Курс «Компиляторные технологии»

Лекция 3

Flex: генератор лексических анализаторов

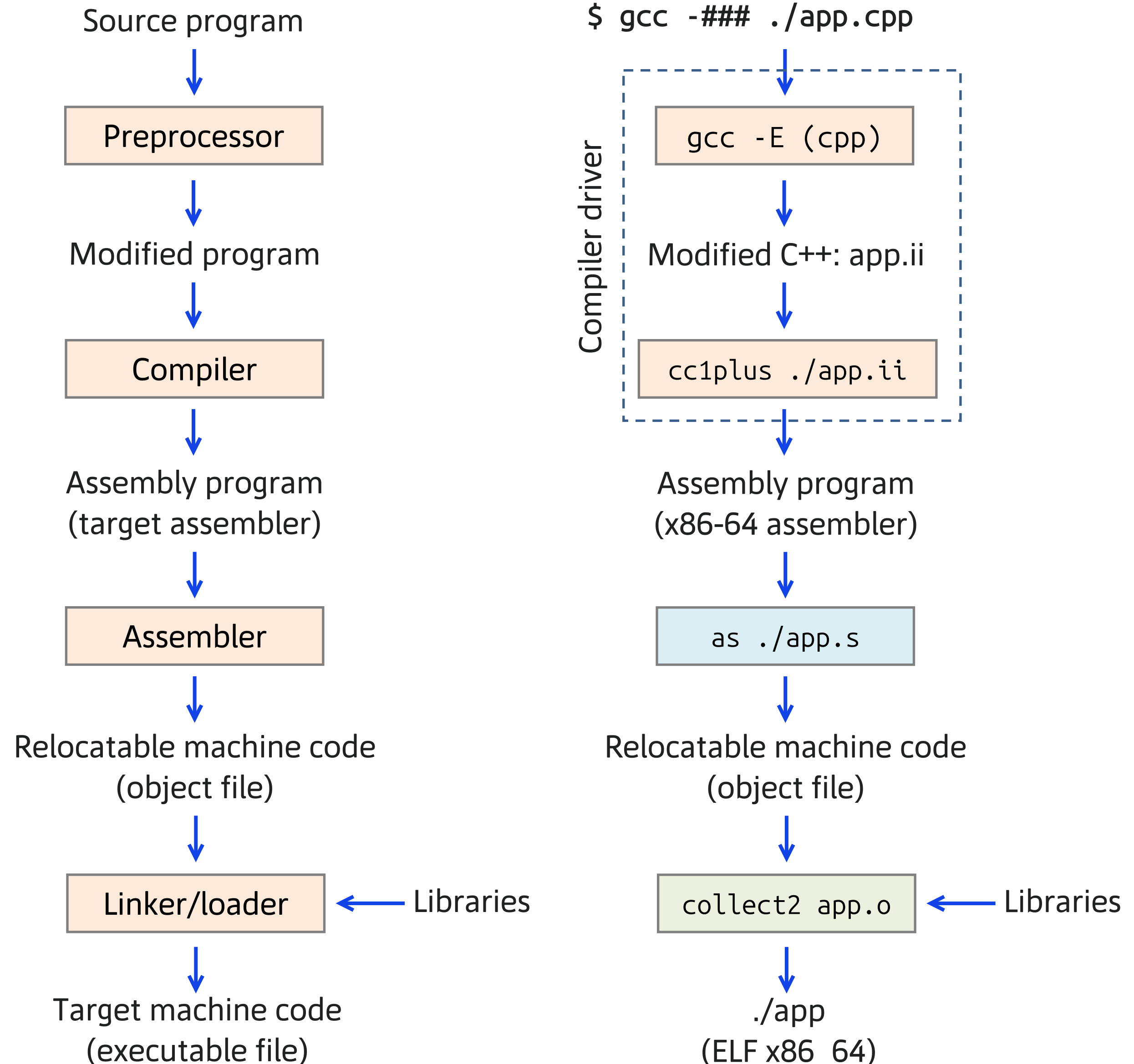
Курносов Михаил Георгиевич

www.mkurnosov.net

Сибирский государственный университет телекоммуникаций и информатики
Осенний семестр

Процесс компиляции (повторение)

