UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

INGENIERÍA EN COMPUTACIÓN

COMPILADORES

GRUPO 01

PROYECTO 02

ANALIZADOR LÉXICO Y SINTÁCTICO DESCENDENTE RECURSIVO

SEMESTRE 2024-2

PROFESORA: Elba Karen Saenz García

INTEGRANTES: Barragán Pilar Diana

Ramírez Martínez Valeria

Silverio Martínez Andrés

FECHA DE ENTREGA 13 / MAYO / 2024

ÍNDICE

OBJETIVO

Construir los analizadores Léxico y Sintáctico Descendente Recursivo para que revisen programas escritos en el lenguaje definido por la gramática dada.

INTRODUCCIÓN

DEFINICIÓN

El analizador léxico se encarga de buscar los componentes léxicos o palabras que componen el programa fuente, según unas reglas o patrones, la entrada del analizador léxico podemos definirla como una secuencia de caracteres.

CARACTERÍSTICAS

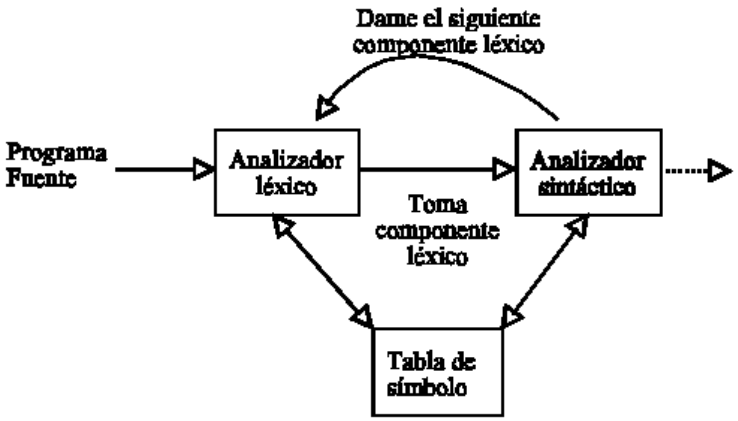
- Lee caracteres.
- Produce componentes léxicos (tokens).
- Filtra comentarios.
- Filtra separadores múltiples (espacios, tabuladores y saltos de línea).
- Lleva el contador de línea y columna del texto fuente.
- Genera errores en caso de que la entrada no corresponda a ninguna categoría léxica.

FUNCIÓN

El analizador es la primera fase de un compilador, su principal función consiste en leer los caracteres de entrada y elaborar como salida una secuencia de componentes léxicos que utiliza el analizador sintáctico para hacer el análisis. Esta interacción, suele aplicarse convirtiendo al analizador léxico en una subrutina o corrutina del analizador sintáctico.

Otras funciones que realizan se listan a continuación:

- Eliminar los comentarios del programa.
- Eliminar espacios en blanco, tabuladores y en general todo aquello que carezca de significado según la sintaxis del lenguaje.
- Reconocer los identificadores de usuario, números, palabras reservadas del lenguaje y tratarlos correctamente con respecto a la tabla de símbolos.
- Puede hacer funciones de pre-procesador.

