SOLUCIONES

Parte I: PREGUNTAS TIPO TEST. 30 %.

1. c)
2. c)
3. c)
4. b)
5. a)
6. b)
7. c)
8. b)
9. b)
10. c)
11. c)
12. a)
13. c)
14. b)
15. b)

Parte II: PREGUNTAS CORTAS. 15 %.

Apartado 1.

La regla $E' \rightarrow \lambda$ aparece en las casillas $\{E',]\}$ y $\{E', \$\}$ porque tanto] como \$ pertenecen al conjunto *Predict* de dicha regla. Como el lado derecho de la regla es λ, el conjunto *Predict* de la regla coincide con el conjunto SIGUIENTE(E'). A su vez, dicho conjunto coincide con SIGUIENTE(E) por la regla $E \rightarrow id\ E'$. Como E es el símbolo inicial de la gramática, \$ $\in$ SIGUIENTE(E). Por otra parte, por la regla $E \rightarrow [\ E\ ]\ E'$, se tiene que [$\in$ SIGUIENTE(E).

Apartado 2.

La casilla $\{E,]\}$ corresponde a la situación en la que se espera encontrar un token *id* que permita expandir el no terminal E, y en su lugar se encuentra un token]. Esta situación se puede producir si la cadena de entrada contiene la secuencia de tokens [ ] si ninguna expresión (derivada de E) correcta en el interior de los corchetes.

El método de recuperación a *nivel de frase* implica, por un lado, la generación de un mensaje de error en el que se indique al usuario el fallo que ha cometido. Y en segundo lugar, el análisis debe continuar resolviendo la configuración errónea. Para ello, teniendo en cuenta que [$\in SIGUIENTE(E)$, podemos extraer de la pila el no terminal E y continuar el análisis como si se hubiese reconocido una expresión. Por tanto, la rutina podría ser la siguiente:

```
void error_falta_E() {
        printf("Error: se esperaba una expresión antes de ']'\n");
        extraer_de_pila('E');
        continuar_analisis();
}
```

Apartado 3.

A continuación se muestra la simulación de la cadena de entrada $id\hat{}.id[\ ]\$$, usando recuperación de errores a nivel de frase al detectar el error en la cadena de entrada:

Pila	Entrada	Salida
$\$\ E$	$id\ \hat{}\ .\ id\ [\ ]\ \$$	
$\$\ E'\ id$	$id\ \hat{}\ .\ id\ [\ ]\ \$$	$E \to\ id\ E'$
$\$\ E'$	$\hat{}\ .\ id\ [\ ]\ \$$	
$\$\ E'\ \hat{}$	$\hat{}\ .\ id\ [\ ]\ \$$	$E' \to\ \hat{}E'$
$\$\ E'$	$.\ id\ [\ ]\ \$$	
$\$\ E'\ id\ .$	$.\ id\ [\ ]\ \$$	$E' \to\ .\ id\ E'$
$\$\ E'\ id$	$id\ [\ ]\ \$$	
$\$\ E'$	$[\ ]\ \$$	
$\$\ E'\ ]\ E\ [$	$[\ ]\ \$$	$E' \to\ [\ E\ ]\ E'$
$\$\ E'\ ]\ E$	$]\ \$$	Error: se esperaba una expresión antes de ']'
$\$\ E'\ ]$	$]\ \$$	
$\$\ E'$	$\$$	
$\$$	$\$$	$E' \to\ \lambda$
$\$$	$\$$	Aceptar

Parte III: PROBLEMA. 55 %

Apartado 1.

La gramática no es LL(1) porque es recursiva por la izquierda debido a la regla $LP \to LP\ ;\ num$. La recursividad por la izquierda provoca siempre conflictos en la función *Predict* del no terminal afectado.

Apartado 2.

Los conjuntos PRIMERO y SIGUIENTE de la gramática son los siguientes:

PRIMERO$(S) = \{\ num\ \}$
PRIMERO$(LP) = \{\ num\ \}$
PRIMERO$(E) = \{\ num,\ var,\ fun\ \}$

SIGUIENTE$(S) = \{\ \$\ \}$
SIGUIENTE$(LP) = \{\ ;\ \}$
SIGUIENTE$(E) = \{\ ),\ \$\}$

Apartado 3.

Los conjuntos de ítems de la colección LR(0) son los siguientes:

$$I_0 = \{\ S' \to \cdot\ S \quad S \to \cdot\ LP\ ;\ E \quad LP \to \cdot\ LP\ ;\ num \quad LP \to \cdot\ num\ \}$$

$$I_1 = \text{Goto}(I_0, S) = \{\ S' \to S \cdot\ \}$$

$$I_2 = \text{Goto}(I_0, LP) = \{\ S \to LP \cdot\ ;\ E \quad LP \to LP \cdot\ ;\ num\ \}$$

$$I_3 = \text{Goto}(I_0, num) = \{\ LP \to num \cdot\ \}$$

$$I_4 = \text{Goto}(I_2, ;) = \{\ S \to LP\ ;\ \cdot\ E \quad LP \to LP\ ;\ \cdot\ num \quad E \to \cdot\ num \quad E \to \cdot\ var \quad E \to \cdot\ fun\ (\ E\ )\ \}$$