- **Препроцессор** (preprocessor) — обрабатывает директивы (#include, #define, #ifdef)
- **Компилятор** (compiler) — программа, транслирующая код на *исходном языке* (source launguage) в текст программы на *целевом языке* (target language) с сохранением семантики
- **Ассемблер** (assembler) — транслятор с ассемблера в машинный код
- **Компоновщик** (linker) — программа объединяющая объектные файлы в исполняемый файл
- **Исполняемая программа** (program) — файл на носителе информации в исполняемом формате (ELF, PE, Mach-O) с секциями кода для целевой архитектуры (target architecture)
- **Загрузчик** (loader) — загружает секции исполняемого файла в память, загружает требуемые библиотеки динамической компоновки, передает управление на точку старта

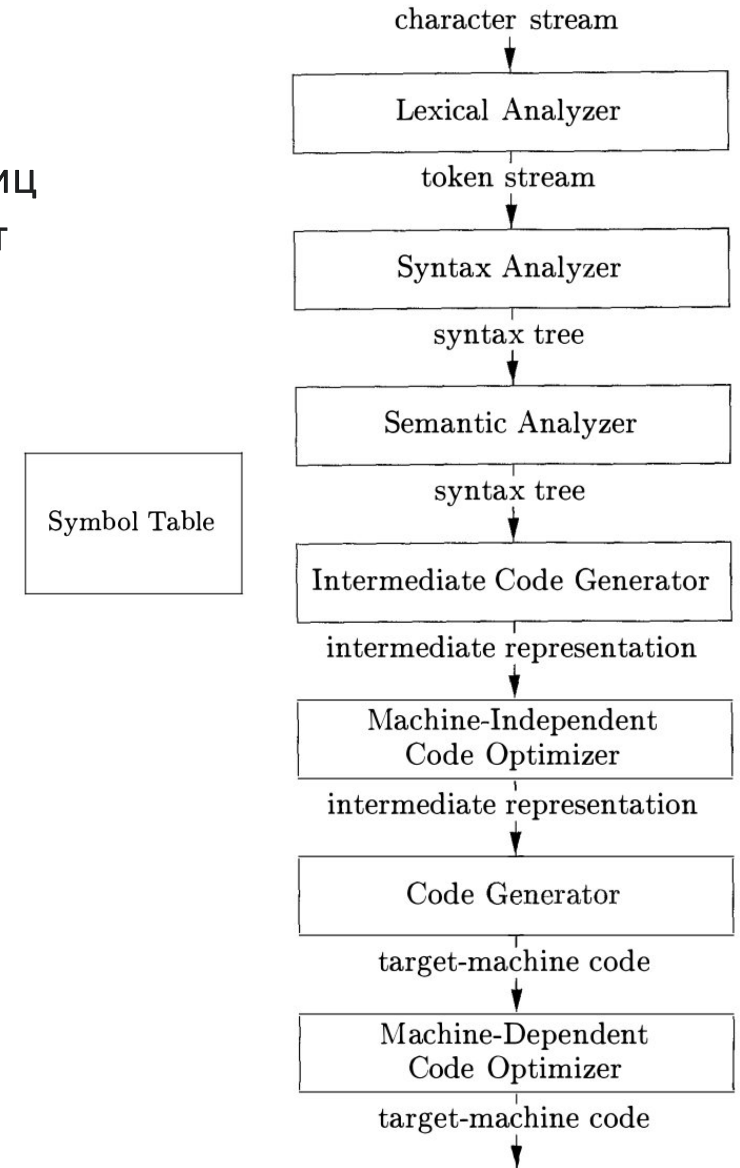


Структура компилятора (повторение)

- **Фаза анализа** (frontend, начальная стадия) —
  разбивает программу на последовательность минимально значимых единиц языка, накладывает на них грамматическую структуру языка, обнаруживает синтаксические и семантические ошибки, формирует таблицу символов, генерирует промежуточное представление программы

- **Фаза синтеза** (backend, заключительная стадия) —
  транслирует программу на основе таблицы символов и промежуточного представления в код целевой архитектуры

- **Общий процесс компиляция включает фазы** (phases):
  - лексический анализ
  - синтаксический анализ
  - семантический анализ
  - генерация (синтез) промежуточного представления
  - машинно-независимая оптимизация промежуточного представления
  - генерация машинного кода
  - машинно-зависимые оптимизации кода



[*] Aho A., Lam M., Sethi R., Ullman J. **Compilers: principles, techniques, and tools**. — 2nd ed., 2007

Лексический анализ

- **Лексический анализатор** (lexical analyzer, lexer, scanner) —
  разбивает входную программу на последовательность *лексем* (lexeme), минимально значимых единиц входного языка
- Тип допустимых лексем определяется грамматикой языка
- Игнорирует пробельные символы, комментарии, отслеживает номер текущей строки для корректного информирования о положении возможных ошибок

