Compiladores
Roteiro de Laboratório 01 – Construindo *Scanners*

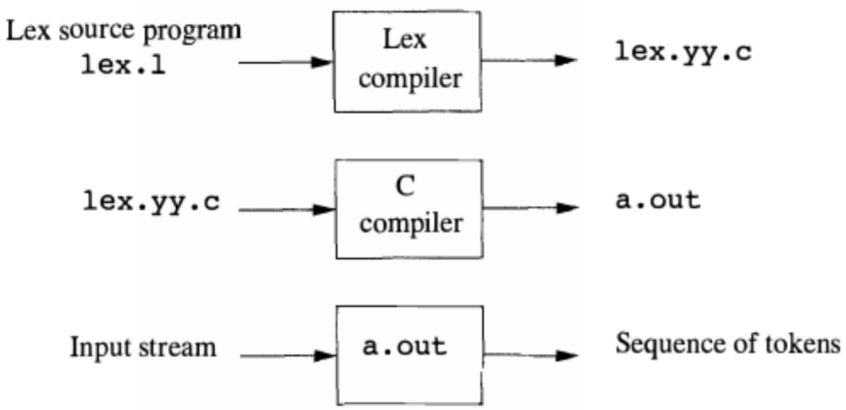
Parte I
Utilizando o `flex`

1 Introdução

- No conteúdo teórico do Módulo 01 vimos que:
  - *Scanners* (analisadores léxicos) são o primeiro componente do *front-end* de um compilador.
  - *Scanners* podem ser *hand-coded*, i.e., programados de forma dedicada para uma dada linguagem.
  - Mas também podem ser gerados automaticamente através de uma ferramenta para construção de *scanners*.
- Neste laboratório vamos ter um contato prático com uma dessas ferramentas: o `flex`.
- O `flex` é a versão GNU do venerável `lex`, escrito na década de 1970 por Mike Lesk e Eric Schmidt (sim, o cara do Google!).
- O `flex` serve para gerar automaticamente *scanners* em C.
- O programa gerado varre a entrada e produz uma sequência de *tokens*.
- Os *tokens* gerados pelo *scanner* geralmente são processados posteriormente por um *parser* (próximo módulo).

2 Utilizando o `flex`

- **Entrada**: arquivo de descrição do *scanner*: `*.l`. Contém um conjunto de expressões regulares (ERs) que são usadas para reconhecer os *tokens*.
- **Saída**: programa na linguagem C que implementa o *scanner* especificado. (Arquivo *default*: `lex.yy.c`).



- Compilando um arquivo `flex`:
  - Passo 1: `flex -o <arq_saida.c> <arq_definic.l>`
  - Passo 2: `gcc <arq_saida.c> -o <arq_exec> -lfl`

- Observações importantes:
  - A opção `-lfl` é necessária para o processo de ligação do executável com a biblioteca `fl`.
  - Programa gerado pode ser executado, entrada vem do `stdin` (teclado).
  - Para evitar repetição de vários comandos o tempo todo é bom usar um `Makefile`.

3 Arquivo de entrada do `flex`

- Um arquivo de especificação do `flex` possui três partes:

```
seção de definições
%%
regras de tradução (produções)
%%
funções auxiliares
```

  onde a seção de definições e as funções auxiliares podem ser omitidas.
- O segundo `%%` é opcional mas o primeiro indica o começo das regras e portanto é obrigatório.
- Ao lado de cada regra você pode colocar uma ação (trecho de código em C), que é executada sempre que a ER indicada é casada com a entrada (ou uma parte dela).
- As regras de tradução têm a seguinte forma:

```
ER    { Ação }
```

- **Exemplo 01** – Considere o seguinte arquivo, `basic.l`.

```
%%
a            { ECHO; printf(" -> Rule 1\n"); }
ab           { ECHO; printf(" -> Rule 2\n"); }
a*b+         { ECHO; printf(" -> Rule 3\n"); }
(a|b)*abb    { ECHO; printf(" -> Rule 4\n"); }
.*           { ECHO; printf(" -> Not recognised!\n"); }
\n           { printf("Next input please => "); }
%%
```

