

Processamento de Linguagem (3º ano de Engenharia Informática)

**Trabalho Prático 1**

Relatório de Desenvolvimento

Adriana Meireles  
(a82582)

Mariana Pereira  
(a81146)

Pedro Pinto  
(a82535)

26 de Maio de 2020

## **Resumo**

Este relatório tem como objetivo documentar o primeiro trabalho prático desenvolvido na unidade curricular de Processamento de Linguagens, utilizando a ferramenta Flex como auxiliar para gerar filtros de texto. Deste modo, é apresentada uma introdução ao projeto bem como a descrição do enunciado. De seguida é explicada de que forma é implementada a solução e as decisões que foram tomadas durante a sua realização. No final é feita uma apreciação crítica ao trabalho desenvolvido.

# Conteúdo

<b>1</b>	<b>Introdução</b>	<b>2</b>
<b>2</b>	<b>Análise e Especificação</b>	<b>3</b>
2.1	Descrição do Problema . . . . .	3
2.1.1	Criar um filtro extraiGIC . . . . .	3
2.1.2	Criar um filtro lexgen . . . . .	4
2.2	Especificação dos Requisitos . . . . .	4
<b>3</b>	<b>Desenho e implementação da Solução</b>	<b>5</b>
3.1	Estruturas de Dados . . . . .	5
3.2	Filtro extraGIC . . . . .	5
3.3	Filtro lexgen . . . . .	6
<b>4</b>	<b>Codificação e Testes</b>	<b>9</b>
4.1	Alternativas, Decisões e Problemas de Implementação . . . . .	9
4.2	Testes realizados e Resultados . . . . .	9
4.2.1	alínea a) . . . . .	9
4.2.2	alínea b) . . . . .	11
4.3	Guia de Utilização . . . . .	12
4.3.1	Compilação . . . . .	12
4.3.2	Execução . . . . .	12
<b>5</b>	<b>Conclusão</b>	<b>13</b>

# Capítulo 1

## Introdução

A realização deste trabalho prático da unidade curricular de Processamento de Linguagens consiste no desenvolvimento de um filtro de texto para fazer o reconhecimento dos padrões identificados e proceder à transformação pretendida, com recurso ao Gerador Flex.

Neste relatório é apresentado o enunciado do problema, bem como todas as decisões tomadas pelo grupo na resolução do problema proposto. De seguida, são expostos alguns exemplos de utilização dos filtros implementados.

O enunciado escolhido foi o número 2 - Filtro para gramáticas.

## Capítulo 2

# Análise e Especificação

### 2.1 Descrição do Problema

Como referido anteriormente, dentro dos vários enunciados disponíveis, decidimos escolher o número 2, referente a filtros de gramáticas. Assim, tendo em conta um ficheiro disponibilizado contendo um extrato de yacc pretende-se:

- Escrever um filtro flex que extraia do ficheiro apenas a gramática pura.
- Construir um analisador léxico associado à gramática que permita identificar os respetivos tokens.

#### 2.1.1 Criar um filtro extraiGIC

O objetivo desta fase será criar um filtro que seja capaz de extrair a gramática pura de um excerto de código como este, deixando de fora caracteres desnecessários.

---

```
1 ....
2 %union ....
3 %token ID MKLISTA NULA
4 %type ....
5 %%
6     lista : MKLISTA '(' ids ')' { ... a o sem ntica em C }
7           | NULA
8           ;
9     ids : ID ',' ids { ignorar as constantes char (ex: '!') das a es sem nticas }
10        | ID { if ( ... == 'a') { ... } else { ...} }
11        ;
12 %%
13 ... codigo C...
```

---

Dado o input apresentado acima o resultado esperado será o seguinte:

---

```
1 lista : MKLISTA '(' ids ')'
2       | NULA
3       ;
4 ids : ID ',' ids
5      | ID
6      ;
```

---

### 2.1.2 Criar um filtro lexgen

Nesta parte, deve ser desenvolvido um analisador léxico associado à gramática que permita identificar os tokens e os símbolos terminais. Para isso, utilizamos um filtro flex que gera o esqueleto de texto de um analisador léxico para essa gramática, a partir de um texto contendo gramáticas na notação yacc.

Depois de executado, o output do analisador léxico deverá ser semelhante ao excerto apresentado em baixo, sendo que as palavras 'FIXME' são substituídas pelas expressões regulares que definem os tokens.

