

abr 24, 13 15:47

## word\_mangling.h

Page 1/1

```

1  /* *****
2  * *****
3  * Clase WORDMANGLING
4  * .....
5  * Modela el módulo encargado de llevar a cabo el word mangling a partir de una
6  * lista de reglas sobre palabras recibidas a través de un receptor.
7  *
8  * *****
9  * *****/
10
11
12 #ifndef WORD_MANGLING_H
13 #define WORD_MANGLING_H
14
15
16 #include "lista.h"
17 #include "regla.h"
18 #include "receptor.h"
19
20
21
22 class WordMangling {
23 private:
24     Lista< Regla >& lReglas;    // Lista de reglas a aplicar
25
26 public:
27
28     // Constructor
29     explicit WordMangling(Lista< Regla >& lReglas);
30
31     // Ejecuta el alterador de palabras.
32     // PRE: 'rxPalabras' es un Receptor de palabras sobre las que se
33     // aplicarán las reglas.
34     void ejecutar(Receptor* rxPalabras);
35 };
36
37
38 #endif

```

abr 24, 13 15:47

## word\_mangling.cpp

Page 1/1

```

1  /* *****
2  * *****
3  * Clase WORDMANGLING
4  * .....
5  * Modela el módulo encargado de llevar a cabo el word mangling a partir de una
6  * lista de reglas sobre palabras recibidas a través de un receptor.
7  *
8  * *****
9  * *****/
10
11
12 #include "word_mangling.h"
13 #include "listaref.h"
14
15
16
17
18 // Constructor:
19 // 'lReglas' es una Lista de Reglas que se desean aplicar
20 WordMangling::WordMangling(Lista< Regla >& lReglas) : lReglas(lReglas) { }
21
22
23 // Ejecuta el alterador de palabras.
24 // PRE: 'rxPalabras' es un Receptor de palabras sobre las que se
25 // aplicarán las reglas.
26 void WordMangling::ejecutar(Receptor *rxPalabras) {
27     // Aplicamos reglas a cada palabra entrante recibida
28     std::string palabra = rxPalabras->recibir();
29
30     while(!palabra.empty()) {
31         // Creamos la lista de transformaciones
32         ListaRef< std::string > lTransformaciones;
33         // Insertamos la palabra original
34         lTransformaciones.insertarUltimo(palabra);
35
36         // Aplicamos reglas a palabra
37         for(int i = 0; i < this->lReglas.tamano(); i++)
38             this->lReglas[i]->aplicar(lTransformaciones);
39
40         palabra = rxPalabras->recibir();
41     }
42 }

```

abr 24, 13 15:47

tx\_salida\_estandar.h

Page 1/1

```

1  /* *****
2  * *****
3  * Clase TXSALIDAESTANDAR
4  * .....
5  * Transmisor que emite resultados hacia la salida estándar.
6  *
7  * *****
8  * *****/
9
10
11 #ifndef TX_SALIDA_ESTANDAR_H
12 #define TX_SALIDA_ESTANDAR_H
13
14
15 #include <string>
16 #include "transmisor.h"
17
18
19
20 class TxSalidaEstandar:public Transmisor {
21 public:
22     TxSalidaEstandar();
23
24     // Se ejecuta la transmisión de un dato hacia la salida estándar.
25     virtual void transmitir(const std::string& palabra);
26 };
27
28
29 #endif

```

abr 24, 13 15:47

tx\_salida\_estandar.cpp

Page 1/1

```

1  /* *****
2  * *****
3  * Clase TXSALIDAESTANDAR
4  * .....
5  * Transmisor que emite resultados hacia la salida estándar.
6  *
7  * *****
8  * *****/
9
10
11 #include "tx_salida_estandar.h"
12 #include <iostream>
13
14
15
16 // Constructor
17 TxSalidaEstandar::TxSalidaEstandar() {
18     activar();
19 }
20
21
22 // Se ejecuta la transmisión de un dato hacia la salida estándar.
23 void TxSalidaEstandar::transmitir(const std::string& palabra) {
24     std::cout << palabra << std::endl;
25 }
26

```

abr 24, 13 15:47

tx\_archivo.h

Page 1/1

```

1  /* *****
2  * *****
3  * Clase TXARCHIVO
4  * .....
5  * Transmisor que transmite los datos hacia un archivo de salida. El archivo de
6  * salida se crea si no existe, y de existir, no se trunca, sino que se siguen
7  * escribiendo las transmisiones al final del archivo.
8  * .....
9  * *****
10 * *****/
11
12
13 #ifndef TX_ARCHIVO_H
14 #define TX_ARCHIVO_H
15
16
17 #include "transmisor.h"
18 #include <fstream>
19
20
21 class TxArchivo:public Transmisor {
22 private:
23
24     std::ofstream archivo;    // Archivo en donde se escriben las palabras
25                             // alteradas
26
27 public:
28
29     explicit TxArchivo(const std::string& nombre_archivo);
30
31     virtual ~TxArchivo();
32
33     // Se ejecuta la transmisión de un dato hacia un archivo. Se almacena
34     // una palabra por línea.
35     virtual void transmitir(const std::string& palabra);
36 };
37
38
39 #endif

```

abr 24, 13 15:47

tx\_archivo.cpp

Page 1/1

```

1  /* *****
2  * *****
3  * Clase TXARCHIVO
4  * .....
5  * Transmisor que transmite los datos hacia un archivo de salida. El archivo de
6  * salida se crea si no existe, y de existir, no se trunca, sino que se siguen
7  * escribiendo las transmisiones al final del archivo.
8  * .....
9  * *****
10 * *****/
11
12
13 #include "tx_archivo.h"
14
15
16
17 // Constructor
18 TxArchivo::TxArchivo(const std::string& nombre_archivo) {
19     this->archivo.open(nombre_archivo.c_str(), std::ios::app);
20
21     // Verificamos que se halla abierto correctamente
22     if(this->archivo.is_open()) activar();
23     else desactivar();
24 }
25
26 // Destructor
27 TxArchivo::~TxArchivo() {
28     this->archivo.close();
29 }
30
31
32 // Se ejecuta la transmisión de un dato hacia un archivo.
33 void TxArchivo::transmitir(const std::string& palabra) {
34     this->archivo << palabra << std::endl;
35 }
36

```

abr 24, 13 15:47

transmisor.h

Page 1/1

```

1  /* *****
2  * *****
3  * Clase TRANSMISOR
4  * *****
5  * *****/
6
7  #ifndef TRANSMISOR_H
8  #define TRANSMISOR_H
9
10
11  #include <string>
12
13
14  class Transmisor {
15  private:
16
17      // Estado del transmisor
18      bool activo;
19
20  protected:
21
22      // Activa el transmisor
23      void activar();
24
25      // Desactivar el transmisor
26      void desactivar();
27
28  public:
29
30      // Destructor
31      virtual ~Transmisor() { };
32
33      // Verifica si el transmisor esta activo.
34      // POST: devuelve true si está activo o false en caso contrario
35      bool estaActivo();
36
37      // Se ejecuta la transmisión de un dato.
38      virtual void transmitir(const std::string& palabra) = 0;
39  };
40
41
42
43  #endif

```

abr 24, 13 15:47

transmisor.cpp

Page 1/1

```

1  /* *****
2  * *****
3  * Clase TRANSMISOR
4  * *****
5  * *****/
6
7
8  #include "transmisor.h"
9
10
11  // Verifica si el transmisor esta activo.
12  // POST: devuelve true si está activo o false en caso contrario
13  bool Transmisor::estaActivo() {
14      return this->activo;
15  }
16
17
18  // Activa el transmisor
19  void Transmisor::activar() {
20      this->activo = true;
21  }
22
23
24  // Desactiva el transmisor
25  void Transmisor::desactivar() {
26      this->activo = false;
27  }

