

Генератор лексических анализаторов Lex

Lex представляет собой программу для генерации лексических анализаторов.

Лексический анализатор – часть транслятора, которая читает исходную программу и выделяет в ее тексте лексемы входного языка.

Лексема (лексическая единица языка) – структурная единица языка, которая состоит из элементарных символов языка и не содержит в своем составе других структурных единиц языка.

Лексемами языков программирования являются идентификаторы, константы, ключевые слова языка, знаки операций и разделители. Состав возможных лексем конкретного языка определяется его синтаксисом.

Пример:

Для языка с подобной грамматикой:

<Программа>:=<Объявление переменных> <Описание вычислений>.

<Описание вычислений> ::= Begin <Список присваиваний> End

<Объявление переменных> ::= Var <Список переменных> : Logical;

<Список переменных> ::= <Идент> | <Идент>, <Список переменных> и

т.д., следует описать следующие лексемы:

Лексема	Тип лексемы
Begin	Ключевое слово
End	Ключевое слово
Var	Ключевое слово
Logical	Ключевое слово
:	Знак двоеточие
;	Знак точка с запятой
,	Знак запятой
::=	Знак присваивания
.	Знак точки и т.д.

Основная задача лексического анализатора заключается в определении значения (не путать с семантическим, это третий этап анализа) входной лексемы. Соответственно, на начальном этапе разработки ЛА необходимо описать для Lex имеющиеся лексемы. Фактически это сводится к простому наименованию лексем понятным для программиста, создающего транслятор, способом. На основе выше приведенной таблицы получим следующий набор значений лексем:

Значение
KEYWORD
KEYWORD
KEYWORD
KEYWORD
COLON
SEMICOLON
COMMA
APPROP
DOT

После определения значений лексем следует приступить к описанию lex-спецификации.

Lex-спецификация – файл, на основе которого будет сгенерирован лексический анализатор.

Lex-спецификация состоит из трех секций: определений, правил и подпрограмм пользователя. Секция правил является обязательной. Порядок следования секций также установленный и не терпит изменений (рисунок 1).

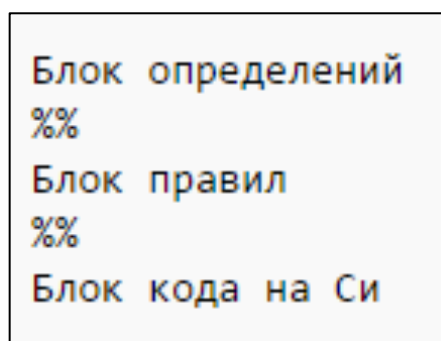
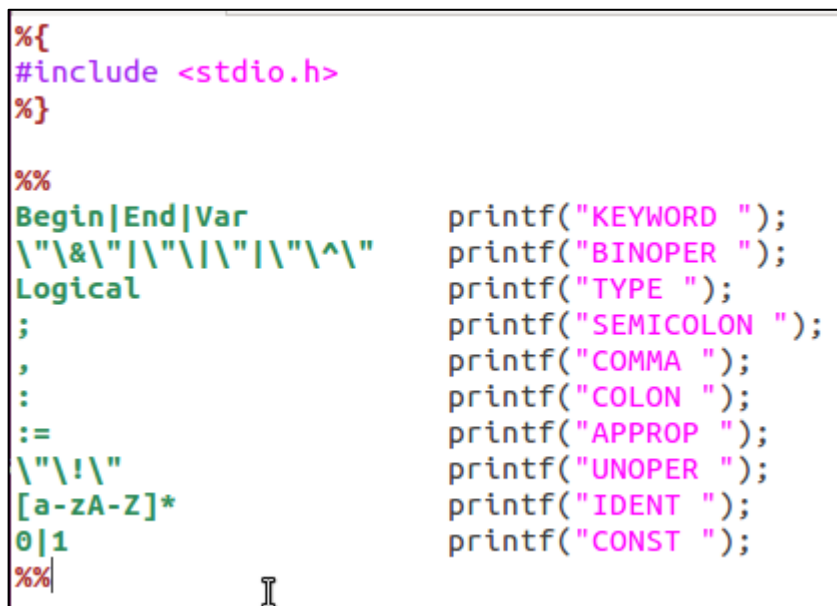


Рисунок 1 – Структура
lex-спецификации

Секция правил открывается посредством разделителя `%%`. В случае, если в описании имеется раздел подпрограмм пользователя, в конце описания правил также необходимо поставить разделитель `%%`.