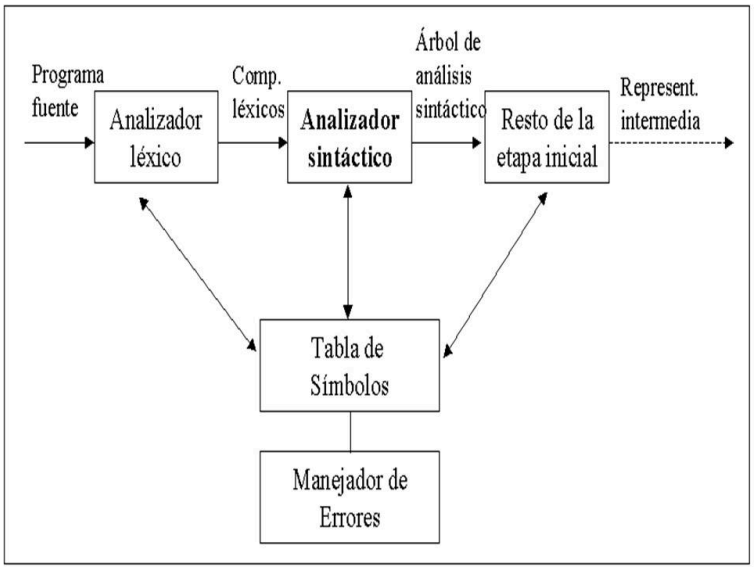


DEFINICIÓN

Un analizador sintáctico, también conocido como Parser, es una parte fundamental de un compilador o intérprete de lenguaje de programación. El analizador sintáctico verifica si la secuencia de tokens (símbolos) generada por el analizador léxico (que divide el código en tokens) cumple con la gramática del lenguaje. Existen diferentes tipos de analizadores sintácticos, como los analizadores sintácticos descendentes (top-down) y los analizadores sintácticos ascendentes (bottom-up), cada uno con sus propias técnicas y algoritmos para realizar el análisis sintáctico.

FUNCIONES

- Obtener una cadena de componentes léxicos de A.L y comprobar si la cadena puede ser generada por la gramática del lenguaje fuente.
- Informar los errores sintácticos en forma precisa y significativa. Deberá ser dotado de un mecanismo de recuperación de errores para continuar con el análisis.



ANÁLISIS SINTÁCTICO DESCENDENTE

Se puede considerar como un intento de encontrar una derivación más a la izquierda para la cadena de entrada. También puede considerarse como un intento por construir un árbol de análisis sintáctico para la entrada comenzando desde la raíz y creando los nodos del árbol en orden previo.

- ANÁLISIS SINTÁCTICO DESCENDENTE RECURSIVO
  - Usa retroceso para resolver la incertidumbre
  - Sencillo de implementar
  - Muy ineficiente

DESCRIPCIÓN DEL PROBLEMA

- Ambos analizadores estarán en el mismo programa, esto es, se utilizará el proyecto 1 y se agregará el analizador sintáctico recursivo como se indica en el anexo B.
- Así la entrada es un archivo con el programa fuente a analizar que deberá estar escrito en el lenguaje definido por la gramática dada en el Anexo A de este documento. Este archivo de entrada se indicará desde la línea de comandos.
- El analizador léxico deberá generar además de los tokens, la cadena de átomos que será la entrada del analizador sintáctico. Los átomos se pueden ir generando al mismo tiempo que los tokens y almacenarlos en una sola cadena.
- Los átomos están definidos en este documento por cada componente léxico y corresponden a los elementos terminales de la gramática.
- La tabla de clases de componentes léxicos con sus correspondientes átomos es:

Clase	Descripción	átomo
0	Palabras reservadas (ver tabla).	(ver tabla)
1	Identificadores. Iniciar con $ y le sigue al menos una letra minúscula o mayúscula. Ejemplos: $ejemplo, $Varible, $OtraVariable, $XYZ	i
2	Constantes numéricas enteras. En base 10 (secuencia de dígitos del 0-9 sin 0's a la izquierda, excepto para el número 0), en base 8 (inicien con O u o y le sigan dígitos del 0 al 7).	n
3	Constantes numéricas reales. Siempre deben llevar parte decimal y es opcional la parte entera. Ejemplos: 73.0, .0, 10.2 No aceptados: . , 12 , 4.	r
4	Constantes cadenas. Encerrado entre comillas (“) cualquier secuencia de más de un carácter que no contenga “ ni ‘. Para cadenas de un solo carácter, encerrarlo entre apóstrofos (‘). La cadena de unas comillas debe ser encerrada entre apóstrofos: ‘“’ . La cadena de un apóstrofo debe ser encerrada por comillas: “‘” . No se aceptan cadenas vacías. Ejemplos NO válidos: “ejemplo no “valido” , “” , “”” , “hola ‘mundo”	s
5	Símbolos especiales [ ] ( ){ } , : ;	mismo símbolo
6	Operadores aritméticos + - * / % \ ^	mismo símbolo
7	Operadores relacionales (ver tabla).	(ver tabla)
8	Operador de asignación =	=

- El valor en los tokens y los átomos se indican en las siguientes tablas.

