Prácticas de Autómatas y Lenguajes. Curso 2017/18
Práctica 1: configuración de autómata a pila no determinista

Descripción del enunciado

Esta práctica tiene dos objetivos principales:

1. Que desarrolles con el lenguaje de programación C los tipos de datos necesarios para implementar una configuración de un autómata a pila (potencialmente no determinista)
2. Que esa librería esté preparada para integrarse en el futuro en un simulador de autómatas a pila.

Descripciones instaneas de autómatas a pila vs configuración no determinista.

En esta práctica describiremos el estado de cualquier autómata en un momento determinado mediante descripciones instantáneas. Habrás visto en la parte de teoría que una descripción instantánea de un autómata a pila es una terna:

- Estado
- PIla
- Cadena pendiente de procesar.

Cuando se simula el funcionamiento de un autómata a pila potencialmente no determinista se debe poder guardar el conjunto de configuraciones activo en cada momento.

Por lo tanto, la configuración no determinista de un autómata a pila será una colección de las descripciones instantáneas activas en un momento determinado.

En esta parte de la práctica se te pide que implementes

- Un tipo de dato para la descripción instantánea
- Un tipo de dato (colección genérica) para implementar la configuración no determinista del autómata mediante todas las descripciones instantáneas del instante en consideración.

A continuación se describe con más detalle cada uno de estos tipos de dato.

Configuración de autómata a pila no determinista (ConfiguracionApnd)

Puedes basarte en cualquier colección genérica que hayas implementado previamente aunque el tipo de dato que le corresponde es el conjunto (set). Recuerda que un conjunto es una colección de elementos del mismo tipo en el que el orden en el que están dispuestos no tiene importancia y en el que cada elemento sólo puede estar una vez.

Las consideraciones generales para su implementación son análogas a las que se especificaron para la pila (stack)

La funcionalidad básica de este tipo de datos debe proporcionar mecanismos para

- Crear una configuración no determinista nueva.
- Insertar una configuración determinista (asegurándote de que no hay duplicados).
- Extraer una de ellas (no es relevante en qué orden)
- Mostrar (imprimir a un FILE *) la configuración no determinista.
- Destruir la configuración no determinista.
- Indicar si una configuración no determinista está vacía.

A continuación especificamos las cabeceras de estas funciones

```
ConfiguracionApnd * configuracionApndIni( );
/* Inicializa una configuración no determinista */
```

```
ConfiguracionApnd* configuracionApndInsert(ConfiguracionApnd* capnd, const
ConfiguracionAp * p_cap);
/* Inserta una configuración determinista en una no determinista. Se realiza una
copia en memoria propia de la colección para el nuevo elemento y se asegura que
no haya duplicados*/

ConfiguracionAp * configuracionApndExtract(ConfiguracionApnd * capnd);
/* Se obtiene una configuración determinista de la no determinista, que
desaparece de la colección.
No se especifica el orden en el que se extrae */

int configuracionApndIsEmpty(const ConfiguracionApnd* capnd);
/* Se devuelve 1 si está vacía y 0 en caso contrario */

int configuracionApndSize(const ConfiguracionApnd * capnd);
/* Se devuelve el número de configuraciones deterministas que hay dentro de la no
determinista */

int configuracionApndPrint(FILE *fd, const ConfiguracionApnd* capnd);
/* Se imprime todas las configuraciones deterministas. No se especifica en qué
orden */

void configuracionApndDestroy( ConfiguracionApnd* capnd);
/* Se libera toda la memoria asociada con la configuracion no determinista */
```

Configuración determinista (descripción instantánea)

Una configuración (determinista) de un autómata a pila no determinista cualquiera se puede representar mediante su descripción instantánea, que es la combinación del estado (sólo uno) en el que se puede encontrar, el contenido de la pila y la cadena de entrada que le queda aún por procesar.

```
typedef struct _ConfiguracionAp ConfiguracionAp;

ConfiguracionAp * configuracionApNueva( Estado  * estado, Stack * pila, Palabra *
cadena);
/* Se crea una configuración reservando memoria nueva para ella y para copiar
todas las componentes que se proporcionan como argumento */

void configuracionApElimina( ConfiguracionAp * p_cap);
/* Se libera toda la memoria asociada con la configuracion */

void configuracionApImprime( FILE * fd, ConfiguracionAp * p_cap);
/* Se muestra por pantalla la configuración */

ConfiguracionAp * configuracionApCopia( ConfiguracionAp * p_cap);
/* Se devuelve una copia en memoria nueva de la configuración */

int configuracionCompara( ConfiguracionAp * p_cap1, ConfiguracionAp * p_cap2);
/* Se utiliza para comparar dos configuraciones deterministas, devolviendo un
valor negativo, 0 o positivo en función de la comparación de sus componentes en
este orden: estados, pilas, cadenas. En el caso de que no sean iguales, devuelve
el valor que devuelva la comparación de la primera que sea distinta*/
```

