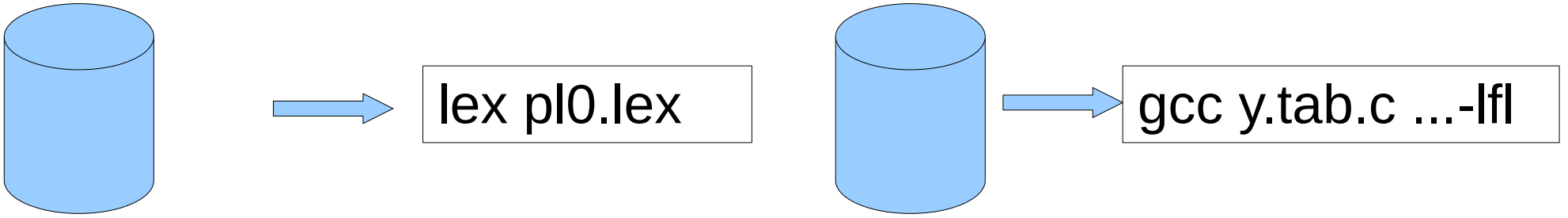
lex / flex

https://www.cs.virginia.edu/~cr4bd/flex-manual/

- 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



Pl0.lex

y.tab.c

- Optionen 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)
- Macros:

[0-9]+

```
DIGIT  [0-9]
.....
{DIGIT}+
```

DIGIT steht für Ziffern 0..9

Definition von Zahlen
{DIGIT}: Anwendung des Macros DIGIT.
{DIGIT}+ : wenigstens eine Ziffer

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 Regelanfang (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

```
[ \t]+
```

Reihenfolge der Pattern

- Reihenfolge der Patternlines ist relevant
- Patternlines werden von oben nach unten verarbeitet.
- 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

lex erlaubt Kommentierung im C-Stil mit /* … */ in allen drei Sektionen.
Die Kommentarzeilen müssen, wie c-Code mit einem whitespace beginnen!

Definition division

- Enthält lex-macros
- C-Code
  - c-includes
  - c-defines
  - Variablendefinitionen
  - c-Code, der am Anfang des generierten Codes eingesetzt wird.
- c_code muss in %{ und %} geklammert werden.

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 mindestens einem Leerzeichen anschließen, bei mehr als einer Zeile als Block

Rules division

- Wiederholungen, wobei r wiederrum für einen regulären Ausdruck steht
- r* r kann mehrfach, einmal oder gar nicht auftreten.
- r+ r kann mehrfach, muss aber mindestens einmal auftreten.
- r? r kann einmal auftreten oder auch nicht (optional).
- r{n,m} r muss mindestens n mal, und darf höchstens m mal auftreten.
- r{n,} r muss mindestens n mal auftreten.
- r{n} r muss genau n mal auftreten.
- {name} extract Macro name

Rules division

- Anwendung von Optionen auf reguläre (Teil-)ausdrücke
  - (?o:pattern) wobei o eine oder mehrere der nachfolgenden Optionen sein kann:

    i case-insensitive

    -i case-sensitive

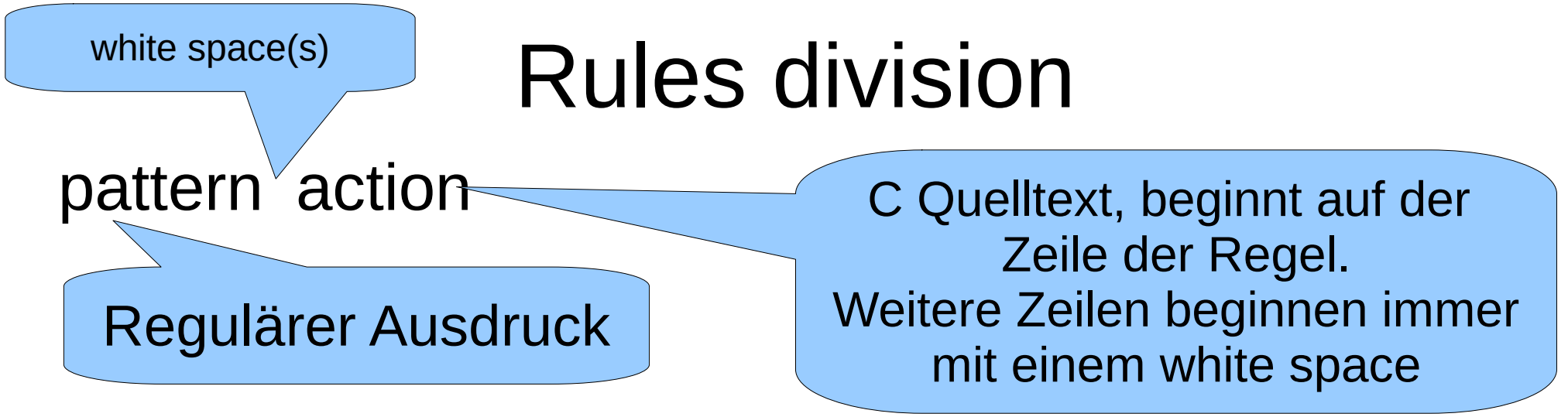
    s Das Zeichen . umfasst alle Zeichen von 0x00 bis 0xFF

    -s Das Zeichen . umfasst alle Zeichen außer \n

    x Leerzeichen und Kommentare im pattern werden ignoriert, außer, Leerzeichen nach \, Leerzeichen eingeschlossen in “” oder Leerzeichen, enthalten in einer Zeichenklasse.
- Beispiel (t5.lex):

```
(?i:([a-zäöüß])+)  ++num_words; num_chars+=strlen(yytext); printf("<%s>",yytext);
```

Rules division



- Reihenfolge der pattern lines ist relevant.
- Aufbau der patterns
  - x ein einzelnes Zeichen, in diesem Fall das Zeichen x
  - . ein beliebiges Zeichen, außer \n
  - […] ein Zeichen der Zeichenklasse
  - [^...] negierte Zeichenklasse ([^1-9], jedes Zeichen, außer Ziffern)
  - \t, \n, \b … (escape Zeichen, wie in C)
  - \0x41, \101 (octal !!), \0

Vordefinierte Symbole

Symbol	Bedeutung
char *yytext	pointer to matched string
yyleng	length of matched string
yylval	value associated with token
FILE *yyout	output file
FILE *yyin	input file

Vordefinierte Funktionen

Function	
int yylex(void)	call to invoke lexer
int yywrap(void)	wrapup, return 1 if done, 0 if not done
yy_scan_string(const char* pstr)	Produces a scanner, scanning the string pstr
yy_scan_bytes(const char *bytes, int len)	
ECHO	write matched string
INITIAL	initial start condition
BEGIN	condition switch start condition

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
```

Beispiel 1 mit yy_scan_string

```
 /* Definitionsteil */

%%
 /* Regelteil */
[ \t]+   printf(" ");
%%
 /* Funktionenteil */

int main(int argc, char*argv[])
{
  yy_scan_string(argv[1]);
  yylex();
  puts("");
  return 0;
}
```

Im Unterschied zum vorigen Beispiel liest yylex die Eingabe aus dem übergebenen String (hier argv[1]), statt aus stdin.

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 (t6.lex)

```
%{
#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
```

Startconditions

- Start conditions
  - Müssen in definition division deklariert sein mit %x oder %s

    %x starta startb
  - Mit <starta> werden Regeln maskiert, so dass sie nur gültig sind, wenn die Start condition eingestellt ist
  - Mit **Begin**(starta) wird eine start condition aktiviert. Sie bleibt so lange gültig, bis eine neue start condition eingestellt wird.
  - begin(INITIAL) setzt start conditionen zurück

Startconditions

- 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 '/'

Beispiel Startconditions

```
%START AA BB CC
%%
^a   { BEGIN AA;}
^b   { BEGIN BB;}
^c   { BEGIN CC;}
\n   {ECHO; BEGIN 0;};
<AA>magic printf("first");
<BB>magic printf("second");
<CC>magic printf("third");
```

