abr 24, 13 15:47 **word_mangling.h** Page 1/1

```
/* ******************************************************************************
 * ******************************************************************************
 * Clase WORDMANGLING
 * ..............................................................................
 * Modela el módulo encargado de llevar a cabo el word mangling a partir de una
 * lista de reglas sobre palabras recibidas a través de un receptor.
 *
 * ******************************************************************************
 * ******************************************************************************/


#ifndef WORD_MANGLING_H
#define WORD_MANGLING_H


#include "lista.h"
#include "regla.h"
#include "receptor.h"



class WordMangling {
private:
  Lista< Regla >& lReglas;    // Lista de reglas a aplicar

public:

  // Constructor
  explicit WordMangling(Lista< Regla >& lReglas);

  // Ejecuta el alterador de palabras.
  // PRE: 'rxPalabras' es un Receptor de palabras sobre las que se
  // aplicarán las reglas.
  void ejecutar(Receptor* rxPalabras);
};


#endif
```

abr 24, 13 15:47 **word_mangling.cpp** Page 1/1

```
/* ******************************************************************************
 * ******************************************************************************
 * Clase WORDMANGLING
 * ..............................................................................
 * Modela el módulo encargado de llevar a cabo el word mangling a partir de una
 * lista de reglas sobre palabras recibidas a través de un receptor.
 *
 * ******************************************************************************
 * ******************************************************************************/


#include "word_mangling.h"
#include "listaref.h"




// Constructor:
// 'lReglas' es una Lista de Reglas que se desean aplicar
WordMangling::WordMangling(Lista< Regla >& lReglas) : lReglas(lReglas) { }


// Ejecuta el alterador de palabras.
// PRE: 'rxPalabras' es un Receptor de palabras sobre las que se
// aplicarán las reglas.
void WordMangling::ejecutar(Receptor *rxPalabras) {
  // Aplicamos reglas a cada palabra entrante recibida
  std::string palabra = rxPalabras→recibir();

  while(¬palabra.empty()) {
    // Creamos la lista de transformaciones
    ListaRef< std::string > lTransformaciones;
    // Insertamos la palabra original
    lTransformaciones.insertarUltimo(palabra);

    // Aplicamos reglas a palabra
    for(int i = 0; i < this→lReglas.tamanio(); i++)
      this→lReglas[i]→aplicar(lTransformaciones);

    palabra = rxPalabras→recibir();
  }
}
```

abr 24, 13 15:47 **tx_salida_estandar.h** Page 1/1

```
/* ******************************************************************************
 * ******************************************************************************
 * Clase TXSALIDAESTANDAR
 * ..............................................................................
 * Transmisor que emite resultados hacia la salida estándar.
 *
 * ******************************************************************************
 * ******************************************************************************/


#ifndef TX_SALIDA_ESTANDAR_H
#define TX_SALIDA_ESTANDAR_H


#include <string>
#include "transmisor.h"



class TxSalidaEstandar:public Transmisor {
public:

  TxSalidaEstandar();

  // Se ejecuta la transmisión de un dato hacia la salida estándar.
  virtual void transmitir(const std::string& palabra);
};

#endif
```

abr 24, 13 15:47 **tx_salida_estandar.cpp** Page 1/1

```
/* ******************************************************************************
 * ******************************************************************************
 * Clase TXSALIDAESTANDAR
 * ..............................................................................
 * Transmisor que emite resultados hacia la salida estándar.
 *
 * ******************************************************************************
 * ******************************************************************************/


#include "tx_salida_estandar.h"
#include <iostream>



// Constructor
TxSalidaEstandar::TxSalidaEstandar() {
  activar();
}


// Se ejecuta la transmisión de un dato hacia la salida estándar.
void TxSalidaEstandar::transmitir(const std::string& palabra) {
  std::cout << palabra << std::endl;
}

```

abr 24, 13 15:47 **tx_archivo.h** Page 1/1

```
/* ******************************************************************************
 * ******************************************************************************
 * Clase TXARCHIVO
 * ..............................................................................
 * Transmisor que transmite los datos hacia un archivo de salida. El archivo de
 * salida se crea si no existe, y de existir, no se trunca, sino que se siguen
 * escribiendo las transmisiones al final del archivo.
 *
 * ******************************************************************************
 * ******************************************************************************/


#ifndef TX_ARCHIVO_H
#define TX_ARCHIVO_H


#include "transmisor.h"
#include <fstream>



class TxArchivo:public Transmisor {
private:

  std::ofstream archivo;    // Archivo en donde se escriben las palabras
                // alteradas

public:

  explicit TxArchivo(const std::string& nombre_archivo);

  virtual ~TxArchivo();

  // Se ejecuta la transmisión de un dato hacia un archivo. Se almacena
  // una palabra por linea.
  virtual void transmitir(const std::string& palabra);
};

#endif
```

abr 24, 13 15:47 **tx_archivo.cpp** Page 1/1

```
/* ******************************************************************************
 * ******************************************************************************
 * Clase TXARCHIVO
 * ..............................................................................
 * Transmisor que transmite los datos hacia un archivo de salida. El archivo de
 * salida se crea si no existe, y de existir, no se trunca, sino que se siguen
 * escribiendo las transmisiones al final del archivo.
 *
 * ******************************************************************************
 * ******************************************************************************/


#include "tx_archivo.h"



// Constructor
TxArchivo::TxArchivo(const std::string& nombre_archivo) {
  this→archivo.open(nombre_archivo.c_str(), std::ios::app);

  // Verificamos que se halla abierto correctamente
  if(this→archivo.is_open()) activar();
  else desactivar();
}


// Destructor
TxArchivo::~TxArchivo() {
  this→archivo.close();
}


// Se ejecuta la transmisión de un dato hacia un archivo.
void TxArchivo::transmitir(const std::string& palabra) {
  this→archivo << palabra << std::endl;
}
```

abr 24, 13 15:47 **transmisor.h** Page 1/1

```
/* ************************************************************************
 * ************************************************************************
 * Clase TRANSMISOR
 * ************************************************************************
 * ************************************************************************/

#ifndef TRANSMISOR_H
#define TRANSMISOR_H


#include <string>



class Transmisor {
private:

  // Estado del transmisor
  bool activo;

protected:

  // Activa el transmisor
  void activar();

  // Desactivar el transmisor
  void desactivar();

public:

  // Destructor
  virtual ~Transmisor() { };

  // Verifica si el transmisor esta activo.
  // POST: devuelve true si está activo o false en caso contrario
  bool estaActivo();

  // Se ejecuta la transmisión de un dato.
  virtual void transmitir(const std::string& palabra) = 0;
};


#endif
```