Se te recomienda que implementes también los siguientes tipos de datos auxiliares.

Estado

Es necesario guardar los estados y la información mínima que los describe ha de ser

- Un char * para el nombre
- El tipo de estado (codificado en un entero)
    - Inicial
    - Final
    - Inicial y final
    - Normal (ninguno de los anteriores)

Si implementas todas las funciones siguientes mantén las cabeceras que se muestran a continuación:

```
#define INICIAL 0
#define FINAL   1
#define INICIAL_Y_FINAL 2
#define NORMAL 3



typedef struct _Estado Estado;

Estado * estadoNuevo( char * nombre, int tipo);
/* Reserva memoria nueva para el estado y para la cadena de su nombre, crea un
estado con las dos componentes que se le proporcionan como argumento */

void estadoElimina( Estado * p_s);
/* Libera la memoria asociada con el estado */

void estadoImprime( FILE * fd, Estado * p_s);
/* Muestra por el FILE * el estado */

int estadoEs(Estado * p_s, char * nombre);
/* Función de comparación de estados por nombre, devuelve 1 si el estado tiene el
nombre proporcionado */

char * estadoNombre(Estado * p_s);
/* Devuelve (un puntero) al nombre del estado */

int estadoTipo(Estado * p_s);
/* Devuelve el tipo de estado */


int estadoCompara(Estado * p_s1, Estado * p_s2);
/* Se usa para comparar estados por su nombre y devuelve el strcmp de sus nombres
*/



Estado * estado_copy(Estado * p_s);
/* Se crea una copia en memoria nueva del estado proporcionado como argumento */
```

Palabra

Implementa la cadena de símbolos de entrada que el autómata debe procesar.

Ten en cuenta que los símbolos deben poder tener como nombre cualquier cadena de caracteres.

Debes decidir qué tipo de dato C utilizaras para guardar “la colección” de char * (cada una de ellas será una letra)

A continuación se describen las funciones típicas de este tipo de dato.

```
typedef struct _Palabra Palabra;


Palabra * palabraNueva();
/* Crea una palabra nueva */

void palabraElimina(Palabra * p_p);
/* Libera la memoria asociada con la palabra */

void palabraImprime(FILE * fd, Palabra * p_p);
/* Muestra por el FILE * la palabra */

Palabra * palabraInsertaLetra(Palabra * p_p, char * letra);
/* Inserta una letra (que es un string - char * - ) en la palabra proporcionada
como argumento (modificándola, por tanto) y devuelve la palabra resultante. Para
ello debe hacer una copia en memoria nueva para la nueva letra */

int palabraTamano(Palabra * p_p);
/* Devuelve la longitud de la palabra */


Palabra * palabraCopia (Palabra * p_p);
/* Hace en memoria nueva una copia de la palabra y la devuelve */

int palabraCompara( Palabra * p_p1, Palabra * p_p2);
/* Para comparar dos palabras, devuelve un valor negativo, cero o positivo en
función de que todas las letras de la misma posición y de izquierda a derecha
sean iguales. En el caso de que no lo sean, se devuelve el strcmp de la primera
pareja de letras que no sean iguales */
```

Pila

Debes utilizar la pila que has desarrollado en la parte anterior de esta práctica.
Sin embargo será necesario que realices los siguientes cambios.