- Note o comando `printf` padrão do C. Além disso, foi utilizada a macro `ECHO` do `flex`, que exibe o *token* reconhecido na tela.
- Para executar esse programa, abra um terminal e execute os seguintes comandos no diretório `ex01` do arquivo de exemplos.

```
$ flex basic.l
$ gcc lex.yy.c -lfl
$ ./a.out
```

- Não esqueça a opção `-lfl` ao compilar, pois senão isso leva a um erro como abaixo.

```
$ gcc lex.yy.c
[...]
lex.yy.c: undefined reference to 'yywrap'
collect2: error: ld returned 1 exit status
```

- Após executar o programa (`./a.out`), digite qualquer sequência de caracteres que você queira analisar, por exemplo:

```
$ ./a.out
ab
ab -> Rule 2
Next input please => abbb
abbb -> Rule 3
Next input please => abbbb
abbbb -> Rule 3
Next input please => aaa
aaa -> Not recognised!
Next input please =>
```

  e termine com `<ctrl>D` para sair do programa.

4 Expressões Regulares no `flex`

- Os meta-símbolos utilizados para especificação das expressões regulares são os abaixo.

  `" \ [ ] ? - . * + | ( ) / { } % < >`

  Se você quiser que esses símbolos representem os seus respectivos caracteres, você deve escapá-los com `\` ou colocá-los entre aspas duplas. Na dúvida, coloque toda a *string* que você quer reconhecer entre aspas, como por exemplo, `"xyz++"`.
- Os padrões para entrada do `flex` utilizam um conjunto de ERs estendidas. Os comandos para construção dos padrões são:
  - `xyz`: reconhece a sequência de caracteres `xyz`.
  - `.`: reconhece qualquer caractere **exceto enter**.
  - `[xyz]`: uma *classe de caracteres*, nesse caso, a ER casa com o caractere `x`, **ou** o caractere `y` **ou** `z`.
  - `[abj-oZ]`: uma classe de caracteres contendo uma *faixa*. Casa com `a`, ou `b`, ou qualquer letra de `j` a `o`, ou por fim `Z`.
  - `[a-zA-Z]`: reconhece uma letra do alfabeto (maiúscula ou minúscula).
  - `[ \t\n]` reconhece um espaço em branco ou um *tab* ou uma quebra de linha.
  - `[^A-Z]`: uma classe de caracteres *negada*, isto é, qualquer caractere que não esteja na classe. Nesse exemplo, casa com qualquer caractere exceto letras maiúsculas.
  - `[^A-Z\n]`: qualquer caractere que não seja uma letra maiúscula ou enter.
  - `r*`: zero ou mais ocorrências da ER `r`.
  - `r+`: uma ou mais ocorrências de `r`.
  - `r?`: zero ou uma ocorrência de `r` (isto é, um `r` opcional).
  - `r{n}`: reconhece exatamente n ocorrências de `r`.
  - `r{n,}`: reconhece no mínimo n ocorrências de `r`.
  - `r{n,m}`: reconhece no mínimo n e no máximo m ocorrências de `r`.
  - `{nome}`: a expansão do *fragmento* `nome`. (Veja o Exemplo 4, abaixo.)
  - `"[xyz]\"foo"`: reconhece exatamente a string `[xyz]"foo`. Muito usado para meta-caracteres, como explicado acima. Exemplo: `"+"` para representar o símbolo de soma.
  - `(r)`: parênteses são usados para agrupar sub-expressões.
  - `r|s`: reconhece `r` ou `s`.

  - `^r`: reconhece `r` se estiver no início da linha.
  - `r$`: reconhece `r` se estiver no final da linha.
  - `<<EOF>>`: marcador de fim de arquivo na entrada.
- *Obs.:* Note que, dentro de uma classe de caracteres, todos os meta-símbolos do `flex` são tratados como caracteres normais, com exceção dos caracteres `\`, `-`, e `^`.