abr 24, 13 15:47 **transmisor.cpp** Page 1/1

```
/* ************************************************************************
 * ************************************************************************
 * Clase TRANSMISOR
 * ************************************************************************
 * ************************************************************************/


#include "transmisor.h"


// Verifica si el transmisor esta activo.
// POST: devuelve true si está activo o false en caso contrario
bool Transmisor::estaActivo() {
  return this→activo;
}


// Activa el transmisor
void Transmisor::activar() {
  this→activo = true;
}


// Desactiva el transmisor
void Transmisor::desactivar() {
  this→activo = false;
}
```

```
/* ************************************************************************
 * WORD MANGLING
 * ************************************************************************
 *
 * Facultad de Ingeniería – UBA
 * 75.42 Taller de Programación I
 * Trabajo Práctico N°3
 *
 * ALUMNO: Federico Martín Rossi
 * PADRÓN: 92086
 * EMAIL:  federicomrossi@gmail.com
 *
 * ************************************************************************
 *
 * Programa que realiza el proceso de word mangling a partir de un conjunto de
 * reglas especificadas por el usuario en un archivo de entrada. El ingreso de
 * palabras puede realizarse a través de otro archivo de entrada o mediante la
 * entrada estándar. Por cada palabra leída se le aplicará la serie de reglas
 * modificadoras, emitiendose las palabras resultantes hacia un archivo de
 * salida, o bien, directamente en la salida estándar.
 *
 *
 * FORMA DE USO
 * ============
 *
 * Deberá ejecutarse el programa en la línea de comandos de la siguiente
 * manera:
 *
 *    # ./tp [archivo_reglas] [entrada] [salida]
 *
 * donde,
 *
 *    archivo_reglas: nombre del archivo (incluyendo su extensión) en donde
 *              se almacenan las reglas a aplicar;
 *    entrada: especificación de la entrada de palabras. Debe especificarse
 *         un nombre de archivo o el símbolo '–' si se desea que la
 *         entrada de palabras sea a través de la entrada estándar;
 *    salida: especificación de la salida de palabras. Debe especificarse
 *        un nombre de archivo o el símbolo '–' si se desea que la
 *        salida de palabras sea a través de la salida estándar.
 *
 *
 * INSTRUCCIONES DEL ARCHIVO DE REGLAS
 * ===================================
 *
 * A continuación se muestran las instrucciones válidas para las reglas que
 * se desean aplicar. Cada una de ellas se debe finalizar con un ";".
 *
 * – "uppercase n m": Transforma los caracteres que están en la posición 'n'
 *   hasta la posición 'm' (inclusive) a mayúsculas;
 *
 * – "lowercase n m": Transforma los caracteres que están en la posición 'n'
 *   hasta la posición 'm' (inclusive) a minúsculas;
 *
 * – "repeat n m r i": Copia el substring delimitado por 'n' y 'm', lo repite
 *   'r' veces y lo inserta en la posición 'i';
 *
 * – "rotate n": Mueve los caracteres 'n' lugares hacia la derecha en forma
 *   circular. Si 'n' es negativo, la rotación se hace hacia la izquierda;
 *
 * – "insert i mmm": Agrega el string 'mmm' en la posición 'i'. 'mmm' no
 *   debe contener ningún espacio ni salto de línea ni el caracter ';';
 *
 * – "revert i": Revierte los efectos de las modificaciones de las últimas
 *   'i' reglas. El parámetro 'i' es siempre mayor o igual a 1.
 *
```



```
 */


#include "parser_entrada.h"
#include "parser_salida.h"
#include "parser_reglas.h"
#include "word_mangling.h"




/* ************************************************************************
 * PROGRAMA PRINCIPAL
 * ************************************************************************/


int main(int argc, char* argv[]) {
  // Si la cantidad de argumentos no es la correcta, lanzamos código de
  // retorno
  if(argc ≠ 4) return 1;

  // Declaramos parsers
  ParserEntrada pEntrada;
  ParserSalida pSalida;
  ParserReglas pReglas;

  // Parseamos argumentos de entrada
  Receptor *rxPalabras = pEntrada.parsear(argv[2]);
  Transmisor *tx = pSalida.parsear(argv[3]);
  Lista< Regla > lReglas = pReglas.parsear(argv[1], tx);

  // Verificamos que esten activos el transmisor y el receptor
  if(¬rxPalabras→estaActivo() ∨ ¬tx→estaActivo() ∨ lReglas.estaVacia()) {
    // Liberamos memoria
    delete rxPalabras;
    delete tx;

    // Retornamos código de error
    return 2;
  }

  // Ejecutamos el proceso de alteración de palabras
  WordMangling wordMangling(lReglas);
  wordMangling.ejecutar(rxPalabras);

  delete rxPalabras;
  delete tx;

  return 0;
}
```

abr 24, 13 15:47 **rx_entrada_estandar.h** Page 1/1

```
/* ******************************************************************************
 * ******************************************************************************
 * Clase RXENTRADAESTANDAR
 * ..............................................................................
 * Receptor que recibe los datos desde la entrada estándar.
 *
 * ******************************************************************************
 * ******************************************************************************/


#ifndef RX_ENTRADA_ESTANDAR_H
#define RX_ENTRADA_ESTANDAR_H


#include "receptor.h"



class RxEntradaEstandar:public Receptor {
public:

  // Constructor
  RxEntradaEstandar();

  // Se ejecuta la recepción de un dato desde la entrada estándar.
  // PRE: Se debe ingresar una palabra sin espacios.
  // POST: se devuelve la palabra recibida. Debe considerarse terminada la
  // recepción de palabras cuando se recibe una cadena vacía, es decir, nada.
  // Debe enviarse una única palabra a la vez, quedando terminantemente prohibid
o
  // enviar una cadena que conste de mas de una palabra.
  virtual std::string recibir();
};


#endif
```

abr 24, 13 15:47 **rx_entrada_estandar.cpp** Page 1/1

```
/* ******************************************************************************
 * ******************************************************************************
 * Clase RXENTRADAESTANDAR
 * ..............................................................................
 * Receptor que recibe los datos desde la entrada estándar.
 *
 * ******************************************************************************
 * ******************************************************************************/


#include <iostream>
#include "rx_entrada_estandar.h"



// Constructor
RxEntradaEstandar::RxEntradaEstandar() {
  activar();
}


// Se ejecuta la recepción de un dato desde la entrada estándar.
// PRE: Se debe ingresar una palabra sin espacios.
// POST: se devuelve la palabra recibida. Debe considerarse terminada la
// recepción de palabras cuando se recibe una cadena vacía, es decir, nada.
// Debe enviarse una única palabra a la vez, quedando terminantemente prohibido
// enviar una cadena que conste de mas de una palabra.
std::string RxEntradaEstandar::recibir() {
  std::string palabra;
  getline(std::cin, palabra);

  return palabra;
}
```