Valor	Palabra reservada	Equivale en C	átomo
0	alternative	case	a
1	big	long	b
2	evaluate	if	f
3	instead	else	t
4	large	double	g
5	loop	while	w
6	make	do	m
7	number	int	#
8	other	default	o
9	real	float	x
10	repeat	for	j
11	select	switch	h
12	small	short	p
13	step	continue	c
14	stop	break	q
15	symbol	char	y
16	throw	return	z

Valor	Op. relacional	átomo
0	<	<
1	>	>
2	<=	l
3	>=	u
4	==	e
5	!=	d

- El analizador sintáctico deberá mostrar todos los errores sintácticos que encuentre, indicando qué se esperaba.
- Como resultados, el analizador léxico-sintáctico deberá mostrar el contenido de la tabla de símbolos, las tablas de literales, los tokens y la cadena de átomos. Finalmente deberá indicar si está sintácticamente correcto el programa fuente.
- Los errores que vaya encontrando el analizador léxico, los podrá ir mostrando en pantalla o escribirlos en un archivo, así como él o los errores sintácticos. Es conveniente que cuando encuentre un error sintáctico se indique en qué átomo de la cadena se encontró (con ubicación).
- El programa deberá estar comentado, con una descripción breve de lo que hace (puede ser el objetivo indicado en este documento), el nombre de quienes elaboraron el programa y fecha de elaboración.

DESARROLLO

PROPUESTA DE SOLUCIÓN Y FASES DEL DESARROLLO DEL SISTEMA

ANÁLISIS

Para llevar a cabo este proyecto, fue esencial realizar un exhaustivo análisis de los requerimientos del nuevo programa. Esta tarea fundamental fue llevada a cabo por todo el equipo, que colaboró en cada etapa del proceso. Este proceso involucró una cuidadosa investigación previa, que demandó retomar y aplicar los conceptos aprendidos en clase, así como combinarlos con nuestros conocimientos de programación. Exploramos detalladamente cada aspecto del proyecto, desde sus objetivos fundamentales hasta los posibles desafíos técnicos que podríamos enfrentar.

Este enfoque integrado nos permite desarrollar una estrategia sólida y bien fundamentada para la implementación del programa, asegurando así que cumplirá con las expectativas y necesidades esperadas.

DISEÑO E IMPLEMENTACIÓN

A partir del programa implementado que dió solución al problema planteado del proyecto anterior, se le realizaron las modificaciones pertinentes para integrar un analizador sintáctico, creando en conjunto, un analizador léxico - sintáctico. Primero, fue necesaria la implementación de varias funciones, que al momento de obtener un componente léxico, a su vez, se obtuviera el átomo de este, el cual es guardado en una cadena, que al final, se imprimirá mostrando la cadena de átomos resultante de los componentes léxicos generados.

Para ello, a aquellos componentes léxicos, los cuales tenían diferentes átomos para cada uno de estos, se crearon funciones específicas para poder definir sus átomos, en cuanto a aquellos componentes léxicos que, sin importar cuál era, su átomo se le asignó en la parte de las acciones, entonces, quedaron de la siguiente manera:

- Para las palabras reservadas:

```
char asignarAtomoPalRes(char *palabra) {
    if (strcmp(palabra, "case") == 0) {
        return 'a';
    } else if (strcmp(palabra, "long") == 0) {
        return 'b';
    } else if (strcmp(palabra, "if") == 0) {
        return 'f';
    } else if (strcmp(palabra, "else") == 0) {
        return 't';
    } else if (strcmp(palabra, "double") == 0) {
        return 'g';
    } else if (strcmp(palabra, "while") == 0) {
        return 'w';
    } else if (strcmp(palabra, "do") == 0) {
        return 'm';
    } else if (strcmp(palabra, "int") == 0) {
        return '#';
    } else if (strcmp(palabra, "default") == 0) {
        return 'o';
    } else if (strcmp(palabra, "float") == 0) {
        return 'x';
    } else if (strcmp(palabra, "for") == 0) {
        return 'j';
    } else if (strcmp(palabra, "switch") == 0) {
        return 'h';
    } else if (strcmp(palabra, "short") == 0) {
        return 'p';
    } else if (strcmp(palabra, "continue") == 0) {
        return 'c';
    } else if (strcmp(palabra, "break") == 0) {
        return 'q';
    } else if (strcmp(palabra, "char") == 0) {
        return 'y';
    } else if (strcmp(palabra, "return") == 0) {
        return 'z';
    } else {
        return 'N'; // Valor por defecto si no se encuentra la palabra
    }
```

