**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 | | |
| 93016 | Anthony Pereira | | |
| 93406 | Fábio Carmelino | | |
| 93078 | João Soares | | |
| 89318 | Pedro Iglésias | | |
| 89069 | Pedro Marques | | |
| Grupo | P4G8 |  |  |

|  |  |
| --- | --- |
| Data | 14/06/2020 |
| 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.

**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;

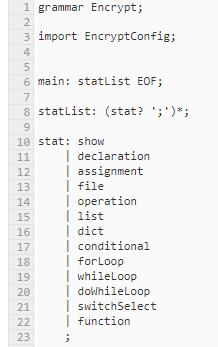
**Outras Operações:** Chamada de funções já declaradas, escrevendo o seu nome seguido de parêntesis, com uma lista opcional no seu interior de parâmetros de entrada, caso se adeque. Uma última operação é ainda chamar um método de uma classe, através da sua expressão, seguida de ponto e do nome do método.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

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

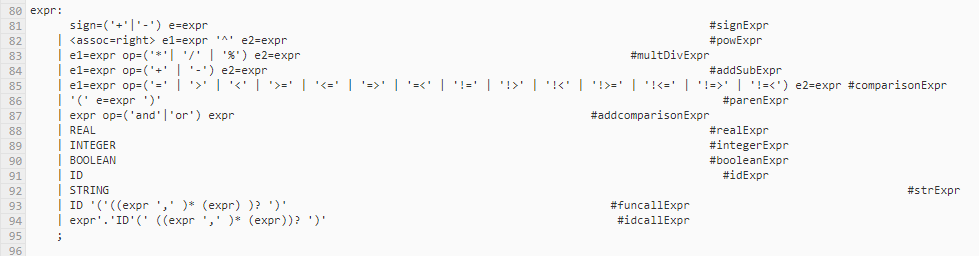
dict.add(num1);

**Gramática 1**

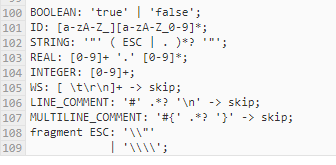
Esta gramática corre uma statList, que é um conjunto de 0 ou mais stats, seguidas de ponto e vírgula. Estes, tal como vimos anteriormente, stats podem ser prints (show), declaração de variáveis, atribuição de valores às mesmas, declaração de ficheiros, listas ou dicionários, operações de leitura e escrita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



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.



**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.



**Testes da Gramática 1**

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

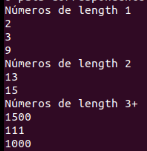
No primeiro exemplo são declaradas e atribuídas duas variáveis reais weight e height, de valores 75 e 1.70 respetivamente. É depois criada uma função getIMC com 2 parâmetros de entrada do tipo real e que retorna outro valor real, que é o quadrado da divisão do segundo parâmetro com o primeiro. Por fim, 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…



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”.



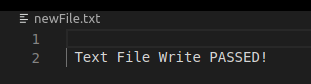
O terceiro exemplo declara e atribui uma lista de números inteiros chamada nums com os valores 2, 1500, 3, 9, 13, 111, 15 e 1000. Declara depois mais 3 listas vazias chamadas one\_char, two\_char e three\_char\_more. É criado um for loop que coloca a variável elem a percorrer os elementos da lista nums. 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.

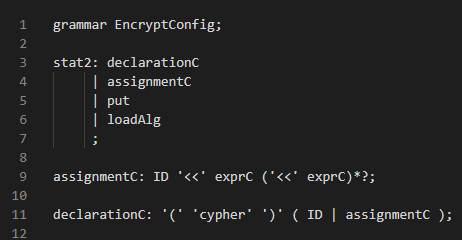
**Escolha da Linguagem**

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.



(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.



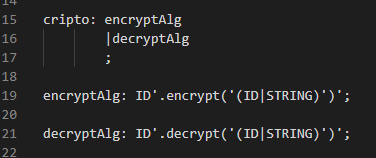
c.put(alg: Algorithm);

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



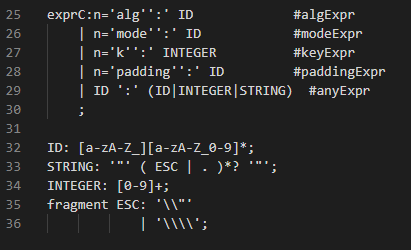
c.load(Algorithm);

**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.



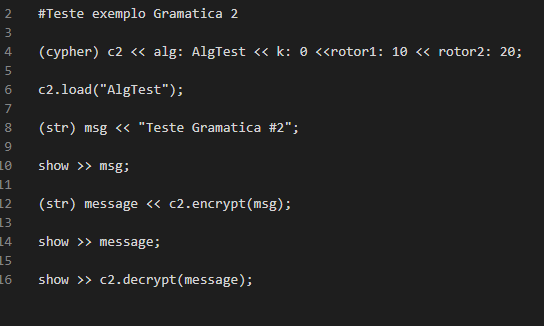
(str) message << c.encrypt(“Isto é uma string.”);

show >> c.decrypt(message);



**Testes da Gramática 1**

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.





**Encriptação**