```

abr 24, 13 15:47

tp.cpp

Page 1/2

```

1  /* *****
2  * WORD MANGLING
3  * *****
4  *
5  * Facultad de Ingeniería - UBA
6  * 75.42 Taller de Programación I
7  * Trabajo Práctico N°3
8  *
9  * ALUMNO: Federico Martín Rossi
10 * PADRÓN: 92086
11 * EMAIL: federicomrossi@gmail.com
12 *
13 * *****
14 *
15 * Programa que realiza el proceso de word mangling a partir de un conjunto de
16 * reglas especificadas por el usuario en un archivo de entrada. El ingreso de
17 * palabras puede realizarse a través de otro archivo de entrada o mediante la
18 * entrada estándar. Por cada palabra leída se le aplicará la serie de reglas
19 * modificadoras, emitiéndose las palabras resultantes hacia un archivo de
20 * salida, o bien, directamente en la salida estándar.
21 *
22 *
23 * FORMA DE USO
24 * =====
25 *
26 * Deberá ejecutarse el programa en la línea de comandos de la siguiente
27 * manera:
28 *
29 * # ./tp [archivo_reglas] [entrada] [salida]
30 *
31 * donde,
32 *
33 *     archivo_reglas: nombre del archivo (incluyendo su extensión) en donde
34 *     se almacenan las reglas a aplicar;
35 *     entrada: especificación de la entrada de palabras. Debe especificarse
36 *     un nombre de archivo o el símbolo '-' si se desea que la
37 *     entrada de palabras sea a través de la entrada estándar;
38 *     salida: especificación de la salida de palabras. Debe especificarse
39 *     un nombre de archivo o el símbolo '-' si se desea que la
40 *     salida de palabras sea a través de la salida estándar.
41 *
42 *
43 * INSTRUCCIONES DEL ARCHIVO DE REGLAS
44 * =====
45 *
46 * A continuación se muestran las instrucciones válidas para las reglas que
47 * se desean aplicar. Cada una de ellas se debe finalizar con un ";".
48 *
49 * - "uppercase n m": Transforma los caracteres que están en la posición 'n'
50 * hasta la posición 'm' (inclusive) a mayúsculas;
51 *
52 * - "lowercase n m": Transforma los caracteres que están en la posición 'n'
53 * hasta la posición 'm' (inclusive) a minúsculas;
54 *
55 * - "repeat n m r i": Copia el substring delimitado por 'n' y 'm', lo repite
56 * 'r' veces y lo inserta en la posición 'i';
57 *
58 * - "rotate n": Mueve los caracteres 'n' lugares hacia la derecha en forma
59 * circular. Si 'n' es negativo, la rotación se hace hacia la izquierda;
60 *
61 * - "insert i mmm": Agrega el string 'mmm' en la posición 'i'. 'mmm' no
62 * debe contener ningún espacio ni salto de línea ni el carácter ';';
63 *
64 * - "revert i": Revierte los efectos de las modificaciones de las últimas
65 * 'i' reglas. El parámetro 'i' es siempre mayor o igual a 1.
66 *

```

abr 24, 13 15:47

tp.cpp

Page 2/2

```

67  */
68
69
70  #include "parser_entrada.h"
71  #include "parser_salida.h"
72  #include "parser_reglas.h"
73  #include "word_mangling.h"
74
75
76
77
78  /* *****
79  * PROGRAMA PRINCIPAL
80  * *****
81
82
83  int main(int argc, char* argv[]) {
84  // Si la cantidad de argumentos no es la correcta, lanzamos código de
85  // retorno
86  if(argc != 4) return 1;
87
88  // Declaramos parsers
89  ParserEntrada pEntrada;
90  ParserSalida pSalida;
91  ParserReglas pReglas;
92
93  // Parseamos argumentos de entrada
94  Receptor *rxPalabras = pEntrada.parsear(argv[2]);
95  Transmisor *tx = pSalida.parsear(argv[3]);
96  Lista< Regla > lReglas = pReglas.parsear(argv[1], tx);
97
98  // Verificamos que esten activos el transmisor y el receptor
99  if(!rxPalabras->estaActivo() || !tx->estaActivo() || lReglas.estaVacia()) {
100  // Liberamos memoria
101  delete rxPalabras;
102  delete tx;
103
104  // Retornamos código de error
105  return 2;
106  }
107
108  // Ejecutamos el proceso de alteración de palabras
109  WordMangling wordMangling(lReglas);
110  wordMangling.ejecutar(rxPalabras);
111
112  delete rxPalabras;
113  delete tx;
114
115  return 0;
116  }

```

abr 24, 13 15:47

rx\_entrada\_estandar.h

Page 1/1

```

1  /* *****
2  * *****
3  * Clase RXENTRADAESTANDAR
4  * .....
5  * Receptor que recibe los datos desde la entrada estándar.
6  *
7  * *****
8  * *****/
9
10
11 #ifndef RX_ENTRADA_ESTANDAR_H
12 #define RX_ENTRADA_ESTANDAR_H
13
14
15 #include "receptor.h"
16
17
18
19 class RxEntradaEstandar:public Receptor {
20 public:
21
22     // Constructor
23     RxEntradaEstandar();
24
25     // Se ejecuta la recepción de un dato desde la entrada estándar.
26     // PRE: Se debe ingresar una palabra sin espacios.
27     // POST: se devuelve la palabra recibida. Debe considerarse terminada la
28     // recepción de palabras cuando se recibe una cadena vacía, es decir, nada.
29     // Debe enviarse una única palabra a la vez, quedando terminantemente prohibid
30     o
31     // enviar una cadena que conste de mas de una palabra.
32     virtual std::string recibir();
33 };
34
35 #endif

```

abr 24, 13 15:47

rx\_entrada\_estandar.cpp

Page 1/1

```

1  /* *****
2  * *****
3  * Clase RXENTRADAESTANDAR
4  * .....
5  * Receptor que recibe los datos desde la entrada estándar.
6  *
7  * *****
8  * *****/
9
10
11 #include <iostream>
12 #include "rx_entrada_estandar.h"
13
14
15
16 // Constructor
17 RxEntradaEstandar::RxEntradaEstandar() {
18     activar();
19 }
20
21
22 // Se ejecuta la recepción de un dato desde la entrada estándar.
23 // PRE: Se debe ingresar una palabra sin espacios.
24 // POST: se devuelve la palabra recibida. Debe considerarse terminada la
25 // recepción de palabras cuando se recibe una cadena vacía, es decir, nada.
26 // Debe enviarse una única palabra a la vez, quedando terminantemente prohibido
27 // enviar una cadena que conste de mas de una palabra.
28 std::string RxEntradaEstandar::recibir() {
29     std::string palabra;
30     getline(std::cin, palabra);
31
32     return palabra;
33 }

```

abr 24, 13 15:47

rx\_archivo.h

Page 1/1

```

1  /* *****
2  * *****
3  * Clase RXARCHIVO
4  * .....
5  * Receptor que recibe los datos desde un archivo.
6  *
7  * *****
8  * *****/
9
10
11 #ifndef RX_ARCHIVO_H
12 #define RX_ARCHIVO_H
13
14
15 #include "receptor.h"
16 #include <fstream>
17
18
19 class RxArchivo:public Receptor {
20 private:
21
22     std::ifstream archivo;      // Archivo del que se toman las palabras
23
24
25 public:
26
27     // Constructor
28     explicit RxArchivo(const std::string& nombre_archivo);
29
30     // Destructor
31     ~RxArchivo();
32
33     // Se ejecuta la recepción de un dato desde la entrada estándar.
34     // POST: se devuelve la palabra recibida. Debe considerarse terminada la
35     // recepción de palabras cuando se recibe una cadena vacía, es decir, nada. En
36     // este caso debe limitarse estrictamente a una palabra por línea del archivo.
37     virtual std::string recibir();
38 };
39
40
41 #endif

```

abr 24, 13 15:47

rx\_archivo.cpp

Page 1/1

```

1  /* *****
2  * *****
3  * Clase RXARCHIVO
4  * .....
5  * Receptor que recibe los datos desde un archivo.
6  *
7  * *****
8  * *****/
9
10
11 #include "rx_archivo.h"
12
13
14
15 // Constructor
16 RxArchivo::RxArchivo(const std::string& nombre_archivo) {
17     // Abrimos archivo
18     this->archivo.open(nombre_archivo.c_str());
19
20     // Verificamos que se halla abierto correctamente
21     if(this->archivo.is_open()) activar();
22     else desactivar();
23 }
24
25
26 // Destructor
27 RxArchivo::~RxArchivo() {
28     this->archivo.close();
29 }
30
31
32 // Se ejecuta la recepción de un dato desde la entrada estándar.
33 // POST: se devuelve la palabra recibida. Debe considerarse terminada la
34 // recepción de palabras cuando se recibe una cadena vacía, es decir, nada. En
35 // este caso debe limitarse estrictamente a una palabra por línea del archivo.
36 std::string RxArchivo::recibir() {
37     std::string palabra;
38     this->archivo >> palabra;
39
40     return palabra;
41 }

```

abr 24, 13 15:47

## ruppercase.h

Page 1/1

```

1  /* *****
2  * *****
3  * Clase RUPPERCASE
4  * .....
5  * La regla transforma los caracteres que están en la posición 'n' hasta la
6  * posición 'm' (inclusive) a mayúsculas.
7  * .....
8  * *****
9  * *****/
10
11
12 #ifndef RUPPERCASE_H
13 #define RUPPERCASE_H
14
15
16 #include "regla.h"
17
18
19
20 class RUppercase:public Regla {
21 private:
22
23     int n;    // Posición inicial de transformación
24     int m;    // Posición final de transformación
25
26 public:
27
28     // Constructor
29     RUppercase(int n, int m);
30
31     // Destructor
32     ~RUppercase();
33
34     // Aplica la regla sobre una lista de transformaciones.
35     virtual void aplicar(ListaRef< std::string >& lTransformaciones);
36 };
37
38 #endif

```

abr 24, 13 15:47

## ruppercase.cpp

Page 1/1

```

1  /* *****
2  * *****
3  * Clase RUPPERCASE
4  * .....
5  * La regla transforma los caracteres que están en la posición 'n' hasta la
6  * posición 'm' (inclusive) a mayúsculas.
7  * .....
8  * *****
9  * *****/
10
11
12 #include "ruppercase.h"
13
14
15
16 // Constructor
17 RUppercase::RUppercase(int n, int m) : n(n), m(m) { }
18
19
20 // Destructor
21 RUppercase::~RUppercase() { }
22
23
24 // Aplica la regla sobre una lista de transformaciones
25 void RUppercase::aplicar(ListaRef< std::string >& lTransformaciones) {
26     // Tomamos la palabra sobre la cual debemos aplicar
27     std::string s = lTransformaciones.verUltimo();
28     int sTamanio = s.size();
29
30     // Verificamos si estamos dentro del rango de la palabra
31     if(estaFueraDeRango(sTamanio, this->n, this->m)) return;
32
33     // Convertimos posiciones en posiciones válidas respecto al string
34     int nn = convertirEnPosicionValida(sTamanio, this->n);
35     int mm = convertirEnPosicionValida(sTamanio, this->m);
36
37     // Transformamos en mayúscula las letras del conjunto
38     for(int i = nn; i <= mm; i++)
39         s[i] = toupper(s[i]);
40
41     // Insertamos la transformación en la lista
42     lTransformaciones.insertarUltimo(s);
43 }