5 Mais exemplos simples

- **Exemplo 02 – Contagem de linhas e caracteres**. O exemplo abaixo faz uso das outras seções do arquivo `.l`.

```
%{
  #include<stdio.h>
  int lines = 0, chars = 0;  /* Global vars */
%}
%%
\n      { lines++; chars++; }
.       { chars++; }
%%
int main() {
  yylex();
  printf("Line count: %d, chars count: %d\n", lines, chars);
  return 0;
}
```

Inspecione o arquivo `lex.yy.c` para ver o código gerado automaticamente pelo `flex`. O código desse arquivo implementa um DFA (autômato finito determinístico) que reconhece as ERs especificadas. Tente encontrar os trechos de código C do arquivo `.l` no arquivo `.c`. Para executar:

```
$ flex count.l
$ gcc lex.yy.c -lfl
$ ./a.out < count.l
Line count: 13, chars count: 229
```

- **Exemplo 03 – Reconhecendo palavras de interesse**.

```
%{ /* Reconhece os artigos em ingles a, an, the */
  #include <stdio.h>
%}
%%
[ \t]+      { } /* pula espaco em branco - acao: fazer nada. */
a|an|the    { printf("%s: eh um artigo.\n", yytext); }
[a-zA-Z]+   { printf("%s: ???\n", yytext); }
%%
```

O programa acima usa uma variável especial do `flex` chamada `yytext`, que é um ponteiro para o primeiro caractere do *token* reconhecido. A macro `ECHO` utiliza essa variável, mas podemos usar `yytext` diretamente também. Uma segunda variável `yyleng` indica o número de caracteres do *token*. Executando o programa:

```
$ ./a.out <<< "Hi, this is a test!"
Hi: ???
,this: ???
is: ???
a: eh um artigo.
test: ???
!
```

A execução ilustra um comportamento adicional do `flex`: se um caractere não casa com nenhuma ER, ele é exibido de volta no terminal. Isso explica porque `,` e `!` aparecem na saída do *scanner*.

- **Exemplo 04 – Reconhecendo números naturais**. O exemplo abaixo usa a seção de definições do `flex` para especificar *fragmentos* de ERs. Isso é útil para evitar repetições de ERs e deixar o código mais legível.

```
%{  /* Reconhece numeros naturais arbitrarios. */
  #include <stdio.h>
%}
/* As definicoes abaixo sao chamadas de fragmentos, que podem
   ser usadas para formar ERs. */
digito  [0-9]
naozero [1-9]
numero  ({naozero}{digito}*)|0

%%
{numero}    { printf("Encontrado numero: %s\n", yytext); }
[^ \t\n]+   { printf("Encontrado nao-numero: %s\n", yytext); }
[ \t\n]     { /* ignorados */ }
<<EOF>>     { printf("Fim de dados\n"); return 0; }
%%

int main() {
  yylex();
  return 0;
}

/* Definindo esta funcao nao precisa compilar com -lfl */
int yywrap() {
  return 1;
}
```

Compilando e executando:

```
$ flex naturais.l
$ gcc lex.yy.c
$ ./a.out <<< "0 123 0123"
Encontrado numero: 0
Encontrado numero: 123
Encontrado nao-numero: 0123
Fim de dados
```

6 Resolução de conflitos no `flex`

- Você pode ter reparado que em alguns dos exemplos anteriores há uma "sobreposição" de ERs, isto é, uma mesma sequência de caracteres pode ser reconhecida por mais de uma ER. O `flex` usa as seguintes regras nesses casos:
  - Tentar consumir a maior quantidade de caracteres de uma vez (isto é, com uma única ER). Isso se chama *construção do prefixo mais longo*.
  - Se o prefixo mais longo casa com duas ou mais ERs, escolher a ER que aparece primeiro na especificação.
- Esse é um detalhe muito importante, que costuma dar dor de cabeça a alguns alunos. Por exemplo, veja as regras abaixo:

```
[a-z]*   { /* Regra 1 */ }
"foo"    { /* Regra 2 */ }
```

