**EA872 - Laboratório de Programação de Software Básico**

**Atividade 2**

# 1. Tema

Interpretadores de comandos, compiladores e ferramentas para geração de novas linguagens.

# 2. Objetivos

Familiarização com processos de geração automática de analisadores léxicos e sintáticos. Compreensão dos papéis complementares entre a análise léxica e a sintática no processo de reconhecimento de uma sentença válida. Uso de gramáticas formais para especificação das funcionalidades de analisadores. Associação de ações semânticas (síntese de resultados) aos comandos interpretados. Implementação de um *parser* para os comandos do protocolo http que fará parte do servidor HTTP em desenvolvimento.

# 3. Interpretadores e Compiladores

Interpretadores e compiladores são programas que processam dados de entrada (instruções de programas ou comandos isolados) e geram ações ou dados de saída (programa compilado). Normalmente os dados de entrada consistem de uma sequência de caracteres (por exemplo, digitados pelos usuários ou provenientes de um arquivo-texto) e a saída, de uma série de ações estabelecidas pela aplicação.

Basicamente, distinguem-se duas fases no processo de compilação ou interpretação:

- **análise**: validação da sequência de caracteres de entrada de acordo com a sintaxe da linguagem. Os caracteres são agrupados de forma a permitir decidir quais ações semânticas são aplicáveis sobre eles;
- **síntese**: aplicação das ações semânticas sobre grupos de caracteres. As ações semânticas mais comuns são a geração de códigos de máquina (código-objeto) em um arquivo, no caso de compiladores, ou a execução de comandos analisados e apresentação dos resultados, no caso de interpretadores.

Sob o ponto de vista de implementação, é mais eficiente separar a fase de análise em duas etapas: análise léxica e análise sintática. As principais razões para tal procedimento são:

- significativa simplificação na implementação, tanto da análise léxica como sintática, quando ambas são realizadas separadamente. Com isso, aumenta-se a possibilidade de utilização de ferramentas de automatização das análises léxica e sintática;
- a eficiência e a portabilidade do compilador ou interpretador são melhoradas.

Se um compilador ou interpretador tivesse que processar apenas programas corretos, seu projeto e implementação seriam enormemente simplificados. No entanto, eles possuem a importante tarefa de assessorar o programador na identificação e localização de erros. Por outro lado, a grande maioria das especificações de linguagem de programação não descrevem como os compiladores ou interpretadores devem responder a erros, que podem ser léxicos, sintáticos, semânticos ou lógicos. A resposta é deixada a cargo do projetista do compilador ou interpretador. Além disso, as linguagens de programação exigem muito mais precisão e rigor que as linguagens faladas (por exemplo, não pode existir ambiguidade nas linguagens de programação).

O primeiro compilador Fortran foi construído por um grupo de especialistas que levou vários anos para escrevê-lo. Hoje, um único programador com conhecimentos incipientes pode obter o mesmo resultado em poucas semanas. O que torna isto possível são os avanços na ciência de computação e nas ferramentas de desenvolvimento de software.

# 4. Sistemas formais

## 4.1. Gramáticas Formais

As regras que estabelecem concatenações válidas de caracteres na sequência de entrada constituem a sintaxe de uma linguagem de programação. Quando uma linguagem abrange um conjunto finito de concatenações, pode-se validar qualquer cadeia de caracteres especificada através de uma busca exaustiva contendo todas as sequências válidas preestabelecidas. Este método, além de ser ineficiente, não consegue processar uma linguagem que contém um número infinito, porém enumerável, de concatenações válidas, como é o caso das linguagens de alto nível.

Portanto, é necessário **formalizar** o processo de construção dos programas em uma linguagem, visando definir um algoritmo para gerá-los sistematicamente por meio de uma gramática formal. Um sistema formal consiste de um **conjunto de palavras** (alfabeto) e um conjunto finito de relações denominadas **regras de inferência**.