```



abr 24, 13 15:47

rrotate.h

Page 1/1

```

1  /* *****
2  * *****
3  * Clase RROTATE
4  * .....
5  * La regla mueve los caracteres 'n' lugares hacia la derecha en forma
6  * circular. Si 'n' es negativo, la rotación se hace hacia la izquierda.
7  *
8  * *****
9  * *****/
10
11
12 #ifndef RROTATE_H
13 #define RROTATE_H
14
15 #include "regla.h"
16
17
18
19 class RRotate:public Regla {
20 private:
21
22     int n;          // Cantidad de lugares que deben moverse los caracteres
23     bool derecha;   // Rotar hacia la derecha
24
25 public:
26
27     // Constructor
28     explicit RRotate(int n);
29
30     // Destructor
31     ~RRotate();
32
33     // Aplica la regla sobre una lista de transformaciones
34     virtual void aplicar(ListaRef< std::string >& lTransformaciones);
35 };
36
37
38
39 #endif

```

abr 24, 13 15:47

rrotate.cpp

Page 1/1

```

1  /* *****
2  * *****
3  * Clase RROTATE
4  * .....
5  * La regla mueve los caracteres 'n' lugares hacia la derecha en forma
6  * circular. Si 'n' es negativo, la rotación se hace hacia la izquierda.
7  *
8  * *****
9  * *****/
10
11
12 #include "rrotate.h"
13
14
15
16 // Constructor
17 RRotate::RRotate(int n) {
18     if(n >= 0) {
19         this->derecha = true;
20         this->n = n;
21     }
22     else {
23         this->derecha = false;
24         this->n = n * (-1);
25     }
26 }
27
28
29 // Destructor
30 RRotate::~RRotate() { }
31
32
33 // Aplica la regla sobre una lista de transformaciones
34 void RRotate::aplicar(ListaRef< std::string >& lTransformaciones) {
35     // Tomamos la palabra sobre la cual debemos aplicar
36     std::string s = lTransformaciones.verUltimo();
37     int sTamanio = s.size();
38
39     for(int j = 0; j < this->n; j++) {
40         // Rotamos hacia la derecha
41         if(this->derecha) {
42             // Guardamos el último carácter que será el último en moverse
43             char u = s[sTamanio-1];
44             for(int i = sTamanio-1; i >= 1; i--) s[i] = s[i-1];
45             s[0] = u;
46         }
47         // Rotamos hacia la izquierda
48         else {
49             // Guardamos el primer carácter que será el último en moverse
50             char u = s[0];
51             for(int i = 0; i < (sTamanio-1); i++) s[i] = s[i+1];
52             s[sTamanio-1] = u;
53         }
54     }
55
56     // Insertamos la transformación en la lista
57     lTransformaciones.insertarUltimo(s);
58 }
59

```

abr 24, 13 15:47

rrevert.h

Page 1/1

```

1  /* *****
2  * *****
3  * Clase RREVERT
4  * .....
5  * La regla revierte los efectos de las modificaciones de las últimas i reglas.
6  *
7  * *****
8  * *****/
9
10
11 #ifndef RREVERT_H
12 #define RREVERT_H
13
14 #include "regla.h"
15
16
17
18
19 class RRevert:public Regla {
20 private:
21
22     int i;        // Cantidad de modificaciones de reglas a revertir
23
24 public:
25
26     // Constructor
27     explicit RRevert(int i);
28
29     // Destructor
30     ~RRevert();
31
32     // Aplica la regla sobre una lista de transformaciones
33     virtual void aplicar(ListaRef< std::string >& lTransformaciones);
34 };
35
36
37 #endif

```

abr 24, 13 15:47

rrevert.cpp

Page 1/1

```

1  /* *****
2  * *****
3  * Clase RPRINT
4  * .....
5  * La regla transmite el resultado de la transformación mas reciente.
6  *
7  * *****
8  * *****/
9
10
11 #include "rrevert.h"
12
13
14
15 // Constructor
16 RRevert::RRevert(int i) {
17     this->i = i;
18 }
19
20
21 // Destructor
22 RRevert::~RRevert() { }
23
24
25 // Aplica la regla sobre una lista de transformaciones
26 // POST: Si la lista contiene menos cantidad de transformaciones que las que se
27 // desean revertir no se revierte nada, queda como estaba la lista.
28 void RRevert::aplicar(ListaRef< std::string >& lTransformaciones) {
29     // Calculamos la posición a la que hay que revertir
30     int pos = lTransformaciones.tamano() - this->i - 1;
31
32     // Verificamos si hay elementos y que la cantidad a revertir sea válida
33     if(lTransformaciones.estaVacia() ∨ pos < 0) return;
34
35     // Insertamos la transformación en la lista
36     lTransformaciones.insertarUltimo(lTransformaciones[pos]);
37 }

```

abr 24, 13 15:47

rrepeat.h

Page 1/1

```

1  /* *****
2  * *****
3  * Clase RREPEAT
4  * .....
5  * La regla copia el substring delimitado por 'n' y 'm', lo repite 'r' veces
6  * y lo inserta en la posición 'i'.
7  * Si 'i' es positivo, se insertará el substring antes del carácter en la
8  * posición 'i'. Si es negativo, se insertará después del carácter en la
9  * posición 'i'.
10 * Si 'i' es positivo y excede las posiciones válidas de la palabra, se
11 * insertará al final, mientras que, si es negativo, se insertará al
12 * principio.
13 *
14 * *****
15 * *****/
16
17
18 #ifndef RREPEAT_H
19 #define RREPEAT_H
20
21
22 #include "regla.h"
23
24
25
26 class RRepeat:public Regla {
27 private:
28
29     int n;    // Posición inicial del substring
30     int m;    // Posición final del substring
31     int r;    // Cantidad de repeticiones del substring
32     int i;    // Posición en donde se debe insertar los substrings
33
34 public:
35
36     // Constructor
37     RRepeat(int n, int m, int r, int i);
38
39     // Destructor
40     ~RRepeat();
41
42     // Aplica la regla sobre una lista de transformaciones
43     virtual void aplicar(ListaRef< std::string >& lTransformaciones);
44 };
45
46
47 #endif

```

abr 24, 13 15:47

rrepeat.cpp

Page 1/2

```

1  /* *****
2  * *****
3  * Clase RREPEAT
4  * .....
5  * La regla copia el substring delimitado por 'n' y 'm', lo repite 'r' veces
6  * y lo inserta en la posición 'i'.
7  * Si 'i' es positivo, se insertará el substring antes del carácter en la
8  * posición 'i'. Si es negativo, se insertará después del carácter en la
9  * posición 'i'.
10 * Si 'i' es positivo y excede las posiciones válidas de la palabra, se
11 * insertará al final, mientras que, si es negativo, se insertará al
12 * principio.
13 *
14 * *****
15 * *****/
16
17
18 #include "rrepeat.h"
19
20
21
22 // Constructor
23 RRepeat::RRepeat(int n, int m, int r, int i) {
24     this->n = n;
25     this->m = m;
26     this->r = r;
27     this->i = i;
28 }
29
30
31 // Destructor
32 RRepeat::~RRepeat() { }
33
34
35 // Aplica la regla sobre una lista de transformaciones
36 void RRepeat::aplicar(ListaRef< std::string >& lTransformaciones) {
37     // Tomamos la palabra sobre la cual debemos aplicar
38     std::string s = lTransformaciones.verUltimo();
39     int sTamanio = s.size();
40
41     // Verificamos si estamos dentro del rango de la palabra
42     if(estaFueraDeRango(sTamanio, this->n, this->m)) return;
43
44     // Convertimos posiciones en posiciones válidas respecto al string
45     int nn = convertirEnPosicionValida(sTamanio, this->n);
46     int mm = convertirEnPosicionValida(sTamanio, this->m);
47     int ii = convertirEnPosicionValida(sTamanio, this->i);
48
49     // Tomamos el substring y generamos el string repetido a insertar
50     std::string sToRepeat = s.substr(nn, mm-nn+1);
51     std::string sToAppend = sToRepeat;
52     for(int i = 0; ++i < this->r; sToAppend.append(sToRepeat));
53
54     // Si 'i' es positivo, se insertará antes del caracter en la posición 'i',
55     // a menos que esté fuera de rango, en cuyo caso, se posicionará al final
56     // y se insertará antes del caracter de dicha posición final.
57     // Si 'i' es negativo, se insertará después del caracter en la posición
58     // 'i', a menos que esté fuera de rango, en cuyo caso, se posicionará al
59     // inicio y se insertará después del caracter de dicha posición inicial.
60     if((estaFueraDeRango(sTamanio, this->i) ^ (this->i < 0)) ∨ (this->i < 0))
61         ++ii;
62
63     // Insertamos repetición en string
64     s.insert(ii, sToAppend);
65
66     // Insertamos la transformación en la lista