  A Regra 2 nunca será executada, *mesmo quando a entrada for* `foo`, porque a entrada também casa com a Regra 1 e esta tem prioridade por ter sido declarada primeiro. **Lembre disso quando estiver criando seu *scanner* para o trabalho, na hora de definir palavras reservadas e nomes de variáveis (identificadores).**

7 Exercícios de aquecimento

0. Faça o download dos arquivos de exemplo. Compile-os e execute-os como explicado acima. Usando o `flex`, crie e teste os *scanners* pedidos nos items a seguir. (*Obs.:* Trabalhe somente com `stdin` e `stdout`. Não é preciso ficar abrindo e fechando arquivos. Use redirecionamentos do *shell* como exemplificado acima.)
1. Remova da entrada todas as ocorrências de `#` e o restante da linha. Útil para eliminar comentários em *scripts shell*. *Obs.*: é possível resolver esse item com apenas duas linhas de código no `flex`.
2. Encontre letras maiúsculas na entrada e substitua pelas suas equivalentes em minúsculas. Não modifique os demais caracteres.
3. Reconheça inteiros 32-bit em notação hexadecimal. Os números começam com `0x` ou `0X` e podem ter no máximo 8 dígitos hexadecimais. As letras podem ser em qualquer caixa (alta ou baixa).
4. Reconheça placas de carros antigas no formato `AAA-0000`.

Parte II
Construindo um *scanner* para EZLang

8 A Linguagem EZLang

Um programa em EZLang tem uma estrutura bastante simples: ele é composto apenas por uma sequência de declarações (*statements*) separadas por ponto-e-vírgula, com uma sintaxe similar a de Ada e Pascal. Não existem funções ou procedimentos, somente o corpo do programa principal. Existem apenas quatro tipos primitivos (inteiro, real, Booleano e *string*) e não é possível criar novos tipos. Existem apenas dois comandos de controle: `if` e `repeat`. Ambos podem conter um bloco de comandos. Um comando `if` pode ter uma parte `else` opcional

e deve terminar com a palavra-chave `end`. Existem também comandos `read` e `write` para realização de operações de entrada e saída básicas. Comentários são escritos entre chaves e não podem ser aninhados.

Expressões em EZLang são limitadas a expressões sobre os tipos primitivos. Uma expressão Booleana consiste de uma comparação entre duas expressões aritméticas usando um dos dois operadores de comparação, < ou =. Uma expressão aritmética pode envolver constantes numéricas, variáveis, parênteses e quaisquer dos quatro operadores aritméticos +, −, * e /, com as propriedades matemáticas usuais.

Apesar da linguagem EZLang não possuir muitas das características necessárias a linguagens de programação reais (procedimentos, vetores e estruturas estão entre as omissões mais sérias), a linguagem ainda é suficientemente grande para exemplificar a maioria das características essenciais de um compilador.

Na listagem abaixo temos um exemplo de um programa em EZLang para o cálculo da função fatorial.

```
{ Sample program in EZ language -
  computes factorial
}

program fact;
var
    int x;
    int fact;
begin
    read x; { input an integer }
    if 0 < x then { don't compute if x <= 0 }
        fact := 1;
        repeat
            fact := fact * x;
            x := x - 1;
        until x = 0
        write fact; { output factorial of x }
    end
end
```

9 Convenções Léxicas da linguagem EZLang

A seção anterior fez apenas uma breve introdução informal da linguagem EZLang. A tarefa deste laboratório é construir um *scanner* para a linguagem, e portanto, a sua estrutura léxica deve ser devidamente descrita. Em outras palavras, é necessário definir quais são os tipos de *tokens* e seus respectivos lexemas. Essas informações estão apresentadas abaixo:

```
{ Palavras reservadas: }
begin   bool   else    end     false   if      int     program
read    real   repeat  string  then    true    until   var     write

{ Símbolos especiais: }
:=  =   <   +   -   *   /   (   )   ;
```

```
{ Constantes numéricas e strings (exemplos): }
42      4.2     "abc"
```