abr 24, 13 15:47 **rx_archivo.h** Page 1/1

```
/* ******************************************************************************
 * ******************************************************************************
 * Clase RXARCHIVO
 * ..............................................................................
 * Receptor que recibe los datos desde un archivo.
 *
 * ******************************************************************************
 * ******************************************************************************/


#ifndef RX_ARCHIVO_H
#define RX_ARCHIVO_H


#include "receptor.h"
#include <fstream>



class RxArchivo:public Receptor {
private:

  std::ifstream archivo;      // Archivo del que se toman las palabras

public:

  // Constructor
  explicit RxArchivo(const std::string& nombre_archivo);

  // Destructor
  ~RxArchivo();

  // Se ejecuta la recepción de un dato desde la entrada estándar.
  // POST: se devuelve la palabra recibida. Debe considerarse terminada la
  // recepción de palabras cuando se recibe una cadena vacía, es decir, nada. En
  // este caso debe limitarse estrictamente a una palabra por línea del archivo.
  virtual std::string recibir();
};


#endif
```

abr 24, 13 15:47 **rx_archivo.cpp** Page 1/1

```
/* ******************************************************************************
 * ******************************************************************************
 * Clase RXARCHIVO
 * ..............................................................................
 * Receptor que recibe los datos desde un archivo.
 *
 * ******************************************************************************
 * ******************************************************************************/


#include "rx_archivo.h"



// Constructor
RxArchivo::RxArchivo(const std::string& nombre_archivo) {
  // Abrimos archivo
  this→archivo.open(nombre_archivo.c_str());

  // Verificamos que se halla abierto correctamente
  if(this→archivo.is_open()) activar();
  else desactivar();
}


// Destructor
RxArchivo::~RxArchivo() {
  this→archivo.close();
}


// Se ejecuta la recepción de un dato desde la entrada estándar.
// POST: se devuelve la palabra recibida. Debe considerarse terminada la
// recepción de palabras cuando se recibe una cadena vacía, es decir, nada. En
// este caso debe limitarse estrictamente a una palabra por línea del archivo.
std::string RxArchivo::recibir() {
  std::string palabra;
  this→archivo >> palabra;

  return palabra;
}
```

abr 24, 13 15:47 **ruppercase.h** Page 1/1

```
/* ************************************************************************
 * ************************************************************************
 * Clase RUPPERCASE
 * ........................................................................
 * La regla transforma los caracteres que están en la posición 'n' hasta la
 * posición 'm' (inclusive) a mayúsculas.
 *
 * ************************************************************************
 * ************************************************************************/


#ifndef RUPPERCASE_H
#define RUPPERCASE_H


#include "regla.h"



class RUppercase:public Regla {
private:

  int n;    // Posición inicial de transformación
  int m;    // Posición final de transformación

public:

  // Constructor
  RUppercase(int n, int m);

  // Destructor
  ~RUppercase();

  // Aplica la regla sobre una lista de transformaciones.
  virtual void aplicar(ListaRef< std::string >& lTransformaciones);
};

#endif
```

abr 24, 13 15:47 **ruppercase.cpp** Page 1/1

```
/* ************************************************************************
 * ************************************************************************
 * Clase RUPPERCASE
 * ........................................................................
 * La regla transforma los caracteres que están en la posición 'n' hasta la
 * posición 'm' (inclusive) a mayúsculas.
 *
 * ************************************************************************
 * ************************************************************************/


#include "ruppercase.h"



// Constructor
RUppercase::RUppercase(int n, int m) : n(n), m(m) { }


// Destructor
RUppercase::~RUppercase() { }


// Aplica la regla sobre una lista de transformaciones
void RUppercase::aplicar(ListaRef< std::string >& lTransformaciones) {
  // Tomamos la palabra sobre la cual debemos aplicar
  std::string s = lTransformaciones.verUltimo();
  int sTamanio = s.size();

  // Verificamos si estamos dentro del rango de la palabra
  if(estaFueraDeRango(sTamanio, this→n, this→m)) return;

  // Convertimos posiciones en posiciones válidas respecto al string
  int nn = convertirEnPosicionValida(sTamanio, this→n);
  int mm = convertirEnPosicionValida(sTamanio, this→m);

  // Transformamos en mayúscula las letras del conjunto
  for(int i = nn; i ≤ mm; i++)
    s[i] = toupper(s[i]);

  // Insertamos la transformación en la lista
  lTransformaciones.insertarUltimo(s);
}
```

abr 24, 13 15:47 **rrotate.h** Page 1/1

```
/* ************************************************************************
 * ************************************************************************
 * Clase RROTATE
 * ........................................................................
 * La regla mueve los caracteres 'n' lugares hacia la derecha en forma
 * circular. Si 'n' es negativo, la rotación se hace hacia la izquierda.
 *
 * ************************************************************************
 * ************************************************************************/


#ifndef RROTATE_H
#define RROTATE_H


#include "regla.h"



class RRotate:public Regla {
private:

  int n;        // Cantidad de lugares que deben moverse los caracteres
  bool derecha;   // Rotar hacia la derecha

public:

  // Constructor
  explicit RRotate(int n);

  // Destructor
  ~RRotate();

  // Aplica la regla sobre una lista de transformaciones
  virtual void aplicar(ListaRef< std::string >& lTransformaciones);
};


#endif
```

abr 24, 13 15:47 **rrotate.cpp** Page 1/1

```
/* ************************************************************************
 * ************************************************************************
 * Clase RROTATE
 * ........................................................................
 * La regla mueve los caracteres 'n' lugares hacia la derecha en forma
 * circular. Si 'n' es negativo, la rotación se hace hacia la izquierda.
 *
 * ************************************************************************
 * ************************************************************************/


#include "rrotate.h"



// Constructor
RRotate::RRotate(int n) {
  if(n ≥ 0) {
    this→derecha = true;
    this→n = n;
  }
  else {
    this→derecha = false;
    this→n = n * (−1);
  }
}


// Destructor
RRotate::~RRotate() { }


// Aplica la regla sobre una lista de transformaciones
void RRotate::aplicar(ListaRef< std::string >& lTransformaciones) {
  // Tomamos la palabra sobre la cual debemos aplicar
  std::string s = lTransformaciones.verUltimo();
  int sTamanio = s.size();

  for(int j = 0; j < this→n; j++) {
    // Rotamos hacia la derecha
    if(this→derecha) {
      // Guardamos el último caractér que será el último en moverse
      char u = s[sTamanio−1];
      for(int i = sTamanio−1; i ≥ 1; i−−) s[i] = s[i−1];
      s[0] = u;
    }
    // Rotamos hacia la izquierda
    else {
      // Guardamos el primer caractér que será el último en moverse
      char u = s[0];
      for(int i = 0; i < (sTamanio−1); i++) s[i] = s[i+1];
      s[sTamanio−1] = u;
    }
  }

  // Insertamos la transformación en la lista
  lTransformaciones.insertarUltimo(s);
}
```

