Departamento de Eletrónica, Telecomunicações e Informática

Curso	8295 – Licenciatura em Engenharia Informática
Disciplina	41469 – Compiladores
Ano letivo	2019/20

Relatório Trabalho Prático

Encriptação

Autores:

93346	Alexandra de Carvalho 1/6
93016	Anthony Pereira 1/6
93406	Fábio Carmelino 1/6
93078	João Soares 1/6
89318	Pedro Iglésias 1/6
89069	Pedro Marques 1/6
Grupo	P4G8

Data	15/06/2020
Docentes	Miguel Oliveira e Silva, Artur Pereira e André Zúquete

Resumo: Neste relatório pretende-se documentar a linguagem desenvolvida e justificar todas as opções tomadas durante o desenvolvimento do trabalho, em concordância com o tema escolhido.

Contextualização

O tema a que nos propomos solucionar é o número VI, que consiste em desenvolver uma linguagem que permita a encriptação e desencriptação de sequências de caracteres apresentados em bytes, uma vez ter sido este aquele que nos interessou mais e que possivelmente será mais importante no nosso futuro.

Documentação da Linguagem

Instruções: Todas as instruções definidas na linguagem terminam com o caracter ' ; '.

Tipos de variáveis: Para a nossa linguagem, definimos os seguintes tipos de varável: **integer** para representar números inteiros, **real** para representar números decimais, **boolean** para os valores lógicos de verdadeiro e falso, **str** para representar sequências de caracteres e **bytes** para representar sequências de bits.

Declaração e atribuição de variáveis: Podem ser efetuadas em simultâneo ou separadamente. Para a declaração, é necessário referir o tipo da variável entre parêntesis seguido do seu nome. A atribuição faz corresponder a uma variável o seu valor de tipo correspondente, através do sinal **'<<'**.

```
(integer) num;

num << 3;

(integer) dobro << 6;
```

Listas e dicionários: Listas são declaradas através da palavra **'list'**, seguida do seu tipo, expresso dentro parêntesis retos, e do nome da variável que a vai representar. De seguida é usado o sinal de atribuição **'<<'** para atribuir à variável a lista pretendida, expressa através de zero ou mais elementos dentro de parêntesis retos. Analogamente, a declaração de dicionários inicia com a palavra **'dict'**, seguida do tipo da chave e do valor, expressos dentro de parêntesis retos e separados por vírgula, e do nome da variável que representa o dicionário. Após a utilização do sinal **'<<'**, a atribuição de valores ao dicionário é realizada dentro de chavetas, numa lista de pares chave-valor, separados entre si por parêntesis e pela utilização da vírgula. Dentro destes parêntesis, separando a chave e o valor, encontra-se o caracter **':'**.

```
list [str] frutas << [“pera”, “maçã”, “banana”];

dict [str, int] pauta << {(“João” : 20) , (“Anthony” :
19), (“Alexandra” : 12)};
```

Ficheiros: Declarados através da palavra **'file'**, seguida do nome da variável que o vai representar. De seguida é usado o sinal de atribuição **'<<'** seguido da palavra **'open'** e do path (string) dentro de parêntesis.

```
file f << open ("textfile.txt");
```

Print: Para imprimir na consola é necessário escrever a palavra **'show'** seguida do sinal **'>>'** e, depois, da expressão a ser apresentada.

```
show >> x;
```

Comentários: Comentários in-line começam pelo caracter **'#'** e seguem até ao final da linha. Comentários multi-linha iniciam pelo mesmo caracter e estão circunscritos por chavetas.

```
(intger) i; #isto é um comentário inline

(integer) i; #{isto

é um comentário

multilinha}
```

Instruções Condicionais: Iniciados com a palavra **'if'**, seguida de uma expressão booleana a ser avaliada dentro de parêntesis e da palavra **'then'**. Dentro das chavetas que seguem, encontramos instruções a ser realizadas caso a expressão avaliada tenha valor positivo. É possível que de seguida, se encontrem alternativas. Estas têm a mesma estrutura descrita anteriormente, mas começam com a palavra **'else'**. Por fim, pode existir uma última condição, iniciada pela expressão **'else then'** e por instruções dentro de chavetas a ser realizadas caso todas as expressões booleanas anteriores sejam falsas.

```
(str) nome << "Vítor";

(str) sobrenome;


if (nome = "João") then{
     sobrenome << "Soares";
} else if (nome = "Anthony") then{
     sobrenome << "Pereira";
else then{
     sobrenome << "Desconhecido";
};
```