Os *tokens* de EZLang podem ser classificados em três categorias típicas: palavras reservadas, símbolos especiais e outras. Há 17 palavras reservadas, com os significados usuais (embora a sua semântica só será propriamente definida em laboratórios futuros). Existem 10 símbolos especiais, para as quatro operações aritméticas básicas, duas operações de comparação (igual e menor que), parênteses, ponto-e-vírgula e atribuição. Todos os símbolos especiais têm comprimento de um caractere, exceto a atribuição, que tem dois caracteres. Os outros *tokens* são números, que são sequências com um ou mais dígitos, *strings*, e identificadores, que (para simplificar) são sequências com uma ou mais letras.

Além dos *tokens*, EZLang segue as convenções léxicas a seguir. Comentários são cercados por chaves e não podem ser aninhados. O formato do código é livre, isto é, não existem colunas ou posições específicas para uma dada operação. Espaços são formados por branco, tabulações e quebras de linhas.

10 Implementado um *Scanner* para a linguagem EZLang

Você deve criar um *scanner* que reconhece todos os elementos léxicos da linguagem EZLang. Para tal, utilize o `flex`. Para cada *token* reconhecido no programa de entrada, exiba no terminal:

1. Número da linha do *token* seguido de `:`.
2. Lexema reconhecido seguido de `->`.
3. Tipo do *token* associado ao lexema. Caso o lexema identificado não esteja associado a nenhum tipo de *token*, o *scanner* termina com uma mensagem de erro.

A saída do seu *scanner* para o programa do exemplo anterior deve ficar da seguinte forma:

```
5: program -> PROGRAM
5: fact -> ID
5: ; -> SEMI
6: var -> VAR
7: int -> INT
7: x -> ID
7: ; -> SEMI
8: int -> INT
8: fact -> ID
8: ; -> SEMI
9: begin -> BEGIN
10: read -> READ
10: x -> ID
10: ; -> SEMI
11: if -> IF
11: 0 -> INT_VAL
11: < -> LT
11: x -> ID
11: then -> THEN
12: fact -> ID
```

```
12: := -> ASSIGN
12: 1 -> INT_VAL
12: ; -> SEMI
13: repeat -> REPEAT
14: fact -> ID
14: := -> ASSIGN
14: fact -> ID
14: * -> TIMES
14: x -> ID
14: ; -> SEMI
15: x -> ID
15: := -> ASSIGN
15: x -> ID
15: - -> MINUS
15: 1 -> INT_VAL
15: ; -> SEMI
16: until -> UNTIL
16: x -> ID
16: = -> EQ
16: 0 -> INT_VAL
17: write -> WRITE
17: fact -> ID
17: ; -> SEMI
18: end -> END
19: end -> END
```

Algumas observações importantes:

- Lembre-se que o lexema reconhecido fica guardado na variável global `yytext`.
- Não esqueça de colocar a regra de reconhecimento de identificadores *depois* das regras de palavras reservadas para evitar sobreposição.
- O `flex` possui uma variável global para contagem de linha chamada `yylineno`. No entanto, essa variável não é incrementada automaticamente. Use a opção `%option yylineno` do `flex` para tal.
- Veja os exemplos no arquivo de entrada (`in.zip`) com as respectivas saídas (`out01.zip`) no AVA. A partir das saídas fornecidas deve ser trivial inferir todos os tipos de *tokens* utilizados, levando-se em conta as informações iniciais apresentadas na Seção 9.
- Para testar a saída do seu *scanner* contra a fornecida pelo professor, use um *script* como abaixo:

```
#!/bin/bash
EXE=./lab01
IN=in
OUT=out01
for infile in `ls $IN/*.ezl`; do
    base=$(basename $infile)
    outfile=$OUT/${base/.ezl/.out}
    echo $infile
    $EXE < $infile | diff -w $outfile -
done
```