abr 24, 13 15:47 **rrevert.h** Page 1/1

```
/* ************************************************************************
 * ************************************************************************
 * Clase RREVERT
 * ........................................................................
 * La regla revierte los efectos de las modificaciones de las últimas i reglas.
 *
 * ************************************************************************
 * ************************************************************************/


#ifndef RREVERT_H
#define RREVERT_H


#include "regla.h"



class RRevert:public Regla {
private:

  int i;      // Cantidad de modificaciones de reglas a revertir

public:

  // Constructor
  explicit RRevert(int i);

  // Destructor
  ~RRevert();

  // Aplica la regla sobre una lista de transformaciones
  virtual void aplicar(ListaRef< std::string >& lTransformaciones);
};


#endif
```

abr 24, 13 15:47 **rrevert.cpp** Page 1/1

```
/* ************************************************************************
 * ************************************************************************
 * Clase RPRINT
 * ........................................................................
 * La regla transmite el resultado de la transformación mas reciente.
 *
 * ************************************************************************
 * ************************************************************************/


#include "rrevert.h"



// Constructor
RRevert::RRevert(int i) {
  this→i = i;
}


// Destructor
RRevert::~RRevert() { }


// Aplica la regla sobre una lista de transformaciones
// POST: Si la lista contiene menos cantidad de transformaciones que las que se
// desean revertir no se revierte nada, queda como estaba la lista.
void RRevert::aplicar(ListaRef< std::string >& lTransformaciones) {
  // Calculamos la posición a la que hay que revertir
  int pos = lTransformaciones.tamanio() − this→i − 1;

  // Verificamos si hay elementos y que la cantidad a revertir sea válida
  if(lTransformaciones.estaVacia() ∨ pos < 0) return;

  // Insertamos la transformación en la lista
  lTransformaciones.insertarUltimo(lTransformaciones[pos]);
}
```

```
/* ************************************************************************
 * ************************************************************************
 * Clase RREPEAT
 * ........................................................................
 * La regla copia el substring delimitado por 'n' y 'm', lo repite 'r' veces
 * y lo inserta en la posición 'i'.
 * Si 'i' es positivo, se insertará el substring antes del caractér en la
 * posición 'i'. Si es negativo, se insertará después del caractér en la
 * posición 'i'.
 * Si 'i' es positivo y excede las posiciones válidas de la palabra, se
 * insertará al final, mientras que, si es negativo, se insertará al
 * principio.
 *
 * ************************************************************************
 * ************************************************************************/


#ifndef RREPEAT_H
#define RREPEAT_H


#include "regla.h"



class RRepeat:public Regla {
private:

  int n;    // Posición inicial del substring
  int m;    // Posición final del substring
  int r;    // Cantidad de repeticiones del substring
  int i;    // Posición en donde se debe insertar los substrings

public:

  // Constructor
  RRepeat(int n, int m, int r, int i);

  // Destructor
  ~RRepeat();

  // Aplica la regla sobre una lista de transformaciones
  virtual void aplicar(ListaRef< std::string >& lTransformaciones);
};


#endif
```



```
/* ************************************************************************
 * ************************************************************************
 * Clase RREPEAT
 * ........................................................................
 * La regla copia el substring delimitado por 'n' y 'm', lo repite 'r' veces
 * y lo inserta en la posición 'i'.
 * Si 'i' es positivo, se insertará el substring antes del caractér en la
 * posición 'i'. Si es negativo, se insertará después del caractér en la
 * posición 'i'.
 * Si 'i' es positivo y excede las posiciones válidas de la palabra, se
 * insertará al final, mientras que, si es negativo, se insertará al
 * principio.
 *
 * ************************************************************************
 * ************************************************************************/


#include "rrepeat.h"



// Constructor
RRepeat::RRepeat(int n, int m, int r, int i) {
  this→n = n;
  this→m = m;
  this→r = r;
  this→i = i;
}


// Destructor
RRepeat::~RRepeat() { }


// Aplica la regla sobre una lista de transformaciones
void RRepeat::aplicar(ListaRef< std::string >& lTransformaciones) {
  // Tomamos la palabra sobre la cual debemos aplicar
  std::string s = lTransformaciones.verUltimo();
  int sTamanio = s.size();

  // Verificamos si estamos dentro del rango de la palabra
  if(estaFueraDeRango(sTamanio, this→n, this→m)) return;

  // Convertimos posiciones en posiciones válidas respecto al string
  int nn = convertirEnPosicionValida(sTamanio, this→n);
  int mm = convertirEnPosicionValida(sTamanio, this→m);
  int ii = convertirEnPosicionValida(sTamanio, this→i);

  // Tomamos el substring y generamos el string repetido a insertar
  std::string sToRepeat = s.substr(nn, mm−nn+1);
  std::string sToAppend = sToRepeat;
  for(int i = 0; ++i < this→r; sToAppend.append(sToRepeat));

  // Si 'i' es positivo, se insertará antes del caracter en la posición 'i',
  // a menos que esté fuera de rango, en cuyo caso, se posicionará al final
  // y se insertará antes del caracter de dicha posición final.
  // Si 'i' es negativo, se insertará despues del caracter en la posición
  // 'i', a menos que esté fuera de rango, en cuyo caso, se posicionará al
  // inicio y se insertará después del caracter de dicha posición inicial.
  if((estaFueraDeRango(sTamanio, this→i) ∧ (this→i < 0)) ∨ (this→i < 0))
    ++ii;

  // Insertamos repetición en string
  s.insert(ii, sToAppend);

  // 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

```cpp
/* ************************************************************************
 * ************************************************************************
 * Clase RPRINT
 * ........................................................................
 * La regla transmite el resultado de la transformación mas reciente.
 *
 * ************************************************************************
 * ************************************************************************/


#include "rprint.h"



// Constructor
RPrint::RPrint(Transmisor *tx) : tx(tx) { }


// Destructor
RPrint::~RPrint() { }


// Aplica la regla sobre una lista de transformaciones. La regla emite la
// última modificación realizada.
void RPrint::aplicar(ListaRef< std::string >& lTransformaciones) {
  // Insertamos una copia del primer elemento de la lista para poder
  // considerar a print como una modificación
  lTransformaciones.insertarUltimo(lTransformaciones.verUltimo());
  tx→transmitir(lTransformaciones.verUltimo());
}
```

abr 24, 13 15:47 **rlowercase.h** Page 1/1

```cpp
/* ************************************************************************
 * ************************************************************************
 * Clase RLOWERCASE
 * ........................................................................
 * La regla transforma los caracteres que están en la posición 'n' hasta la
 * posición 'm' (inclusive) a minúsculas.
 *
 * ************************************************************************
 * ************************************************************************/


#ifndef RLOWERCASE_H
#define RLOWERCASE_H


#include "regla.h"



class RLowercase:public Regla {
private:
  int n;    // Posición inicial de transformación
  int m;    // Posición final de transformación

public:

  // Constructor
  RLowercase(int n, int m);

  // Destructor
  ~RLowercase();

  // Aplica la regla sobre una lista de transformaciones
  virtual void aplicar(ListaRef< std::string >& lTransformaciones);
};


