Lex и Yacc

1. Lex

Lex — программа для генерации лексических анализаторов, обычно используемая совместно с генератором синтаксических анализаторов yacc. Lex был первоначально написан Эриком Шмидтом (*Eric Schmidt*) и Майком Леском (*Mike Lesk*) и является стандартным генератором лексических анализаторов в операционных системах Unix, а также включен в стандарт POSIX. Lex читает входной поток, описывающий лексический анализатор, и даёт на выходе исходный код на языке программирования C.

Структура входного файла

Структура lex-файла сделана подобно структуре yacc-файла; файл разделяется на три блока, разделённые строками, содержащими два символа процента:

```
Блок определений
%%
Блок правил
%%
Блок кода на Си
```

- В блоке **определений** задаются макросы и заголовочные файлы. Здесь также допустимо писать любой код на Си — он будет скопирован в результирующий файл.
- Блок **правил** — наиболее важная часть; она описывает шаблоны и ассоциирует их с вызовами. Шаблоны представляют собой регулярные выражения. Когда анализатор видит текст, подходящий под шаблон, он выполняет указанный код.
- Блок **кода** содержит операторы и функции на Си, которые копируются в генерируемый файл. Предполагается, что эти операторы содержат код, вызываемый правилами из предыдущего блока. Для сложных анализаторов бывает более удобно поместить этот код в отдельный файл, подключающийся на стадии компиляции.

Записывать правила можно в любом текстовом редакторе, сохраняем файл с расширением *.lex. И можно приступать к компиляции.

Пример лексического анализатора:

```
%%
([0-9])+                    printf("Константа ");
(Var)                       printf("Ключевое слово ");
([a-zA-Z]){1}([a-zA-Z0-9]|\_)?+    printf("Идентификатор ");
(\-|\*|\+|\/|\=){1}         printf("Операция ");
(\,|\;|\(|\)){1}            printf("Разделительный символ ");
\n                          printf("\n");
[\t]+                       printf("\t");
%%
```

Чтобы установить флекс используем следующую команду:

$ sudo apt-get install flex

Затем запускаем лекс:

$ lex [имя_файла].lex

На выходе получаем файл lex.yy.c. Его компилируем и запускаем программу:

$ gcc lex.yy.c -lfl

Если требуется установить gcc, устанавливаем.

Далее, запускаем программу:

$ ./a.out

Результат можно увидеть на рисунке ниже:

2. Yacc

yacc — компьютерная программа, служащая стандартным генератором синтаксических анализаторов (парсеров) в Unix-системах. Название является акронимом «Yet Another Compiler Compiler» («ещё один компилятор компиляторов»). Yacc генерирует парсер на основе аналитической грамматики, описанной в нотации BNF (форма Бэкуса-Наура) или контекстно-свободной грамматики. На выходе yacc выдаётся код парсера на языке программирования Си.

Yacc работает на основе Lex. Тоесть Lex возвращает значения или тип лексем, а Yacc уже с ними работает.

Поэтому нам немного придется переписать наш Lex-файл.

```
%{
#include <stdio.h>
#include "y.tab.h"
%}


%%
[0-9]+                          return NUMBER;
(Var)                           return KEY_VAR;
([a-zA-Z]){1}([a-zA-Z0-9]|\_)+  return IDENTIFIER;
(\*|\/){1}                      return OPERATION;
(\;){1}                         return SEMICOLON;
(\,){1}                         return COMMA;
(\=){1}                         return SIGN;
(\-){1}                         return MINUS;
(\+){1}                         return PLUS;
(\(){1}                         return OBRACKET;
(\)){1}                         return EBRACKET;
\n                              ;
[ \t]+
%%
```

Теперь нам нужно написать еще файл для Yacc.

В начале у нас будет вставка с кода, там прописываются дополнительные параметры.

Затем мы перечисляем лексемы, возвращаемые Lex, которые будут использоваться в дальнейшем.

Далее записывается грамматика будущего языка.

Пример Yacc-файла приведен ниже:

```
%{
#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
;
```

Затем производим сборку и запускаем программу:

$ yacc -d file.y && lex 1.lex && gcc lex.yy.c y.tab.c

$ ./a.out

На выходе получим примерно следующее:

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

3. Семантический анализатор

Следующий шаг анализа текста программы – семантический, существенно отличается от двух предыдущих – 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,** т.е. внешними, текстовыми конструкциями языка. Семантика же, ориентированная на содержательную интерпретацию, имеет дело с внутренним представлением «смысла» объектов, описанных в программе. Для любого, имеющего опыт практического программирования, ясно, что формальные конструкции языка дают описание свойств и действий над **внутренними объектами**, с которыми имеет дело программа.

Теперь нам надо, чтобы Lex возвращал не просто тип лексемы, но и ее значение. В связи с этим дополняем Lex-файл.

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

#ifndef YYSTYPE
#define YYSTYPE char *
#endif
#define INTEGER 288
extern YYSTYPE yylval;
%}

%%
[0-9]+          yylval=strdup(yytext); return NUMBER;
(Var)           yylval=strdup(yytext); return KEY_VAR;
([a-zA-Z]){1}([a-zA-Z0-9]|\_)+  yylval=strdup(yytext); return IDENTIFIER;
(\*|\/){1}      yylval=strdup(yytext); return OPERATION;
(\;){1}         return SEMICOLON;
(\,){1}         return COMMA;
(\=){1}         return SIGN;
(\-){1}         return MINUS;
(\+){1}         return PLUS;
(\(){1}         return OBRACKET;
(\)){1}         return EBRACKET;
\n              ;
[ \t]+
%%
```

Так же немного изменяем вторую часть yacc-файла:

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

%%
PROGRAM: VARS OPS
;
// Объявляем переменные
VARS:   VAR
{
  printf("\n%s",$1);
}
|       VARS VAR
{
  printf("%s",$2);
}
;
VAR:    KEY_VAR IDS SEMICOLON
{
  $$ = strcat($1," ");
  $$ = strcat($$,$2);
  $$ = strcat($$,"\n");
}
;
IDS:    IDENTIFIER
{
  $$= $1;
}
|       IDS COMMA IDENTIFIER
 {
   $$ = strcat($1,",");
   $$ = strcat($$, $3);
 }
;
// Работа с присваиваниями
OPS:    OP
{
  printf("%s",$1);
}
|       OPS OP
{
  printf("%s",$2);
}
;
OP:     IDENTIFIER SIGN EXP SEMICOLON
```

Затем производим сборку и запускаем программу:

$ yacc -d file.y && lex 1.lex && gcc lex.yy.c y.tab.c

$ ./a.out

На выходе получаем следующее:

```
Var ioui;
Var adfasf, adsfadsf;
asdfs = 5;
adsf = 78;
Var ioui
Var adfasf,adsfadsf
asdfs := 5

adsf := 78
```

Дополнительную информацию можно найти по следующим ссылкам:

1. http://rus-linux.net/lib.php?name=/MyLDP/algol/lex-yacc-howto.html
2. http://epaperpress.com/lexandyacc/download/LexAndYaccTutorial.pdf
3. http://ermak.cs.nstu.ru/trans/Trans411.htm
4. http://www.cyberforum.ru/cpp-beginners/