# Lex & Yacc[1]

- objetivo genérico: ferramentas de auxílio na escrita de programas que promovem transformações sobre entradas estruturadas.
- objetivo específico: ferramentas desenvolvidas para programadores de compiladores e interpretadores.
- objetivos específicos secundários: ferramentas válidas também para outras aplicações, como detecção de padrões em arquivos de dados, linguagens de comandos, etc.
- vantagens sobre ferramentas alternativas: permitem um rápido desenvolvimento de protótipos e uma manutenção simples do software.
- tanto o Lex como o Yacc foram desenvolvidos nos Bell Laboratories.

[1] Este material está baseado em Levine, Mason & Brown, "lex & yacc", O'Reilly & Associates, Inc., 1995.





- Yacc (Yet another compiler-compiler): Stephen C. Johnson (1975)
- Lex: Mike Lesk & Eric Schmidt → desenvolvido posteriormente para trabalhar junto com o Yacc
- Free Software Foundation → Projeto GNU → flex & bison
- bison: Robert Corbett & Richard Stallman
- versão padronizada: POSIX 1003.2
- existem versões também para arquiteturas padrão IBM/PC
- gcc (GNU C compiler): foi desenvolvido com Lex & Yacc
- etapas:
  - Análise léxica: dividir as entradas em unidades coerentes (*tokens*)
  - Análise sintática: descobrir o relacionamento entre os *tokens*
- toda tarefa que envolve ambas ou uma das etapas acima é candidata a ser resolvida via Lex e/ou Yacc.

- papel do Lex: toma um conjunto de descrições de possíveis *tokens* e produz uma rotina em C que irá identificar estes *tokens* → analisador léxico
- descrições de possíveis *tokens*: expressões regulares (especificação léxica)

- papel do Yacc: toma uma descrição concisa de uma gramática e produz uma rotina em C que irá executar a análise sintática ou *parsing*.

- nota1: um analisador léxico desenvolvido usando Lex é quase sempre mais rápido do que um analisador léxico escrito diretamente em C.
- nota2: um analisador sintático desenvolvido utilizando Yacc é geralmente mais lento do que um analisador sintático escrito diretamente em C. Mas o ganho no desenvolvimento e manutenção do programa é enorme.





# 1. Geradores de Analisadores Léxicos

- a análise léxica pode ser vista como a primeira fase do processo de compilação.
- sua principal tarefa é ler uma seqüência de caracteres de entrada, geralmente associados a um código-fonte, e produzir como saída uma seqüência de itens léxicos.
- por outro lado, a análise sintática tem por objetivo agrupar os itens léxicos em blocos de comandos válidos, procurando reconstruir uma árvore sintática, conforme ilustrado na figura 1.
- os itens léxicos são geralmente denominados *tokens* e correspondem a palavras-chave, operadores, símbolos especiais, símbolos de pontuação, identificadores (variáveis) e literais (constantes) presentes em uma linguagem.

- o analisador léxico pode executar também algumas tarefas secundárias de interface com o usuário, como reconhecer comentários no código-fonte, eliminar caracteres de separação (espaço, tabulação e fim de linha), associar mensagens de erro provenientes de outras etapas do processo de compilação com as linhas correspondentes no código-fonte, realizar pré-processamento, etc.