Definição: Uma gramática formal $G$ é uma quádrupla ordenada $G = (N,T,\Sigma,P)$, onde:
1. $N$ é um conjunto finito de símbolos não-terminais;
2. $T$ é um conjunto finito de símbolos terminais;
3. $\Sigma \in N$ é o símbolo não-terminal inicial;
4. $P$ é o conjunto finito de produções.

Exemplo: $N = \{\Sigma, \text{inteiro, digito}\}$
$T = \{1,0\}$
$P = \{\Sigma \rightarrow \text{inteiro};$
inteiro $\rightarrow$ inteiro digito;
inteiro $\rightarrow$ digito;
digito $\rightarrow 1$;
digito $\rightarrow 0\}$

## 4.2. Expressões Regulares

Uma expressão regular permite a escrita concisa de uma sequência válida de caracteres, sem recorrer a gramáticas formais. Para tanto, são utilizados operadores na forma de metacaracteres, sendo que os mais comuns são apresentados na Tabela 1 (mais detalhes podem ser obtidos em sistemas unix com o comando `man grep`). Mesmo sendo possível descrever a sintaxe léxica de uma linguagem utilizando gramáticas formais livres de contexto, esta descrição geralmente é feita por meio de expressões regulares pelas seguintes razões:
- as regras léxicas de uma linguagem são geralmente muito simples, não necessitando uma descrição gramatical (recurso demasiadamente poderoso para tal fim);
- expressões regulares geralmente fornecem uma notação mais concisa e fácil de entender do que a correspondente representação gramatical;
- 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 (mais detalhes: `man regexp`)

| 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 caractere, exceto *linefeed* | |
| ^ | casa com o início de uma linha, exceto quando está entre [ ], quando significa complemento. | |
| $ | fim de uma linha | |
| [ ] | qualquer caractere especificado | [abc] ≡ {a,b,c} |
| - | dentro de [ ], qualquer caractere 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 caractere como caractere comum. É também utilizado para representar caracteres não-imprimíveis | . ≡ \.<br>\t ≡ tabulação<br>\b ≡ *blank*<br>\n ≡ *linefeed* |
| / | especifica um conjunto de sequências seguida de uma expressão | [012]+/Y aceita qualquer sequência composta de 0, 1 e 2 seguida de Y |

# 5. Planejando um compilador

A Fig. 1 ilustra as várias etapas pelas quais passa um compilador, que nada mais é que um programa que envolve três linguagens em seu desenvolvimento:

- a linguagem-fonte a ser compilada;
- a linguagem-objeto do código a ser gerado pelo compilador;
- a linguagem de implementação do compilador (na qual ele é escrito e compilado).