```

abr 24, 13 15:47

rrepeat.cpp

Page 2/2

```

67     lTransformaciones.insertarUltimo(s);
68 }

```

abr 24, 13 15:47

rprint.h

Page 1/1

```

1  /* *****
2  * *****
3  * Clase RPRINT
4  * .....
5  * La regla transmite el resultado de la transformación mas reciente.
6  * .....
7  * *****
8  * *****/
9
10
11 #ifndef RPRINT_H
12 #define RPRINT_H
13
14
15 #include "regla.h"
16 #include "transmisor.h"
17
18
19
20 class RPrint:public Regla {
21 private:
22
23     Transmisor *tx;    // Transmisor del dato
24
25 public:
26
27     // Constructor
28     explicit RPrint(Transmisor *tx);
29
30     // Destructor
31     ~RPrint();
32
33     // Aplica la regla sobre una lista de transformaciones. La regla emite la
34     // última modificación realizada.
35     virtual void aplicar(ListaRef< std::string >& lTransformaciones);
36 };
37
38
39 #endif

```

abr 24, 13 15:47

rprint.cpp

Page 1/1

```

1  /* *****
2  * *****
3  * Clase RPRINT
4  * .....
5  * La regla transmite el resultado de la transformación mas reciente.
6  *
7  * *****
8  * *****/
9
10
11 #include "rprint.h"
12
13
14
15 // Constructor
16 RPrint::RPrint(Transmisor *tx) : tx(tx) { }
17
18
19 // Destructor
20 RPrint::~RPrint() { }
21
22
23 // Aplica la regla sobre una lista de transformaciones. La regla emite la
24 // última modificación realizada.
25 void RPrint::aplicar(ListaRef< std::string >& lTransformaciones) {
26     // Insertamos una copia del primer elemento de la lista para poder
27     // considerar a print como una modificación
28     lTransformaciones.insertarUltimo(lTransformaciones.verUltimo());
29     tx->transmitir(lTransformaciones.verUltimo());
30 }

```

abr 24, 13 15:47

rlowercase.h

Page 1/1

```

1  /* *****
2  * *****
3  * Clase RLOWERCASE
4  * .....
5  * La regla transforma los caracteres que están en la posición 'n' hasta la
6  * posición 'm' (inclusive) a minúsculas.
7  *
8  * *****
9  * *****/
10
11
12 #ifndef RLOWERCASE_H
13 #define RLOWERCASE_H
14
15
16 #include "regla.h"
17
18
19
20 class RLowercase:public Regla {
21 private:
22     int n;    // Posición inicial de transformación
23     int m;    // Posición final de transformación
24
25 public:
26
27     // Constructor
28     RLowercase(int n, int m);
29
30     // Destructor
31     ~RLowercase();
32
33     // Aplica la regla sobre una lista de transformaciones
34     virtual void aplicar(ListaRef< std::string >& lTransformaciones);
35 };
36
37
38 #endif

```

abr 24, 13 15:47

rlowercase.cpp

Page 1/1

```

1  /* *****
2  * *****
3  * Clase RLOWERCASE
4  * .....
5  * La regla transforma los caracteres que están en la posición 'n' hasta la
6  * posición 'm' (inclusive) a minúsculas.
7  * .....
8  * *****
9  * *****/
10
11
12 #include "rlowercase.h"
13
14
15
16 // Constructor
17 RLowercase::RLowercase(int n, int m) : n(n), m(m) { }
18
19
20 // Destructor
21 RLowercase::~RLowercase() { }
22
23
24 // Aplica la regla sobre una lista de transformaciones
25 void RLowercase::aplicar(ListaRef< std::string >& lTransformaciones) {
26     // Tomamos la palabra sobre la cual debemos aplicar
27     std::string s = lTransformaciones.verUltimo();
28     int sTamaño = s.size();
29
30     // Verificamos si estamos dentro del rango de la palabra
31     if(estaFueraDeRango(sTamaño, this->n, this->m)) return;
32
33     // Convertimos posiciones en posiciones validas respecto al string
34     int nn = convertirEnPosicionValida(sTamaño, this->n);
35     int mm = convertirEnPosicionValida(sTamaño, this->m);
36
37     // Transformamos en minúscula las letras del conjunto
38     for(int i = nn; i ≤ mm; i++)
39         s[i] = tolower(s[i]);
40
41     // Insertamos la transformación en la lista
42     lTransformaciones.insertarUltimo(s);
43 }

```

abr 24, 13 15:47

rinsert.h

Page 1/1

```

1  /* *****
2  * *****
3  * Clase RINSERT
4  * .....
5  * La regla agrega el string 'mmm' en la posición 'i' de una palabra.
6  * Si 'i' es positivo, se insertará el substring antes del carácter en la
7  * posición 'i'. Si es negativo, se insertará después del carácter en la
8  * posición 'i'.
9  * Si 'i' es positivo y excede las posiciones válidas de la palabra, se
10 * insertará al final, mientras que, si es negativo, se insertará al
11 * principio.
12 * .....
13 * *****
14 * *****/
15
16
17 #ifndef RINSERT_H
18 #define RINSERT_H
19
20
21 #include "regla.h"
22
23
24
25 class RInsert:public Regla {
26 private:
27
28     int i;           // Posición donde se debe insertar el string
29     std::string mmm; // String a agregar
30
31 public:
32
33     // Constructor
34     RInsert(int i, const std::string& mmm);
35
36     // Destructor
37     ~RInsert();
38
39     // Aplica la regla sobre una lista de transformaciones
40     virtual void aplicar(ListaRef< std::string >& lTransformaciones);
41 };
42
43
44 #endif

```

abr 24, 13 15:47

rinsert.cpp

Page 1/1

```

1  /* *****
2  * *****
3  * Clase RINSERT
4  * .....
5  * La regla agrega el string 'mmm' en la posición 'i' de una palabra.
6  * Si 'i' es positivo, se insertará el substring antes del carácter en la
7  * posición 'i'. Si es negativo, se insertará después del carácter en la
8  * posición 'i'.
9  * Si 'i' es positivo y excede las posiciones válidas de la palabra, se
10 * insertará al final, mientras que, si es negativo, se insertará al
11 * principio.
12 *
13 * *****
14 * *****/
15
16 #include "rinsert.h"
17
18
19
20
21 // Constructor
22 RInsert::RInsert(int i, const std::string& mmm) {
23     this->i = i;
24     this->mmm = mmm;
25 }
26
27
28 // Destructor
29 RInsert::~RInsert() { }
30
31
32 // Aplica la regla sobre una lista de transformaciones
33 void RInsert::aplicar(ListaRef< std::string >& lTransformaciones) {
34     // Tomamos la palabra sobre la cual debemos aplicar
35     std::string s = lTransformaciones.verUltimo();
36     int sTamanio = s.size();
37
38     // Convertimos posiciones en posiciones válidas respecto al string
39     int ii = convertirEnPosicionValida(sTamanio, this->i);
40
41     // Si 'i' es positivo, se insertará antes del caracter en la posición 'i',
42     // a menos que esté fuera de rango, en cuyo caso, se posicionará al final
43     // y se insertará antes del caracter de dicha posición final.
44     // Si 'i' es negativo, se insertará después del caracter en la posición
45     // 'i', a menos que esté fuera de rango, en cuyo caso, se posicionará al
46     // inicio y se insertará después del caracter de dicha posición inicial.
47     if((estaFueraDeRango(sTamanio, this->i) ^ (this->i < 0)) ^ (this->i < 0))
48         ++ii;
49
50     // Insertamos repetición en string
51     s.insert(ii, this->mmm);
52
53     // Insertamos la transformación en la lista
54     lTransformaciones.insertarUltimo(s);
55 }

```

abr 24, 13 15:47

regla.h

Page 1/1

```

1  /* *****
2  * *****
3  * Clase REGLA
4  * *****
5  * *****/
6
7
8  #ifndef REGLA_H
9  #define REGLA_H
10
11 #include <string>
12 #include "listaref.h"
13
14
15
16 class Regla {
17 public:
18
19     // Destructor
20     virtual ~Regla();
21
22     // Aplica la regla sobre una lista de transformaciones.
23     virtual void aplicar(ListaRef< std::string >& lTransformaciones) = 0;
24
25 protected:
26
27     // Convierte una posición de entrada en una posición válida de un string
28     // PRE: 'sTamanio' es el tamaño del string al que se refiere; 'pos' es la
29     // posición del string.
30     // POST: si 'pos' es negativa, se convierte en una posición válida
31     // positiva, considerando que en caso de desborde se devolverá la primer
32     // posición del string; si 'pos' es positiva y supera el tamaño del string,
33     // se devuelve la última posición de este.
34     int convertirEnPosicionValida(int sTamanio, int pos);
35
36     // Verifica si existe intersección entre el intervalo sobre el cual se
37     // quiere aplicar la arregla y el tamaño real de la palabra.
38     // PRE: 'sTamanio' es el tamaño de la palabra; 'n' es la posición inicial
39     // del substring; 'm' es la posición final del substring. 'n' debe ser
40     // menor o igual que 'm'.
41     // POST: devuelve true si el intervalo se encuentra dentro del rango o
42     // false en su defecto.
43     bool estaFueraDeRango(int sTamanio, int n, int m);
44
45     // Verifica si la posición 'i' se encuentra fuera de rango en referencia a
46     // la palabra.
47     // PRE: 'sTamanio' es el tamaño de la palabra; 'i' es una posición.
48     // POST: devuelve true si la posición es una posición dentro de la cadena o
49     // false en su defecto.
50     bool estaFueraDeRango(int sTamanio, int i);
51 };
52
53 #endif