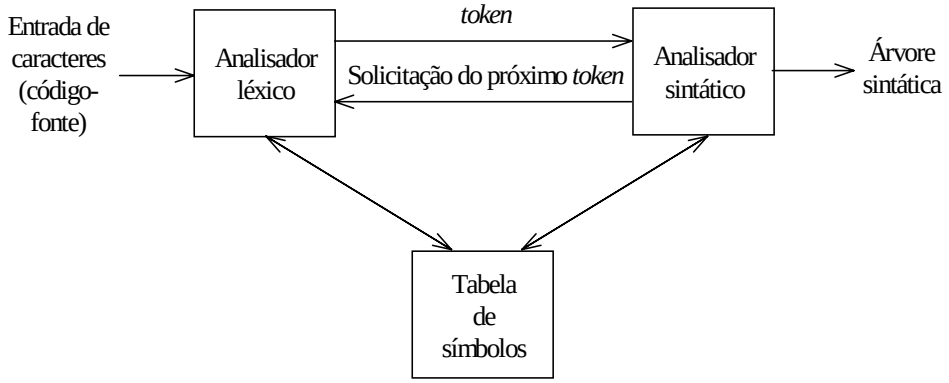


Figura 1 - Interação entre análise léxica e sintática





## 1.1. Lex

- a ferramenta de automatização da análise léxica a ser estudada utiliza o comando **`lex`** disponível no UNIX (ou seu correspondente **`flex`** da GNU). O programa Lex produz automaticamente um analisador léxico a partir de especificações de **expressões regulares**, passíveis de representação na forma de **autômatos finitos**.
- o programa Lex é geralmente utilizado conforme ilustrado na figura 2. O formato geral para a especificação de uma gramática regular em Lex é

```
definições          /* opcional */
%%
regras
%%
rotinas do usuário  /* opcional */
```

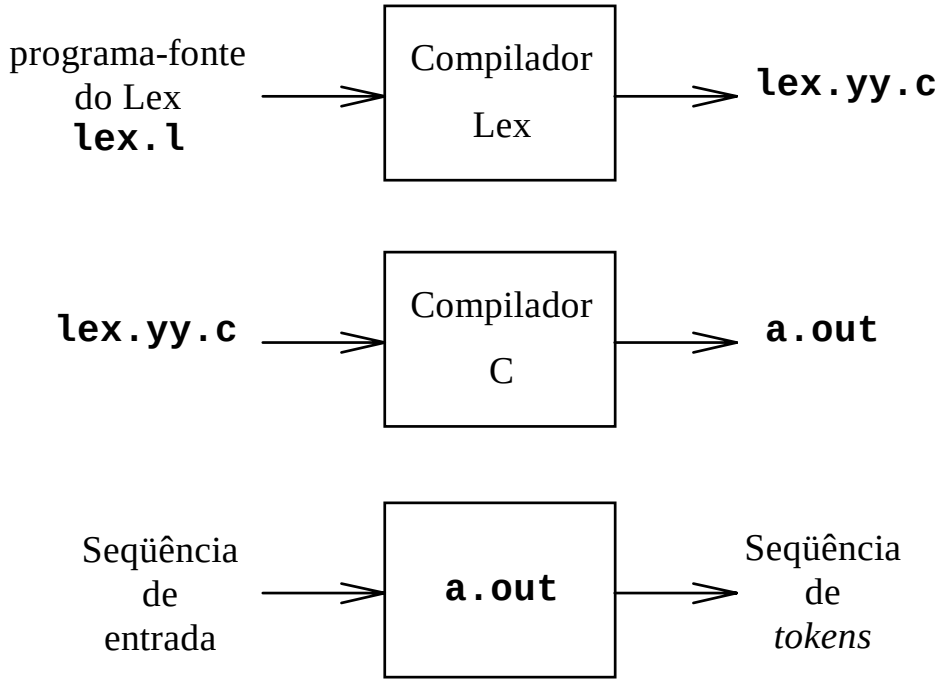


Figura 2- Criando um analisador léxico com Lex





- a seção de definições inclui:

1. a definição de macros como:

```
digito [01]+        /* substituir {digito} por [01]+ ao processar as regras */
frac   .[0-9]+      /* substituir {frac} por .[0-9]+ ao processar as regras */
nl     \n           /* substituir {nl} por \n ao processar as regras */
```

2. a inclusão das linhas de comando em C, que devem ser delimitadas por <%{> e <%}>, como:

```
%{
#include <y.tab.h>
extern int yylval;
%}
```

- a redefinição dos tamanhos das tabelas utilizadas pelo gerador, tais como número de nós da árvore sintática, número de transições e número de estados.