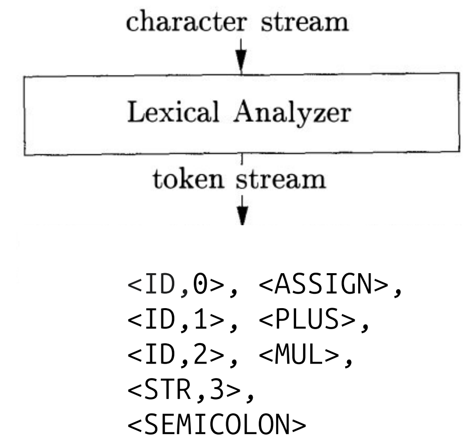
```
// Увеличить сумму
globalSum = localSum + г * 16;
```

Лексемы: «globalSum», «=», «localSum», «+», «г», «*», «16», «;»

- Для каждой найденной лексемы анализатор формирует *токен* (token) —
  пара <*имя-токена*, *значение-атрибута*>, *имя-токена* — тип/класс лексемы, *значение-атрибута* — непосредственно лексема или ссылка на запись в таблице символов

Tokens: <ID,0>, <ASSIGN>, <ID,1>, <PLUS>, <ID,2>, <MUL>, <STR,3>, <SEMICOLON>

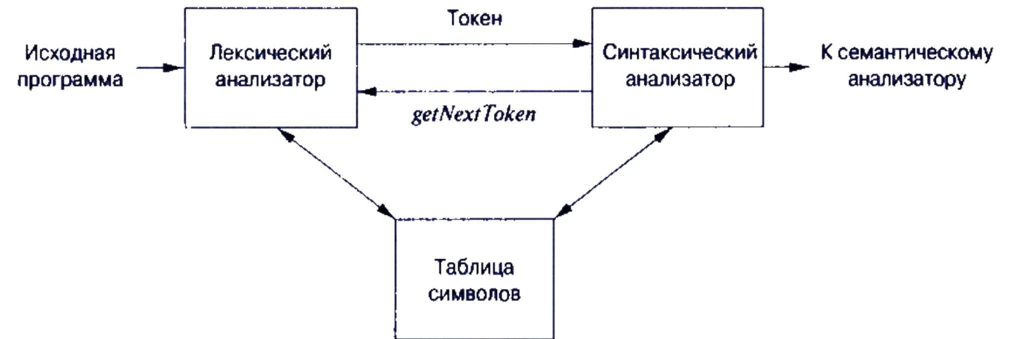
Token-names: ID, ASSIGN, PLUS, MUL, STR, SEMICOLON



Symbol table	
ID	Symbol
0	globalSum
1	localSum
2	г
3	16

Лексический анализатор (lexer, scanner)

- **Лексический анализатор** (lexical analyzer, lexer, scanner) читает из входного потока символов (единицы трансляции) очередную лексему и возвращает токен



- **Токен** (token) — пара <*имя-токена*, *значение-атрибута*>
  - *имя-токена* — тип/класс лексемы
  - *значение-атрибута* — непосредственно лексема или ссылка на запись в таблице символов

- Синтаксический анализатор вызывает лексический анализатор (управляет им):

```
o Token getNextToken()  ->  {TOKEN_TYPE, LEXEME/SYMTAB_ID)
o int yylex()
o token_t lexer_next_token()
```

- Управление лексическим анализатором
  - инициализации при открытии нового файла (единицы трансляции)
  - поддержка номеров строк для корректного указания позиций обнаруженных лексических, синтаксических и семантических ошибок