Для рассмотренного выше примера блок правил и блок определений примут следующий вид (пример дополнен лексемами, не описанными в примерах ранее):

A screenshot of a text editor showing a Lex specification file. The file is divided into two sections by a double percent sign (`%%`). The first section contains rules for identifying keywords, operators, types, semicolons, commas, colons, assignment operators, and identifiers/constant literals. The second section contains corresponding actions that print the name of the matched lexeme. The rules are: `Begin|End|Var` (KEYWORD), `"\&\"|\"\\\"|\"^\"` (BINOPER), `Logical` (TYPE), `;` (SEMICOLON), `,` (COMMA), `:` (COLON), `:=` (APPROP), `"!\"` (UNOPER), `[a-zA-Z]*` (IDENT), and `0|1` (CONST). The actions are `printf("KEYWORD ");`, `printf("BINOPER ");`, `printf("TYPE ");`, `printf("SEMICOLON ");`, `printf("COMMA ");`, `printf("COLON ");`, `printf("APPROP ");`, `printf("UNOPER ");`, `printf("IDENT ");`, and `printf("CONST ");`. The file ends with another `%%` separator.

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

%%
Begin|End|Var          printf("KEYWORD ");
"\&\"|\"\\\"|\"^\"      printf("BINOPER ");
Logical               printf("TYPE ");
;                     printf("SEMICOLON ");
,                     printf("COMMA ");
:                     printf("COLON ");
:=                    printf("APPROP ");
"!\"                  printf("UNOPER ");
[a-zA-Z]*             printf("IDENT ");
0|1                   printf("CONST ");
%%
```

Рисунок 2 – Файл-описание лексем грамматики

Спецификация описывается в простом текстовом редакторе, после чего файл сохраняется с расширением `.lex`: `example.lex`.

Далее запускаем Lex посредством ввода в терминал следующей команды: `lex example.lex`. В результате работы Lexa будет сгенерирован файл `lex.yy.cc`, который появится в той же папке, где лежит файл-спецификация. Необходимо скомпилировать данный файл командой: `gcc lex.yy.cc -ll` или же `gcc lex.yy.cc -lfl` (в зависимости от того, какой именно инструмент вы используете: Lex или же производный от него Flex).

Теперь следует запустить программу: `./a.out`. После запуска ЛА курсор терминала будет ожидать входной последовательности, которую требуется проанализировать. При введении описанных в спецификации лексем будут выводиться соответствующие им названия (рисунок 3).

```
anastasia@anastasia-VirtualBox:~/Рабочий стол$ ./a.out
|
BINOPER
&
BINOPER
^
BINOPER
Var a,b,c:Logical;
KEYWORD IDENT COMMA IDENT COMMA IDENT COLON TYPE SEMICOLON
Begin End
KEYWORD KEYWORD
█
```

Рисунок 3 – Пример работы ЛА

При появлении неописанной ранее комбинации символов, она просто продублируется ниже. Вывод сообщения об ошибке будет рассмотрен в следующем разделе на этапе синтаксического анализа.

Генератор синтаксических анализаторов Yacc.

Синтаксический анализ в лингвистике и информатике – это процесс сопоставления линейной последовательности лексем рассматриваемого формального языка с его формальной грамматикой.

Синтаксический анализатор – это программа или часть программы, выполняющая синтаксический анализ (что логично, не правда ли?).

Yacc представляет собой программу, которая служит стандартным генератором синтаксических анализаторов в Unix-системах. Название является акронимом «Yet Another Compiler Compiler» («ещё один компилятор компиляторов»). Yacc генерирует анализатор на основе аналитической грамматики, описанной в нотации BNF (форма Бэкуса-Наура) или контекстно-свободной грамматики. На выходе yacc выдаётся код так называемого парсера на языке программирования Си.

Для описания будущего анализатора необходимо создать файл с расширением «.y». Структура такого файла та же, что и у lex-файла:

Секция описаний

%%

Секция грамматических правил

%%

Секция процедур

Секция описаний содержит:

- описания переменных языка Си, которые используются при описании грамматики; эти описания заключаются в скобки `%{ ... }%`, они будут перенесены в текст результирующей программы без изменения;
- определения типов, значения которых возвращаются как значения семантик; эти типы определяются как элементы объединенного типа `%union { type1 id1; ... };`
- объявления терминальных символов (лексических классов, `%tokens`) грамматики;

– объявления нетерминальных символов грамматики в форме %type name;

– определения ассоциативности и приоритетов операций.

Секция грамматических правил состоит из правил, которые записываются следующим образом:

A: production_body,

где A – имя нетерминала, production_body – последовательность нуля или большего количества имен и литералов. Имена могут быть произвольной длины и содержать буквы, цифры (цифра не может быть первой литерой имени), подчеркивания и точки. Если имеется несколько грамматических правил с одинаковой левой частью, то может использоваться литера вертикальная черта для объединения всех правил в одно.

Секция описаний процедур предназначена для процедур, которые пользователь использует при написании семантических действий. Впрочем, эти процедуры могут быть размещены и в других файлах и откомпилированы отдельно. Таким образом, эта секция необязательна, в отличие от секции описаний и грамматических правил.

В синтаксическом анализе используется результат работы лексического анализатора. Однако для осуществления совместной работы двух анализаторов необходимо внести коррективы в lex-файл:


```

%%
#include <stdio.h>
#include <string.h>

void yyerror(const char *str)
{
    fprintf(stderr, "error: %s\n", str);
}

int yywrap()
{
    return 1;
}

main()
{
    yyparse();
}

%%

%token NUMBER IDENTIFIER KEY_VAR OPERATION SEMICOLON COMMA SIGN MINUS PLUS OBRACKET EBRACKET

%%
PROGRAM: VARS OPS
;
// Объявляем переменные
VARS: = VAR
| = VARS VAR
;
VAR: = KEY_VAR IDS SEMICOLON
;
IDS: = IDENTIFIER
| = IDS COMMA IDENTIFIER
;

```

Рисунок 2 – Файл синтаксического анализатора

Помимо описания правил грамматики и терминалов, в файле также присутствуют функции вывода сообщения обо ошибке (в случае несоответствия входной комбинации какому-либо правилу), завершения чтения входной комбинации и запуска самого анализатора.

Для запуска анализатора в терминале используется сочетание следующих команд: `уасс -d file.y && lex example.lex && gcc lex.yy.c y.tab.c`.

Запускается анализатор аналогично предыдущему: `./a.out`.

Результат работы синтаксического анализатора:

```

Var asfd;
Var adsf,adsf;
sdf=-9;
ghji
k
k678ihu
error: syntax error

```

Рисунок 3 – Пример работы СА

Важные замечания:

1. Внимательно изучите описанные в вариантах задания грамматики, т.к. они могут содержать правила, которые будут восприниматься в Yacc, как порождающие конфликты (в основном сдвиг-свертка). Если не исправить подобные правила, попытка запуска Yacc закончится большим списком предупреждений.

2. Правило называется *рекурсивным*, если нетерминал его *результата* появляется также в его правой части. Почти все грамматики Yacc должны использовать рекурсию, потому что это единственный способ определить последовательность из произвольного числа элементов. Рассмотрим рекурсивное определение последовательности одного или более выражений, разделённых запятыми:

```
expseq1: exp
        | expseq1 ',' exp
        ;
```

Поскольку рекурсивный символ `expseq1` – самый левый в правой части, это называется *левой рекурсией*. И наоборот, вот та же конструкция, определённая с использованием *правой рекурсии*:

```
expseq1: exp
        | exp ',' expseq1
        ;
```

Последовательность любого вида может быть определена с использованием как левой, так и правой рекурсии, но вам следует **всегда использовать леворекурсивные правила**, потому что они могут разобрать последовательность из любого числа элементов, используя ограниченное стековое пространство. Размер используемого праворекурсивными правилами стека Yacc пропорционален числу элементов последовательности, поскольку все эти элементы должны быть помещены в стек перед тем, как правило будет применено в первый раз

Семантический анализатор.

Семантика программы – внутренняя модель (база данных) множества именованных объектов, с которыми работает программа, с описанием их свойств, характеристик и связей.

Семантический анализ существенно отличается от лексического и синтаксического. И дело не столько в том, что фаза семантического анализа реализуется не формальными, а содержательными методами (т.е. на данный момент нет универсальных математических моделей и формальных средств описания «смысла» программы). Лексический и синтаксический анализ имеют дело со структурными, текстовыми конструкциями языка. Семантика же, ориентированная на содержательную интерпретацию, имеет дело с внутренним представлением «смысла» объектов, описанных в программе.

Задача семантического анализатора – установить соответствие между комбинацией входных символов языка и действием, которое транслятору необходимо произвести в случае появления такой входной комбинации.

В данной работе целью применения семантического анализатора является «перевод» конкретных лексем входного языка, в соответствующие им лексемы целевого языка.

Основным действием описываемых семантик будет замещение конкретной лексемы исходного языка строкой, содержащей соответствующую лексему целевого языка.

Реализация семантического анализа в Yacc заключается в описании действий для каждого правила описанной ранее грамматики (рисунок 1).