```
$ lex -o magic.c magic.lex
$ gcc magic.c -ll
$ ./a.out
a magic hallo lex!
 first hallo lex!
b magic Fun with lex!
 second Fun with lex!
c magic lexspass
 third lexspass
$
```

Bei Eingabe von ’a magic’ mit ’a’ als erstes Zeichen wird

- a überlesen
- Startcondition AA eingestellt
- ’magic’ akzeptiert
- ’first’ ausgegeben

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" );
```

Bei Erkennung von ‘floats‘ wird die Startcondition aktiviert Es sind nur noch die Regeln, die mit <expect> markiert sind, aktiv

```
"/*"         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);
%%

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

Lexer für PL/0-Compiler

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

#include "lex.h"
#define OK 1
#define FAIL 0

extern tMorph Morph;   /* globale Morphemvariable */
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;}
```

```
 /* Sonderzeichen */
"?"          {MorSo('?' );return 0;}
"!"          {MorSo('!' );return 0;}
"+"          {MorSo('+' );return 0;}
"-"          {MorSo('-' );return 0;}
"*"          {MorSo('*' );return 0;}
"/"          {MorSo('/' );return 0;}
"="          {MorSo('=' );return 0;}
">"          {MorSo('>' );return 0;}
"<"          {MorSo('<' );return 0;}
":="         {MorSo(zErg);return 0;}
"<="         {MorSo(zle );return 0;}
">="         {MorSo(zge );return 0;}
";"          {MorSo(';' );return 0;}
"."          {MorSo('.' );return 0;}
","          {MorSo(',' );return 0;}
"("          {MorSo('(' );return 0;}
")"          {MorSo(')' );return 0;}

 /* String */
(\".*\")     {Morph.MC=mcStrng;
              Morph.Val.pStr=yytext;
              Morph.MLen=strlen(yytext);
              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;
        }
```

```
%%
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;
}
```

Angepasstes LexTest

```
#include <stdio.h>
#include <ctype.
#include "lex.h"
tMorph Morph={0};
int main(int argc, void*argv[])
{
  initLex(argv[1]);
  do
  {
    Lex();
    printf("Line%4d, Col%3d: ",Morph.PosLine, Morph.PosCol);
    switch(Morph.MC)
    {
       case mcSymb :
         if (Morph.Val.Symb==zErg)   printf("Symbol,:=\n");    else
         if (Morph.Val.Symb==zle )   printf("Symbol,<=\n");    else
         if (Morph.Val.Symb==zge )   printf("Symbol,>=\n");    else
         if (Morph.Val.Symb==zBGN)   printf("Symbol,_BEGIN\n");else
         if (Morph.Val.Symb==zCLL)   printf("Symbol,_CALL\n"); else
         if (Morph.Val.Symb==zCST)   printf("Symbol,_CONST\n");else
         if (Morph.Val.Symb==zDO )   printf("Symbol,_DO\n");   else
         if (Morph.Val.Symb==zEND)   printf("Symbol,_END\n");  else
         if (Morph.Val.Symb==zIF )   printf("Symbol,_IF\n");   else
         if (Morph.Val.Symb==zODD)   printf("Symbol,_ODD\n");  else
         if (Morph.Val.Symb==zPRC)   printf("Symbol,_PROCEDURE\n");else
         if (Morph.Val.Symb==zTHN)   printf("Symbol,_THEN\n"); else
         if (Morph.Val.Symb==zVAR)   printf("Symbol,_VAR\n");  else
         if (Morph.Val.Symb==zWHL)   printf("Symbol,_WHILE\n");
          if (isprint(Morph.Val.Symb))
               printf("Symbol,%c\n",(char)Morph.Val.Symb);
                    break;
       case mcNum :
            printf("Zahl  ,%ld\n",Morph.Val.Numb);
                    break;
       case mcIdent:
         printf("Ident ,%s\n",(char*)Morph.Val.pStr);
                    break;
    }
  }while (!(Morph.MC==mcSymb && Morph.Val.Symb=='.')) ;
  puts("");
  return 0;
}
```