- Para los identificadores

```
tokens[numTokens].atomo = 'i';
```

- Para las constantes numéricas enteras

```
tokens[numTokens].atomo = 'n'
```

- Para las constantes numéricas reales

```
tokens[numTokens].atomo = 'r';
```

- Para las constantes cadena

```
tokens[numTokens].atomo = 's';
```

- Para los símbolos especiales

```
char asignarAtomoSimEspe(const char* palabra){
    if (strcmp(palabra, "[") == 0) {
        return '[';
    } else if (strcmp(palabra, "]") == 0) {
        return ']';
    } else if (strcmp(palabra, "(") == 0) {
        return '(';
    } else if (strcmp(palabra, ")") == 0) {
        return ')';
    } else if (strcmp(palabra, "{") == 0) {
        return '{';
    } else if (strcmp(palabra, "}") == 0) {
        return '}';
    }else if (strcmp(palabra, ",") == 0) {
        return ',';
    } else if (strcmp(palabra, ":") == 0) {
        return ':';
    } else if (strcmp(palabra, ";") == 0) {
        return ';';
    } else {
        return 'N'; // Valor por defecto si no se encuentra la palabra
    }
```

- Para los operadores aritméticos

```
char asignarAtomoOpArit(const char* palabra) {
    if (strcmp(palabra, "+") == 0) {
        return '+';
    } else if (strcmp(palabra, "-") == 0) {
        return '-';
    } else if (strcmp(palabra, "*") == 0) {
        return '*';
    } else if (strcmp(palabra, "/") == 0) {
        return '/';
    } else if (strcmp(palabra, "%") == 0) {
        return '%';
    } else if (strcmp(palabra, "\\") == 0) { // Verifica la barra invertida correctamente
        return '\\';
    } else if (strcmp(palabra, "^") == 0) {
        return '^';
    } else {
        return 'N'; // Valor por defecto si no se encuentra la palabra
    }
```

- Para los operadores relacionales

```
char *asignarAtomoOpRel(const char* palabra) {
    if (strcmp(palabra, "<") == 0) {
        return "<";
    } else if (strcmp(palabra, ">") == 0) {
        return ">";
    } else if (strcmp(palabra, "<=") == 0) {
        return "<=";
    } else if (strcmp(palabra, ">=") == 0) {
        return ">=";
    } else if (strcmp(palabra, "==") == 0) {
        return "==";
    } else if (strcmp(palabra, "!=") == 0) {
        return "!=";
    } else {
        return "na"; // Valor por defecto si no se encuentra la palabra
    }
```

- Para el operador de asignación

```
tokens[numTokens].atomo = '=';
```

Y que dichos átomos se guardan en una misma cadena gracias a la siguiente función:

```
strcat(cadenaAtomos, &tokens[numTokens].atomo);
```

Haciendo que, del siguiente ejemplo (expresión relacional):

```
int $main(int $numero){
    int $numero;
    float $pi;
    int $resultado;

    $numero = 42;
    $pi = 3.14;
    $resultado = $numero * $pi + 10;

    if ($resultado > 50)
        return (1);
    else
        return (0);
    :
}
```

Su cadena de átomos sea:

```
#i(#i){#i;xi;#i;i=n;i=r;i=i*i+n;f(i>n)z(n);tz(n);:}
```

