



# Flex + SED

Analizador léxico



# Análise léxica

A análise léxica envolve receber o texto de um código e transformar ele em uma sequência de “tokens”.

Tokens são elementos indivisíveis de um código, por exemplo em C:

```
int var=x+2;
```

Apesar de `var=x+2;` não conter nenhum espaço separando os seus elementos nós sabemos que: “var” é o nome da variável sendo criada, “=” é um operador de atribuição, “x” é o nome de uma variável que já existia, “+” é um operador de soma, “2” é uma constante e “;” é o final de uma linha de comando.



# Análise léxica

```
int var=x+2;
```

Vira uma sequência de tokens: “int”, “identificador:var”, “=”, “identificador:x”, “+”, “constante:2”, “;”

int

var                    =x

+                    2

;

Vira exatamente a mesma sequência.



# Análise léxica

O trabalho de um analisador léxico é transformar uma sequência de caracteres legíveis em uma sequência de elementos (tokens) mais fácil para uma parte seguinte do compilador poder trabalhar.



# Flex

Flex é uma ferramenta que ajuda a implementar analisadores léxicos em C.

Ela funciona recebendo um arquivo X.l que vai conter a descrição dos tokens e um pouco de código em C para ajudar a lidar com cada uma das tokens.



## Flex - Exemplo

```
func function2(a, b, c) : (((12 + 5) * ((a * b) + c)) - (5 * function1(2)))
```

Uma linguagem que define funções algébricas simples. A função é declarada pela palavra reservada `func`, seguida do nome da função, parênteses contendo os argumentos da função, dois pontos “:” e a expressão dentro de parênteses. Um dos elementos da função pode ser a chamada de outra função.



# Flex - Exemplo

Deve existir um .h contendo a definição dos tokens: token.h

```
enum token { FUNC, PLUS, MINUS, DIV, MULT, NUMBER, ID, COLON, OPEN_P, CLOSE_P, ERROR, EOF};
```

O arquivo .l vai conter descrição de tokens, e um código em C para ser executado quando o token for encontrado:



## Flex - Exemplo (scanner.l)

```
enum token { FUNC, PLUS, MINUS, DIV, MULT, NUMBER, ID, COLON, OPEN_P, CLOSE_P, ERROR, EOF};
```

```
func  return(FUNC);
```

```
"+"  return(PLUS);
```

```
"-"  return(MINUS);
```

```
[0-9]+return(NUMBER);
```

```
[a-z][a-z0-9]*  return(ID);
```

```
.      return(ERROR);
```

```
...
```





## Flex - Exemplo (scanner.l)

A parte do código responsável pela análise léxica deve ficar entre

```
%%
```

Análise

```
%%
```



## Flex - Exemplo

O comando abaixo usa o scanner.l para criar o scanner.c que contém o código que será responsável por fazer a análise léxica.

```
flex -i -o scanner.c scanner.l
```

Normalmente nós não nos preocupamos com o arquivo scanner.c, pois ele deveria ser usado pelo programa de “parser” (existe uma ferramenta chamada “bison” que é usada para fazer o “parsing” e usar as funções geradas pelo flex), mas nós vamos usar as funções declaradas por ele para fazer a nossa análise sem passar pelo parte de parseamento.



## Flex - Exemplo

O arquivo scanner.c vai implementar a função global `yylex()`, que tenta extrair tokens do `stdin` e coloca a string responsável pela geração da token em um ponteiro de char global nome `*yytext`.

Dessa forma, basta chamar em loop a função `yylex` para obter todos os tokens de um código (informado pelo `stdin`), e usar a variável `yytext` quando apropriado.



# sed

Sed vem de Stream EDitor. É uma ferramenta muito usada em sistemas UNIX que recebe comandos e um arquivo por parâmetro e edita o arquivo de acordo com os comandos, ou cria um arquivo novo para guardar o resultado.

O sed não é uma ferramenta usada para fazer a parte de análise léxica de um compilador, mas o seu princípio é o mesmo, ela é uma ferramenta que tem sua funcionalidade completamente baseada em gramáticas regulares.



# Sed - exemplos

Sed -e "s/EXPRESSION/NEW\_EXPRESSION/" arquivo

Muda todas as ocorrências de EXPRESSION por NEW\_EXPRESSION



# Sed - exemplos

Sed -e “/EXPRESSION/d” arquivo

Deleta todas as linhas onde EXPRESSION é encontrado



# Sed - exemplos

Sed -n “/EXPRESSION/p” arquivo

Deixa somente as linhas do arquivo onde EXPRESSION é encontrado



## Sed - exemplos

Usando a opção -E (ou -r) é possível usar algumas capacidades interessantes:

```
Sed -E -e "s/menin([oa])/gat\1" arquivo
```

Substitui ocorrências de menino por gato e menina por gata.

A o que der “match” com as coisas dentro dos parênteses fica disponível pelo \1.

Se tivesse um segundo par de parênteses ele estaria no \2, etc





## Sed - exemplos

Usando a opção -E (ou -r) é possível usar algumas capacidades interessantes:

Sed -E -e “/cachorro|gato/d” arquivo

Nesse caso, o | funciona como um “ou”. Esse comando deleta todas as linhas que contenham “cachorro” ou “gato”



## lab2

Nesse lab, será fornecido um .zip com a pasta de exemplo mat\_lang

Nessa pasta contem um exemplo de flex com um script exec.sh que gera o executável scanner\_test

Usando o comando

```
scanner_test < code.ml
```

Ele imprime todas as tokens de code.ml



## lab2

Também será fornecido a pasta `c_lang`.