$$I_5 = \text{Goto}(I_4, E) = \{\ S \to LP\ ;\ E \cdot\ \}$$

$$I_6 = \text{Goto}(I_4, num) = \{\ LP \to LP\ ;\ num \cdot \quad E \to num \cdot\ \}$$

$$I_7 = \text{Goto}(I_4, var) = \{\ E \to var \cdot\ \}$$

$$I_8 = \text{Goto}(I_4, fun) = \{\ E \to fun\ \cdot\ (\ E\ )\ \}$$

$$I_9 = \text{Goto}(I_8, (\ ) = \{\ E \to fun\ (\ \cdot\ E\ ) \quad E \to \cdot\ num \quad E \to \cdot\ var \quad E \to \cdot\ fun\ (\ E\ )\ \}$$

$$I_{10} = \text{Goto}(I_9, E) = \{\ E \to fun\ (\ E \cdot\ )\ \}$$

$$I_{11} = \text{Goto}(I_9, num) = \{\ E \to num \cdot\ \}$$

$$\text{Goto}(I_9, var) = I_7$$

$$\text{Goto}(I_9, fun) = I_8$$

$$I_{12} = \text{Goto}(I_{10},)\ ) = \{\ E \to fun\ (\ E\ ) \cdot\ \}$$

La tabla de análisis es la siguiente:

Estado	Acción							Ir-a		
	(	)	;	*fun*	*num*	*var*	$	*E*	*LP*	*S*
0					d3				2	1
1							acc			
2			d4							
3			r3							
4				d8	d6	d7		5		
5							r1			
6		r4	r2				r4			
7		r5					r5			
8	d9									
9				d8	d11	d7		10		
10		d12								
11		r4					r4			
12		r6					r6			

La gramática es SLR, ya que la tabla no contiene ningún conflicto.

Apartado 4.

Al ser SLR, la gramática es también LALR y LR-Canónica.

Apartado 5.

Emplearemos los siguientes atributos:

Símbolo	**Atributo**	**Tipo**	**Comentario**
num	val	`float`	Valor numérico del token.
fun	texto	`char[]`	Cadena de texto de la función.
var	texto	`char[]`	Cadena de texto de la variable.
E	puntos	`float[]`	Lista con los puntos en los que hay que evaluar la derivada de la expresión.
	val	`float[]`	Lista con la evaluación de la expresión sin derivar en los puntos indicados.
	valder	`float[]`	Lista con la evaluación de la derivada de la expresión en los puntos indicados.
	deriv	`char[]`	Cadena de texto con la derivada de la expresión.
	texto	`char[]`	Cadena de texto con la expresión sin derivar.
LP	puntos	`lista`	Lista con los puntos en los que hay que evaluar la derivada.

La definición dirigida por la sintaxis (DDS) para obtener la información solicitada puede ser la siguiente:

Regla de producción	**Acción**

$S \to LP\ ;\ E$	E.puntos = LP.puntos;
	for (int i=0; i<size(E.valder); i++) {
	  print(E.valder[i]+";");
	}
	print $E.deriv$;

$LP \to LP_1\ ;\ num$	LP.puntos = LP_1.puntos; LP.puntos[size(LP_1.puntos)] = num.val;
$LP \to num$	LP.puntos[0] = num.val;

$E \to num$	E.deriv = "0";
	E.texto = float_to_text(num.val);
	for (int i=0; i<size(E.puntos); i++) E.val[i] = num.val;
	for (int i=0; i<size(E.puntos); i++) E.valder[i] = 0;

$E \to var$	E.deriv = "1";
	E.texto = var.texto;
	for (int i=0; i<size(E.puntos); i++) E.val[i] = E.puntos[i];
	for (int i=0; i<size(E.puntos); i++) E.valder[i] = 1;

(continúa en la siguiente hoja)

Regla de producción	Acción

$E \rightarrow fun$ (E_1)	E_1.puntos = E.puntos;
	E.texto = fun.texto + "(" + E_1.texto + ")";
	if (fun.texto == "sin") {
	  E.deriv = "cos(" + E_1.texto + ")*" + E_1.deriv;
	  for (int i=0; i<size(E_1.val); i++) {
	  E.val[i] = sin(E_1.val[i]);
	  E.valder[i] = cos(E_1.val[i])*E_1.valder[i];
	  }
	}
	if (fun.texto == "cos") {
	  E.deriv = "-sin(" + E_1.texto + ")*" + E_1.deriv;
	  for (int i=0; i<size(E_1.val); i++) {
	  E.val[i] = cos(E_1.val[i]);
	  E.valder[i] = -sin(E_1.val[i])*E_1.valder[i];
	  }
	}

En el pseudocódigo de la definición dirigida por la sintaxis se tienen en cuenta las siguientes consideraciones:

- Las listas son tratadas como arrays, con índice 0 en su primer elemento, y no se considera nada relativo a su tamaño máximo.
- Existe una función `size` aplicable a las listas que devuelve la cantidad de entradas utilizadas.
- Existe una función `float_to_text` que convierte un número flotante a una cadena de caracteres.
- El operador + se emplea como concatenación de cadenas de caracteres.

El atributo `puntos` del no terminal E es heredado. Los demás atributos empleados son sintetizados. Se trata, por tanto, de una gramática L-atribuida.

El árbol sintáctico decorado para la entrada `0,1 ; 0,2 ; 0,3 ; sin(sin(x))` es el siguiente (valores en radianes):