Donde, sí se hace el análisis de los átomos que debería de generar cada uno de los tokens generados, los hace correctamente, además, en la tabla de los tokens generados, se implementó la muestra de su átomos, como se muestra a continuación:

```
Tokens generados:
Clase | Valor | Info     | Atomo
----------------------
    0 | 7     | int      | #
    1 | 1     | $main    | i
    5 | (     | (        | (
    0 | 7     | int      | #
    1 | 2     | $numero          | i
    5 | )     | )        | )
    5 | {     | {        | {
    0 | 7     | int      | #
    1 | 2     | $numero          | i
    5 | ;     | ;        | ;
    0 | 9     | float    | x
    1 | 3     | $pi      | i
    5 | ;     | ;        | ;
    0 | 7     | int      | #
    1 | 4     | $resultado       | i
    5 | ;     | ;        | ;
    1 | 2     | $numero          | i
    8 | =     | =        | =
    2 | 1     | 42       | n
    5 | ;     | ;        | ;
    1 | 3     | $pi      | i
    8 | =     | =        | =
    3 | 2     | 3.14     | r
    5 | ;     | ;        | ;
    1 | 4     | $resultado       | i
    8 | =     | =        | =
    1 | 2     | $numero          | i
    6 | *     | *        | *
    1 | 3     | $pi      | i
    6 | +     | +        | +
    2 | 3     | 10       | n
    5 | ;     | ;        | ;
    0 | 2     | if       | f
    5 | (     | (        | (
    1 | 4     | $resultado       | i
    7 | 1     | >        | >
    2 | 4     | 50       | n
    5 | )     | )        | )
    0 | 16    | return   | z
    5 | (     | (        | (
    2 | 5     | 1        | n
    5 | )     | )        | )
    5 | ;     | ;        | ;
    0 | 3     | else     | t
    0 | 16    | return   | z
```

```
    5 | (     | (        | (
    2 | 6     | 0        | n
    5 | )     | )        | )
    5 | ;     | ;        | ;
    5 | :     | :        | :
    5 | }     | }        | }
```

Ahora bien, pasando al análisis sintáctico, en el programa se definieron las siguientes funciones para que lo pudiese realizar el análisis de manera correcta:

Para “Program()”

```
void Program(){
    printf("Program\n");
    if(next_atomo=='#' || next_atomo=='b' || next_atomo=='g' || next_atomo=='x' || next_atomo=='y'){
        Func();
        if(next_atomo=='}'&&(tokens[n+1].atomo=='#'||tokens[n+1].atomo=='b'||tokens[n+1].atomo=='g'||tokens[n+1].atomo=='x'||tokens[n+1].atomo=='y')){
            n++;
            next_atomo= tokens[n].atomo;
            otraFunc();
        }else if(next_atomo=='}'&& (n==numTokens-1)){

        }else{
            printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba otra funcion \n",next_atomo,n);
            exit(0);
        }
    }else{
        printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba un tipo de dato \n",next_atomo,n);
        exit(0);
    }
}
```

Para “otraFunc()”

```
void otraFunc(){
    //strcmp(palabra, ">") == 0
    if(next_atomo=='#' || next_atomo=='b' || next_atomo=='g' || next_atomo=='x' || next_atomo=='y'){
        Func();
    }
}
```

Para “Func()”

```
void Func(){
    printf("Func:  next_atomo %c",next_atomo);
    if(next_atomo=='#' || next_atomo=='b' || next_atomo=='g' || next_atomo=='x' || next_atomo=='y'){
        Tipo();
        if(next_atomo=='i'){
            n++;
            next_atomo= tokens[n].atomo;
            if(next_atomo=='('){
                n++;
                next_atomo= tokens[n].atomo;
                if(next_atomo=='#' || next_atomo=='b' || next_atomo=='g' || next_atomo=='x' || next_atomo=='y'){
                    Param();
                    if(next_atomo==')'){
                        n++;
                        next_atomo= tokens[n].atomo;
                        if(next_atomo=='{'){
                            n++;
                            next_atomo= tokens[n].atomo;
                            if(next_atomo=='#' || next_atomo=='b' || next_atomo=='g' || next_atomo=='x' || next_atomo=='y'){
                                Cuerpo();
                            }else if(next_atomo=='i'||next_atomo=='f'||next_atomo=='h'||next_atomo=='w'||next_atomo=='j'||next_atomo=='['||next_atomo=='z'||next_atomo=='c'){
                                Cuerpo();
                            }else{
                                printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba una declaracion o una lista de parametros\n",next_atomo,n);
                                exit(0);
                            }
                        }else{
                            printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba un { \n",next_atomo,n);
                            exit(0);
                        }
                    }else{
                        printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba un ) \n",next_atomo,n);
                        exit(0);
                    }
                }else if(next_atomo==')'){
                    n++;
                    next_atomo= tokens[n].atomo;
                    if(next_atomo=='{'){
                        n++;
                        next_atomo= tokens[n].atomo;
                        if(next_atomo=='{'){
                            n++;
                            next_atomo= tokens[n].atomo;
                            if(next_atomo=='#' || next_atomo=='b' || next_atomo=='g' || next_atomo=='x' || next_atomo=='y'){
                                Cuerpo();
                            }else if(next_atomo=='i'||next_atomo=='f'||next_atomo=='h'||next_atomo=='w'||next_atomo=='j'||next_atomo=='['||next_atomo=='z'||next_atomo=='c'){
                                Cuerpo();
                            }else{
                                printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba una declaracion o una lista de parametros\n",next_atomo,n);
                                exit(0);
                            }
                        }else{
                            printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba un { \n",next_atomo,n);
                            exit(0);
                        }
                    }else{
                        printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba un tipo de dato \n",next_atomo,n);
                        exit(0);
                    }
                }else{
                    printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba un ( \n",next_atomo,n);
                    exit(0);
                }
            }else{
                printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba un identificador \n",next_atomo,n);
                exit(0);
            }
        }
    }
```