Ciclos While e For: O ciclo **while** é iniciado com a palavra **'while'** seguida da expressão booleana a avaliar antes de cada iteração dentro de parêntesis e da palavra **'do'**. Depois entre chavetas estão as instruções a realizar para cada iteração na qual a expressão seja verdadeira. Já o **do while** inicia com **'do while'** seguido da expressão booleana a avaliar depois de cada iteração dentro de parêntesis e ainda dentro de chavetas estão as instruções a realizar pelo menos uma vez e depois disso se a condição analisada anteriormente for verdadeira. O **for** começa com a palavra **'for'** e, dentro de parêntesis, uma variável já existente ou ali declarada. A palavra **'in'** segue a expressão, juntamente com o nome de uma variável que contenha uma lista ou com um intervalo de valores, expresso entre parêntesis com uma vírgula a separar o valor inicial e o final. Esta lista ou intervalo encontra-se dentro de parêntesis, bem como a possível palavra **'jump'** e um valor para modificar o passo, que por *default* é 1. De seguida, a palavra **'do'** introduz instruções dentro de parêntesis, a realizar enquanto a variável inicial percorre a lista / intervalo.

```
while ((integere) i << 1 < 10) do{
     show >> i;
     i << i + 1;
};

do while ((integer) i << 1 < 10) do{
     show >> i;
     i << i + 1;
};

for ((integer) i) in ((0,10) jump 2) do {
     show >> i;
};
```

Switch: Inicia com a palavra ‘switch’ seguida de uma variável declarada anteriormente. Dentro de chavetas, podemos encontrar algumas ocorrências da seguinte sintaxe: dentro de parêntesis é escrita uma expressão com a qual a variável acima mencionada se vai confrontar. Se o seu valor coincidir, são realizadas as instruções que seguem o carater ‘ : ‘. No final, pode existir uma última linha, que começa com a palavra ‘default’ seguida de ‘ : ‘ e de instruções que serão realizadas caso o valor da variável não coincida com nenhuma das propostas acima.

```
(integer) double;
switch (num){
     (1)  : double << 2;
     (2)  : double << 4;
     (3)  : double << 6;
     default : double << 0;
```

Funções: Iniciam com a palavra ‘function’ seguida do seu nome e, possivelmente, da palavra ‘uses’ com uma lista de tipo-nome de variáveis que serão necessárias como parâmentro de entrada. É também possível que tal sintaxe seja seguida da palavra ‘returns’ e do tipo que a variável de retorno apresenta. Depois dentro de chavetas encontra-se o corpo da função, um conjunto de instruções. Se a função retornar algum valor, há uma instrução constituída pela palavra ‘returns’ e o valor.

```
function comparevalues uses integer valor1, integer valor2
returns integer{
     if (valor1>valor2) then {
          returns valor1;
     } else if (valor1=valor2) then {
          returns 0;
     } else then {
          returns valor2;
     };
};
```

Operações sobre Ficheiros: Iniciam com o tipo de operação, que pode ser “READB”, “WRITEB” - para leitura e escrita binária, respetivamente – “READ”, “WRITE” – para leitura e escrita como string, respetivamente.

```
READ resultado << my_file;

WRITE conteudo >> my_file;

READB resultado << my_file;

WRITEB conteudo >> my_file;
```

Cypher: Pode ter parâmetros de entrada, tais como o nome doalgoritmo e 2 rotores.

```
cypher) c2 << alg: AlgTest << k: 0 <<rotor1: 10 << rotor2: 20;
```

Load: Serve para “carregar” o algoritmo de entrada. O algoritmo de entrada é elaborado em JAVA pelo utilizador e tem que ter 2 funções, encrypt e decrypt.

```
c2.load("AlgTest");
```

Encrypt: Serve para encriptar a mensagem, conforme o algoritmo de entrada definido.

```
c2.encrypt(msg);
```

Decrypt: Serve para desencriptar a mensagem, conforme o algoritmo de entrada definido.

```
c2.decrypt(message);
```