```

abr 24, 13 15:47

regla.cpp

Page 1/1

```

1  /* *****
2  * *****
3  * Clase REGLA
4  * *****
5  * *****/
6
7
8  #include "regla.h"
9
10
11 // Destructor
12 Regla::~Regla() { }
13
14
15 // Convierte una posición de entrada en una posición válida de un string
16 // PRE: 'sTamaño' es el tamaño del string al que se refiere; 'pos' es la
17 // posición del string.
18 // POST: si 'pos' es negativa, se convierte en una posición válida
19 // positiva, considerando que en caso de desborde se devolverá la primer
20 // posición del string; si 'pos' es positiva y supera el tamaño del string,
21 // se devuelve la última posición de este.
22 int Regla::convertirEnPosicionValida(int sTamaño, int pos) {
23     if(pos < 0) {
24         int posValida = sTamaño + pos;
25         if(posValida < 0) return 0;
26         return posValida;
27     }
28     else if (pos > sTamaño - 1) return (sTamaño - 1);
29
30     return pos;
31 }
32
33
34 // Verifica si existe intersección entre el intervalo sobre el cual se
35 // quiere aplicar la arregla y el tamaño real de la palabra.
36 // PRE: 'sTamaño' es el tamaño de la palabra; 'n' es la posición inicial
37 // del substring; 'm' es la posición final del substring. 'n' debe ser
38 // menor o igual que 'm'.
39 // POST: devuelve true si el intervalo se encuentra dentro del rango o
40 // false en su defecto.
41 bool Regla::estaFueraDeRango(int sTamaño, int n, int m) {
42     // Caso en que ambos son negativos y están fuera de rango
43     if((n < 0) ^ (m < 0) ^ ((n+sTamaño) < 0) ^ ((m+sTamaño) < 0))
44         return true;
45     // Caso en que ambos son positivos y están fuera de rango
46     else if ((n > 0) ^ (m > 0) ^ (n > sTamaño-1) ^ (m > sTamaño-1))
47         return true;
48
49     return false;
50 }
51
52
53 // Verifica si la posición 'i' se encuentra fuera de rango en referencia a
54 // la palabra.
55 // PRE: 'sTamaño' es el tamaño de la palabra; 'i' es una posición.
56 // POST: devuelve true si la posición es una posición dentro de la cadena o
57 // false en su defecto.
58 bool Regla::estaFueraDeRango(int sTamaño, int i) {
59     // Caso en que es negativo y está fuera de rango
60     if((i < 0) ^ ((i+sTamaño) < 0)) return true;
61     // Caso en que es positivo y está fuera de rango
62     else if ((i > 0) ^ (i > sTamaño-1)) return true;
63
64     return false;
65 }

```

abr 24, 13 15:47

receptor.h

Page 1/1

```

1  /* *****
2  * *****
3  * Clase RECEPTOR
4  * *****
5  * *****/
6
7
8  #ifndef RECEPTOR_H
9  #define RECEPTOR_H
10
11 #include <string>
12
13
14
15 class Receptor {
16 private:
17
18     // Estado del receptor
19     bool activo;
20
21 protected:
22
23     // Activa el receptor
24     void activar();
25
26     // Desactiva el receptor
27     void desactivar();
28
29 public:
30
31     // Destructor
32     virtual ~Receptor() { };
33
34     // Verifica si el receptor esta activo.
35     // POST: devuelve true si está activo o false en caso contrario
36     bool estaActivo();
37
38     // Se ejecuta la recepción de un dato desde la entrada estándar.
39     // POST: se devuelve la palabra recibida. Debe considerarse terminada la
40     // recepción de palabras cuando se recibe una cadena vacía, es decir, nada.
41     virtual std::string recibir() = 0;
42 };
43
44
45 #endif

```



abr 24, 13 15:47

## receptor.cpp

Page 1/1

```

1  /* *****
2  * *****
3  * Clase RECEPTOR
4  * *****
5  * *****/
6
7
8  #include "receptor.h"
9
10
11 // Verifica si el receptor esta activo.
12 // POST: devuelve true si está activo o false en caso contrario
13 bool Receptor::estaActivo() {
14     return this->activo;
15 }
16
17
18 // Activa el receptor
19 void Receptor::activar() {
20     this->activo = true;
21 }
22
23
24 // Desactiva el receptor
25 void Receptor::desactivar() {
26     this->activo = false;
27 }

```

abr 24, 13 15:47

## pila.h

Page 1/3

```

1  /* *****
2  * *****
3  * Clase PILA
4  * .....
5  * Implementación de la clase Pila.
6  * *****
7  * *****/
8
9
10
11 #ifndef PILA_H
12 #define PILA_H
13
14
15
16 /* *****
17 * DECLARACIÓN DE LA CLASE
18 * *****/
19
20
21 template < typename Tipo >
22 class Pila {
23 private:
24
25     struct Nodo {
26         Tipo dato;           // Dato al que referencia el nodo
27         Nodo *siguiente;     // Puntero al siguiente nodo
28
29         // Constructor
30         explicit Nodo(Tipo& dato) : dato(dato), siguiente(0) { }
31     };
32
33     int cantElementos;       // Número de elementos en la pila
34     Nodo *primero;          // Puntero al primer elemento de la pila
35     Nodo *ultimo;           // Puntero al último elemento de la pila
36
37 public:
38
39     // Constructor
40     Pila();
41
42     // Destructor
43     ~Pila();
44
45     // Verifica si la pila tiene o no elementos.
46     // POST: devuelve true si la pila esta vacía o false en su defecto.
47     bool estaVacía();
48
49     // Agrega un nuevo elemento a la pila.
50     // PRE: 'dato' es el dato que se desea apilar.
51     // POST: se agregó el nuevo elemento, el cual se encuentra en el tope de
52     // la pila.
53     void apilar(Tipo& dato);
54
55
56     // Saca el primer elemento de la pila.
57     // POST: si no hay elementos que desapilar no hace nada.
58     void desapilar();
59
60     // Obtiene el valor del primer elemento de la pila.
61     // POST: se devuelve el dato que se encuentra en el tope de la pila
62     Tipo& verTope();
63 };
64
65
66

```

abr 24, 13 15:47

pila.h

Page 2/3

```

67
68 /* *****
69 * DEFINICIÓN DE LA CLASE
70 * *****/
71
72
73 // Constructor
74 template <typename Tipo >
75 Pila< Tipo >::Pila() {
76     this->cantElementos = 0;
77     this->primero = 0;
78     this->ultimo = 0;
79 }
80
81
82 // Destructor
83 template <typename Tipo >
84 Pila< Tipo >::~~Pila() {
85     Nodo *nodo_actual, *nodo;
86
87     // Si hay elementos, comenzamos la eliminación
88     if(this->cantElementos != 0) {
89         nodo_actual = this->primero;
90
91         // Eliminación de los nodos uno a uno
92         while(nodo_actual->siguiente) {
93             nodo = nodo_actual->siguiente;
94             delete nodo_actual;
95             nodo_actual = nodo;
96         }
97
98         delete nodo_actual;
99     }
100 }
101
102
103 // Verifica si la pila tiene o no elementos.
104 // POST: devuelve true si la pila esta vacía o false en su defecto.
105 template <typename Tipo >
106 bool Pila< Tipo >::estaVacía() {
107     return (this->cantElementos == 0);
108 }
109
110
111 // Agrega un nuevo elemento a la pila.
112 // PRE: 'dato' es el dato que se desea apilar.
113 // POST: se agregó el nuevo elemento, el cual se encuentra en el tope de
114 // la pila.
115 template <typename Tipo >
116 void Pila< Tipo >::apilar(Tipo& dato) {
117     // Creamos un nuevo nodo
118     Nodo *nodo = new Nodo(dato);
119
120     // Enlazamos el nodo en el tope de la pila.
121     if(this->primero)
122         nodo->siguiente = this->primero;
123     else
124         this->ultimo = nodo;
125
126     this->primero = nodo;
127     this->cantElementos++;
128 }
129
130
131 // Saca el primer elemento de la pila.
132 // POST: si no hay elementos que desapilar no hace nada.