#endif
```

abr 24, 13 15:47 **rlowercase.cpp** Page 1/1

```cpp
/* ************************************************************************
 * ************************************************************************
 * Clase RLOWERCASE
 * ........................................................................
 * La regla transforma los caracteres que están en la posición 'n' hasta la
 * posición 'm' (inclusive) a minúsculas.
 *
 * ************************************************************************
 * ************************************************************************/


#include "rlowercase.h"



// Constructor
RLowercase::RLowercase(int n, int m) : n(n), m(m) { }


// Destructor
RLowercase::~RLowercase() { }


// Aplica la regla sobre una lista de transformaciones
void RLowercase::aplicar(ListaRef< std::string >& lTransformaciones) {
  // Tomamos la palabra sobre la cual debemos aplicar
  std::string s = lTransformaciones.verUltimo();
  int sTamanio = s.size();

  // Verificamos si estamos dentro del rango de la palabra
  if(estaFueraDeRango(sTamanio, this→n, this→m)) return;

  // Convertimos posiciones en posiciones validas respecto al string
  int nn = convertirEnPosicionValida(sTamanio, this→n);
  int mm = convertirEnPosicionValida(sTamanio, this→m);

  // Transformamos en minúscula las letras del conjunto
  for(int i = nn; i ≤ mm; i++)
    s[i] = tolower(s[i]);

  // Insertamos la transformación en la lista
  lTransformaciones.insertarUltimo(s);
}
```

abr 24, 13 15:47 **rinsert.h** Page 1/1

```cpp
/* ************************************************************************
 * ************************************************************************
 * Clase RINSERT
 * ........................................................................
 * La regla agrega el string 'mmm' en la posición 'i' de una palabra.
 * Si 'i' es positivo, se insertará el substring antes del caractér en la
 * posición 'i'. Si es negativo, se insertará después del caractér en la
 * posición 'i'.
 * Si 'i' es positivo y excede las posiciones válidas de la palabra, se
 * insertará al final, mientras que, si es negativo, se insertará al
 * principio.
 *
 * ************************************************************************
 * ************************************************************************/


#ifndef RINSERT_H
#define RINSERT_H


#include "regla.h"



class RInsert:public Regla {
private:

  int i;          // Posición donde se debe insertar el string
  std::string mmm;    // String a agregar

public:

  // Constructor
  RInsert(int i, const std::string& mmm);

  // Destructor
  ~RInsert();

  // Aplica la regla sobre una lista de transformaciones
  virtual void aplicar(ListaRef< std::string >& lTransformaciones);
};


#endif
```

```
/* ************************************************************************
 * ************************************************************************
 * Clase RINSERT
 * ........................................................................
 * La regla agrega el string 'mmm' en la posición 'i' de una palabra.
 * Si 'i' es positivo, se insertará el substring antes del caractér en la
 * posición 'i'. Si es negativo, se insertará después del caractér en la
 * posición 'i'.
 * Si 'i' es positivo y excede las posiciones válidas de la palabra, se
 * insertará al final, mientras que, si es negativo, se insertará al
 * principio.
 *
 * ************************************************************************
 * ************************************************************************/


#include "rinsert.h"



// Constructor
RInsert::RInsert(int i, const std::string& mmm) {
  this→i = i;
  this→mmm = mmm;
}


// Destructor
RInsert::~RInsert() { }


// Aplica la regla sobre una lista de transformaciones
void RInsert::aplicar(ListaRef< std::string >& lTransformaciones) {
  // Tomamos la palabra sobre la cual debemos aplicar
  std::string s = lTransformaciones.verUltimo();
  int sTamanio = s.size();

  // Convertimos posiciones en posiciones válidas respecto al string
  int ii = convertirEnPosicionValida(sTamanio, this→i);

  // Si 'i' es positivo, se insertará antes del caracter en la posición 'i',
  // a menos que esté fuera de rango, en cuyo caso, se posicionará al final
  // y se insertará antes del caracter de dicha posición final.
  // Si 'i' es negativo, se insertará despues del caracter en la posición
  // 'i', a menos que esté fuera de rango, en cuyo caso, se posicionará al
  // inicio y se insertará después del caracter de dicha posición inicial.
  if((estaFueraDeRango(sTamanio, this→i) ∧ (this→i < 0)) ∨ (this→i < 0))
    ++ii;

  // Insertamos repetición en string
  s.insert(ii, this→mmm);

  // Insertamos la transformación en la lista
  lTransformaciones.insertarUltimo(s);
}
```



```
/* ************************************************************************
 * ************************************************************************
 * Clase REGLA
 * ************************************************************************
 * ************************************************************************/


#ifndef REGLA_H
#define REGLA_H

#include <string>
#include "listaref.h"



class Regla {
public:

  // Destructor
  virtual ~Regla();

  // Aplica la regla sobre una lista de transformaciones.
  virtual void aplicar(ListaRef< std::string >& lTransformaciones) = 0;


protected:

  // Convierte una posición de entrada en una posición válida de un string
  // PRE: 'sTamanio' es el tamaño del string al que se refiere; 'pos' es la
  // posición del string.
  // POST: si 'pos' es negativa, se convierte en una posición válida
  // positiva, considerando que en caso de desborde se devolverá la primer
  // posición del string; si 'pos' es positiva y supera el tamaño del string,
  // se devuelve la última posición de este.
  int convertirEnPosicionValida(int sTamanio, int pos);

  // Verifica si existe intersección entre el intervalo sobre el cual se
  // quiere aplicar la arregla y el tamaño real de la palabra.
  // PRE: 'sTamanio' es el tamaño de la palabra; 'n' es la posición inicial
  // del substring; 'm' es la posición final del substring. 'n' debe ser
  // menor o igual que 'm'.
  // POST: devuelve true si el intervalo se encuentra dentro del rango o
  // false en su defecto.
  bool estaFueraDeRango(int sTamanio, int n, int m);

  // Verifica si la posición 'i' se encuentra fuera de rango en referencia a
  // la palabra.
  // PRE: 'sTamanio' es el tamaño de la palabra; 'i' es una posición.
  // POST: devuelve true si la posición es una posición dentro de la cadena o
  // false en su defecto.
  bool estaFueraDeRango(int sTamanio, int i);
};

#endif
```

abr 24, 13 15:47 **regla.cpp** Page 1/1

```cpp
/* ************************************************************************
 * ************************************************************************
 * Clase REGLA
 * ************************************************************************
 * ************************************************************************/


#include "regla.h"