---

```
1 %%  
2 FIXME {return ID;}  
3 FIXME {return MKLISTA;}  
4 FIXME {return NULA;}  
5 [( ) ,] {return yytext[0];}  
6 %%
```

---

## 2.2 Especificação dos Requisitos

O enunciado que escolhemos, para além das indicações específicas do próprio tema deve também seguir algumas indicações gerais:

- Através de Expressões Regulares, devemos especificar os padrões de frases que queremos encontrar no texto-fonte;
- Identificar as Estruturas de Dados globais utilizadas para armazenar a informação extraída do texto-fonte;
- Identificar as reações semânticas a realizar relacionadas com os padrões previamente identificados;
- Utilizando o Gerador FLex, desenvolver um filtro de texto que faça o reconhecimento dos padrões identificados e proceda à transformação pretendida.

## Capítulo 3

# Desenho e implementação da Solução

### 3.1 Estruturas de Dados

No que diz respeito às estruturas de dados, de modo a armazenar a informação relativa aos tokens e aos símbolos terminais, foram utilizados dois arrays dinâmicos, sendo que um guarda os nomes dos tokens e o outro os símbolos terminais existentes nas variáveis **tokens** e **terminais** respetivamente.

### 3.2 Filtro extraGIC

O começo de uma gramática pura é identificado por **%%** e, portanto, iniciamos o estado **gram** sempre que encontramos estes caracteres. Dentro disto, até se encontrar uma chaveta, plica ou percentagem o texto é processado. Posteriormente, entra numa das 3 condições conforme o que aparecer. Se for uma chaveta o conteúdo é ignorado. Se for uma plica tudo o que se encontra dentro das mesmas à exceção do espaço é processado. Se for uma percentagem, é executado o **BEGIN 0**, permitindo retornar ao estado inicial. Tudo o que se encontre fora deste estado deve ser ignorado.

Este processo encontra-se refletido no excerto de **FLEX** que se segue:

---

```
1  \%\\%                BEGIN  gram;
2
3  <gram>\\%\\%        BEGIN  0;
4
5  <gram> '[^ ]+'      printf("%s", yytext);
6
7  <gram> \{.*\}        ;
8
9  <gram> [^{'%]+      printf("%s", yytext);
10
11 <*> .|\n|\r|\t      ;
```

---

Para processar o conteúdo do ficheiro, o input é redirecionado para a gramática passada como parâmetro e o output irá também ser redirecionado para um ficheiro(gram.txt):

---

```
1 int main(int argc, char *argv[]) {
2
3     if ( argc != 2 ) {
```

---

```

4     printf ("Usage : ./ exe <ficheiro yacc> \n");
5         return 0;
6     }
7
8     ifd = open(argv[1],ORDONLY);
9     ofd = open("gram.txt", O_CREAT | O_WRONLY | O_TRUNC, 0666);
10
11     int res = dup2(ifd, STDIN_FILENO);
12     int res2 = dup2(ofd, STDOUT_FILENO);
13
14     yylex();
15
16     return 0;
17 }

```

---

### 3.3 Filtro lexgen

Para gerar o esqueleto de um analisador léxico dividimos o algoritmo em três partes: guardar os tokens, guardar os símbolos terminais (o que está dentro de plicas) e ler tudo para um ficheiro output. Relativamente ao primeiro ponto, para filtrar os tokens decidimos guardar cada um num array de strings(*char \*\*tokens*). Sempre que se encontra **%token** é iniciado o estado **tok**. Dentro disto, cada token encontra-se separado por espaços pelo que sempre que se encontra um espaço este deve ser ignorado. Caso contrário são aplicadas expressões que permitem filtrar os tokens e guardá-los na variável referida anteriormente. Por fim é executado o **BEGIN 0** quando o **\n** é capturado, permitindo sair da *start condition* dos tokens e retornar ao estado inicial.

Este processo encontra-se refletido no **FLEX** que se segue:

---

```

1  \%token BEGIN tok;
2
3  <tok>\n BEGIN 0;
4
5  <tok>\ ;
6
7  <tok>[^ \n]+ {
8
9                      if(ntokens == 0){
10                         tokens = malloc(ntokensmax*sizeof(char*));
11                     }
12
13                     if(ntokens == ntokensmax){
14                         alocaMem(tokens, ntokens, ntokensmax);
15                     }
16
17                     tokens[ntokens] = malloc(yyleng*sizeof(char));
18                     tokens[ntokens] = strdup(yytext);
19                     ntokens++;
20                 }

```

---

Relativamente ao segundo ponto, de modo a filtrar os símbolos terminais seguimos uma abordagem semelhante à dos tokens. Quando se encontra **%%** é iniciado o estado **gen**, sendo aqui feita a filtragem dos símbolos terminais. Deste modo, criou-se um filtro que quando encontra duas plicas com algo dentro



excluindo o espaço, guarda na variável **terminais** o que encontra dentro destas. Encontrando de novo %% é executado o BEGIN 0 voltando ao estado inicial, como anteriormente referido.