```

PROGRAM: VARS OPS
;

OPS: BEG APPS END
{
    printf("%s", $1);
    printf("\n%s", $2);
    printf("\n%s", $3);
}
;
VARS: VAR
|
    VARS VAR
;
VAR: KEY_VAR IDS COLON TYPE SEMICOLON
{
    printf("%s %s %s %s %s", $1, $2, $3, "Boolean", $5);
}
;

IDS: IDENTIFIER
{
    $$=$1;
}
|
    IDS COMMA IDENTIFIER
{
    $$=strcat($1, ",");
    $$=strcat($$, $3);
}

```

Рисунок 1 – Часть кода семантик в спецификации синтаксического анализатора

Каждый вызов функции `yylex()` возвращает целое значение, представляющее из себя тип токена. Таким образом YACC понимает, какого типа токен был считан. Этот токен может обладать значением (атрибутом), которое помещается в переменную `yylval`.

По умолчанию переменная `yylval` имеет тип `int`, но это можно переопределить (`#define YYSTYPE`).

```

#ifdef YYSTYPE
#define YYSTYPE char*
#endif
extern YYSTYPE yylval;

```

Рисунок 2 – Заголовок файла лексического анализатора

Лексический анализатор должен иметь доступ к переменной `yylval`. Для этого она должна быть объявлена в области видимости файла `lex` как `extern-переменная` (рисунок 2).

Изменениям подвергается и спецификация файла лексического анализатора (рисунок 3).

```
%%
(\\\"!\\\"){1}      {yylval.str=strdup(yytext);return UNOPER;}
(\\\"&\\\"){1}      {yylval.str=strdup(yytext);return AND;}
(\\\"|\\\"){1}      {yylval.str=strdup(yytext);return OR;}
(\\\"^\\\"){1}      {yylval.str=strdup(yytext);return XOR;}
(\\:){1}          {yylval.str=strdup(yytext);return APPROP;}
(\\)){1}          {yylval.str=strdup(yytext);return RBRET;}
(\\){1}          {yylval.str=strdup(yytext);return LBRET;}
(Logical)        {yylval.str=strdup(yytext);return TYPE;}
(Begin)          {yylval.str=strdup(yytext);return BEG;}
(End)            {yylval.str=strdup(yytext);return END;}
(Var)            {yylval.str=strdup(yytext);return KEY_VAR;}
(\\;){1}          {yylval.str=strdup(yytext);return SEMICOLON;}
(\\,){1}          {yylval.str=strdup(yytext);return COMMA;}
(\\:){1}          {yylval.str=strdup(yytext);return COLON;}
(\\.){1}          {yylval.str=strdup(yytext);return DOT;}
([a-zA-Z])+      {yylval.str=strdup(yytext);return IDENTIFIER;}
([0-1])+         {yylval.str=strdup(yytext);return CONST;}
\\n               ;
[ \\t]+
```

Рисунок 3 – Изменение спецификации лексического анализатора

Теперь каждая лексема будет восприниматься Yacc как последовательность строкового типа данных. При необходимости введения арифметических операций, ряду лексем будет необходимо изменить тип данных.

В заголовке спецификации синтаксического анализатора необходимо определить тип рассматриваемых токенов и нетерминальных символов (рисунок 4).