- A los punteros a función que almacene la pila, hay que añadir una función para comparar los elementos de la pila

```
    destroy_element_function_type   destroy_element_function;
    copy_element_function_type      copy_element_function;
    print_element_function_type     print_element_function;
    cmp_element_function_type     cmp_element_function;
```

- Y, por lo tanto, también debes modificar la cabecera de la función de inicialización

```
Stack * stack_ini( destroy_element_function_type f1,
                   copy_element_function_type f2,
                   print_element_function_type f3,
                   cmp_element_function_type f4);
```

Ejemplos

A modo de ejemplo, tu librería debería ser compatible con el siguiente programa principal

```
#include <string.h>
#include <stdio.h>
#include "configuracion_ap.h"
#include "estado.h"
#include "stack.h"
#include "palabra.h"
#include "generic_string.h"
#include "dynamic_node.h"
#include "configuracion_apnd.h"



int main(int argc, char ** argv)
{
	ConfiguracionAp * p_cap;
	ConfiguracionAp * p_cap2;
	ConfiguracionAp * p_cap_aux;
	char texto[1024];

	ConfiguracionApnd * capnd;

	Stack * pila;
	Estado * estado;
	Palabra * cadena;



	capnd = configuracionApndIni ();



	cadena = palabraNueva();
	palabraInsertaLetra(cadena,"a1");
	palabraInsertaLetra(cadena,"a2");
	palabraInsertaLetra(cadena,"a3");
	palabraInsertaLetra(cadena,"a4");


	pila = stack_ini( destroy_p_string, copy_p_string,
	                  print_p_string,(cmp_element_function_type)strcmp);
	sprintf(texto,"z");
	stack_push(pila,texto);
	sprintf(texto,"a");
	stack_push(pila,texto);
	sprintf(texto,"b");
	stack_push(pila,texto);

	estado = estadoNuevo("q1",2);

	p_cap = configuracionApNueva(estado,pila,cadena);

	p_cap2 = configuracionApCopia(p_cap);



	configuracionApndInsert(capnd,p_cap);
	configuracionApndInsert(capnd,p_cap2);
```

```
	fprintf(stdout,"\nCONFIGURACION 1\n");
	configuracionApImprime(stdout,p_cap);
	configuracionApElimina(p_cap);

	estadoElimina(estado);
	stack_destroy(pila);
	palabraElimina(cadena);

	fprintf(stdout,"\nCONFIGURACION 2\n");
	configuracionApImprime(stdout,p_cap2);
	configuracionApElimina(p_cap2);



	fprintf(stdout,"\nCONFIGURACION NO DETERMINISTA\n");
	configuracionApndPrint(stdout,capnd);

	fprintf(stdout,"\nCONFIGURACION NO DETERMINISTA, IMPRESA ELEMENTO A
ELEMENTO\n");
	while( configuracionApndIsEmpty(capnd) != 1 )
	{
		p_cap_aux = configuracionApndExtract(capnd);
		fprintf(stdout,"\nELEMENTO\t");
		configuracionApImprime(stdout,p_cap_aux);
		configuracionApElimina(p_cap_aux);
	}

	configuracionApndDestroy(capnd);

	return 0;
}
```

Este programa en concreto genera la siguiente salida (puedes tomar este ejemplo para determinar el formato de salida de las funciones que imprimen cada tipo de dato.

```
CONFIGURACION 1
(->q1*  <b a z > [(4) a1 a2 a3 a4])
CONFIGURACION 2
(->q1*  <b a z > [(4) a1 a2 a3 a4])
CONFIGURACION NO DETERMINISTA
Lista con 1 elementos:
(->q1*  <b a z > [(4) a1 a2 a3 a4])

CONFIGURACION NO DETERMINISTA, IMPRESA ELEMENTO A ELEMENTO

ELEMENTO	(->q1*  <b a z > [(4) a1 a2 a3 a4])
```

Observa que aunque se ha insertado dos veces la misma configuración determinista (descripción instantánea) en la no determinista sólo aparece una vez.

En esta práctica se pide:

- Cada grupo debe realizar una entrega
- Tu profesor te indicará la tarea Moodle donde debes hacer la entrega.
- Es imprescindible que mantengas los nombres especificados en este enunciado **sólo para las funciones que se piden de manera obligatoria**.
- Cualquier práctica que no compile (con los flags de compilación -Wall -ansi -pedantic), o no ejecute correctamente será considerada como no entregada.
- Cualquier práctica que deje "lagunas de memoria" será penalizada de manera explícita en función de la gravedad de esas lagunas.