- a seção de regras define a funcionalidade do analisador léxico. Cada regra compreende uma seqüência válida de caracteres (utilizando literais e expressões regulares) e as ações semânticas associadas a ela.
- Lex armazena temporariamente a subseqüência de caracteres identificada na variável **`yytext`** do tipo <char *>. Podemos, então, usar a função **`sscanf()`** da biblioteca de C para convertê-la em outros tipos de dados. A variável reservada pelo Lex para armazenar o resultado da conversão é **`yylval`**. A seção de rotinas do usuário é opcional e ela é usada para incluir rotinas em C desenvolvidas pelos usuários, como o programa **`yylex()`** que chama o analisador léxico gerado pelo Lex.
- quando uma seqüência de caracteres de entrada casa com mais de uma regra, Lex adota uma das seguintes estratégias para resolver ambigüidades:
- escolher a regra que consegue casar a maior seqüência de caracteres possível;





- quando há mais de uma regra que case com a maior seqüência de caracteres, escolher aquela que aparece primeiro na seção de regras.
- embora o Lex seja relativamente simples de entender, ele é freqüentemente associado ao Yacc (já não tão simples) em aplicações no domínio dos compiladores. No entanto, o Lex pode (e deve) ser utilizado também como uma ferramenta única.

# 2. Geradores de Analisadores Sintáticos

- Ferramentas similares e complementares aos geradores de analisadores léxicos.
- Ferramenta mais comum: **yacc** *(**y**et **a**nother **c**ompiler-**c**ompiler)*
- **lex** gera a função **yylex( )** que retorna o identificador de um item léxico reconhecido.

- **yacc** gera a função **yyparse( )**, que analisa os itens léxicos e decide se eles formam ou não uma sentença válida.

## 2.1. Passos para gerar analisador sintático

- Escrever programa analisador léxico ou usar ferramenta **lex** para tal.
- Escrever gramática da linguagem em arquivo com extensão ***.y*** .
- Escrever programa analisador sintático ou usar a ferramenta **yacc** para tal.
- Compilar e ligar os programas-fonte.
- Para **yylex( )** e **yyparse( )** trabalharem em conjunto é preciso garantir que os tokens e seus valores sejam comuns às duas funções.





## 2.2. Formato geral da gramática de Yacc

- Similar ao da gramática de **lex**
- três seções no arquivo ***.y***:

**definições** (declaração de nomes e tipos de tokens, de variáveis, etc)

**%%**

**regras** (contém as produções da gramática em BNF)

**símbolo : derivações {ações semânticas}**

**%%**

**rotinas do usuário** (contém a função principal main( ) e outras rotinas do usuário)

## 3. Cooperação entre LEX e YACC

- para obter um analisador sintático com o uso de Lex e Yacc é necessário (veja figura 3):

1. especificar a gramática de definição dos itens léxicos em linguagem de especificação de Lex num arquivo, por convenção com extensão <.l>, e gerar o analisador léxico em C (**`yylex()`**) através do seguinte comando: **`flex <`*`filename`*`>.l`**. O arquivo que contém **`yylex()`** é denominado <**lex.yy.c**>.
2. especificar a estrutura sintática da linguagem através da linguagem de especificação de Yacc num arquivo, por convenção com extensão <.y>, e gerar o analisador sintático em C (**`yyparse()`**) através do seguinte comando: **`yacc -d <`*`filename`*`>.y`**. A chave <**`d`**> indica que um arquivo de definição de itens léxicos e tipo de dados da variável global <**`yylval`**>,





***filename*.tab.h**, deve ser gerado. Este arquivo estabelece a dependência simbólica entre o analisador léxico e o sintático. Yacc inclui automaticamente as chamadas a **`yylex()`** para obter os itens léxicos e os seus valores. O arquivo que contém **`yyparse()`** é chamado <***filename*.tab.c**>;

3. compilar e ligar os programas-fonte com outros programas adicionais através do seguinte comando:

   **`cc -o `*`filename filename`*`.tab.c lex.yy.c -ly -ll`**

   para gerar o código executável, por exemplo <*filename*>. Os indicadores <**`y`**> e <**`l`**> correspondem à inclusão de rotinas das bibliotecas em C <**liby.a**> e <**libl.a**>, respectivamente.