```

abr 24, 13 15:47

pila.h

Page 3/3

```

133 template <typename Tipo >
134 void Pila< Tipo >::desapilar() {
135     // Corroboramos que hayan elementos en la pila
136     if(this->cantElementos == 0) return;
137
138     // Tomamos dato del nodo
139     Nodo *nodo = this->primero;
140
141     // Desenlazamos nodo y liberamos memoria.
142     if(nodo->siguiente)
143         this->primero = nodo->siguiente;
144     else
145         this->primero = this->ultimo = 0;
146
147     delete nodo;
148     this->cantElementos--;
149 }
150
151
152 // Obtiene el valor del primer elemento de la pila.
153 // POST: se devuelve el dato que se encuentra en el tope de la pila.
154 template <typename Tipo >
155 Tipo& Pila< Tipo >::verTope() {
156     return (this->primero->dato);
157 }
158
159
160 #endif

```

abr 24, 13 15:47

## parser\_salida.h

Page 1/1

```

1  /* *****
2  * *****
3  * Clase PARSEERSALIDA
4  * *****
5  * *****/
6
7
8  #ifndef PARSER_SALIDA_H
9  #define PARSER_SALIDA_H
10
11 #include <string>
12 #include "transmisor.h"
13
14
15 class ParserSalida {
16 public:
17
18     // Parsea la especificación del tipo de salida que se utilizará.
19     // PRE: 'tipoSalida' es el tipo de salida que se desea usar. Debe
20     // pasársele el nombre de archivo en caso de desear utilizar un archivo de
21     // palabras alteradas o el caracter "-" si se desea enviar las palabras
22     // a través de la salida estándar.
23     // POST: se devuelve un puntero a un Transmisor.
24     Transmisor* parsear(const std::string& tipoSalida);
25 };
26
27
28 #endif

```

abr 24, 13 15:47

## parser\_salida.cpp

Page 1/1

```

1  /* *****
2  * *****
3  * Clase PARSEERENTRADA
4  * *****
5  * *****/
6
7
8  #include "parser_salida.h"
9  #include "tx_archivo.h"
10 #include "tx_salida_estandar.h"
11
12
13
14 // Parsea la especificación del tipo de salida que se utilizará.
15 // PRE: 'tipoSalida' es el tipo de salida que se desea usar. Debe
16 // pasársele el nombre de archivo en caso de desear utilizar un archivo de
17 // palabras alteradas o el caracter "-" si se desea enviar las palabras
18 // a través de la salida estándar.
19 // POST: se devuelve un puntero a un Transmisor.
20 Transmisor* ParserSalida::parsear(const std::string& tipoSalida) {
21     // La emisión será a través de la salida estándar
22     if(tipoSalida == "-")
23         return new TxSalidaEstandar();
24     // La emisión se hará a través de un archivo de de salida
25     return new TxArchivo(tipoSalida);
26 }

```

abr 24, 13 15:47

parser\_reglas.h

Page 1/1

```

1  /* *****
2  * *****
3  * Clase PARSEERREGLAS
4  * *****
5  * *****/
6
7
8  #ifndef PARSEERREGLAS_H
9  #define PARSEERREGLAS_H
10
11 #include <fstream>
12 #include "lista.h"
13 #include "regla.h"
14 #include "transmisor.h"
15
16
17
18 class ParserReglas {
19 public:
20
21     // Parsea el archivo de reglas.
22     // PRE: 'archivo' es el nombre de archivo que contiene las reglas; 'tx' es
23     // un puntero a un Transmisor el cual puede ser usado por alguna regla
24     // para emitir palabras.
25     // POST: se devuelve una referencia a una lista que contiene, en orden
26     // de aparición en el archivo, los objetos que son Regla. La lista se
27     // encontrará vacía de producirse un error en la apertura del archivo o si
28     // este último no contenía instrucciones válidas, debiéndose considerarse como
29     // erróneo.
30     Lista< Regla > parsear(const std::string& nombre_archivo, Transmisor* tx);
31 };
32
33 #endif

```

abr 24, 13 15:47

parser\_reglas.cpp

Page 1/2

```

1  /* *****
2  * *****
3  * Clase PARSEERREGLAS
4  * *****
5  * *****/
6
7
8  #include "parser_reglas.h"
9
10 // Reglas
11 #include "uppercase.h"
12 #include "lowercase.h"
13 #include "repeat.h"
14 #include "rotate.h"
15 #include "insert.h"
16 #include "revert.h"
17 #include "print.h"
18
19
20 namespace {
21     // Constantes para los nombres de instrucciones
22     const std::string S_UPPERCASE = "uppercase";
23     const std::string S_LOWERCASE = "lowercase";
24     const std::string S_REPEAT = "repeat";
25     const std::string S_ROTATE = "rotate";
26     const std::string S_INSERT = "insert";
27     const std::string S_REVERT = "revert";
28     const std::string S_PRINT = "print";
29
30     // Constante para caracter de fin de instrucción
31     const std::string S_FIN_INSTRUCCION = ";";
32 }
33
34
35
36
37 /* *****
38 * DEFINICIONES DE MÉTODOS DE LA CLASE
39 * *****/
40
41
42 // Parsea el archivo de reglas.
43 // PRE: 'archivo' es el nombre de archivo que contiene las reglas; 'tx' es
44 // un puntero a un Transmisor el cual puede ser usado por alguna regla
45 // para emitir palabras.
46 // POST: se devuelve una referencia a una lista que contiene, en orden
47 // de aparición en el archivo, los objetos que son Regla. La lista se
48 // encontrará vacía de producirse un error en la apertura del archivo o si
49 // este último no contenía instrucciones válidas, debiéndose considerarse como
50 // erróneo.
51 Lista< Regla > ParserReglas::parsear(const std::string& nombre_archivo,
52 Transmisor *tx) {
53     // Creamos la lista de reglas
54     Lista< Regla > lReglas;
55
56     // Abrimos archivo
57     std::ifstream archivo;
58     archivo.open(nombre_archivo.c_str());
59
60     // Verificamos que se halla abierto correctamente
61     if(!archivo.is_open()) return lReglas;
62
63
64     // Variables auxiliares para parseo
65     std::string instruccion, mmm, fin;
66     int n, m, r, i;

```

abr 24, 13 15:47

parser\_reglas.cpp

Page 2/2

```

67 // Procesamos cada instrucción. Si alguna no esta definida, no se
68 // considera y se sigue procesando las restantes.
69 while(archivo >> instruccion) {
70     // Filtros
71     if(instruccion == S_UPPERCASE) {
72         archivo >> n >> m >> fin;
73         lReglas.insertarUltimo(new RUppercase(n, m));
74     }
75     else if (instruccion == S_LOWERCASE) {
76         archivo >> n >> m >> fin;
77         lReglas.insertarUltimo(new RLowercase(n, m));
78     }
79     else if (instruccion == S_REPEAT) {
80         archivo >> n >> m >> r >> i >> fin;
81         lReglas.insertarUltimo(new RRepeat(n, m, r, i));
82     }
83     else if (instruccion == S_ROTATE) {
84         archivo >> n >> fin;
85         lReglas.insertarUltimo(new RRotate(n));
86     }
87     else if (instruccion == S_INSERT) {
88         archivo >> i >> mmm >> fin;
89         lReglas.insertarUltimo(new RInsert(i, mmm));
90     }
91     else if (instruccion == S_REVERT) {
92         archivo >> i >> fin;
93         lReglas.insertarUltimo(new RRevert(i));
94     }
95     else if (instruccion == S_PRINT) {
96         archivo >> fin;
97         lReglas.insertarUltimo(new RPrint(tx));
98     }
99     // Caso en que no matchea con ninguna instrucción. Lo consideramos un
100     // error.
101     else {
102         // Vaciamos la lista y la devolvemos vacía
103         while(!lReglas.estaVacia())
104             delete lReglas.eliminarPrimero();
105         break;
106     }
107 }
108
109 // Cerramos el archivo
110 archivo.close();
111
112 return lReglas;
113 }

```

abr 24, 13 15:47

parser\_entrada.h

Page 1/1

```

1  /* *****
2  * *****
3  * Clase PARSEENTRADA
4  * *****
5  * *****/
6
7
8  #ifndef PARSER_ENTRADA_H
9  #define PARSER_ENTRADA_H
10
11 #include <string>
12 #include <iostream>
13 #include "receptor.h"
14
15
16 class ParserEntrada {
17 public:
18
19     // Parsea la especificación del tipo de entrada se utilizará.
20     // PRE: 'tipoEntrada' es el tipo de entrada que se desea usar. Debe
21     // pasársele el nombre de archivo en caso de desear utilizar un archivo de
22     // palabras iniciales o el caracter "-" si se desea recibir las palabras
23     // a través de la entrada estándar.
24     // POST: se devuelve un puntero a un Receptor.
25     Receptor* parsear(const std::string& tipoEntrada);
26 };
27
28
29
30 #endif

```

abr 24, 13 15:47

## parser\_entrada.cpp

Page 1/1

```

1  /* *****
2  * *****
3  * Clase PARSEERENTRADA
4  * *****
5  * *****/
6
7
8  #include "parser_entrada.h"
9  #include "rx_archivo.h"
10 #include "rx_entrada_estandar.h"
11
12
13
14 // Parsea la especificación del tipo de entrada se utilizará.
15 // PRE: 'tipoEntrada' es el tipo de entrada que se desea usar. Debe
16 // pasársele el nombre de archivo en caso de desear utilizar un archivo de
17 // palabras iniciales o el caracter "-" si se desea recibir las palabras
18 // a través de la entrada estándar.
19 // POST: se devuelve un puntero a un Receptor.
20 Receptor* ParserEntrada::parsear(const std::string& tipoEntrada) {
21     // El ingreso será a través de la entrada estándar
22     if(tipoEntrada == "-")
23         return new RxEntradaEstandar();
24
25     // El ingreso se hará a través de un archivo de entrada
26     return new RxArchivo(tipoEntrada);
27 }

