



## Trabalho 01 – Análise Léxica

### Descrição geral

O objetivo deste trabalho é construir um analisador léxico para uma linguagem de programação de alto nível. Como objeto de trabalho, utilizaremos um subconjunto da linguagem C, o C--.

### Análise Léxica

O programa deve **ler um arquivo** contendo o código fonte em C-- e gerar uma relação dos tokens encontrados neste programa.

Deverá ser criada uma função `analex()` que fará a análise léxica do programa fonte. Cada vez que esta função for chamada a partir do programa principal, ela retornará um par (token, lexema) para ele.

### Alfabeto

O Compilador deve ler um arquivo de entrada que contém símbolos válidos da linguagem. No caso do C--, estes símbolos são:

- Letras: **ab ... zAB ... Z**
- Dígitos: **0123456789**
- Símbolos Especiais: **, ; ( ) = < > + - \* / % [ ] " ' \_ \$ { } ? : ! . etc**
- Separadores: espaço, enter, tab

### Tokens

Deverão ser gerados tokens para:

- todas as palavras reservadas (cada uma delas um token diferente) =
  - {void, int, float, char, bool, if, else, for, while, do, return, break, continue, goto, true, false}
- todos os operadores utilizados na linguagem (cada um deles um token diferente)
  - **+ - \* / % ? : ! & . -> < > == != <= -> >= = += -= \*= /= %= ++ && || etc**
- todos os sinais de pontuação utilizados na linguagem (cada um deles um token diferente)
  - **, ; ( ) [ ] { }**
- todos os literais básicos

- inteiros, numero = digito{+}
- reais, número . número
- caracteres, um símbolo entre ''
- strings, tudo entre ""
- Booleanos true | false
- identificadores =
  - [\$ \_]\* letra (letra | digito | \$ \_)\*

Não geram tokens:

- comentários tudo que estiver entre /\* e \*/ ou depois de //
- separadores

Dicas:

- os tokens podem ser criados como constantes (#define)
- Indicar os erros possíveis (caractere invalido, literais inconsistentes, ...)
- Parar a execução após encontrar o primeiro erro

## Saída do programa

Gerar um arquivo no qual serão gravados cada token reconhecido e o seu respectivo lexema, e em caso de erro, a mensagem correspondente (escrever estas ações de forma clara).

Exemplos:

Entrada:	Saída:	
a = 2;	Tokens	Lexemas
	ID OP_ATRIB NUM_INT PV	a = 2 ;
'a' "palavra * a9 987.5	Tokens	Lexemas
	CARACTERE ERRO string incompleta	'a' "palavra
+ ++ += == void	Tokens	Lexemas



	OPAD	+
	OPINC	++
	OPADATRIB	+=
	OPIG	==
	VOID	void

## Principais funções que devem ser implementadas

- **analex():** função que implementa o diagrama de transição do analisador léxico.
- **prox\_char():** esta função deve retornar apenas um caractere do arquivo de entrada por vez quando for chamada. Ela faz a interface entre o arquivo de entrada e o programa.
- **grava\_token():** faz a gravação do token e do seu lexema no arquivo de saída

## Estrutura básica para o programa principal

```
void main(){
    ch = prox_char();
    while(não fim) {
        (token, lexeme) = analex();
        ...
        grava_token(token, lexeme);
    }
}
```

A gramática a ser utilizada está descrita abaixo.

## Equipes

O trabalho pode ser feito em grupos com no máximo 2 alunos.

## O que deve ser entregue:

Além da entrega do código fonte, na plataforma do colabweb, o aluno deverá produzir:

1. **Manual do usuário** (uma página) Num arquivo chamado **mu.txt** ou **mu.doc**, contendo uma explicação de como se utilizar o analisador (explicar o formato da entrada e da saída do programa).



2. **Manual do programador** Num arquivo chamado **mp.txt** ou **mp.doc**, contendo (na ordem das letras):
  - a. A lista completa (descritiva e explicativa) dos tokens utilizados
  - b. Os diagramas de transição implementados
  - c. Demonstrar passo a passo as fatorações, eliminação de recursividade e análise do primeiro símbolo ou outras técnicas que forem utilizadas sobre a gramática dada acima.
  - d. Deverá ser criada uma lista de mensagens de erro, uma para cada caso, como feito num compilador real (mensagens consistentes).
3. **um vídeo (ou mais)** explicando como as funções foram pensadas. As principais funções devem também ser explicadas. No vídeo, há a necessidade de apresentar o código fonte, a compilação e a execução do programa. Não há necessidade do aluno aparecer no vídeo.