Para “Tipo()”

```
void Tipo(){
    if(next_atomo=='#' || next_atomo=='b' || next_atomo=='g' || next_atomo=='x' || next_atomo=='y'){
        n++;
        next_atomo= tokens[n].atomo;
    }
}
```

Para “Param()”

```
void Param(){
    if(next_atomo=='#' || next_atomo=='b' || next_atomo=='g' || next_atomo=='x' || next_atomo=='y'){
        Tipo();
        if(next_atomo=='i'){
            n++;
            next_atomo= tokens[n].atomo;
            if(next_atomo==',' ){
                otroParam();
            }else if(next_atomo==')'){

            }
        }
    }
}
```

Para “otroParam()”

```
void otroParam(){
    if(next_atomo==','){
        n++;
        next_atomo= tokens[n].atomo;
        Param();
    }
}
```

Para “Cuerpo()”

```
void Cuerpo(){
    printf("Cuerpo:  next_atomo %c\n",next_atomo);
    if(next_atomo=='#' || next_atomo=='b' || next_atomo=='g' || next_atomo=='x' || next_atomo=='y'){
        Decl();
        if(next_atomo=='i'||next_atomo=='f'||next_atomo=='h'||next_atomo=='w'||next_atomo=='j'||next_atomo=='['||next_atomo=='z'||next_atomo=='c'){
            listaP();
        }
    }else if(next_atomo=='i'||next_atomo=='f'||next_atomo=='h'||next_atomo=='w'||next_atomo=='j'||next_atomo=='['||next_atomo=='z'||next_atomo=='c'){
        listaP();
    }else if(next_atomo=='}'){

    }else{
        printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba una declaracion de variables o una lista de parametros \n",next_atomo,n);
        exit(0);
    }
}
```

Para “Decl()”

```
void Decl(){
    if(next_atomo=='#' || next_atomo=='b' || next_atomo=='g' || next_atomo=='x' || next_atomo=='y'){
        D();
        if(next_atomo=='#' || next_atomo=='b' || next_atomo=='g' || next_atomo=='x' || next_atomo=='y'){
            Decl();
        }
    }
}
```

Para “D()”

```
void D(){
    if(next_atomo=='#' || next_atomo=='b' || next_atomo=='g' || next_atomo=='x' || next_atomo=='y'){
        printf("D_antTipo:  next_atomo %c\n",next_atomo);
        Tipo();
        if(next_atomo=='i'){
            printf("D_desTipo:  next_atomo %c\n",next_atomo);
            K();
            if(next_atomo==';'){
                n++;
                next_atomo= tokens[n].atomo;
            }
        }else{
            printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba un identificador \n",next_atomo,n);
            exit(0);
        }
    }else{
        printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba un tipo de dato \n",next_atomo,n);
        exit(0);
    }
}
```

Para “K()”

```
void K(){
    if(next_atomo=='i'){
        n++;
        next_atomo= tokens[n].atomo;
        printf("K:  next_atomo %c\n",next_atomo);
        if(next_atomo=='='||next_atomo==','){
            Q();
        }else if(next_atomo==';'){

        }
    }else{
        printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba un identificador \n",next_atomo,n);
        exit(0);
    }
}
```