```

abr 24, 13 15:47

## listaref.h

Page 1/4

```

1  /* *****
2  * *****
3  * Clase LISTAREF
4  * .....
5  * Implementación de la clase ListaRef la cual modela una lista de referencias.
6  * .....
7  * *****
8  * *****/
9
10
11
12 #ifndef LISTAREF_H
13 #define LISTAREF_H
14
15
16 /* *****
17 * DECLARACIÓN DE LA CLASE
18 * *****/
19
20
21 template < typename Tipo >
22 class ListaRef {
23 private:
24
25     struct Nodo {
26         Tipo dato;           // Dato al que referencia el nodo
27         Nodo *siguiente;     // Puntero al siguiente nodo
28
29         // Constructor
30         explicit Nodo(Tipo& dato) : dato(dato), siguiente(0) { }
31     };
32
33     Nodo *primero;           // Puntero al primer elemento de la lista
34     Nodo *ultimo;           // Puntero al último elemento de la lista
35     int largo;               // Tamaño que representa la cantidad de
36                             // elementos que contiene la lista
37
38 public:
39
40     // Constructor
41     ListaRef();
42
43     // Destructor
44     ~ListaRef();
45
46     // Verifica si una lista se encuentra vacía.
47     // POST: Devuelve verdadero si la lista se encuentra vacía o falso en
48     // caso contrario.
49     bool estaVacía();
50
51     // Devuelve el tamaño actual de la lista.
52     int tamaño();
53
54     // Inserta un elemento al principio de la lista.
55     // PRE: 'dato' es el dato a insertar.
56     void insertarPrimero(Tipo& dato);
57
58     // Inserta un elemento en el último lugar de la lista.
59     // PRE: 'dato' es el dato a insertar.
60     void insertarUltimo(Tipo& dato);
61
62     // Obtiene el valor del primer elemento de la lista.
63     // POST: se devuelve el dato que se encuentra primero en la lista.
64     Tipo& verPrimero();
65
66     // Obtiene el valor del último elemento de la lista.

```

abr 24, 13 15:47

listaref.h

Page 2/4

```

67 // POST: se devuelve el dato que se encuentra último en la lista.
68 Tipo& verUltimo();
69
70 // Elimina el primer elemento de la lista.
71 // POST: se retorna el elemento eliminado de la lista.
72 void eliminarPrimero();
73
74 // Operador []
75 // Permite acceder a los índices de la lista mediante la notación
76 // lista[i], donde i es un número entero comprendido entre [0, n-1],
77 // siendo n el tamaño de la lista.
78 Tipo& operator[] (const int indice);
79 };
80
81
82
83
84 /* *****
85 * DEFINICIÓN DE LA CLASE
86 * *****/
87
88 // Constructor
89 template <typename Tipo >
90 ListaRef< Tipo >::ListaRef() {
91     this->primero = 0;
92     this->ultimo = 0;
93     this->largo = 0;
94 }
95
96 // Destructor
97 template <typename Tipo >
98 ListaRef< Tipo >::~~ListaRef() {
99     Nodo *nodo;
100
101     // Recorremos los nodos y los destruimos
102     while(this->primero) {
103         nodo = this->primero;
104         this->primero = nodo->siguiente;
105         delete nodo;
106     }
107 }
108
109 // Verifica si una lista se encuentra vacía.
110 // POST: Devuelve verdadero si la lista se encuentra vacía o falso en
111 // caso contrario.
112 template <typename Tipo >
113 bool ListaRef< Tipo >::estaVacía() {
114     return (this->largo == 0);
115 }
116
117 // Devuelve el tamaño actual de la lista.
118 template <typename Tipo >
119 int ListaRef< Tipo >::tamano() {
120     return this->largo;
121 }
122
123 // Inserta un elemento al principio de la lista.
124 // PRE: 'dato' es el dato a insertar.
125 template <typename Tipo >
126 void ListaRef< Tipo >::insertarPrimero(Tipo& dato) {
127     // Creamos un nuevo nodo

```

abr 24, 13 15:47

listaref.h

Page 3/4

```

133     Nodo *nodo = new Nodo(dato);
134
135 // Seteamos los campos del nodo
136 nodo->dato = dato;
137 nodo->siguiente = this->primero;
138 this->primero = nodo;
139
140 // Si no hay ningún elemento, el primero también es el último
141 if(!this->largo) this->ultimo = nodo;
142 this->largo++;
143 }
144
145 // Inserta un elemento en el último lugar de la lista.
146 // PRE: 'dato' es el dato a insertar.
147 template <typename Tipo >
148 void ListaRef< Tipo >::insertarUltimo(Tipo& dato) {
149     // Creamos un nuevo nodo
150     Nodo *nodo = new Nodo(dato);
151
152 // Seteamos los campos del nodo
153 nodo->dato = dato;
154 nodo->siguiente = 0;
155
156 // Si no hay elementos, el último también es el primero
157 if(!this->ultimo)
158     this->primero = nodo;
159 // Sino, insertamos el nuevo nodo a continuación del que
160 // se encontraba último
161 else
162     this->ultimo->siguiente = nodo;
163
164 this->ultimo = nodo;
165 this->largo++;
166 }
167
168 // Obtiene el valor del primer elemento de la lista.
169 // POST: se devuelve el dato que se encuentra primero en la lista.
170 template <typename Tipo >
171 Tipo& ListaRef< Tipo >::verPrimero() {
172     return (this->primero->dato);
173 }
174
175 // Obtiene el valor del último elemento de la lista.
176 // POST: se devuelve el dato que se encuentra último en la lista.
177 template <typename Tipo >
178 Tipo& ListaRef< Tipo >::verUltimo() {
179     return (this->ultimo->dato);
180 }
181
182 // Elimina el primer elemento de la lista.
183 // POST: se retorna el elemento eliminado de la lista.
184 template <typename Tipo >
185 void ListaRef< Tipo >::eliminarPrimero() {
186     // Tomamos el nodo a borrar
187     Nodo *nodo = this->primero;
188
189 // El segundo elemento pasa a ser el primero
190 this->primero = this->primero->siguiente;
191 this->largo--;
192
193 // Liberamos la memoria usada por el nodo.
194 delete nodo;

```

abr 24, 13 15:47

listaref.h

Page 4/4

```

199
200 // Verificamos si quedan mas elementos en la lista
201 if (this->largo == 0) this->ultimo = 0;
202 }
203
204
205 // Operador []
206 // Permite acceder a los índices de la lista mediante la notación lista[i],
207 // donde i es un número entero comprendido entre [0, n-1], siendo n el tamaño
208 // de la lista.
209 // POST: Si el índice se encuentra fuera de rango, se lanza una excepción.
210 template <typename Tipo >
211 Tipo& ListaRef< Tipo >::operator[] (const int indice) {
212 // Corroboramos que el índice sea válido
213 if(indice ≥ this->largo) throw std::string("ERROR: Indice fuera de rango");
214
215 int i;
216 Nodo *nodo = this->primero;
217
218 for(i = 0; i < indice; i++)
219     nodo = nodo->siguiente;
220
221 return nodo->dato;
222 }
223
224
225 #endif

```

abr 24, 13 15:47

lista.h

Page 1/4

```

1  /* *****
2  * *****
3  * Clase LISTA
4  * .....
5  * Implementación de la clase Lista.
6  *
7  * *****
8  * *****/
9
10
11
12 #ifndef LISTA_H
13 #define LISTA_H
14
15
16 /* *****
17 * DECLARACIÓN DE LA CLASE
18 * *****/
19
20
21 template < typename Tipo >
22 class Lista {
23 private:
24
25     struct Nodo {
26         Tipo *dato;        // Dato al que referencia el nodo
27         Nodo *siguiente;    // Puntero al siguiente nodo
28
29         // Constructor
30         explicit Nodo(Tipo *dato) : dato(dato), siguiente(0) { }
31     };
32
33     Nodo *primero;        // Puntero al primer elemento de la lista
34     Nodo *ultimo;        // Puntero al último elemento de la lista
35     int largo;            // Tamaño que representa la cantidad de
36                           // elementos que contiene la lista
37
38 public:
39
40     // Constructor
41     Lista();
42
43     // Destructor
44     ~Lista();
45
46     // Verifica si una lista se encuentra vacía.
47     // POST: Devuelve verdadero si la lista se encuentra vacía o falso en
48     // caso contrario.
49     bool estaVacía();
50
51     // Devuelve el tamaño actual de la lista.
52     int tamaño();
53
54     // Inserta un elemento al principio de la lista.
55     // PRE: 'dato' es el dato a insertar.
56     void insertarPrimero(Tipo *dato);
57
58     // Inserta un elemento en el último lugar de la lista.
59     // PRE: 'dato' es el dato a insertar.
60     void insertarUltimo(Tipo *dato);
61
62     // Obtiene el valor del primer elemento de la lista.
63     // POST: se devuelve el dato que se encuentra primero en la lista.
64     Tipo* verPrimero();
65
66     // Obtiene el valor del último elemento de la lista.