Essa pasta contém:

`codeX.c` -> exemplo de código que vocês devem fazer análise léxica.

`Tokens.h` -> um arquivo `.h` com todas as tokens que vocês devem gerar.



## lab2

Nesse lab vocês devem:

- Criar o arquivo `scanner.l` que contém a descrição léxica da linguagem C reduzida. (os elementos presentes podem ser inferidos pela lista de tokens)
- Criar o arquivo que vai chamar `yylex` e imprimir todas as tokens. Diferente do exemplo em `mat_lang` todas as tokens devem ser impressas com o `yytext` (a “matching string”) que leva ela e deve terminar com a linha: `“lines:%u\n”` onde `%u` é o número de linhas presentes no arquivo.
- Um script (descrito em um slide a frente)



## lab2

Nesse lab, os arquivos .c estão “corrompidos”. As keywords da linguagem estão todas em “upper case”, mas o analizador léxico deve receber apenas keywords lowercase. Fora isso, em alguns arquivos vão existir trechos de código com variáveis ou funções contendo a palavra TEST que não deveriam fazer parte do código.

Parte do seu trabalho será usar o sed para trocar todas as keywords que estão em maiúsculo para minúsculo, e para remover as linhas de contendo a palavra TEST.



# lab2

Alguns detalhes:

Toda as ocorrências de operação com TEST ocupam apenas uma linha

Comentários são apenas aceitos do tipo que a partir do `//` até o `\n` tudo é considerado comentário.



## lab2

O script a ser entregue deve fazer:

- Usar o flex para transformar o .l em .c
- Compilar o scanner
- Receber um arquivo parâmetro e a partir dele usar o sed para remover as linhas com TEST e transformar as keywords em lower case. O resultado deve ficar em um arquivo chamado "corrected.c"
- Usar o scanner em corrected.c e colocar o resultado em tokens.txt
- Criar um arquivo que contém apenas as linhas de token.txt contendo os tokens T\_ID, T\_STR e T\_NUM (usar sed) chamado selected.txt



## lab2

O script pode gerar outros arquivos temporários para guardar resultados intermediários, mas eles devem estar SEMPRE dentro da pasta onde o script é executado.

Se o script escrever em algum diretório além de onde ele for executado a nota do lab é 0 (e possivelmente do curso se identificada má índole).

Apenas um scrip pode ser enviado.

O script deve ser executado por pelo programa bash.





## Lab2 - Entrega

Os alunos devem entregar um .zip com o nome [RAaluno1]\_[RAaluno2].zip contendo uma pasta chamada lab2 com arquivos:

- O arquivo .l
- O código que será o scanner .c
- O script bash

O arquivo .h será o que foi fornecido e não precisa estar no zip (o script pode assumir que ele vai estar dentro da pasta lab2 onde ele será executado).



# Lab2

Tutorial de sed: <https://www.tutorialspoint.com/sed/>

Documentação do flex: [https://ftp.gnu.org/old-gnu/Manuals/flex-2.5.4/html\\_mono/flex.html](https://ftp.gnu.org/old-gnu/Manuals/flex-2.5.4/html_mono/flex.html)

Manual de bash: <https://www.gnu.org/software/bash/manual/bash.pdf>



# Lab2

Dicas de bash:

Os argumentos passados podem ser usado pelas variáveis \$1, \$2, etc.

Ex: `bash script.sc valor1 valorB`

\$1 vai ser a string valor1

\$2 vai ser a string valorB



# Lab2

Dicas de bash:

É possível fazer funções em bash

```
Function_name () { BASH COMMANDS }
```

Ela é chamada por:

```
Function_name argumento1 argumento2
```

Argumento1 pode ser acessado como \$1 e argumento2 \$2 como os argumentos do script. Isso quer dizer que dentro de uma função, \$1 NÃO é o argumento do script, e sim da função.



## Lab2

Expressões regulares são usadas para descrever “tokens” tanto no flex quanto no sed (quando usando -E/-r).

Dicas de expressões regulares:

. simboliza um caracter qualquer.

^a simboliza qualquer caracter menos o a.

[a-z] simboliza qualquer caracter que esteja entre a-z



## Lab2

Expressões regulares são usadas para descrever “tokens” tanto no flex quanto no sed (quando usando -E/-r).

Dicas de expressões regulares:

`[aOf]` simboliza um caracter que pode ser a O ou f

`[aOf]*` simboliza n caracteres que podem ser a O ou f,  $n \geq 0$

`[aOf]+` semelhante ao anterior mas  $n \geq 1$



# Lab2

Dicas de sed:

Em sed não existe `\n`. Logo, se quiser fazer match com algo que esteja no início da linha, usa-se `^` como primeiro elemento da expressão. Se quiser fazer match com algo que esteja no fim da linha, usa-se `$` no final da expressão.



## Lab2

Dentro da pasta `c_lang/results` já estão os resultados que o script de vocês deveria gerar. Para passar no lab, o diff dos arquivos que vocês geram e eles devem retornar que os arquivos são iguais.





## Lab2

Você pode perceber no programa exemplo que algumas funções a mais são declaradas, como yywrap e yyerror. Isso ocorre pois essas funções normalmente são declaradas por outras ferramentas. Não precisa se preocupar com elas, apenas declare elas como no exemplo fornecido.



## Lab2

ATENÇÃO: diferentes implementações de sed podem não aceitar os mesmos comandos.

Para garantir que o seu lab funcione, execute o seu script pelo menos uma vez em uma máquina fedora do IC com o sed já presente nela.

Se ele não funcionar nas máquinas do ic você corre o risco do seu lab não ser aceito.