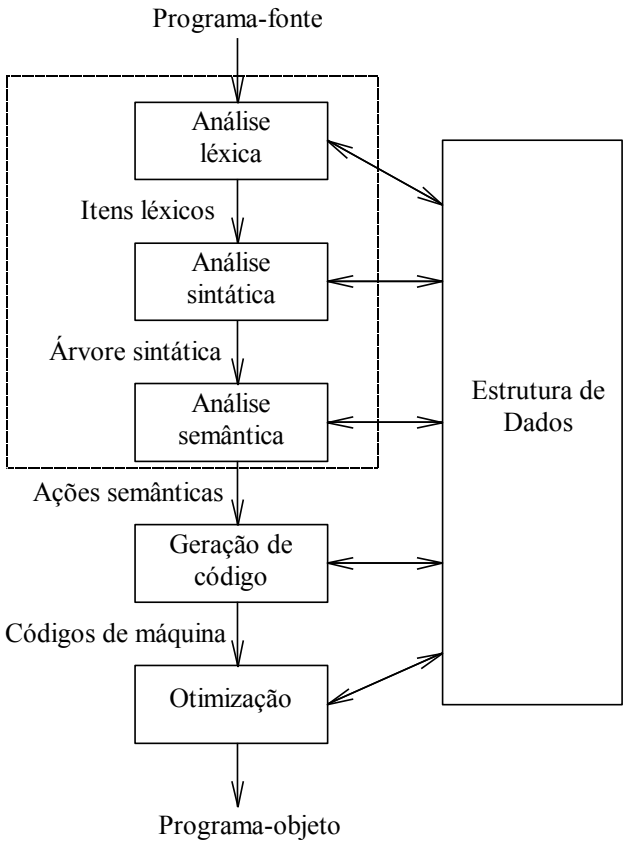


Figura 1 - Tarefas básicas de um compilador

No projeto de um compilador, deve-se considerar as seguintes especificações: "tamanho" da linguagem, extensão das mudanças na linguagem-fonte durante a construção do compilador e critérios de desempenho (velocidade do compilador, qualidade do código, diagnóstico de erros, portabilidade e manutenção). Além disso, é necessário ter presente que:

- um compilador portável pode não ser tão eficiente quanto um compilador projetado para uma máquina específica;
- a escrita de um compilador é um projeto complexo que precisa de recursos de engenharia de software;
- raramente é necessário desenvolver uma organização de compilador completamente nova. Sendo assim, o desenvolvedor pode adotar a organização de um compilador conhecido (de uma linguagem semelhante);
- um compilador é um programa suficientemente complexo para justificar seu desenvolvimento em uma linguagem mais amigável que a linguagem assembly;
- no ambiente de programação do UNIX, os compiladores são frequentemente escritos em C. Mesmo os compiladores C são escritos em C;
- um compilador pode rodar em uma arquitetura e produzir código-objeto para outra arquitetura (*cross-compiler*).

# 6. Análise léxica

A análise léxica pode ser vista como a primeira fase do processo de compilação. Sua principal tarefa é ler uma sequência de caracteres de entrada, geralmente associados a um código-fonte, e produzir como saída uma sequência 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 2. Os itens léxicos são denominados *tokens* e correspondem a palavras-chave, operadores, símbolos especiais, símbolos de pontuação, identificadores (variáveis) e literais (constantes) presentes em uma linguagem.

Ferramentas de software que automatizam a construção de analisadores léxicos e sintáticos tornam essas tarefas acessíveis a usuários que não sejam especialistas em compiladores.

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.

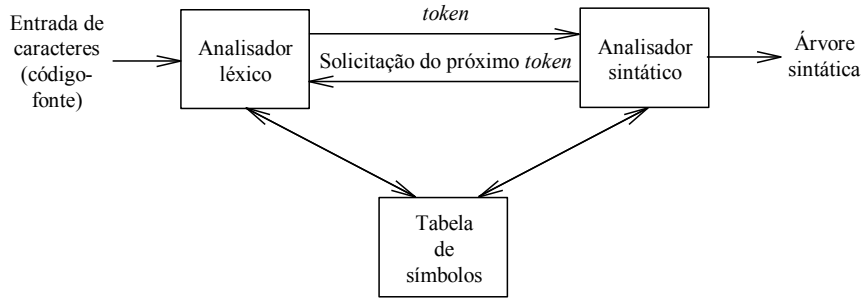


Figura 2 - Interação entre análise léxica e sintática

## 6.1. LEX

A ferramenta de automatização da análise léxica a ser estudada nesta atividade 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**.

O programa LEX é geralmente utilizado conforme ilustrado na Figura 3. O formato geral para a especificação de uma gramática regular em LEX é

```
definições              /* opcional */

%%

regras

%%

rotinas do usuário      /* opcional */
```

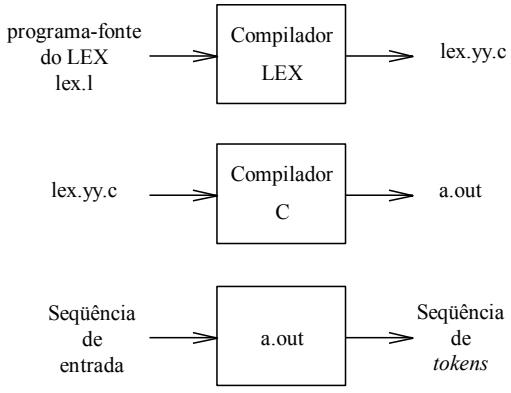


Figura 3 - Criando um analisador léxico com LEX

A seção de definições inclui:

- 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 (newline) ao processar as regras */
```

- 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 sequência válida de caracteres (utilizando literais e expressões regulares) e as ações semânticas associadas a ela.

LEX armazena temporariamente a subsequência 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 e para chamar, por meio da função `yylex()`, o analisador léxico gerado pelo LEX.

Quando uma sequência de caracteres de entrada casa com mais de uma regra, LEX adota uma das seguintes estratégias para resolver ambiguidades:

- escolher a regra que consegue casar a maior sequência de caracteres possível;
- quando há mais de uma regra que case com a maior sequência de caracteres, escolher aquela que aparece primeiro na seção de regras.

Embora o LEX seja frequentemente associado ao YACC (ou `bison`) em aplicações no domínio dos compiladores, ele pode (e deve) ser utilizado também como uma ferramenta única, conforme será abordado nas atividades práticas.

# 7. Análise sintática

Existem três tipos de analisadores sintáticos:

- universal: pode analisar qualquer tipo de gramática, sendo ineficiente na produção de compiladores;
- *top-down*: começa a análise da raiz, caminhando para as folhas da árvore sintática;
- *bottom-up*: começa a análise pelas folhas, caminhando para a raiz da análise sintática.

Os analisadores sintáticos mais eficientes trabalham apenas com subclasses de gramáticas. Subclasses como LL (Left-Left) e LR (Left-Right) são suficientes para descrever a maioria das construções sintáticas em linguagens de programação.

## 7.1. YACC

A ferramenta de automatização da análise sintática a ser estudada nesta atividade utiliza o comando `yacc` disponível no UNIX (ou `bison` da GNU). O programa YACC produz automaticamente um analisador sintático LR a partir de uma descrição gramatical de uma linguagem. É bem mais fácil produzir um analisador sintático utilizando uma descrição gramatical de uma linguagem e um gerador do analisador sintático (a partir desta descrição gramatical) do que implementar diretamente um analisador sintático.

YACC é uma abreviatura de "*yet another compiler-compiler*". Este nome reflete a popularidade de geradores automáticos de analisadores sintáticos no início dos anos 70, quando a primeira versão de YACC foi criada por Johnson (1975). YACC tem sido utilizado para auxiliar na implementação de centenas de compiladores, ao transformar uma extensa classe de gramáticas livres de contexto em analisadores sintáticos (*parsers*). Além disso, através da linguagem de especificação de YACC, é possível associar, a cada produção em BNF (Backus-Naur Form), ações semânticas a serem executadas. Essas ações, como atribuir o resultado de processamento de símbolos no lado direito ao símbolo não-terminal no lado esquerdo de uma produção, podem ser blocos de programas em C. A linguagem de especificação de YACC considera como unidades atômicas os itens léxicos (*tokens*) retornados pelo analisador léxico criado por LEX.

O formato geral para a especificação de uma gramática livre de contexto em YACC é bastante similar ao do LEX.

```
definições              /* opcional */

%%

regras

%%

rotinas do usuário      /* opcional */
```

A seção de definições inclui declarações em C delimitadas por <%{> e <%}> e informações que afetam a operação do gerador YACC através do uso das palavras-chaves apresentadas na tabela a seguir.

A seção de regras contém o conjunto de produções da gramática. Cada produção é descrita num formato similar ao formato BNF:

```
símbolo :  derivações
           {ações semânticas}
           ;
```

O uso de literais delimitados por ' é permitido nas derivações e a especificação de ações semânticas é opcional. A seção de rotinas do usuário é opcional e ela é usada para incluir rotinas em C desenvolvidas pelos usuários tais como um programa que chama o analisador sintático `yyparse()` gerado.

| Palavra-chave | Tipo de declaração | Exemplo |
|---|---|---|
| %token | os nomes dos tokens | %token INT INVALIDO NL |
| %left | operadores que reduzem pelo lado esquerdo | %left '+', '-' |
| %right | operadores que reduzem pelo lado direito | %right '=' |
| %nonassoc | operadores que não devem se associar | %nonassoc '<=' (A <= B <= C não é permitido) |
| %prec | a mudança do nível de precedência de um operador numa produção particular para o do item léxico que segue a palavra-chave | exp: '-' exp %prec '*' (o operador '-' tem a mesma precedência de '*' nesta regra) |
| %type | tipo de símbolos não-terminais | %type <integer> inteiro digito |
| %union | possíveis tipos de dados dos itens léxicos reconhecíveis pelo LEX | %union { int ival; double dval; } |
| %start | símbolo inicial. Por default é o símbolo da primeira produção na seção de regras | %start sigma |

## 7.2. Cooperação entre LEX e YACC

Para obter um analisador sintático com o uso de LEX e YACC, precisaremos (veja Figura 4):

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()`) usando o seguinte comando: `flex <nome do arquivo>.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 <nome do arquivo>.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:

```
gcc -o analisador analisador.tab.c lex.yy.c -ly -ll  (ou -lfl no caso de FLEX)
```

para gerar o código executável, por exemplo <analisador>. Os parâmetros `-ly` e `-ll` (ou `-lfl`) correspondem à inclusão de rotinas das bibliotecas em C <**liby.a**> e <**libl.a**> (ou <**libfl.a**> no caso de `flex`), respectivamente.

### 7.3. Exemplo

A gramática apresentada como exemplo na Seção 4, descreve completamente a sintaxe de números binários. Ela pode ser desmembrada em duas partes: a definição da formação de itens léxicos - os dígitos - e a especificação da concatenação destes dígitos para formar um número binário. Podemos ter as seguintes definições em linguagem de especificação LEX:

```
%{
#include "y.tab.h"           /* definicao das classes de itens lexicos */
                              /* o nome do arquivo deve ser
%}
%%
[01]+   {return INT; }       /* valores 0 e 1 com repeticao */
\n      {return NL; }        /* line feed */
.       {return INVALIDO; }  /* qualquer outro caracter, exceto linefeed */
```

e em linguagem de especificação de YACC:

```
%token INT INVALIDO NL
%%
start   : inteiro NL
        ;
inteiro : inteiro digito
        | digito
        ;
digito  : INT
        ;
```

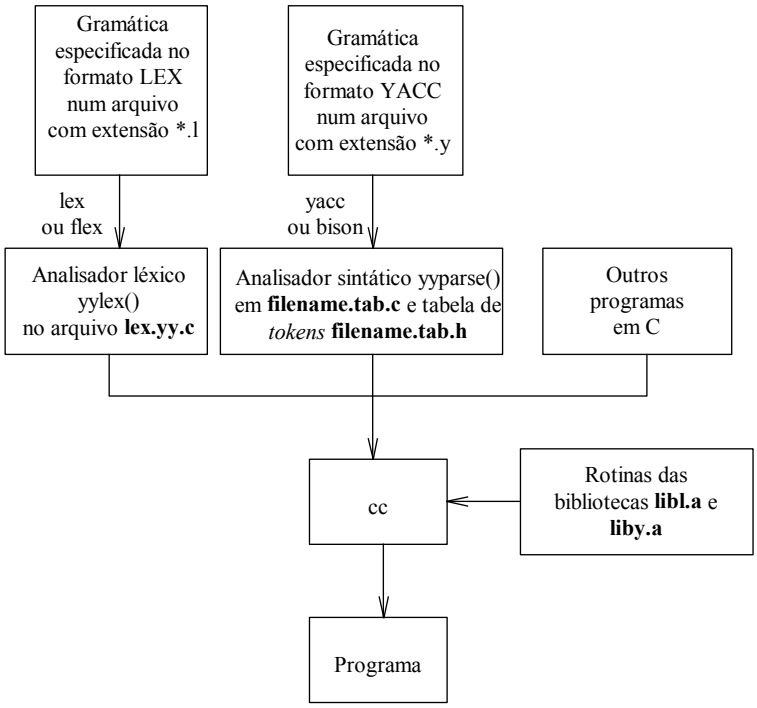


Figura 4 - Esquema do uso de LEX e YACC

A cada produção podemos ainda associar ações semânticas. Neste caso, atribuímos ao símbolo não-terminal os valores intermediários dos números formados pela concatenação dos símbolos no lado direito de cada produção:

```
%token INT INVALIDO NL

%%
```

```
start   : inteiro NL
              {$$ = $1;
               printf ("valor = %d\n", $$); exit(1); }
         ;
inteiro : inteiro digito
              {$$ = 10 * $1 + $2; }
         | digito
              {$$ = $1; }
         ;
digito  : INT
              {$$ = $1; }
         ;
```

Observe que os componentes do lado direito de cada produção são denotados pelas pseudo-variáveis $1, $2, etc. Os valores 1, 2, etc. indicam o primeiro, o segundo, etc. elemento do lado direito da produção. O valor retornado pelas ações semânticas é atribuído à pseudo-variável $$, que corresponde ao símbolo não-terminal do lado esquerdo da produção. Assim, todas as outras produções que utilizam o tal símbolo não-terminal terão acesso ao valor deste símbolo.

O valor do item léxico INT deve ser passado pela variável `yylval`, cujo tipo de dado é definido na especificação da linguagem YACC. Isso deve ser contemplado pela linguagem de especificação de LEX, como mostra o seguinte bloco:

```
%{
#include "filename.tab.h"    /* definicao das classes de itens lexicos */
extern int yylval;
%}
%%
[01]+   {sscanf (yytext, "%d", &yylval);
         return INT; }       /* valores 0 e 1 com repeticao */
\n      {return NL; }        /* line feed */
.       {return INVALIDO; }  /* qualquer outro caracter, exceto line feed */
```

# 8. Atividades Práticas

Para entender melhor as ferramentas lex (ou flex) e yacc (ou bison), estude os programas abaixo, fazendo referência sempre que necessário às páginas de manual online do Unix.

### Programando com LEX

Os códigos a seguir estão disponíveis no arquivo zip fornecido junto com o roteiro e devem ser compilados com o comando `flex p_*.l`, seguido do comando `gcc lex.yy.c -lfl -o p_*` (`*` deve ser a, b, c, ...).

**Atividades durante a aula**

As atividades (a) e (b) devem ser feitas e entregues até o final da aula.

**(a) p_a.l** (exemplo de execução: `./p_a < p_a.l`)

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

(a.1) Explique por que é necessário usar o "ponto e barra ./" antes do nome do executável produzido e porque o mesmo não é necessário quando se executa o gcc, o flex ou o bison.
(a.2) Mostre uma maneira de se configurar seu shell que permita a execução dos executáveis compilados sem a digitação de "./".
(a.3) Teste o programa de análise léxica executável gerado a partir de p_a.l com diferentes arquivos de entrada e explique como ele faz para produzir a saída visualizada.

**(b) p_b.l** (exemplo de execução: `date | ./p_b`)

```
%{
        int h;
%}
AM      [ ](00|01|02|03|04|05|06|07|08|09|10|11)[:]
PM      [ ](12|13|14|15|16|17|18|19|20|21|22|23)[:]
%%
{PM}    {sscanf(yytext," %d", &h);
         if (h == 12) printf(" PM 12:");
         else printf(" PM %02d:", h-12);}

{AM}    {sscanf(yytext," %d", &h);
         if (h==0) printf(" AM 12:");
         else printf(" AM %02d:", h);}