```



abr 24, 13 15:47

lista.h

Page 2/4

```

67 // POST: se devuelve el dato que se encuentra último en la lista.
68 Tipo* verUltimo();
69
70 // Elimina el primer elemento de la lista.
71 // POST: se retorna el elemento eliminado de la lista.
72 Tipo* eliminarPrimero();
73
74 // Operador []
75 // Permite acceder a los índices de la lista mediante la notación
76 // lista[i], donde i es un número entero comprendido entre [0, n-1],
77 // siendo n el tamaño de la lista.
78 Tipo* operator[] (const int indice);
79 };
80
81
82
83
84 /* *****
85  * DEFINICIÓN DE LA CLASE
86  * *****/
87
88 // Constructor
89 template <typename Tipo >
90 Lista< Tipo >::Lista() {
91     this->primero = 0;
92     this->ultimo = 0;
93     this->largo = 0;
94 }
95
96 // Destructor
97 template <typename Tipo >
98 Lista< Tipo >::~~Lista() {
99     Nodo *nodo;
100
101     // Recorremos los nodos y los destruimos
102     while(this->primero) {
103         nodo = this->primero;
104         this->primero = nodo->siguiente;
105         delete nodo->dato;
106         delete nodo;
107     }
108 }
109
110 // Verifica si una lista se encuentra vacía.
111 // POST: Devuelve verdadero si la lista se encuentra vacía o falso en
112 // caso contrario.
113 template <typename Tipo >
114 bool Lista< Tipo >::estaVacia() {
115     return (this->largo == 0);
116 }
117
118 // Devuelve el tamaño actual de la lista.
119 template <typename Tipo >
120 int Lista< Tipo >::tamano() {
121     return this->largo;
122 }
123
124 // Inserta un elemento al principio de la lista.
125 // PRE: 'dato' es el dato a insertar.
126 template <typename Tipo >
127 void Lista< Tipo >::insertarPrimero(Tipo *dato) {

```

abr 24, 13 15:47

lista.h

Page 3/4

```

133 // Creamos un nuevo nodo
134 Nodo *nodo = new Nodo(dato);
135
136 // Seteamos los campos del nodo
137 nodo->dato = dato;
138 nodo->siguiente = this->primero;
139 this->primero = nodo;
140
141 // Si no hay ningún elemento, el primero también es el último
142 if(!this->largo) this->ultimo = nodo;
143 this->largo++;
144 }
145
146 // Inserta un elemento en el último lugar de la lista.
147 // PRE: 'dato' es el dato a insertar.
148 template <typename Tipo >
149 void Lista< Tipo >::insertarUltimo(Tipo *dato) {
150     // Creamos un nuevo nodo
151     Nodo *nodo = new Nodo(dato);
152
153     // Seteamos los campos del nodo
154     nodo->dato = dato;
155     nodo->siguiente = 0;
156
157     // Si no hay elementos, el último también es el primero
158     if(!this->ultimo)
159         this->primero = nodo;
160     // Sino, insertamos el nuevo nodo a continuación del que
161     // se encontraba último
162     else
163         this->ultimo->siguiente = nodo;
164
165     this->ultimo = nodo;
166     this->largo++;
167 }
168
169 // Obtiene el valor del primer elemento de la lista.
170 // POST: se devuelve el dato que se encuentra primero en la lista.
171 template <typename Tipo >
172 Tipo* Lista< Tipo >::verPrimero() {
173     return (this->primero->dato);
174 }
175
176 // Obtiene el valor del último elemento de la lista.
177 // POST: se devuelve el dato que se encuentra último en la lista.
178 template <typename Tipo >
179 Tipo* Lista< Tipo >::verUltimo() {
180     return (this->ultimo->dato);
181 }
182
183 // Elimina el primer elemento de la lista.
184 // POST: se retorna el elemento eliminado de la lista.
185 template <typename Tipo >
186 Tipo* Lista< Tipo >::eliminarPrimero() {
187     // Tomamos el nodo a borrar
188     Nodo *nodo = this->primero;
189     Tipo *dato = this->primero->dato;
190
191     // El segundo elemento pasa a ser el primero
192     this->primero = this->primero->siguiente;
193     this->largo--;
194 }

```

abr 24, 13 15:47

lista.h

Page 4/4

```

199 // Liberamos la memoria usada por el nodo.
200 delete nodo;
201
202 // Verificamos si quedan mas elementos en la lista
203 if (this->largo == 0) this->ultimo = 0;
204
205 return dato;
206 }
207
208
209 // Operador []
210 // Permite acceder a los índices de la lista mediante la notación lista[i],
211 // donde i es un número entero comprendido entre [0, n-1], siendo n el tamaño
212 // de la lista.
213 template <typename Tipo >
214 Tipo* Lista< Tipo >::operator[] (const int indice) {
215     // Corroboramos que el índice sea válido
216     if(indice >= this->largo) return 0;
217
218     int i;
219     Nodo *nodo = this->primero;
220
221     for(i = 0; i < indice; i++)
222         nodo = nodo->siguiente;
223
224     return nodo->dato;
225 }
226
227
228 #endif

```

abr 24, 13 15:47

Table of Content

Page 1/1

1	<b>Table of Contents</b>			
2	1 word_mangling.h.....	sheets	1 to 1 ( 1) pages	1- 1 39 lines
3	2 word_mangling.cpp...	sheets	1 to 1 ( 1) pages	2- 2 43 lines
4	3 tx_salida_estandar.h	sheets	2 to 2 ( 1) pages	3- 3 30 lines
5	4 tx_salida_estandar.cpp	sheets	2 to 2 ( 1) pages	4- 4 27 lines
6	5 tx_archivo.h.....	sheets	3 to 3 ( 1) pages	5- 5 40 lines
7	6 tx_archivo.cpp.....	sheets	3 to 3 ( 1) pages	6- 6 37 lines
8	7 transmisor.h.....	sheets	4 to 4 ( 1) pages	7- 7 44 lines
9	8 transmisor.cpp.....	sheets	4 to 4 ( 1) pages	8- 8 28 lines
10	9 tp.cpp.....	sheets	5 to 5 ( 1) pages	9- 10 117 lines
11	10 rx_entrada_estandar.h	sheets	6 to 6 ( 1) pages	11- 11 36 lines
12	11 rx_entrada_estandar.cpp	sheets	6 to 6 ( 1) pages	12- 12 34 lines
13	12 rx_archivo.h.....	sheets	7 to 7 ( 1) pages	13- 13 42 lines
14	13 rx_archivo.cpp.....	sheets	7 to 7 ( 1) pages	14- 14 42 lines
15	14 ruppercase.h.....	sheets	8 to 8 ( 1) pages	15- 15 39 lines
16	15 ruppercase.cpp.....	sheets	8 to 8 ( 1) pages	16- 16 44 lines
17	16 rrotate.h.....	sheets	9 to 9 ( 1) pages	17- 17 40 lines
18	17 rrotate.cpp.....	sheets	9 to 9 ( 1) pages	18- 18 60 lines
19	18 rrevert.h.....	sheets	10 to 10 ( 1) pages	19- 19 38 lines
20	19 rrevert.cpp.....	sheets	10 to 10 ( 1) pages	20- 20 38 lines
21	20 rrepeat.h.....	sheets	11 to 11 ( 1) pages	21- 21 48 lines
22	21 rrepeat.cpp.....	sheets	11 to 12 ( 2) pages	22- 23 69 lines
23	22 rprint.h.....	sheets	12 to 12 ( 1) pages	24- 24 40 lines
24	23 rprint.cpp.....	sheets	13 to 13 ( 1) pages	25- 25 31 lines
25	24 rlowercase.h.....	sheets	13 to 13 ( 1) pages	26- 26 39 lines
26	25 rlowercase.cpp.....	sheets	14 to 14 ( 1) pages	27- 27 44 lines
27	26 rinsert.h.....	sheets	14 to 14 ( 1) pages	28- 28 45 lines
28	27 rinsert.cpp.....	sheets	15 to 15 ( 1) pages	29- 29 56 lines
29	28 regla.h.....	sheets	15 to 15 ( 1) pages	30- 30 55 lines
30	29 regla.cpp.....	sheets	16 to 16 ( 1) pages	31- 31 66 lines
31	30 receptor.h.....	sheets	16 to 16 ( 1) pages	32- 32 46 lines
32	31 receptor.cpp.....	sheets	17 to 17 ( 1) pages	33- 33 28 lines
33	32 pila.h.....	sheets	17 to 18 ( 2) pages	34- 36 161 lines
34	33 parser_salida.h.....	sheets	19 to 19 ( 1) pages	37- 37 29 lines
35	34 parser_salida.cpp...	sheets	19 to 19 ( 1) pages	38- 38 27 lines
36	35 parser_reglas.h.....	sheets	20 to 20 ( 1) pages	39- 39 35 lines
37	36 parser_reglas.cpp...	sheets	20 to 21 ( 2) pages	40- 41 115 lines
38	37 parser_entrada.h....	sheets	21 to 21 ( 1) pages	42- 42 31 lines
39	38 parser_entrada.cpp..	sheets	22 to 22 ( 1) pages	43- 43 28 lines
40	39 listaref.h.....	sheets	22 to 24 ( 3) pages	44- 47 226 lines
41	40 lista.h.....	sheets	24 to 26 ( 3) pages	48- 51 229 lines