Para “Q()”

```
void Q(){
    printf("Q:  next_atomo %c\n",next_atomo);
    if(next_atomo=='='){
        n++;
        next_atomo= tokens[n].atomo;
        printf("Q_if1:  next_atomo %c\n",next_atomo);
        if(next_atomo=='n' || next_atomo=='r'|| next_atomo=='s'){
            N();
            if(next_atomo==','){
                C();
            }else if(next_atomo==';'){

            }
        }else{
            printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba una constante cadena, entera o real \n",next_atomo,n);
            exit(0);
        }
    }else if(next_atomo==';'){

    }
}
```

Para “N()”

```
void N(){
    if(next_atomo=='n' || next_atomo=='r'|| next_atomo=='s'){
        n++;
        next_atomo= tokens[n].atomo;
        printf("N_if:  next_atomo %c\n",next_atomo);
    }
}
```

Para “C()”

```
void C(){
    if(next_atomo==','){
        n++;
        next_atomo= tokens[n].atomo;
        K();
    }else if(next_atomo==';'){

    }else{
        printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba una , \n",next_atomo,n);
        exit(0);
    }
}
```

Para “listaP()”

```
void listaP(){
    printf("listaP:  next_atomo %c\n",next_atomo);
    //if(next_atomo==';' && tokens[n+1].atomo=='i'){
    if(next_atomo=='i'||next_atomo=='f'||next_atomo=='h'||next_atomo=='w'||next_atomo=='j'||next_atomo=='['||next_atomo=='z'||next_atomo=='c'){
        P();
        if(next_atomo=='i'||next_atomo=='f'||next_atomo=='h'||next_atomo=='w'||next_atomo=='j'||next_atomo=='['||next_atomo=='z'||next_atomo=='c'){
            listaP();
        }else if(next_atomo=='}'){

        }
    }else if(next_atomo=='}'){

    }else{
        printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba una palabra reservada o un identificador \n",next_atomo,n);
        exit(0);
    }
}
```

Para “P()”

```
void P(){
    printf("P:  next_atomo %c\n",next_atomo);
    if(next_atomo=='i'){
        A();
        if(next_atomo=='i'||next_atomo=='f'||next_atomo=='h'||next_atomo=='w'||next_atomo=='j'||next_atomo=='['||next_atomo=='z'||next_atomo=='c'){
            P();
        }
    }else if(next_atomo=='f'){
        I();
        if(next_atomo=='i'||next_atomo=='f'||next_atomo=='h'||next_atomo=='w'||next_atomo=='j'||next_atomo=='['||next_atomo=='z'||next_atomo=='c'){
            P();
        }
    }else if(next_atomo=='h'){
        H();
        if(next_atomo=='i'||next_atomo=='f'||next_atomo=='h'||next_atomo=='w'||next_atomo=='j'||next_atomo=='['||next_atomo=='z'||next_atomo=='c'){
            P();
        }
    }else if(next_atomo=='w'){
        W();
        if(next_atomo=='i'||next_atomo=='f'||next_atomo=='h'||next_atomo=='w'||next_atomo=='j'||next_atomo=='['||next_atomo=='z'||next_atomo=='c'){
            P();
        }
    }else if(next_atomo=='j'){
        J();
        if(next_atomo=='i'||next_atomo=='f'||next_atomo=='h'||next_atomo=='w'||next_atomo=='j'||next_atomo=='['||next_atomo=='z'||next_atomo=='c'){
            P();
        }
    }else if(next_atomo=='['){
        Llama();
        if(next_atomo=='i'||next_atomo=='f'||next_atomo=='h'||next_atomo=='w'||next_atomo=='j'||next_atomo=='['||next_atomo=='z'||next_atomo=='c'){
            P();
        }
    }else if(next_atomo=='z'){
        Devuelve();
        printf("P()_dDevuelve: %c\n",next_atomo);
        if(next_atomo=='i'||next_atomo=='f'||next_atomo=='h'||next_atomo=='w'||next_atomo=='j'||next_atomo=='['||next_atomo=='z'||next_atomo=='c'){
            P();
        }
    }else if(next_atomo=='c'){
        n++;
        next_atomo= tokens[n].atomo;
        if(next_atomo==';'){
            n++;
            next_atomo= tokens[n].atomo;
            if(next_atomo=='i'||next_atomo=='f'||next_atomo=='h'||next_atomo=='w'||next_atomo=='j'||next_atomo=='['||next_atomo=='z'||next_atomo=='c'){
                P();
            }
        }
    }else{
        printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba una palabra reservada o un identificador \n",next_atomo,n);
        exit(0);
    }
}
```