Разработка лексического анализатора

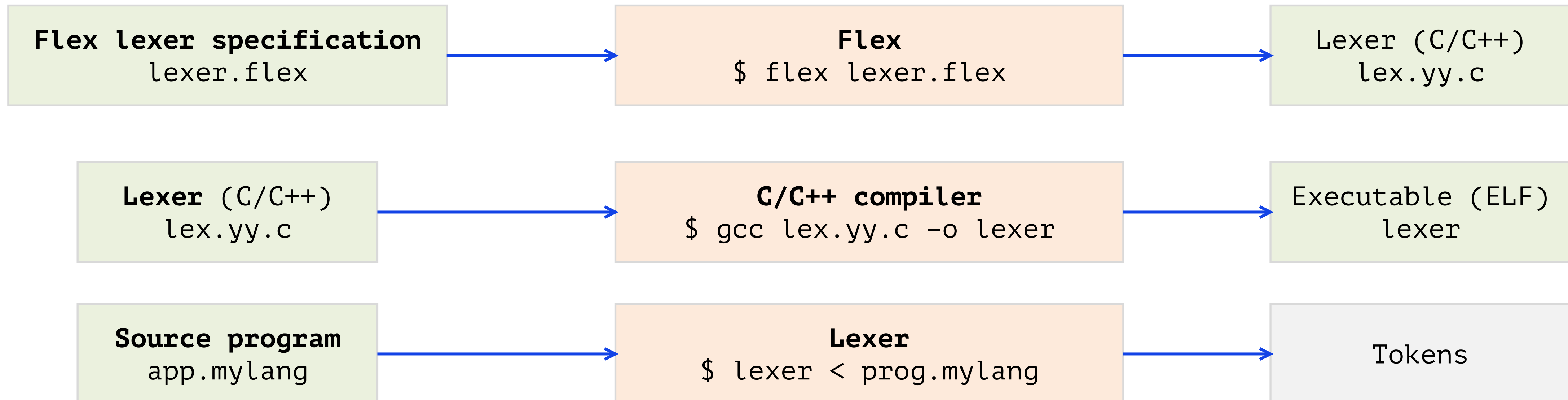
- **Основные классы лексем** выделяются из грамматики языка (пересечение интерфейса с синтаксическим анализатором)
- **Ключевые вопросы** к синтаксису языка
  - чувствительность к регистру символов (ключевые слова, идентификаторы, символы в целочисленных лексемах: 0x10FF == 0x10ff, 1.3e12 == 1.3E12)
  - обработка комментариев: однострочные (single-line), многосторные (multiline), допустимость вложенных комментариев (nested comments)
  - обработка строковых литералов: расширение специальных символов (\n, \t, \r, \\, \", \xF1), сырые строки (raw strings), допустимость многострочные строковых литералов
  - синтаксическая значимость отступов (indentation-sensitive syntax: Python, Haskell, F#)
  - поддержка Unicode: идентификаторы, строковые литералы

- **Варианты разработки и реализации лексического анализатора**
  - использование генератора лексических анализаторов (lex, Flex, RE/Flex, jFlex, ANTLR, Ragel, re2c, …)
  - самостоятельная реализации лексического анализатора (hand-written: gcc, clang)

https://clang.llvm.org/doxygen/Lexer_8cpp_source.html `Lexer::findNextToken()`
https://gcc.gnu.org/onlinedocs/cppinternals/Lexer.html

Flex (fast lexical analyzer generator)

- **lex** (1975, Mike Lesk, Eric Schmidt), Unix, Plan 9, Plan 9: MIT License, включен в POSIX
- **flex** (1987, Vern Paxson), BSD License
- Current release: 2.6.4 (2017), https://github.com/westes/flex
- Поддержка языков C/C++
- **Алгоритм**: генерирует детерминированный конечный автомат (deterministic finite automaton — DFA) на базе таблиц для поиска цепочек символов соответствующих регулярным выражениям



Структура flex-файла

Блок определений (definitions)
- названия шаблонов для использования в блоке правил
- начальные состояния (start condition)

```
<name> <definition>
```

```
%%
```

Блок правил (rules)

```
<pattern> <action>
```

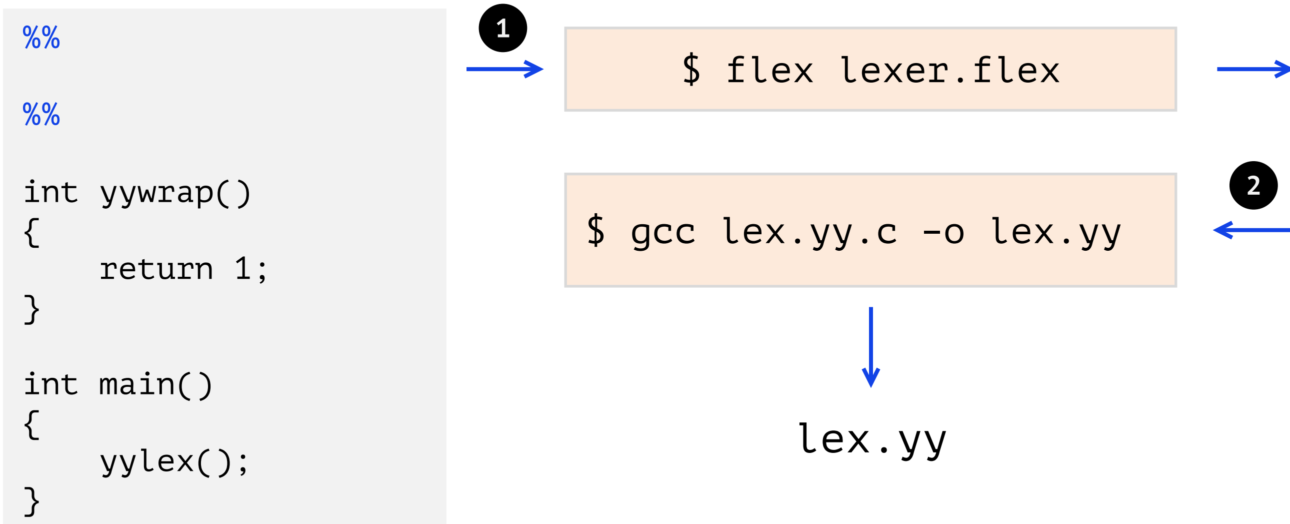
```
%%
```

```
Пользовательский код (C/C++)
```