.       ECHO;
```

(b.1) Explique como o programa gerado por p_b.l consegue produzir os resultados visualizados. Documente alguns casos.
(b.2) Explique o funcionamento do comando pipe ( | ) usado entre date e p_b.
(b.3) Proponha, teste e documente uma outra forma de repassar o resultado do comando date para p_b.

**Atividades para o relatório**

As atividades (c) a (g) deverão fazer parte do relatório a ser entregue até a próxima aula. Procure entender os analisadores léxicos abaixo, execute-os e **explique seu funcionamento no relatório, exemplificando com as várias saídas reais obtidas do computador**. **Observe que não é para dizer apenas o que os programas fazem, mas <u>como eles fazem</u>.**

**(c) p_c.l** (exemplo de execução: `./p_c < input_p_c`)
Explique em detalhes o funcionamento deste analisador léxico e descreva seu propósito.

```
%%
[.]             ;
var             ;
[{]             printf("/*");
[}]             printf("*/");
mod             printf("%%");
or              printf("||");
and             printf("&&");
begin           ;
end             printf("}");
if              printf("if (");
then            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(";");
}
```

**(d) p_d.l** (execute o analisador com ./p_d e digite diversas palavras e quebras de linha, terminando com *ctrl-d*.)
Documente seus testes e explique em detalhes o funcionamento deste analisador léxico, mostrando que compreendeu como ele consegue atingir seus objetivos.

```
%{
int lines=0,characs=0,words=0,pages=1;
%}

%START                  Palavra
NovaLinha               [\n]
Espaco                  [\t ]
NovaPagina              [\f]

%%
{Espaco}            { BEGIN 0; characs++; }
{NovaLinha}         { BEGIN 0; characs++; lines++; }
{NovaPagina}        { BEGIN 0; characs++; pages++; }
<Palavra>.          { characs++; }
.                   { BEGIN Palavra; characs++; words++; }
%%

main()
{
  while (yylex());
  printf("\nResultados: \n%d pagina(s)\n%d linha(s)\n%d palavra(s)\n%d
          caracter(es)\n", pages,lines,words,characs);
}
```

**(e) p_e.l** (exemplo de execução: `./p_e < input_p_e`)
Documente seus testes e explique em detalhes o funcionamento deste analisador léxico, mostrando que compreendeu como ele consegue atingir seus objetivos.

```
%{
  int valor;
%}

OCTAL           [0-7]+O

%%

{OCTAL}         { sscanf(yytext,"%o",&valor); printf("%d",valor); }
.               ECHO;

%%
```

**(f) p_f.l**

Lembrando que comentários em shell script são caracterizados por '#', escreva um programa p_f usando flex para ler um shell script e retirar seus comentários. Explique detalhadamente como seu programa funciona e teste-o utilizando o arquivo input_p_f. Sua saída deve ser idêntica ao arquivo output_p_f (você pode comparar sua saída com a saída output_p_f com o comando diff).

```
./p_f < input_p_f > result_p_f
diff result_p_f output_p_f
```

**(g) p_g.l**

Escreva um programa em flex para ler uma data no formato dd/mm/aa, *usando as declarações já feitas* no arquivo p_g.l, e escrever a mesma por extenso. Por exemplo, a data 19/08/15 deve ser escrita por extenso

como: dezenove de agosto de dois mil e quinze. Sugere-se criar rotinas em C para as conversões de inteiro para string. Documente tudo como indicado no início.

Exemplo de execução: ./p_g < input_p_g

```
%{
char *unidades[]={"primeiro","um","dois","tres","quatro","cinco",
               "seis", "sete","oito","nove"};
char *interm[]={"dez","onze","doze","treze",
                 "quatorze","quinze","dezesseis",
                 "dezessete","dezoito","dezenove"};
char *dezenas[]={"vinte","trinta","quarenta","cinquenta",
                 "sessenta","setenta","oitenta","noventa"};
char *meses[]={"janeiro","fevereiro","marco","abril","maio",
               "junho","julho","agosto","setembro",
               "outubro","novembro","dezembro"};
%}

/* Dica de uma definição que pode ser usada na seção de regras.*/

PrimeiroCampo  [ ]..[/]

%%

%%
```