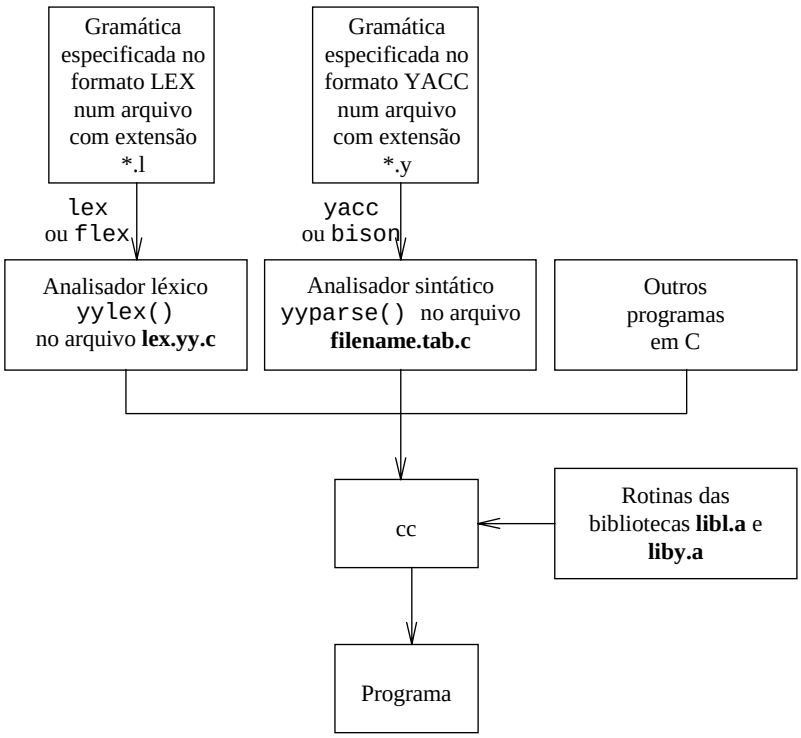


Figura 3 - Esquema do uso de Lex e Yacc





**Exemplos:** Os programas a seguir devem ser compilados com o comando **`flex p<?>.l`**, seguido do comando **`gcc lex.yy.c -ll -o p<?>`**.

**p1.l**

```
%%
[A-Z]      printf("%c", yytext[0]+'a'-'A');
.          ECHO;
```

Execute o seguinte comando: **`p1 < p1.l`**

**p2.l**

```
%{
    int nlin=0;
%}
Qqchar              .*
NovaLinha           [\n]
%%
^{Qqchar}{NovaLinha} {nlin=nlin+1;printf("%4d\t%s", nlin, yytext);}
```

Execute o seguinte comando: **`p2 < p2.l`**

**p3.l**

```
Dia         [" "](00-11)[:]
Tarde       [" "](12-17)[:]
Noite       [" "](18-23)[:]
%%
{Dia}   printf("\nBom Dia !\n");
{Tarde} printf("\nBoa Tarde !\n");
{Noite} printf("\nBoa Noite !\n");
.
```

Execute o seguinte comando: **`date | p3`**

**p4.l**

```
%{
int count=0;
%}
%%
[^ \t\n]+ count++;
%%
main(){
    yylex();
    printf("\n%d palavras contadas.\n", count);
}
```