Добавление кода в генерируемый файл

```
%top {
  // Блок top помещается в начало генерируемого файла (до кода flex)
  #include «parser.h»
}

%{
  // Indented text or text enclosed in ‘%{’ and ‘%}’ is copied verbatim to the output
  // Определение переменных, констант, ...
  enum {
    TOKEN_IDENT,
    TOKEN_KEYWORD,
    ...
  }
%}

%%

%%

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

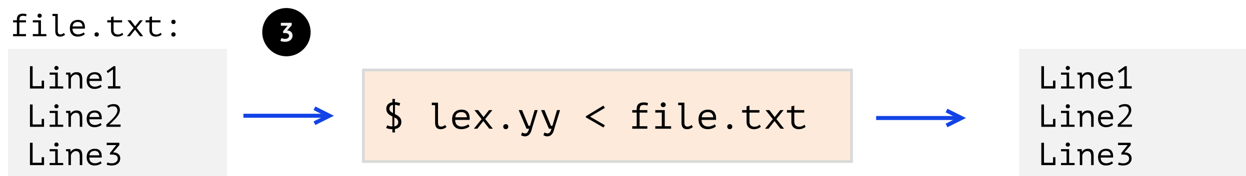
flex-файл без правил



lex.yy.c

```
/* A lexical scanner generated by flex */
#define FLEX_SCANNER
#define BEGIN (yy_start) = 1 + 2 *
#define YY_START (((yy_start) - 1) / 2)
#define YYSTATE YY_START
extern char *yytext;

static const YY_CHAR yy_ec[256] =
    {   0,
        1,    1,    1,    1,    1,    1,    1,    1,    1,    1,
        1,    1,    1,    1,    1,    1,    1,    1,    1,    1,
        1,    1,    1,    1,    1,    1,    1,    1,    1,    1,

#define YY_DECL int yylex (void) {
    ...
    while ( /*CONSTCOND*/1 )
       /* loops until end-of-file is reached */
        {
        yy_cp = (yy_c_buf_p);

   ...
}

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

- При отсутствии правил (регулярных выражений) символы копируются в выходной поток
- yywrap() вызывается при достижении EOF, позволяет открыть следующий файл для чтения, при возврате 1 (TRUE) flex заканчивает чтение на текущем файле



Подсчет числа строк в файле

```
%{
    int nlines = 0;
%}

%option noyywrap

%%

\n {nlines++;}

%%

int main()
{
    yylex();
    printf("Lines: %d\n", nlines);
}
```

```
$ flex ./ex.flex
$ gcc -o lex.yy ./lex.yy.c


$ cat file.txt
Line1
Line2
Line3

$ ./lex.yy < file.txt
Line1Line2Line3Lines: 3
```

- при встрече символа ‘\n’ увеличивается значение nlines, символ ‘\n’ в выходной поток не копируется
- остальные символы копируются в выходной поток

Подсчет числа строк в файле

```
%{
    int nlines = 0;
%}

%option noyywrap

%%

\n {nlines++;}
.  {}

%%

int main()
{
    yylex();
    printf("Lines: %d\n", nlines);
}
```

```
$ flex ./ex.flex
$ gcc -o lex.yy ./lex.yy.c


$ cat file.txt
Line1
Line2
Line3

$ ./lex.yy < file.txt
Lines: 3
```

- при встрече символа ‘\n’ увеличивается значение nlines, символ ‘\n’ в выходной поток не копируется
- второе правило соответствует любому символу
- в выходной поток ничего не копируется, все символы распознаны