// Destructor
Regla::~Regla() { }


// Convierte una posición de entrada en una posición válida de un string
// PRE: 'sTamanio' es el tamaño del string al que se refiere; 'pos' es la
// posición del string.
// POST: si 'pos' es negativa, se convierte en una posición válida
// positiva, considerando que en caso de desborde se devolverá la primer
// posición del string; si 'pos' es positiva y supera el tamaño del string,
// se devuelve la última posición de este.
int Regla::convertirEnPosicionValida(int sTamanio, int pos) {
  if(pos < 0) {
    int posValida = sTamanio + pos;
    if(posValida < 0) return 0;
    return posValida;
  }
  else if (pos > sTamanio − 1) return (sTamanio − 1);

  return pos;
}


// Verifica si existe intersección entre el intervalo sobre el cual se
// quiere aplicar la arregla y el tamaño real de la palabra.
// PRE: 'sTamanio' es el tamaño de la palabra; 'n' es la posición inicial
// del substring; 'm' es la posición final del substring. 'n' debe ser
// menor o igual que 'm'.
// POST: devuelve true si el intervalo se encuentra dentro del rango o
// false en su defecto.
bool Regla::estaFueraDeRango(int sTamanio, int n, int m) {
  // Caso en que ambos son negativos y están fuera de rango
  if((n < 0) ∧ (m < 0) ∧ ((n+sTamanio) < 0) ∧ ((m+sTamanio) < 0))
    return true;
  // Caso en que ambos son positivos y están fuera de rango
  else if ((n > 0) ∧ (m > 0) ∧ (n > sTamanio−1) ∧ (m > sTamanio−1))
    return true;

  return false;
}


// Verifica si la posición 'i' se encuentra fuera de rango en referencia a
// la palabra.
// PRE: 'sTamanio' es el tamaño de la palabra; 'i' es una posición.
// POST: devuelve true si la posición es una posición dentro de la cadena o
// false en su defecto.
bool Regla::estaFueraDeRango(int sTamanio, int i) {
  // Caso en que es negativo y está fuera de rango
  if((i < 0) ∧ ((i+sTamanio) < 0)) return true;
  // Caso en que es positivo y está fuera de rango
  else if ((i > 0) ∧ (i > sTamanio−1)) return true;

  return false;
}
```

abr 24, 13 15:47 **receptor.h** Page 1/1

```cpp
/* ************************************************************************
 * ************************************************************************
 * Clase RECEPTOR
 * ************************************************************************
 * ************************************************************************/


#ifndef RECEPTOR_H
#define RECEPTOR_H

#include <string>



class Receptor {
private:

  // Estado del receptor
  bool activo;

protected:

  // Activa el receptor
  void activar();

  // Desactiva el receptor
  void desactivar();

public:

  // Destructor
  virtual ~Receptor() { };

  // Verifica si el receptor esta activo.
  // POST: devuelve true si está activo o false en caso contrario
  bool estaActivo();

  // Se ejecuta la recepción de un dato desde la entrada estándar.
  // POST: se devuelve la palabra recibida. Debe considerarse terminada la
  // recepción de palabras cuando se recibe una cadena vacía, es decir, nada.
  virtual std::string recibir() = 0;
};


#endif
```

abr 24, 13 15:47 **receptor.cpp** Page 1/1

```cpp
/* ************************************************************************
 * ************************************************************************
 * Clase RECEPTOR
 * ************************************************************************
 * ************************************************************************/


#include "receptor.h"


// Verifica si el receptor esta activo.
// POST: devuelve true si está activo o false en caso contrario
bool Receptor::estaActivo() {
  return this→activo;
}


// Activa el receptor
void Receptor::activar() {
  this→activo = true;
}


// Desactiva el receptor
void Receptor::desactivar() {
  this→activo = false;
}
```

abr 24, 13 15:47 **pila.h** Page 1/3

```cpp
/* ************************************************************************
 * ************************************************************************
 * Clase PILA
 * ........................................................................
 * Implementación de la clase Pila.
 * ************************************************************************
 * ************************************************************************/



#ifndef PILA_H
#define PILA_H



/* ************************************************************************
 * DECLARACIÓN DE LA CLASE
 * ************************************************************************/


template < typename Tipo >
class Pila {
private:

  struct Nodo {
    Tipo dato;        // Dato al que referencia el nodo
    Nodo *siguiente;    // Puntero al siguiente nodo

    // Constructor
    explicit Nodo(Tipo& dato) : dato(dato), siguiente(0) { }
  };

  int cantElementos;      // Número de elementos en la pila
  Nodo *primero;        // Puntero al primer elemento de la pila
  Nodo *ultimo;       // Puntero al último elemento de la pila

public:

  // Constructor
  Pila();

  // Destructor
  ~Pila();

  // Verifica si la pila tiene o no elementos.
  // POST: devuelve true si la pila esta vacía o false en su defecto.
  bool estaVacia();

  // Agrega un nuevo elemento a la pila.
  // PRE: 'dato' es el dato que se desea apilar.
  // POST: se agregó el nuevo elemento, el cual se encuentra en el tope de
  // la pila.
  void apilar(Tipo& dato);


  // Saca el primer elemento de la pila.
  // POST: si no hay elementos que desapilar no hace nada.
  void desapilar();

  // Obtiene el valor del primer elemento de la pila.
  // POST: se devuelve el dato que se encuentra en el tope de la pila
  Tipo& verTope();
};
```

```
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 >::estaVacia() {
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.
```



```
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

```
/* ************************************************************************
 * ************************************************************************
 * Clase PARSERSALIDA
 * ************************************************************************
 * ************************************************************************/


#ifndef PARSER_SALIDA_H
#define PARSER_SALIDA_H

#include <string>
#include "transmisor.h"



class ParserSalida {
public:

  // Parsea la especificación del tipo de salida que se utilizará.
  // PRE: 'tipoSalida' es el tipo de salida que se desea usar. Debe
  // pasársele el nombre de archivo en caso de desear utilizar un archivo de
  // palabras alteradas o el caracter "-" si se desea enviar las palabras
  // a través de la salida estándar.
  // POST: se devuelve un puntero a un Transmisor.
  Transmisor* parsear(const std::string& tipoSalida);
};

#endif
```

abr 24, 13 15:47 **parser_salida.cpp** Page 1/1

```
/* ************************************************************************
 * ************************************************************************
 * Clase PARSERENTRADA
 * ************************************************************************
 * ************************************************************************/


#include "parser_salida.h"
#include "tx_archivo.h"
#include "tx_salida_estandar.h"



// Parsea la especificación del tipo de salida que se utilizará.
// PRE: 'tipoSalida' es el tipo de salida que se desea usar. Debe
// pasársele el nombre de archivo en caso de desear utilizar un archivo de
// palabras alteradas o el caracter "-" si se desea enviar las palabras
// a través de la salida estándar.
// POST: se devuelve un puntero a un Transmisor.
Transmisor* ParserSalida::parsear(const std::string& tipoSalida) {
  // La emisión será a través de la salida estándar
  if(tipoSalida ≡ "-")
    return new TxSalidaEstandar();
  // La emisión se hará a través de un archivo de de salida
  return new TxArchivo(tipoSalida);
}
```

