltbar.
Allein - am längsten, leider, hält
das unhaltbare auf der Welt.

```
beck@Examples> ./a.out < leider.txt
# of lines = 6, # of chars = 148
beck@Examples>
```

3. Beispiel

Wörter zählen

```
%{
#include <string.h>
int num_lines = 0, num_nums = 0;
int num_chars = 0, num_words= 0;
%}
%%
\n          ++num_lines; ++num_chars;
[a-zA-Z]+   ++num_words; num_chars+=strlen(yytext);
[0-9]+      ++num_nums;  num_chars+=strlen(yytext);
%%
main()
{
  yylex();
  printf( "# of lines    = %d\n",num_lines);
  printf( "# of words    = %d\n",num_words);
  printf( "# of numerals = %d\n",num_nums);
  printf( "# of chars    = %d\n",num_chars );
  return 0;
}
```

+: Voranstehendes Zeichen(oder Vertreter der Klasse) muss Mindestens 1x vorkommen

```
beck@Examples> ./a.out <leider.txt
# of lines    = 6
# of words    = 23
# of numerals = 0
# of chars    = 118
```

3. Beispiel

Wörter zählen

```
%{
#include <string.h>
int num_lines = 0, num_nums = 0;
int num_chars = 0, num_words= 0;
%}
LETTER [a-zA-Z]
DIGIT  [0-9]
%%
\n          ++num_lines; ++num_chars;
{LETTER}+   ++num_words; num_chars+=strlen(yytext);
{DIGIT}+    ++num_nums;  num_chars+=strlen(yytext);
%%
main()
{
  yylex();
  printf( "# of lines    = %d\n",num_lines);
  printf( "# of words    = %d\n",num_words);
  printf( "# of numerals = %d\n",num_nums);
  printf( "# of chars    = %d\n",num_chars );
  return 0;
}
```

Beispiel

```
%{
#include <math.h>
%}
%s expect

%%
floats          BEGIN(expect);

<expect>[0-9]+.[0-9]+  {
            printf( "found a float, = %f\n",
                    atof( yytext ) );
            }
<expect>\n  {
            /* end of the line, so we need another "expect-floats"
             * before we'll recognize any more numbers */
            BEGIN(INITIAL);
            }

[0-9]+      {
            printf( "found an integer, = %d\n",
                    atoi( yytext ) );
            }

"."         printf( "found a dot\n" );
```

```
floats 1.3
 found a float, = 1.300000
1.3
found an integer, = 1
found a dot
found an integer, = 3
```

Rules division

- Beispiel start conditions

```
%x comment
%%
        int line_num = 1;

"/*"                    BEGIN(comment);

<comment>[^*\n]*        /* eat anything that's not a '*' */
<comment>"*"+[^*/\n]*   /* eat up '*'s not followed by '/'s
*/
<comment>\n             ++line_num;
<comment>"*"+"/"        BEGIN(INITIAL)
```

Alles, außer '*' und '/n',
Weil diese gesondert behandelt werden.

Viele, mid. ein '*', gefolgt von einem '/'

Vollständiges Beispiel Startconditions

```
%{
#include <math.h>
        int line_num = 1;
%}
%x expect
%s comment
%%
floats          BEGIN(expect);

<expect>[0-9]+.[0-9]+ {
          printf( "found a float, = %f\n", atof( yytext ) );
        }

<expect>\n  {
        /* Beim Zeilenende nach float-Wert  */
        /* zurueckschalten */
        BEGIN(INITIAL);
      }

[0-9]+      {
        printf( "found an integer, = %d\n",atoi( yytext ) );
      }

"."   printf( "found a dot\n" );
```

```
"/*"          BEGIN(comment);

  /* eat anything that's not a '*' */
<comment>[^*\n]*

   /* eat up '*'s not followed by '/'s */
<comment>"*"+[^*/\n]*

<comment>\n             ++line_num;

  /* Kommentarende */
<comment>"*"+"/"        BEGIN(INITIAL);
%%

main()
{
  yylex();
  return 0;
}
```

Lexer für PL/0-Compiler

```
%{
/* Deklarationsteil */
/*
lexikalische Analyse mit lex fuer
graphengesteuerten PL/0 Einpasscompiler
*/

#include "lex.h"
#include "list.h"
#include "debug.h"
extern tMorph Morph;   /* globale Morpemvariable */
FILE * pIF;            /* Eingabedatei */

void MorSo(int Code); /* Function zum Bau eines
                         SymbolTokens  */
%}
%%
```

```
 /* Leer- und Trennzeichen */
[ \t]+

 /* Zeilenwechsel */
[\n]        {Morph.PosLine++;}

 /* Schluesselwoerter (Wortsymbole)   */
 /* werden wie Sonderzeichen behandlt */

"begin"     {MorSo(zBGN);return;}
call        {MorSo(zCLL);return;}
const       {MorSo(zCST);return;}
do          {MorSo(zDO); return;}
else        {MorSo(zELS);return;}
end         {MorSo(zEND);return;}
if          {MorSo(zIF); return;}
odd         {MorSo(zODD);return;}
procedure   {MorSo(zPRC);return;}
then        {MorSo(zTHN);return;}
var         {MorSo(zVAR);return;}
while       {MorSo(zWHL);return;}
```

```
 /**********/
 /* Zahlen */
 /**********/
[0-9]+  {
         Morph.MC=mcNumb;
         Morph.Val.Numb=atol(yytext);
         Morph.MLen=strlen(yytext);
         return;
        }
 /***************/
 /* Bezeichner, */
 /***************/
 /* muessen hinter Schluesselwoertern aufgefuehrt werden */
[A-Za-z]([A-Za-z0-9])*  {
         Morph.MC=mcIdent;
         Morph.Val.pStr=yytext;
         Morph.MLen=strlen(yytext);
         return;
        }
```

```
 /* Sonderzeichen */
("?")       {MorSo('?' );return;}
("!")       {MorSo('!' );return;}
("+")       {MorSo('+' );return;}
("-")       {MorSo('-' );return;}
("*")       {MorSo('*' );return;}
("/")       {MorSo('/' );return;}
("=")       {MorSo('=' );return;}
(">")       {MorSo('>' );return;}
("<")       {MorSo('<' );return;}
(":=")      {MorSo(zErg);return;}
("<=")      {MorSo(zle );return;}
(">=")      {MorSo(zge );return;}
(";")       {MorSo(';' );return;}
(".")       {MorSo('.' );return;}
(",")       {MorSo(',' );return;}
("(")       {MorSo('(' );return;}
(")")       {MorSo(')' );return;}
 /* String */
(\".*\")    {Morph.MC=mcStrng;
             Morph.Val.pStr=yytext;
             Morph.MLen=strlen(yytext);
             return;}
```

```
%%
tMorph* Lex()
{
  yylex();
  return &Morph;
}
void MorSo(int Code)
{
 Morph.MC=mcSymb;
 Morph.Val.Symb=Code;
 Morph.MLen=strlen(yytext);
 return;
}
int initLex(char* fname)
{
  char vName[128+1];
  strcpy(vName,fname);
  if (strstr(vName,".pl0")==NULL) strcat(vName,".pl0");
  pIF=fopen(vName,"rt");
  if (pIF!=NULL) {yyin=pIF; return OK;}
  return FAIL;
}
```