Outras Operações: Chamada de funções já declaradas, escrevendo o seu nome seguido de parêntesis, com uma lista opcional no seu interior de parâmetros de entrada, caso se adeque. Uma última operação é ainda chamar um método de uma classe, através da sua expressão, seguida de ponto e do nome do método. Analogamente, dentro dos seus parêntesis pode existir uma lista de parâmetros de entrada, caso se adeque.

```
(integer) highestValue << comparevalues(num1, num2);

dict.add(num1);
```

Gramática 1

Esta gramática corre uma statList, que é um conjunto de 0 ou mais stats, seguidas de ponto e vírgula. Estes, tal como vimos anteriormente, stats podem ser prints (show), declaração de variáveis, atribuição de valores às mesmas, declaração de ficheiros, listas ou dicionários, operações de leitura e escrita num ficheiro, instruções condicionais, loops for e while, do while, switch e funções.

```
grammar Encrypt;

import EncryptConfig;


main: statList EOF;

statList: (stat? ';')*;

stat: show
    | declaration
    | assignment
    | file
    | operation
    | list
    | dict
    | conditional
    | forLoop
    | whileLoop
    | doWhileLoop
    | switchSelect
    | function
    ;
```

Como expressões possíveis, temos os sinais de + e – (operações de soma e subtração), o sinal ^ para expoentes, com associatividade à direita e os sinais relacionados com multiplicação * (multiplicação), / (divisão) e % (resto da divisão). Temos ainda os sinais '=' '>' '<' e '!' que podem ser utilizados de forma singular ou combinada para comparar dois valores ou variáveis que os representem. Foram ainda incluídos os operadores binários 'and' e 'or'. É importante referir que as expressões podem estar dentro de parêntesis e que o seu sinal pode ser expresso à sua esquerda.

```
expr:
      sign=('+'|'-') e=expr                                                                  #signExpr
    | <assoc=right> e1=expr '^' e2=expr                                                      #powExpr
    | e1=expr op=('*'| '/' | '%') e2=expr                                                  #multDivExpr
    | e1=expr op=('+' | '-') e2=expr                                                       #addSubExpr
    | e1=expr op=('=' | '>' | '<' | '>=' | '<=' | '=>' | '=<' | '!=' | '!>' | '!<' | '!>=' | '!<=' | '!=>' | '!=<') e2=expr #comparisonExpr
    | '(' e=expr ')'                                                                         #parenExpr
    | expr op=('and'|'or') expr                                                            #addcomparisonExpr
    | REAL                                                                                   #realExpr
    | INTEGER                                                                                #integerExpr
    | BOOLEAN                                                                                #booleanExpr
    | ID                                                                                       #idExpr
    | STRING                                                                                                                  #strExpr
    | ID '('((expr ',' )* (expr) )? ')'                                                    #funcallExpr
    | expr'.'ID'(' ((expr ',' )* (expr))? ')'                                               #idcallExpr
    ;

```

Símbolos Terminais: Na nossa gramática, os BOOLEAN são definidos como 'true' ou 'false'. Um ID, nome de variável, é qualquer sequência de caracteres alfanuméricos minúsculos ou maiúsculos que comece por uma letra. Esta sequência pode incluir o caracter '_'. Uma STRING é delimitada por aspas, dentro das quais pode ter qualquer caracter ou sequência de caracteres, ou ainda aparecer vazia. É possível colocar aspas dentro de uma string, se esta for precedida do caracter '\'. Um valor REAL pode conter pelo menos um dígito entre 0-9, seguido de '.' e

opcionalmente de outros dígitos entre 0-9. Tanto os whitespaces " ", \t, \r e \n como os comentários são descartados.

```
100 BOOLEAN: 'true' | 'false';
101 ID: [a-zA-Z_][a-zA-Z_0-9]*;
102 STRING: '"' ( ESC | . )*? '"';
103 REAL: [0-9]+ '.' [0-9]*;
104 INTEGER: [0-9]+;
105 WS: [ \t\r\n]+ -> skip;
106 LINE_COMMENT: '#' .*? '\n' -> skip;
107 MULTILINE_COMMENT: '#{' .*? '}' -> skip;
108 fragment ESC: '\\"'
109             | '\\\\';
```