abr 24, 13 15:47 **parser_reglas.h** Page 1/1

```
/* ************************************************************************
 * ************************************************************************
 * Clase PARSERREGLAS
 * ************************************************************************
 * ************************************************************************/


#ifndef PARSER_REGLAS_H
#define PARSER_REGLAS_H

#include <fstream>
#include "lista.h"
#include "regla.h"
#include "transmisor.h"



class ParserReglas {
public:

  // Parsea el archivo de reglas.
  // PRE: 'archivo' es el nombre de archivo que contiene las reglas; 'tx' es
  // un puntero a un Transmisor el cual puede ser usado por alguna regla
  // para emitir palabras.
  // POST: se devuelve una referencia a una lista que contiene, en orden
  // de aparición en el archivo, los objetos que son Regla. La lista se
  // encontrará vacía de producirse un error en la apertura del archivo o si
  // este último no contenía instrucciones válidas, debiéndose considerarse como
  // erróneo.
  Lista< Regla > parsear(const std::string& nombre_archivo, Transmisor* tx);
};


#endif
```

abr 24, 13 15:47 **parser_reglas.cpp** Page 1/2

```
/* ************************************************************************
 * ************************************************************************
 * Clase PARSERREGLAS
 * ************************************************************************
 * ************************************************************************/


#include "parser_reglas.h"

// Reglas
#include "ruppercase.h"
#include "rlowercase.h"
#include "rrepeat.h"
#include "rrotate.h"
#include "rinsert.h"
#include "rrevert.h"
#include "rprint.h"


namespace {
  // Constantes para los nombres de instrucciones
  const std::string S_UPPERCASE = "uppercase";
  const std::string S_LOWERCASE = "lowercase";
  const std::string S_REPEAT = "repeat";
  const std::string S_ROTATE = "rotate";
  const std::string S_INSERT = "insert";
  const std::string S_REVERT = "revert";
  const std::string S_PRINT = "print";

  // Constante para caracter de fin de instrucción
  const std::string S_FIN_INSTRUCCION = ";";
}




/* ************************************************************************
 * DEFINICIONES DE MÉTODOS DE LA CLASE
 * ************************************************************************/


// Parsea el archivo de reglas.
// PRE: 'archivo' es el nombre de archivo que contiene las reglas; 'tx' es
// un puntero a un Transmisor el cual puede ser usado por alguna regla
// para emitir palabras.
// POST: se devuelve una referencia a una lista que contiene, en orden
// de aparición en el archivo, los objetos que son Regla. La lista se
// encontrará vacía de producirse un error en la apertura del archivo o si
// este último no contenía instrucciones válidas, debiéndose considerarse como
// erróneo.
Lista< Regla > ParserReglas::parsear(const std::string& nombre_archivo,
  Transmisor *tx) {
  // Creamos la lista de reglas
  Lista< Regla > lReglas;

  // Abrimos archivo
  std::ifstream archivo;
  archivo.open(nombre_archivo.c_str());

  // Verificamos que se halla abierto correctamente
  if(¬archivo.is_open()) return lReglas;


  // Variables auxiliares para parseo
  std::string instruccion, mmm, fin;
  int n, m, r, i;
```

```
67
68    // Procesamos cada instrucción. Si alguna no esta definida, no se
69    // considera y se sigue procesando las restantes.
70    while(archivo >> instruccion) {
71      // Filtros
72      if(instruccion ≡ S_UPPERCASE) {
73        archivo >> n >> m >> fin;
74        lReglas.insertarUltimo(new RUppercase(n, m));
75      }
76      else if (instruccion ≡ S_LOWERCASE) {
77        archivo >> n >> m >> fin;
78        lReglas.insertarUltimo(new RLowercase(n, m));
79      }
80      else if (instruccion ≡ S_REPEAT) {
81        archivo >> n >> m >> r >> i >> fin;
82        lReglas.insertarUltimo(new RRepeat(n, m, r, i));
83      }
84      else if (instruccion ≡ S_ROTATE) {
85        archivo >> n >> fin;
86        lReglas.insertarUltimo(new RRotate(n));
87      }
88      else if (instruccion ≡ S_INSERT) {
89        archivo >> i >> mmm >> fin;
90        lReglas.insertarUltimo(new RInsert(i, mmm));
91      }
92      else if (instruccion ≡ S_REVERT) {
93        archivo >> i >> fin;
94        lReglas.insertarUltimo(new RRevert(i));
95      }
96      else if (instruccion ≡ S_PRINT) {
97        archivo >> fin;
98        lReglas.insertarUltimo(new RPrint(tx));
99      }
100     // Caso en que no matchea con ninguna instrucción. Lo consideramos un
101     // error.
102     else {
103       // Vaciamos la lista y la devolvemos vacía
104       while(¬lReglas.estaVacia())
105         delete lReglas.eliminarPrimero();
106       break;
107     }
108   }
109
110   // Cerramos el archivo
111   archivo.close();
112
113   return lReglas;
114 }
```



```
1  /* ************************************************************************
2   * ************************************************************************
3   * Clase PARSERENTRADA
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
17 class ParserEntrada {
18 public:
19
20   // Parsea la especificación del tipo de entrada se utilizará.
21   // PRE: 'tipoEntrada' es el tipo de entrada que se desea usar. Debe
22   // pasársele el nombre de archivo en caso de desear utilizar un archivo de
23   // palabras iniciales o el caracter "-" si se desea recibir las palabras
24   // a través de la entrada estándar.
25   // POST: se devuelve un puntero a un Receptor.
26   Receptor* parsear(const std::string& tipoEntrada);
27 };
28
29
30 #endif
```

abr 24, 13 15:47 **parser_entrada.cpp** Page 1/1

```cpp
/* ******************************************************************************
 * ******************************************************************************
 * Clase PARSERENTRADA
 * ******************************************************************************
 * ******************************************************************************/


#include "parser_entrada.h"
#include "rx_archivo.h"
#include "rx_entrada_estandar.h"



// Parsea la especificación del tipo de entrada se utilizará.
// PRE: 'tipoEntrada' es el tipo de entrada que se desea usar. Debe
// pasársele el nombre de archivo en caso de desear utilizar un archivo de
// palabras iniciales o el caracter "-" si se desea recibir las palabras
// a través de la entrada estándar.
// POST: se devuelve un puntero a un Receptor.
Receptor* ParserEntrada::parsear(const std::string& tipoEntrada) {
  // El ingreso será a través de la entrada estándar
  if(tipoEntrada == "-")
    return new RxEntradaEstandar();

  // El ingreso se hará a través de un archivo de entrada
  return new RxArchivo(tipoEntrada);
}
```