Os vídeos devem ser postados em uma plataforma de vídeo (youtube, por exemplo), de modo que o professor possa acessar, e fazer parte da avaliação do trabalho. Para gravar pode usar serviços gratuitos e online. Por exemplo:

<https://online-screen-recorder.com/pt>

<https://www.veed.io/pt-BR>

<https://streamyard.com/>

ou se quiser, há aplicativos que podem ser baixados:

<https://www.movavi.com/pt/learning-portal/gravadores-de-video.html>

Atenção: Os vídeos não precisam ter alta produção. Para subir os vídeos para o Youtube, há uma necessidade de ajuste no seu perfil do youtube, com pelo menos 24h de antecedência. Portanto, sugiro que esse ajuste seja feito brevemente.

### **A nota será composta assim:**

- De 0 a 0,5 pontos pelo estilo de programação (nomes bem definidos, lugares de declarações, comentários).



- De 0 a 0,5 ponto pela compilação.
- De 0 a 0,5 ponto pelo formato de apresentação dos resultados
- De 0 a 6 pontos pela solução apresentada.
- De 0 a 2,5 pontos pelas informações dos vídeos.

### Comentários Gerais:

1. Comece a fazer este trabalho logo, enquanto o problema está fresco na memória e o prazo para terminá-lo está tão longe quanto jamais poderá estar.
2. Clareza, indentação e comentários no programa também vão valer pontos.
3. Trabalhos copiados serão penalizados conforme anunciado

### Gramática C-:

#### Declarações

1. programa  $\rightarrow$  lista-decl lista-com
2. lista-decl  $\rightarrow$  lista-decl decl | decl
3. decl  $\rightarrow$  decl-var | decl-func
4. decl-var  $\rightarrow$  espec-tipo var ;
5. espec-tipo  $\rightarrow$  INT | VOID | FLOAT
6. decl-func  $\rightarrow$  espec-tipo ID ( params ) com-comp
7. params  $\rightarrow$  lista-param | void |  $\epsilon$
8. lista-param  $\rightarrow$  lista-param , param | param
9. param  $\rightarrow$  espec-tipo var
10. decl-locais  $\rightarrow$  decl-locais decl-var |  $\epsilon$

#### Comandos

11. lista-com  $\rightarrow$  comando lista-com |  $\epsilon$
12. comando  $\rightarrow$  com-expr | com-atrib | com-comp | com-selecao | com-repeticao | com-retorno
13. com-expr  $\rightarrow$  exp ; | ;
14. com-atrib  $\rightarrow$  var = exp ;



- 
15.  $\text{com-comp} \rightarrow \{ \text{decl-locais lista-com} \}$   
16.  $\text{com-selecao} \rightarrow \text{IF ( exp ) comando} \mid \text{IF ( exp ) com-comp ELSE comando}$   
17.  $\text{com-repeticao} \rightarrow \text{WHILE ( exp ) comando} \mid \text{DO comando WHILE ( exp ) ;}$   
18.  $\text{com-retorno} \rightarrow \text{RETURN ;} \mid \text{RETURN exp ;}$

### **Expressões**

19.  $\text{exp} \rightarrow \text{exp-soma op-relac exp-soma} \mid \text{exp-soma}$   
20.  $\text{op-relac} \rightarrow <= \mid < \mid > \mid >= \mid == \mid !=$   
21.  $\text{exp-soma} \rightarrow \text{exp-soma op-soma exp-mult} \mid \text{exp-mult}$   
22.  $\text{op-soma} \rightarrow + \mid -$   
23.  $\text{exp-mult} \rightarrow \text{exp-mult op-mult exp-simples} \mid \text{exp-simples}$   
24.  $\text{op-mult} \rightarrow * \mid / \mid \%$   
25.  $\text{exp-simples} \rightarrow ( \text{exp} ) \mid \text{var} \mid \text{cham-func} \mid \text{literais}$   
26.  $\text{literais} \rightarrow \text{NUM} \mid \text{NUM.NUM}$   
27.  $\text{cham-func} \rightarrow \text{ID ( args )}$   
28.  $\text{var} \rightarrow \text{ID} \mid \text{ID [ NUM ]}$   
29.  $\text{args} \rightarrow \text{lista-arg} \mid \epsilon$   
30.  $\text{lista-arg} \rightarrow \text{lista-arg , exp} \mid \text{exp}$