Testes da Gramática 1

No programa testsGrammar1.txt foram criados exemplos de utilização de todas as estruturas criadas, nomeadamente as que foram apresentadas anteriormente:

No primeiro exemplo são declaradas e atribuídas duas variáveis reais weight e height, de valores 75 e 1.70 respetivamente. É depois criada uma função getIMC com 2 parâmetros de entrada do tipo real e que retorna outro valor real, que é o quadrado da divisão do segundo parâmetro com o primeiro. Por fim, é impressa a frase "O valor de IMC obtido é x" sendo que x chama a função anterior. Aqui o resultado apropriado seria 25.951…

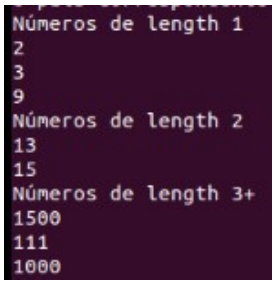
```
Processing ./EncryptConfig
O valor de IMC obtido é 25.95155709342561
```

No segundo exemplo é declarado um dicionário chamado phoneCodes com números inteiros como chaves e strings como valores e um número inteiro userInput, no caso com o valor 61. É depois impressa a frase "O país correspondente ao código introduzido é" seguida de um phoneCodes.get(userInput) que retorna o valor do dicionário correspondente à chave userInput. Como podemos ver no dicionário, o resultado expectável é o valor da chave 61, "Austrália".

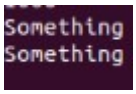
```
O pais correspondente ao código introduzido é Australia
```

O terceiro exemplo declara e atribui uma lista de números inteiros chamada nums com os valores 2, 1500, 3, 9, 13, 111, 15 e 1000. Declara depois mais 3 listas vazias chamadas one_char, two_char e three_char_more. É criado um for loop que coloca a variável elem a percorrer os elementos da lista nums. Cada um deles é então tratado numa estrutura condicional que começa por compará-los com 0, imprimindo "Números negativo detetado--> este será ignorado." se a variável elem for menor que 0. Uma outra condição é avaliada no 'else if' : se a variável for menor do que 10 será adicionada à lista one_char. Analogamente, se a variável for menor que 100 será adicionada à lista two_char, por fim, caso nenhuma das condições se verifique, a variável é adicionada à lista three_char_more. Já fora do ciclo for, introduzimos uma secção para imprimir os resultados obtidos. É impressa a frase "Números de length 1" e é depois

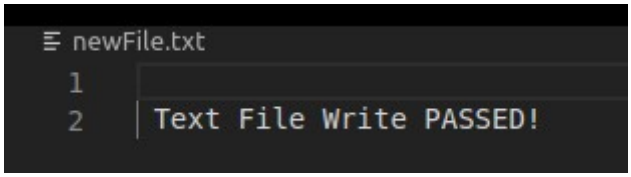
criado um ciclo for para percorrer e imprimir cada elemento da lista one_char. Analogamente, é de seguida impressa a frase "Números de length 2" e criado um segundo ciclo for para percorrer e imprimir cada elemento da lista two_char. Por fim é impressa a frase "Números de length 3+" e é percorrida a lista three_char_more para imprimir cada um dos seus elementos. Para os valores referidos anteriormente, desejamos que a lista one_char acabe com os valores 2, 3 e 9, a lista two_char acabe com os valores 13 e 15 e que os três valores restantes se passem a encontrar na lista three_char_more.

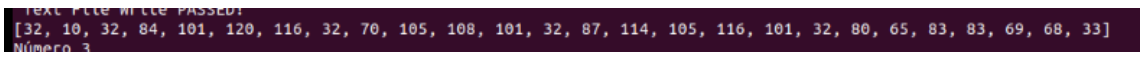


Um outro teste cria uma variável booleana Flag de valor true e um número inteiro de valor 1. É então criado um ciclo while que avalia a Flag. Enquanto esta for true, imprime na consola “Something” e a variável num é incrementada por 1. A estrutura condicional seguinte avalia se o num tem valor superior a 2 e, nesse caso, altera o valor da Flag para false. Então, é espectável que apenas a primeira e a segunda iterações, onde num tem valor inferior ou igual a 2, sejam executadas.