Подсчет числа строк и символов в файле

```
%{
    int nlines = 0; int nchars = 0;
%}

%option noyywrap

%%

\n   {nlines++; nchars++;}
.    {nchars++;}

%%

int main()
{
    yylex();
    printf("Lines: %d, chars %d\n",
           nlines, nchars);
}
```

```
$ flex ./ex.flex
$ gcc -o lex.yy ./lex.yy.c


$ cat file.txt
Line1
Line2
Line3

$ ./lex.yy < file.txt
Lines: 3, chars 18
```

Flex-файл для диалекта языка Pascal

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

DIGIT   [0-9]
ID      [a-zA-Z][a-zA-Z0-9]*

%option noyywrap

%%

{DIGIT}+                               { printf("Integer: %s\n", yytext); }
{DIGIT}+"."{DIGIT}*                    { printf("Float: %s\n", yytext); }
if|then|begin|end|procedure|program    { printf("Keyword: %s\n", yytext); }

{ID}                  { printf("Identifier: %s\n", yytext); }
"+"|"-"|"*"|"/"       { printf("Operator: %s\n", yytext); }
";"                   { printf("Delimiter: %s\n", yytext); }

"{"[^{}\n]*"}"        { /* eat up one-line comments */ }
[ \t\n]+              { /* eat up whitespace */ }
.                     { printf("Lexical error: unrecognized character: %s\n",
                                 yytext); }
%%

int main(int argc, char **argv)
{
    yyin = (argc > 0) ? fopen(argv[1], "r") : stdin;
    yylex();
    return 0;
}
```

```
program Hello;
begin
  x := 12;
  eps := 3.14;
  writeln ('Hello, world.')
end.
```

```
$ ./lex.yy ./prog.pas
Keyword: program
Identifier: Hello
Delimiter: ;
Keyword: begin
Identifier: x
Lexical error: unrecognized character: :
Lexical error: unrecognized character: =
Integer: 12
Delimiter: ;
Identifier: eps
Lexical error: unrecognized character: :
Lexical error: unrecognized character: =
Float: 3.14
Delimiter: ;
Identifier: writeln
Lexical error: unrecognized character: (
Lexical error: unrecognized character: '
Identifier: Hello
Lexical error: unrecognized character: ,
Identifier: world
Lexical error: unrecognized character: .
Lexical error: unrecognized character: '
Lexical error: unrecognized character: )
Keyword: end
Lexical error: unrecognized character: .
```

Диалекта языка Pascal (v2)

```
%{
#include <stdio.h>
#include "parser.h"

int lineno = 0;
void yyerror(char *message);
%}

white_space       [ \t]*
digit             [0-9]
alpha             [A-Za-z_]
alpha_num         ({alpha}|{digit})
hex_digit         [0-9a-fA-F]
identifier        {alpha}{alpha_num}*
unsigned_integer  {digit}+
hex_integer       ${hex_digit}{hex_digit}*
exponent       e[+-]?{digit}+
i              {unsigned_integer}
real           ({i}\.{i}?|{i}?\.{i}){exponent}?

%option noyywrap

%%

and                    return TOKEN_AND;
array                  return TOKEN_ARRAY;
begin                  return TOKEN_BEGIN;
case                   return TOKEN_CASE;
const                  return TOKEN_CONST;
div                    return TOKEN_DIV;
do              return TOKEN_DO;
downto          return TOKEN_DOWNTO;
else            return TOKEN_ELSE;
end             return TOKEN_END;
file            return TOKEN_FILE;
for             return TOKEN_FOR;
function        return TOKEN_FUNCTION;
goto            return TOKEN_GOTO;
if              return TOKEN_IF;
in              return TOKEN_IN;
label           return TOKEN_LABEL;
mod             return TOKEN_MOD;
nil             return TOKEN_NIL;
not             return TOKEN_NOT;
of              return TOKEN_OF;
packed          return TOKEN_PACKED;
procedure       return TOKEN_PROCEDURE;
program         return TOKEN_PROGRAM;
record          return TOKEN_RECORD;
repeat          return TOKEN_REPEAT;
set             return TOKEN_SET;
then            return TOKEN_THEN;
to              return TOKEN_TO;
type            return TOKEN_TYPE;
until           return TOKEN_UNTIL;
var             return TOKEN_VAR;
while           return TOKEN_WHILE;
with            return TOKEN_WITH;
"<="|"=<"           return TOKEN_LEQ;
"=>"|">="           return TOKEN_GEQ;
"<>"                return TOKEN_NEQ;
"="                 return TOKEN_EQ;
".."                return TOKEN_DOUBLEDOT;