Este processo encontra-se refletido no **FLEX** que se segue:

---

```

1
2 \% \% BEGIN gen ;
3
4 <gen> \% \% BEGIN 0;
5
6 <gen> \{.*\} ;
7
8 <gen> '[^ ]+' {
9
10             yytext[yyval-1]='\0';
11             if(terminais == 0){
12                 terminais = malloc(terminaismax*sizeof(char
13                                     *));
14
15             if(terminais == terminaismax){
16
17                 alocaMem(terminais, terminais, terminaismax);
18
19             terminais[terminais] = malloc(yyval*sizeof(char));
20
21             terminais[terminais] = strdup(yytext+1);
22             terminais++;
23         }

```

---

Relativamente ao terceiro ponto, para lermos os dados que se encontram nas variáveis acima referidas recorreremos à função *criarFicheiro()*. Após a criação desta, podemos escrever as informações na estrutura pedida no enunciado e que também foram obtidas através dos filtros anteriormente apresentados.

Este processo encontra-se refletido da maneira seguinte:

---

```

1 void criarFicheiro(){
2     int tam = eliminarrep(terminais);
3     int i;
4
5     printf("\n%s\n", "%%") ;
6
7     for(i=0; i < ntokens; i++){
8
9         printf("FIXME { return %s;}\n", tokens[i]);
10    }
11
12    printf("[");
13    for(i=0; i<tam; i++){
14
15        printf("%s", terminais[i]);
16
17    }
18    printf("] { return yytext[0];}");

```

```
19
20     printf("\n%s\n", "%s") ;
21 }
```

---

Para processar o conteúdo do ficheiro, o input é redirecionado para a gramática passada como parâmetro e o output irá também ser redirecionado para um ficheiro(output.l):

---

```
1 int main(int argc, char *argv[]) {
2
3     if ( argc != 2 ){
4
5         printf ("Usage : ./ exe <ficheiro yacc> \n");
6         return 0;
7     }
8
9     ifd=open(argv[1], O_RDONLY);
10    ofd = open("output.txt", O_CREAT | O_WRONLY | O_TRUNC, 0666);
11
12    int res1 = dup2(ifd, STDIN_FILENO);
13    int res2 = dup2(ofd, STDOUT_FILENO);
14
15    close(ifd);
16    close(ofd);
17
18    yylex();
19
20    return 0;
21 }
```

---

## Capítulo 4

# Codificação e Testes

### 4.1 Alternativas, Decisões e Problemas de Implementação

Ao longo da elaboração deste trabalho foram sendo encontradas algumas dificuldades e tomadas certas decisões.

Primeiramente, surgiu a questão de usar `char**` ou `char*`. Optou-se pela primeira opção uma vez que guarda um token por linha sendo mais acessível depois ler para o ficheiro.

Depois de escolher o formato para guardar, tanto os tokens como os símbolos terminais, surgiu o problema da alocação de memória. Queríamos criar o vetor com o tamanho pretendido, no entanto, à priori não sabemos quantos tokens/símbolos terminais irão ser pelo que foi necessário alocar à medida que o vetor atingia o máximo em cada iteração. De modo a não repetir código de alocação de memória(nos tokens e símbolos terminais) recorreremos à função *alocaMem()*.

Relativamente aos símbolos terminais, surgiu o obstáculo de poder haver símbolos repetidos na gramática. Para solucionar este problema foi criada uma função *eliminarrep(char \*\*)* que exclui todos os símbolos que são repetidos.

Por último, surgiu a dúvida da maneira como se iria ler os dados contidos nas duas variáveis para o ficheiro output. Este problema foi resolvido com o auxílio à função *criarFicheiro()*.

### 4.2 Testes realizados e Resultados

De modo a demonstrar tudo o que foi referido ao longo do relatório apresentamos de seguida algumas gramáticas e os respectivos resultados obtidos.

#### 4.2.1 alínea a)

Dada esta gramática:

---

```
1 ....
2 %union ....
3 %token ID MKLISTA NULA
4 %type ....
5 %%
6 lista : MKLISTA '(' ids ')' { ... a o sem ntica em C }
7      | NULA
```

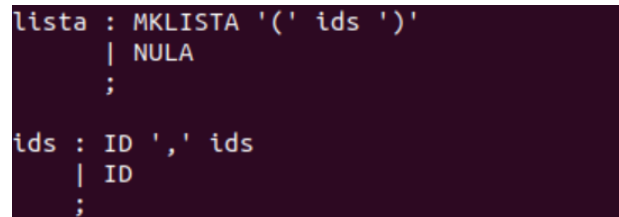
```

8         ;
9
10 ids : ID ',' ids { ignorar as constantes char (ex: '!') das a es sem nticas }
11      | ID { if ( ... == 'a') { ... } else { ...} }
12      ;
13 %%
14 ... codigo C...