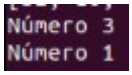


O teste seguinte introduz uma variável content, atribuída com uma string vazia. É introduzida também a variável my_file que representa o ficheiro “newFile.txt”. É então escrito nesse ficheiro " \n Text File Write" e, de seguida, este é lido, o que deve retornar a mesma frase. É então adicionado ao valor da variável content, retornado da leitura do ficheiro, a expressão " PASSED!". É lido novamente o conteúdo do ficheiro para a variável content, que é impressa por fim no terminal. É ainda criada e impressa uma variável fileBytes do tipo bytes, na qual é passado o valor lido em bytes do ficheiro.





O teste seguinte declara valueTeste com um número inteiro de valor 3. De seguida cria um do while que verifica se a variável é superior a 0. Enquanto isto é verdade, o código entra num switch que compara o valor da valueTeste com 1 (e nesse caso é impresso "Número 1" e a variável é decrementada por 1) e com 2 (nesse caso é impresso “Número 3” e a variável é decrementada por 2). Caso nenhuma destas comparações seja verdadeira, é impresso “Número 3”.



Visitors

Foi escolhida a utilização de visitors devido à necessidade de utilizar diretamente o output na segunda gramática. Os visitors dão mais flexibilidade e controlo no que é visitado ao longo do percorrer do programa.

Escolha da Linguagem

Foi escolhida a linguagem JAVA para a compilação da gramática por ser a linguagem com a qual estamos mais familiarizados.

Gramática 2

Esta segunda gramática fornece métodos para declarar e atribuir valores para o algoritmo e para as suas propriedades. Fornece também os meios para encriptar e desencriptar.

Declaração e Atribuição: A declaração é efetuada através da palavra **'cypher'** dentro de parêntesis seguida do nome da variável ou de uma atribuição. Esta é feita através do operador **'<<'** que atribui a um identificador – do lado esquerdo – um respetivo valor – do lado direito. As atribuições podem ser efetuadas de forma encadeada.

```
1   grammar EncryptConfig;
2
3   stat2: declarationC
4        | assignmentC
5        | put
6        | loadAlg
7        ;
8
9   assignmentC: ID '<<' exprC ('<<' exprC)*?;
10
11  declarationC: '(' 'cypher' ')' ( ID | assignmentC );
12
```

```
(cypher) c << alg: Algorithm << k: 0 <<rotor1: 10 << rotor2: 20;
```

Put: Alternativamente, é possível realizar uma atribuição através da keyword **'put'**, com a particularidade de esta forma não permitir encadeamento.

```
23   put: ID'(' exprC ')';
24
```

```
c.put(alg: Algorithm);
```

Carregamento de Algoritmos: Isto é realizado através da função **.load()** que carrega o ficheiro onde se encontra o algoritmo.

```
13   loadAlg: ID'.load(' STRING ')';
```

```
c.load(Algorithm);
```

Encriptação: É tratada através das funções .encrypt() e .depcrypt() – para encriptar e desencriptar, respetivamente. Os parâmetros da encriptação são passados como argumentos da função, apesar de ser possível passá-los também pelo JAVA através de inputs do utilizador do programa.

```
15  cripto: encryptAlg
16          |decryptAlg
17          ;
18
19  encryptAlg: ID'.encrypt('(ID|STRING)')';
20
21  decryptAlg: ID'.decrypt('(ID|STRING)')';
22
```

```
(str) message << c.encrypt("Isto é uma string.");
show >> c.decrypt(message);
```

```
25  exprC:n='alg'':' ID             #algExpr
26      | n='mode'':' ID            #modeExpr
27      | n='k'':' INTEGER          #keyExpr
28      | n='padding'':' ID         #paddingExpr
29      | ID ':' (ID|INTEGER|STRING)  #anyExpr
30      ;
31
32  ID: [a-zA-Z_][a-zA-Z_0-9]*;
33  STRING: '"' ( ESC | . )*? '"';
34  INTEGER: [0-9]+;
35  fragment ESC: '\\"'
36              | '\\\\';
```