```
%union {
    char* str;
}

%token <str> AND OR XOR UNOPER TYPE BEG END KEY_VAR APPROP SEMICOLON COLON COMMA DOT IDENTIFIER CONST RBRET LBRET

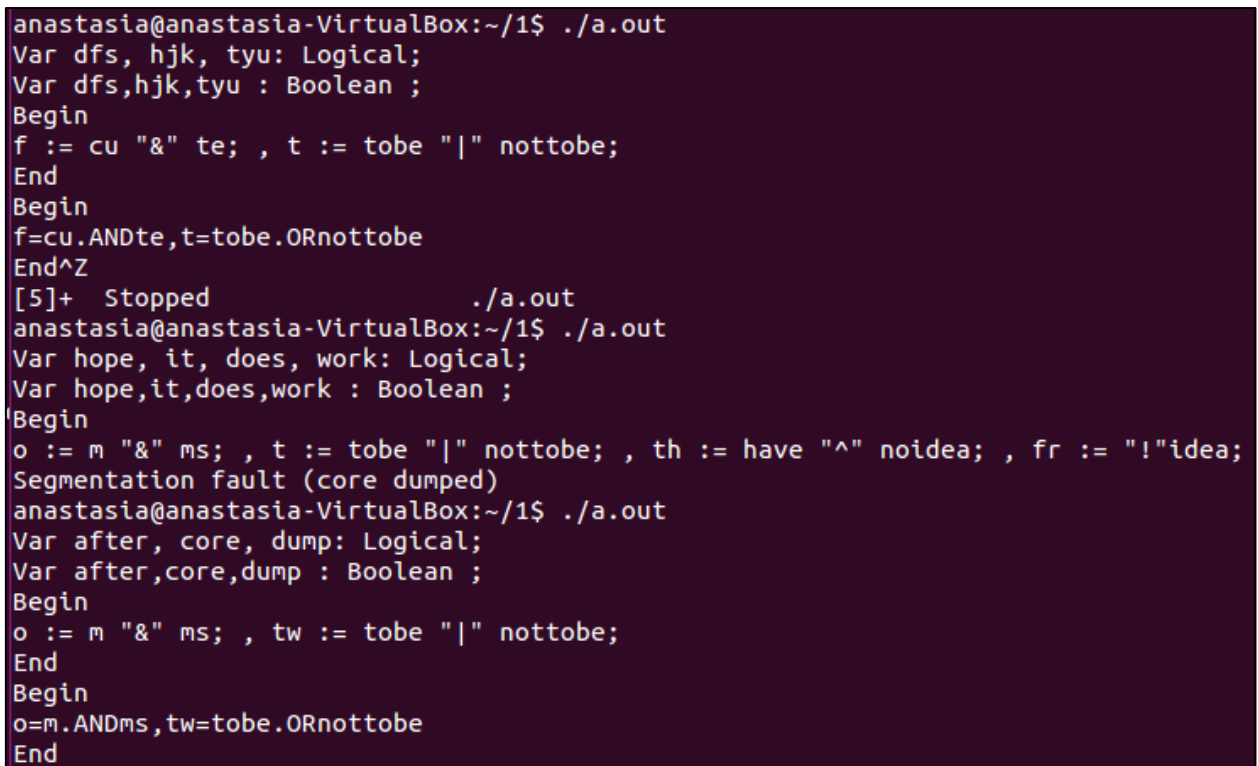
%type <str> PROGRAM VARS OPS VAR APPS IDS APP EXPR SUBEXPR OPERAND
```

Рисунок 5 – Определение типа используемых лексем и нетерминалов

Для осуществления выше указанной задачи необходимо в заголовке использовать следующие объявления:

- %union – объявляет набор типов данных, которые могут иметь семантические значения;
- %token – объявляет терминальный символ (имя типа лексемы) без указания приоритета или ассоциативности;
- %type – объявляет тип семантического значения нетерминального символа.

Полученный в итоге транслятор функционирует следующим образом:



```
anastasia@anastasia-VirtualBox:~/1$ ./a.out
Var dfs, hjk, tyu: Logical;
Var dfs,hjk,tyu : Boolean ;
Begin
f := cu "&" te; , t := tobe "|" nottobe;
End
Begin
f=cu.ANDte,t=tobe.ORnottobe
End^Z
[5]+ Stopped ./a.out
anastasia@anastasia-VirtualBox:~/1$ ./a.out
Var hope, it, does, work: Logical;
Var hope,it,does,work : Boolean ;
Begin
o := m "&" ms; , t := tobe "|" nottobe; , th := have "^" noidea; , fr := "!"idea;
Segmentation fault (core dumped)
anastasia@anastasia-VirtualBox:~/1$ ./a.out
Var after, core, dump: Logical;
Var after,core,dump : Boolean ;
Begin
o := m "&" ms; , tw := tobe "|" nottobe;
End
Begin
o=m.ANDms,tw=tobe.ORnottobe
End
```

Рисунок 6 – Результат работы программы

При появлении на входе анализатора последовательности, не соответствующей описанным правилам, появляется сообщение о наличии синтаксической ошибки.