{unsigned_integer}  return TOKEN_UNSIGNED_INTEGER;
{real}              return TOKEN_REAL;
{hex_integer}       return TOKEN_HEX_INTEGER;

{identifier}        return TOKEN_IDENTIFIER;


[*/+\-,^.;:()\[\]]  return yytext[0];

{white_space}       { /* skip spaces */ }
\n                  lineno++;
.                   yyerror("unrecognized character");

%%

void yyerror(char *message)
{
   fprintf(stderr,"Error (line %d): %s: lexeme %s\n",
           lineno + 1, message, yytext);
   exit(1);
}

int main()
{
    for (int token = yylex(); token; token = yylex()) {
        printf("token %d\n", token);
    }
    return 0;
}
```

- Вызов yylex() в цикле — возвращает тип лексемы (определены в parser.h)
- Вывод номера строки при лексической ошибке (yyerror())
- Отсутствует поддержка комментариев и строковых литералов

Диалекта языка Pascal: строковые литералы

```
%{
#include <stdio.h>
#include "parser.h"

int lineno = 0;
void yyerror(char *message);
%}

white_space        [ \t]*
digit              [0-9]
alpha              [A-Za-z_]
alpha_num          ({alpha}|{digit})
hex_digit          [0-9a-fA-F]
identifier         {alpha}{alpha_num}*
unsigned_integer   {digit}+
hex_integer        ${hex_digit}{hex_digit}*
exponent        e[+-]?{digit}+
i               {unsigned_integer}
real            ({i}\.{i}?|{i}?\.{i}){exponent}?

string         \'([^'\n]|\'\')*\'

%option noyywrap

%%

and                   return TOKEN_AND;
array                 return TOKEN_ARRAY;
begin                 return TOKEN_BEGIN;
case                  return TOKEN_CASE;
const                 return TOKEN_CONST;
```

```
{real}                return TOKEN_REAL;
{hex_integer}         return TOKEN_HEX_INTEGER;

{string}              return TOKEN_STRING;

{identifier}          return TOKEN_IDENTIFIER;


[*/+\-,^.;:()\[\]]    return yytext[0];

{white_space}         { /* skip spaces */ }
\n                    lineno++;
.                     yyerror("unrecognized character");

%%

void yyerror(char *message)
{
   fprintf(stderr,"Error (line %d): %s: lexeme %s\n",
           lineno + 1, message, yytext);
   exit(1);
}

int main()
{
    for (int token = yylex(); token; token = yylex()) {
        printf("token %d `%s`\n", token, yytext);
    }
    return 0;
}
```

```
begin
  writeln('A:  ');
  for i := 1 to N do
    Write(a[i]);
end.
```

```
$ driver < prog.pas
token 2: `begin`
token 43: `writeln`
token 40: `(`
token 42: `'A:  '`
token 41: `)`
token 59: `;`
token 11: `for`
token 43: `i`
token 58: `:`
token 37: `=`
token 39: `1`
token 28: `to`
token 43: `N`
token 6: `do`
token 43: `Write`
token 40: `(`
token 43: `a`
token 91: `[`
token 43: `i`
token 93: `]`
token 41: `)`
token 59: `;`
token 9: `end`
token 46: `.`
```

- Строковый литерал только в пределах одной строки, не может содержать символ ‘, может включать двойной апостроф ‘‘

Диалекта языка Pascal: строковые литералы (ошибка)

```
%{
#include <stdio.h>
#include "parser.h"

int lineno = 0;
void yyerror(char *message);
%}

white_space        [ \t]*
digit              [0-9]
alpha              [A-Za-z_]
alpha_num          ({alpha}|{digit})
hex_digit          [0-9a-fA-F]
identifier         {alpha}{alpha_num}*
unsigned_integer   {digit}+
hex_integer        ${hex_digit}{hex_digit}*
exponent        e[+-]?{digit}+
i               {unsigned_integer}
real            ({i}\.{i}?|{i}?\.{i}){exponent}?

string          \'([^'\n]|\'\')*\'

%option noyywrap

%%

and                   return TOKEN_AND;
array                 return TOKEN_ARRAY;
begin                 return TOKEN_BEGIN;
case                  return TOKEN_CASE;
const                 return TOKEN_CONST;
```

```
{real}                return TOKEN_REAL;
{hex_integer}         return TOKEN_HEX_INTEGER;

{string}              return TOKEN_STRING;

{identifier}          return TOKEN_IDENTIFIER;


[*/+\-,^.;:()\[\]]    return yytext[0];

{white_space}         { /* skip spaces */ }
\n                    lineno++;
.                     yyerror("unrecognized character");