Testes da Gramática 2

Para este exemplo, foi utilizado e declarado um algoritmo de encriptação que criámos para testar chamado “AlgTest” e cuja única funcionalidade de encriptação é devolver a mensagem recebida e acrescentar um valor. Este algoritmo de exemplo exige a utilização de uma chave, e de dois rotores, parâmetros estes cuja soma será o valor acrescentado. Utilizamos no exemplo a chave a 0, o rotor1 a 10 e o rotor2 a 20. O resultado esperado da encriptação será a mensagem passada como argumento, no caso “Teste Gramatica #2” seguida de “- 30”. A desencriptação deverá resultar na mensagem original. O algoritmo é passado através da função load.

```
2   #Teste exemplo Gramatica 2
3
4   (cypher) c2 << alg: AlgTest << k: 0 <<rotor1: 10 << rotor2: 20;
5
6   c2.load("AlgTest");
7
8   (str) msg << "Teste Gramatica #2";
9
10  show >> msg;
11
12  (str) message << c2.encrypt(msg);
13
14  show >> message;
15
16  show >> c2.decrypt(message);
```

```
Processing ./Encrypt
Processing ./EncryptConfig
Teste Gramatica #2
Teste Gramatica #2-30
Teste Gramatica #2
```

Encriptação

O nosso programa de encriptação foi baseado no ENIGMA que foi utilizado pela Alemanha durante a Segunda Guerra Mundial.

Nessa época usavam uma máquina com vários rotores que faziam troca direta de cada caractere por outro durante a encriptação, devendo a pessoa que desencriptar a mensagem saber quais os rotores que foram usados ao encriptar, para assim poder usar os mesmos na desencriptação.

Para simular esta ideia, começámos por criar ficheiros que têm mapas que vão ter um papel semelhante aos rotores. Para criar esses mapas, fizemos um programa chamado "MapaGenerator" que vai gerar 2 arrays de números entre 1 e 128 (valores do ASCII Code) e através de um 'ciclo for' alterar aleatoriamente a ordem de um dos arrays. De seguida, com outro 'ciclo for', convertemos cada um dos números de decimal para binário e escrevemo-los num ficheiro no formato " <arr1[i]> <"-"> <arr2[i]> " (por exemplo:00000011-00001111), ou seja, quando esse mapa for utilizado (carregado para um HashMap em JAVA), um dos valores de

cada linha será usado como 'chave' e o outro como 'valor', permitindo fazer uma troca direta de um caractere para outro caractere.

Neste programa existem 2 Métodos principais que vão ser chamados pela nossa gramática: um método encrypt() e um método decrypt(). Ao longo da sua execução é pedido input ao utilizador perguntando qual o ficheiro que quer encriptar ou desencriptar, assim como também quais mapas quer utilizar no caso da encriptação. Além disso, existem ainda várias métodos auxiliares aos quais o nossos 2 métodos principais recorrem:

FileToMap(HashMap<String, String> map, String choosenmap,boolean reverse).

Na encriptação, este método utiliza boolean reverse com valor falso, lê cada um dos mapas (ficheiros) escolhidos pelo utilizador e carrega-os para HashMaps, utilizando cada valor à esquerda do hífen como 'key' e à direita como 'value'.

Na desencriptação, este método utiliza boolean reverse com valor true e lê cada um dos mapas referidos na primeira linha do ficheiro encriptado (quando se encripta, a primeira linha não é encriptada e tem um expressão do tipo "1-2-3", para quem quiser desencriptar saber quais mapas tem de usar) e carrega-os para HashMaps, utilizando cada valor à esquerda do hífen desta vez como 'value' e à direita como 'key'.

FileToVector(String filename) - carrega o ficheiro que vai ser encriptado ou desencriptado para um vetor.

CypherXDCypher(String filename, Vector<String> frases, String ChoosenMaps, HashMap<String,String> map1,HashMap<String,String> map2,HashMap<String,String> map3) - Esta função é usada tanto pelo Encrypt() como o Decrypt() da mesma maneira, sendo a única diferença os mapas que são passados. O decrypt usa os mapas inversos na ordem inversa do encrypt para chegar aos caracteres originais. Nesta função, cada linha do vetor onde foi carregado o file (para desencriptar ou encriptar) é percorrida caracter a caracter em 3 'ciclos for' (um para cada mapa) onde vai ser procurada a correspondência do caracter nas chaves do mapa para o poder substituir pelo seu 'value'. Após encriptar ou desencriptar todas as linhas, irá percorrer o vetor com as frases e escrever cada uma delas num novo ficheiro.