abr 24, 13 15:47 **listaref.h** Page 1/4

```cpp
/* ******************************************************************************
 * ******************************************************************************
 * Clase LISTAREF
 * ..............................................................................
 * Implementación de la clase ListaRef la cual modela una lista de referencias.
 *
 * ******************************************************************************
 * ******************************************************************************/



#ifndef LISTAREF_H
#define LISTAREF_H


/* ******************************************************************************
 * DECLARACIÓN DE LA CLASE
 * ******************************************************************************/


template < typename Tipo >
class ListaRef {
private:

  struct Nodo {
    Tipo dato;        // Dato al que referencia el nodo
    Nodo *siguiente;    // Puntero al siguiente nodo

    // Constructor
    explicit Nodo(Tipo& dato) : dato(dato), siguiente(0) { }
  };

  Nodo *primero;        // Puntero al primer elemento de la lista
  Nodo *ultimo;       // Puntero al último elemento de la lista
  int largo;          // Tamaño que representa la cantidad de
                // elementos que contiene la lista

public:

  // Constructor
  ListaRef();

  // Destructor
  ~ListaRef();

  // Verifica si una lista se encuentra vacía.
  // POST: Devuelve verdadero si la lista se encuentra vacía o falso en
  // caso contrario.
  bool estaVacia();

  // Devuelve el tamaño actual de la lista.
  int tamanio();

  // Inserta un elemento al principio de la lista.
  // PRE: 'dato' es el dato a insertar.
  void insertarPrimero(Tipo& dato);

  // Inserta un elemento en el último lugar de la lista.
  // PRE: 'dato' es el dato a insertar.
  void insertarUltimo(Tipo& dato);

  // Obtiene el valor del primer elemento de la lista.
  // POST: se devuelve el dato que se encuentra primero en la lista.
  Tipo& verPrimero();

  // Obtiene el valor del último elemento de la lista.
```

```
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
89  // Constructor
90  template <typename Tipo >
91  ListaRef< Tipo >::ListaRef() {
92    this→primero = 0;
93    this→ultimo = 0;
94    this→largo = 0;
95  }
96
97
98  // Destructor
99  template <typename Tipo >
100 ListaRef< Tipo >::~ListaRef() {
101   Nodo *nodo;
102
103   // Recorremos los nodos y los destruimos
104   while(this→primero) {
105     nodo = this→primero;
106     this→primero = nodo→siguiente;
107     delete nodo;
108   }
109 }
110
111
112 // Verifica si una lista se encuentra vacía.
113 // POST: Devuelve verdadero si la lista se encuentra vacía o falso en
114 // caso contrario.
115 template <typename Tipo >
116 bool ListaRef< Tipo >::estaVacia() {
117   return (this→largo ≡ 0);
118 }
119
120
121 // Devuelve el tamaño actual de la lista.
122 template <typename Tipo >
123 int ListaRef< Tipo >::tamanio() {
124   return this→largo;
125 }
126
127
128 // Inserta un elemento al principio de la lista.
129 // PRE: 'dato' es el dato a insertar.
130 template <typename Tipo >
131 void ListaRef< Tipo >::insertarPrimero(Tipo& dato) {
132   // Creamos un nuevo nodo
```



```
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
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 ListaRef< 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
170 // Obtiene el valor del primer elemento de la lista.
171 // POST: se devuelve el dato que se encuentra primero en la lista.
172 template <typename Tipo >
173 Tipo& ListaRef< Tipo >::verPrimero() {
174   return (this→primero→dato);
175 }
176
177
178 // Obtiene el valor del último elemento de la lista.
179 // POST: se devuelve el dato que se encuentra último en la lista.
180 template <typename Tipo >
181 Tipo& ListaRef< Tipo >::verUltimo() {
182   return (this→ultimo→dato);
183 }
184
185
186 // Elimina el primer elemento de la lista.
187 // POST: se retorna el elemento eliminado de la lista.
188 template <typename Tipo >
189 void ListaRef< Tipo >::eliminarPrimero() {
190   // Tomamos el nodo a borrar
191   Nodo *nodo = this→primero;
192
193   // El segundo elemento pasa a ser el primero
194   this→primero = this→primero→siguiente;
195   this→largo--;
196
197   // Liberamos la memoria usada por el nodo.
198   delete nodo;
```

```
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
```



```
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 estaVacia();
50
51   // Devuelve el tamaño actual de la lista.
52   int tamanio();
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.
```

```
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
89  // Constructor
90  template <typename Tipo >
91  Lista< Tipo >::Lista() {
92    this→primero = 0;
93    this→ultimo = 0;
94    this→largo = 0;
95  }
96
97
98  // Destructor
99  template <typename Tipo >
100 Lista< Tipo >::~Lista() {
101   Nodo *nodo;
102
103   // Recorremos los nodos y los destruimos
104   while(this→primero) {
105     nodo = this→primero;
106     this→primero = nodo→siguiente;
107     delete nodo→dato;
108     delete nodo;
109   }
110 }
111
112
113 // Verifica si una lista se encuentra vacía.
114 // POST: Devuelve verdadero si la lista se encuentra vacía o falso en
115 // caso contrario.
116 template <typename Tipo >
117 bool Lista< Tipo >::estaVacia() {
118   return (this→largo ≡ 0);
119 }
120
121
122 // Devuelve el tamaño actual de la lista.
123 template <typename Tipo >
124 int Lista< Tipo >::tamanio() {
125   return this→largo;
126 }
127
128
129 // Inserta un elemento al principio de la lista.
130 // PRE: 'dato' es el dato a insertar.
131 template <typename Tipo >
132 void Lista< Tipo >::insertarPrimero(Tipo *dato) {
```



```
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
147 // Inserta un elemento en el último lugar de la lista.
148 // PRE: 'dato' es el dato a insertar.
149 template <typename Tipo >
150 void Lista< Tipo >::insertarUltimo(Tipo *dato) {
151   // Creamos un nuevo nodo
152   Nodo *nodo = new Nodo(dato);
153
154   // Seteamos los campos del nodo
155   nodo→dato = dato;
156   nodo→siguiente = 0;
157
158   // Si no hay elementos, el último también es el primero
159   if(¬this→ultimo)
160     this→primero = nodo;
161   // Sino, insertamos el nuevo nodo a continuación del que
162   // se encontraba último
163   else
164     this→ultimo→siguiente = nodo;
165
166   this→ultimo = nodo;
167   this→largo++;
168 }
169
170
171 // Obtiene el valor del primer elemento de la lista.
172 // POST: se devuelve el dato que se encuentra primero en la lista.
173 template <typename Tipo >
174 Tipo* Lista< Tipo >::verPrimero() {
175   return (this→primero→dato);
176 }
177
178
179 // Obtiene el valor del último elemento de la lista.
180 // POST: se devuelve el dato que se encuentra último en la lista.
181 template <typename Tipo >
182 Tipo* Lista< Tipo >::verUltimo() {
183   return (this→ultimo→dato);
184 }
185
186
187 // Elimina el primer elemento de la lista.
188 // POST: se retorna el elemento eliminado de la lista.
189 template <typename Tipo >
190 Tipo* Lista< Tipo >::eliminarPrimero() {
191   // Tomamos el nodo a borrar
192   Nodo *nodo = this→primero;
193   Tipo *dato = this→primero→dato;
194
195   // El segundo elemento pasa a ser el primero
196   this→primero = this→primero→siguiente;
197   this→largo--;
198
```

```
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
```