%%

void yyerror(char *message)
{
   fprintf(stderr,"Error (line %d): %s: lexeme %s\n",
           lineno + 1, message, yytext);
   exit(1);
}

int main()
{
    for (int token = yylex(); token; token = yylex()) {
        printf("token %d `%s`\n", token, yytext);
    }
    return 0;
}
```

```
begin
  writeln('A:
  for i := 1 to N do
    Write(a[i]);
end.
```

```
$ driver < prog.pas
token 2: `begin`
token 43: `writeln`
token 40: `(`
Error (line 2):
"unrecognized character": lexeme '`
```

- Строковый литерал только в пределах одной строки, не может содержать символ ‘,
  может включать двойной апостроф ‘’

Диалекта языка Pascal: строковые литералы (ошибка)

```
%{
#include <stdio.h>
#include "parser.h"

int lineno = 0;
void yyerror(char *message);
%}

white_space       [ \t]*
digit             [0-9]
alpha             [A-Za-z_]
alpha_num         ({alpha}|{digit})
hex_digit         [0-9a-fA-F]
identifier        {alpha}{alpha_num}*
unsigned_integer  {digit}+
hex_integer       ${hex_digit}{hex_digit}*
exponent       e[+-]?{digit}+
i              {unsigned_integer}
real           ({i}\.{i}?|{i}?\.{i}){exponent}?

string       \'([^'\n]|\'\')*\'
bad_string   \'([^'\n]|\'\')*

%option noyywrap

%%

and                    return TOKEN_AND;
array                  return TOKEN_ARRAY;
begin                  return TOKEN_BEGIN;
case                   return TOKEN_CASE;
const                  return TOKEN_CONST;
```

```
{real}                return TOKEN_REAL;
{hex_integer}         return TOKEN_HEX_INTEGER;

{string}       return TOKEN_STRING;
{bad_string}   yyerror("unterminated string");

{identifier}          return TOKEN_IDENTIFIER;


[*/+\-,^.;:()\[\]]    return yytext[0];

{white_space}         { /* skip spaces */ }
\n                    lineno++;
.                     yyerror("unrecognized character");

%%

void yyerror(char *message)
{
   fprintf(stderr,"Error (line %d): %s: lexeme %s\n",
           lineno + 1, message, yytext);
   exit(1);
}

int main()
{
    for (int token = yylex(); token; token = yylex()) {
        printf("token %d `%s`\n", token, yytext);
    }
    return 0;
}
```

```
begin
  writeln('A:
  for i := 1 to N do
    Write(a[i]);
end.
```

```
$ driver < prog.pas
token 2: `begin`
token 43: `writeln`
token 40: `(`
Error (line 2): "unterminated
string": lexeme 'A:
```

- Правило {bad_string} упрощает нахождение ошибок в строковых литералах и упрощает вывод сообщений об ошибках

Диалекта языка Pascal: комментарии

```
%{
#include <stdio.h>
#include "parser.h"

int lineno = 0;
void yyerror(char *message);
%}

white_space       [ \t]*
digit             [0-9]
alpha             [A-Za-z_]
alpha_num         ({alpha}|{digit})
hex_digit         [0-9a-fA-F]
identifier        {alpha}{alpha_num}*
unsigned_integer  {digit}+
hex_integer       ${hex_digit}{hex_digit}*
exponent       e[+-]?{digit}+
i              {unsigned_integer}
real           ({i}\.{i}?|{i}?\.{i}){exponent}?
string       \'([^'\n]|\'\')+\'
bad_string   \'([^'\n]|\'\')+

%x COMMENT

%option noyywrap

%%

"{"                  BEGIN(COMMENT);
<COMMENT>[^}\n]+     { /* skip*/ }
<COMMENT>\n          { lineno++; }
<COMMENT><<EOF>>     yyerror("EOF in comment");
<COMMENT>"}"         BEGIN(INITIAL);

and                    return TOKEN_AND;
```

- Многострочный комментарий

```
{
  ...
 }
```

```
{ Tiny app }
begin
  {
    Two-line
    comment
  }
  x := '';
end.
```

```
$ driver < prog.pas
token 2: `begin`
token 43: `x`
token 58: `:`
token 37: `=`
token 42: `''`
token 59: `;`
token 9: `end`
token 46: `.`
```

- <S><pattern>   <action>
  действие action актируется, если сканер в состоянии S (start condition) и обнаружено сопоставление шаблону pattern
- BEGIN(S) – переводит сканер в состояние S
- %x S, %s S – объявление начальных состояний (inclusive, exclusive)