BinaryToString(String binary) e StringToBinary(String s) - estes 2 métodos fazem o oposto um do outro. Para se utilizar bytes em JAVA utilizámos a função "StringToBinary" que retorna uma String com o código binário da frase passada como argumento e a função BinaryToString lê uma String com código binário e converte-a para uma String de caracteres. No método CypherXDCypher referido acima, não se utiliza diretamente os caracteres na sua representação alfabética, mas sim Strings com a sua representação binária.

Testes da Gramática 1 & 2

```
#Teste exemplo Gramatica 2 e 1 utilizando o algoritmo Enigma

#Encrypt ficheiro ALovePoembyAnonymous.txt

(cypher) c2 << alg: AlgEnigma;

c2.load("AlgEnigma");

c2.encrypt();

c2.decrypt();

```

```
import java.util.*;
import java.io.*;
public class Output{
    public static void main(String[] args){
        AlgEnigma c2 = new AlgEnigma();
        c2.encrypt();
        c2.decrypt();
    }

}
```

```
4-2-1
A Love Poem by Anonymous

Roses are red,
Violets are blue,
My perfume is lovely,
And so are you.

Orchids are white,
Ghost ones are rare,
Marbles are shiny,
And so is your hair.

Magnolia grows,
With buds like eggs,
An ocean is beautiful,
And so are your legs.

Sunflowers reach,
Up to the skies,
My sun is dazzling,
And so are your eyes.

Foxgloves in hedges,
Surround the farms,
BPA-free water bottles are safe,
And so are your arms.

Daisies are pretty,
Daffies have style,
The moonlight is illuminating,
And so is your smile.

A is beautiful,
Just like you.

```

```
342d 322d 310a 7a69 2138 3d73 6975 3873
4069 3b66 697a 5338 5366 4038 151e 0a69
0a30 381e 731e 6941 5c73 695c 7378 1c0a
0100 3862 7348 1e69 415c 7369 3b62 1573
1c0a 6166 6977 735c 0f15 4073 6900 1e69
6238 3d73 6266 1c0a 7a53 7869 1e38 6941
5c73 6966 3815 250a 0a6e 5c1b 5d00 781e
6941 5c73 691a 5d00 4873 1c0a 635d 381e
4869 3853 731e 6941 5c73 695c 415c 731c
0a61 415c 3b62 731e 6941 5c73 691e 5d00
5366 1c0a 7a53 7869 1e38 6900 1e69 6638
155c 695d 4100 5c25 0a0a 6141 1753 3862
0041 6917 5c38 1a1e 1c0a 0e00 485d 693b
1578 1e69 6200 4573 6973 1717 1e1c 0a7a
5369 381b 7341 5369 001e 693b 7341 1548
000f 1562 1c0a 7a53 7869 1e38 6941 5c73
6966 3815 5c69 6273 171e 250a 0a10 1553
0f62 381a 735c 1e69 5c73 411b 5d1c 0a06
7769 4838 6948 5d73 691e 4500 731e 1c0a
6166 691e 1553 6900 1e69 7841 6565 6200
5317 1c0a 7a53 7869 1e38 6941 5c73 6966
3815 5c69 7366 731e 250a 0a50 380c 1762
383d 731e 6900 5369 5d73 7817 731e 1c0a
1015 5c5c 3815 5378 6948 5d73 690f 415c
401e 1c0a 1675 7a7d 0f5c 7373 691a 4148
735c 693b 3848 4862 731e 6941 5c73 691e
410f 731c 0a7a 5378 691e 3869 415c 7369
6638 155c 6941 5c40 1e25 0a0a 6041 001e
0073 1e69 415c 7369 775c 7348 4866 1c0a
6041 0f0f 0073 1e69 5d41 3d73 691e 4866
6273 1c0a 795d 7369 4038 3853 6200 175d
4869 001e 6900 6262 1540 0053 4148 0053
171c 0a7a 5378 691e 3869 001e 6966 3815
5c69 1e40 0062 7325 0a0a 7a69 001e 693b
7341 1548 000f 1562 1c0a 7115 1e48 6962
0045 7369 6638 1525 0a
```

Encriptação do ficheiro de teste “A LovePoembyAnonymous.txt” e consequente desencriptação deste, utilizando o algoritmo ENIGMA explicado anteriormente.