**p5.l**

```
%%
[.]             ;
then            ;
var             ;
[{]             printf("/*");
[}]             printf("*/");
mod             printf("%%");
or              printf("||");
and             printf("&&");
begin           printf("{");
end             printf("}");
program.*[(].*$ printf("main()\n{");
[^:><][=]       printf("==");
[:][=]          printf("=");
[<][>]          printf("!=");
^.*integer;     ShuffleInt();
.               ECHO;
%%
ShuffleInt()
{ int i;
    printf("int ");
    for(i=0; yytext[i]!=':'; i++)  printf("%c", yytext[i]);
    printf(";");
}
```

**p6.l**

```
%%
[1-9][0-9]*          printf("Decimal\n");
0[0-7]+              printf("Octal\n");
0[x|X][0-9A-Fa-f]+   printf("\nHexadecimal\n");
.?                   printf("Nao numerico\n");

******************************
Calculadora (lex)
******************************
%{
#include "y.tab.h"
%}
integer    [0-9]+
nl         \n
%%
[ \t]+     ;
{integer}  {sscanf(yytext,"%d",&yylval.integer);return INTEGER;}
{nl}       {extern int lineno; lineno++; return '\n'; }
.          {return yytext[0];}
%%
```





```
******************************
Calculadora (yacc)
******************************
%{
#include <stdio.h>
%}
%union {
   double real;
   int integer;
 }
%token <integer> INTEGER
%type <integer> expr;
%left '+' '-'
%left '*' '/'
%nonassoc UMINUS

%%
lines:  /* nothing */
      | lines line
      ;

line: '\n'
     | expr '\n' {printf("%d\n", $1);}
     | error '\n'  {yyerror();}
     ;
```

```
expr: INTEGER
     | expr '+' expr  {$$ = $1 + $3;}
     | expr '-' expr  {$$ = $1 - $3;}
     | expr '*' expr  {$$ = $1 * $3;}
     | expr '/' expr  {if($3) $$ = $1 / $3;
                       else {
                              yyerror("divide by zero");
                            }
                      }
     | '(' expr ')' {$$ = $2;}
     | '-' expr %prec UMINUS {$$ = - $2;}
     ;

%%
int lineno = 0;
main(){
    yyparse();
}

yyerror(s)
char *s;
{
printf("calc: %s", s);
printf("line %d\n", lineno);
}
```





```
*****************************
Calculadora "científica" (lex)
*****************************

%{
#include "y.tab.h"
%}

integer     [0-9]+
real        [0-9]*"."[0-9]*
nl          \n

%%
[ \t]+  ;
{integer}   {sscanf(yytext, "%d", &yylval.integer);  return INTEGER;}
{real}      {sscanf(yytext, "%lf", &yylval.real);  return REAL;}
{nl}        {extern int lineno; lineno++; return '\n'; }
sin         {return SIN;}
cos         {return COS;}
tan         {return TAN;}
.           {return yytext[0];}
%%
```

```
*****************************
Calculadora "científica" (yacc)
*****************************
%{
#include <stdio.h>
#include <math.h>
%}

%union {
   double real;
   int integer;
 }
%token <integer> INTEGER
%token <real> REAL
%token SIN COS TAN
%type <integer> expr_i;
%type <real> expr;
%left '+' '-'
%left '*' '/'
%nonassoc UMINUS

%%
lines:  /* nothing */
      | lines line
      ;
```





```
line: '\n'
     | expr '\n' {printf("%lf\n", $1);}
     | error '\n'  {yyerror();}
     ;

expr_i: INTEGER;

expr: REAL
     | expr_i {$$ = $1;}
     | SIN '(' expr ')' {$$ = sin($3);}
     | COS '(' expr ')' {$$ = cos($3);}
     | TAN '(' expr ')' {$$ = tan($3);}
     | expr '+' expr  {$$ = $1 + $3;}
     | expr '-' expr  {$$ = $1 - $3;}
     | expr '*' expr  {$$ = $1 * $3;}
     | expr '/' expr  {if($3) $$ = $1 / $3;
                       else {
                              yyerror("divide by zero");
                            }
                      }
     | '(' expr ')' {$$ = $2;}
     | '-' expr %prec UMINUS {$$ = - $2;}
     ;
```

```
%%
int lineno = 0;
main(){
yyparse();
}
yyerror(s)
char *s;
{
    printf("calc: %s", s);
    printf("line %d\n", lineno);
}
```