```

---

O resultado obtido depois da aplicação do filtro extraGIC é:



```

lista : MKLISTA '(' ids ')'
      | NULA
      ;

ids : ID ',' ids
    | ID
    ;

```

Figura 4.1: Resultado da aplicação do filtro extraGIC

Outro exemplo de gramática:

---

```

1 %{
2 #include <stdio.h>
3 extern int yylex();
4 int yyerror();
5 %{
6
7 %token ERRO INT DOISPONTOS
8 %%
9 SeqListas : SeqListas Lista
10           |
11           ;
12
13 Lista    : '[' ']'
14          | '[' Elems ']'
15          ;
16
17 Elems    : Elem
18          | Elems ',' Elem
19          ;
20
21 Elem     : INT
22          | Intervalo
23          | Lista
24          ;
25
26 Intervalo : INT DOISPONTOS INT
27           ;
28
29 %%
30 ... codigo C...

```

---

O resultado obtido depois da aplicação do filtro extraGIC é:

```
SeqListas : SeqListas Lista
          |
          ;

Lista : '[' ']'
      | '[' Elems ']'
      ;

Elems : Elem
      | Elems ',' Elem
      ;

Elem : INT
      | Intervalo
      | Lista
      ;

Intervalo : INT DOISPONTOS INT
          ;
```

Figura 4.2: Resultado da aplicação do filtro extraGIC

#### 4.2.2 alínea b)

Utilizando as gramáticas acima apresentadas, o resultado da aplicação do filtro lexgen é o seguinte:

```
%%
FIXME { return ID;}
FIXME { return MKLISTA;}
FIXME { return NULA;}
[(,),] { return yytext[0];}
%%
```

Figura 4.3: Resultado da aplicação do filtro lexgen na 1ª gramática

```
%%
FIXME { return ERRO;}
FIXME { return INT;}
FIXME { return DOISPONTOS;}
[[],] { return yytext[0];}
%%
```

Figura 4.4: Resultado da aplicação do filtro lexgen na 2ª gramática

## 4.3 Guia de Utilização

### 4.3.1 Compilação

Apresentamos de seguida a makefile que permite a compilação do projeto, correndo o comando **make** que cria dois executáveis `extraiGIC.exe` e `lexgen.exe`. Esta makefile permite também a limpeza dos ficheiros criados na compilação bem como os que resultam da execução do programa. Executa-se o comando **make clean**, para esta funcionalidade.

---

```
1 all: extraiGIC lexgen
2
3 extraiGIC: extraiGIC.l
4             flex extraiGIC.l
5             gcc -o extraiGIC.exe lex.yy.c -ll
6
7 lexgen: lexgen.l
8             flex lexgen.l
9             gcc -o lexgen.exe lex.yy.c -ll
10
11
12 clean:
13         rm -f extraiGIC.exe
14         rm -f lex.yy.c
15         rm -f gram.txt
16         rm -f lexgen.exe
17         rm -f output.l
```

---

### 4.3.2 Execução

Depois de compilar os programas, a sua execução é feita do seguinte modo:

- Programa `extraGIC` :  
./extraiGIC.exe f.y  
cat gram.txt
- Programa `lexgen`:  
./lexgen.exe f.y  
cat output.l

Sendo o `f.y` um exemplo de uma gramática.

## Capítulo 5

# Conclusão

Após ter sido descrito todo o processo de desenvolvimento deste trabalho, desde a descrição do problema em causa até à implementação da solução, resta agora apresentar uma breve conclusão sobre todo o processo.

Este projeto permitiu-nos consolidar a matéria lecionada nas aulas relativamente à ferramenta Flex. Esta ferramenta mostrou-se bastante útil no processamento e filtragem de textos, tornando bastante fácil a sua manipulação. Para além disto, este trabalho permitiu-nos também desenvolver um espírito crítico e ponderado no que toca à tomada de decisões no decorrer do desenvolvimento do projeto.

De uma forma geral, terminamos este trabalho prático confiantes de que o seu objetivo foi cumprido e que todas as alíneas do enunciado foram corretamente respondidas.

Em suma, destaca-se a importância da escolha das estruturas de dados que tem impacto direto na eficácia do programa bem como das expressões regulares.