## 4. Autômatos finitos

Definição: Um autômato finito $M$ é uma quíntupla ordenada $M = \{S, s_0, F, A, g\}$:

1. $S \neq \varnothing$ é um conjunto finito de estados;
2. $s_0 \in S$ é o estado inicial;
3. $F \in S$ é um conjunto de estados finais;
4. $A$ é um alfabeto de entrada;
5. $g$: $S \times A \rightarrow S$ é um conjunto finito de aplicações de transição.





## 5. Expressões regulares

- Uma expressão regular é um objeto matemático específico que permite a escrita concisa de uma seqüência válida de caracteres. Para tanto, são utilizados operadores na forma de metacaracteres, sendo que os mais comuns são apresentados na tabela 1.
- Mesmo sabendo ser possível descrever a sintaxe léxica de uma linguagem utilizando gramáticas livres de contexto, esta descrição geralmente é feita por meio de expressões regulares pelas seguintes razões:

1. as regras léxicas de uma linguagem são geralmente muito simples, não necessitando uma descrição gramatical (recurso demasiadamente poderoso para tal fim);
2. expressões regulares geralmente fornecem uma notação mais concisa e fácil de entender do que a correspondente representação gramatical;
3. analisadores léxicos automáticos construídos a partir de expressões regulares são mais eficientes que aqueles construídos a partir de gramáticas arbitrárias.

**Tabela 1 - Metacaracteres utilizados em expressões regulares**

| Operador | Significado | Exemplo |
|---|---|---|
| ? | a expressão anterior é opcional | 54?1 ≡ 541 ou 51 |
| * | qualquer repetição, inclusive vazio | a* ≡ {∅,a,aa,...} |
| + | qualquer repetição, exclusive vazio | a+ ≡ {a,aa,...} |
| \| | alternativa entre duas expressões | a\|b ≡ a ou b |
| () | agrupamento de expressões | |
| . | qualquer caracter, exceto *linefeed* | |
| ^ | casa com o início de uma linha, exceto quando está entre [ ], quando significa complemento. | |
| $ | fim de uma linha | |
| [ ] | qualquer caracter especificado | [abc] ≡ {a,b,c} |
| - | dentro de [ ], qualquer caracter entre os extremos | [0-9] ≡ {0,1,2,3,4,5,6,7,8,9} |
| { } | indica o número de repetições permitido, ou substitui uma definição macro | a{1,2} ≡ {a,aa}<br>{digito} ≡ [0-1]+ |
| \ | permite interpretar o próximo caracter como caracter comum. É também utilizado para representar caracteres não-imprimíveis | . ≡ \.<br>\t ≡ tabulação<br>\b ≡ *blank*<br>\n ≡ *linefeed* |
| / | especifica um conjunto de seqüências seguida de uma expressão | [012]+/Y aceita qualquer seqüência composta de 0, 1 e 2 seguida de Y |





## Exercícios

1. Apresente a expressão regular mais compacta que seja capaz de descrever qualquer número decimal positivo (e somente eles), com a parte inteira sendo separada da parte fracionária por ponto, quando for o caso. Por exemplo, são seqüências válidas: 0.89 / 12 / 4.0 / .32748 / 34.52

2. Como uma alternativa à expressão regular obtida acima, apresente o autômato finito que permite validar ou não qualquer número decimal positivo. Resolva a questão com base nas seguintes hipóteses:

- existem dois tipos de símbolos terminais ou eventos: "[0-9]" e ".";
- utilize dois estados finais, lembrando que a seqüência de caracteres só é válida se for atingido um estado final **e** não houver mais caracteres a processar.
- para qualquer estado, deve haver uma lei de transição de estados a partir de cada um dos eventos existentes.