Para “A()”

```
void A(){
    printf("A:  next_atomo %c\n",next_atomo);
    if(next_atomo=='i'){
        n++;
        next_atomo= tokens[n].atomo;
        if(next_atomo=='='){
            n++;
            next_atomo= tokens[n].atomo;
            printf("A_if2:  next_atomo %c\n",next_atomo);
            if(next_atomo=='s'||next_atomo=='('||next_atomo=='i'||next_atomo=='n'||next_atomo=='r'||next_atomo=='['){
                AP();
                if(next_atomo==';'){
                    n++;
                    next_atomo= tokens[n].atomo;
                }else{
                    printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba un ;\n",next_atomo,n);
                    exit(0);
                }
            }else{
                printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba una constante cadena o (\n",next_atomo,n);
                exit(0);
            }
        }else{
            printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba un =\n",next_atomo,n);
            exit(0);
        }
    }else{
        printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba un identificador \n",next_atomo,n);
        exit(0);
    }
}
```

Para “AP()”

```
void AP(){
    printf("AP_aif1:  next_atomo %c\n",next_atomo);
    if(next_atomo=='s'){
        n++;
        next_atomo= tokens[n].atomo;
    }else if(next_atomo=='('||next_atomo=='i'||next_atomo=='n'||next_atomo=='r'){
        E();
    }else{
        printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba una constante cadena, numerica, real o un identificador \n",next_atomo,n);
        exit(0);
    }
}
```

Para “E()”

```
void E(){
    if(next_atomo=='('||next_atomo=='i'||next_atomo=='n'||next_atomo=='r'||next_atomo=='['){
        T();
        if(next_atomo=='+'||next_atomo=='-'){
            EP();
        }
    }else{
        printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba una constante numerica, real, un identificador o un ( \n",next_atomo,n);
        exit(0);
    }
}
```

Para “EP()”

```
void EP(){
    if(next_atomo=='+' ||next_atomo=='-'){
        n++;
        next_atomo= tokens[n].atomo;
        T();
        if(next_atomo=='+'||next_atomo=='-'){
            EP();
        }
    }
}
```

Para “T()”

```
void T(){
    if(next_atomo=='('||next_atomo=='i'||next_atomo=='n'||next_atomo=='r'||next_atomo=='['){
        F();
        printf("T()_dF(): next_atomo %c\n",next_atomo);
        if(next_atomo=='*' || next_atomo=='/'||next_atomo=='\\'||next_atomo=='%'||next_atomo=='^'){
            printf("T()_aTP(): next_atomo %c\n",next_atomo);
            TP();
        }else if(next_atomo==';'){

        }else if(next_atomo=='+'||next_atomo=='-'){

        }
    }else{
        printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba una constante numerica, real, un identificador un ( o un [ \n",next_atomo,n);
        exit(0);
    }
}
```

Para “F()”

```
void F(){
    if(next_atomo=='('){
        n++;
        next_atomo= tokens[n].atomo;
        if(next_atomo=='('||next_atomo=='i'||next_atomo=='n'||next_atomo=='r'){
            E();
            if(next_atomo==')'){
                n++;
                next_atomo= tokens[n].atomo;
            }else{
                printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba una ) \n",next_atomo,n);
                exit(0);
            }
        }else{
            printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba una constante numerica, real, un identificador o un ( \n",next_atomo,n);
            exit(0);
        }
    }else if(next_atomo=='i'||next_atomo=='n'||next_atomo=='r'){
        n++;
        next_atomo= tokens[n].atomo;
        printf("F_if2:  next_atomo %c\n",next_atomo);
    }else if(next_atomo=='['){
        Llama();
    }else{
        printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba una constante numerica, real, un identificador, un ( o [ \n",next_atomo,n);
        exit(0);
    }
}
```

Para “Llama()”

```
void Llama(){
    if(next_atomo=='['){
        n++;
        next_atomo= tokens[n].atomo;
        if(next_atomo=='i'){
            n++;
            next_atomo= tokens[n].atomo;
            if(next_atomo=='('){
                n++;
                next_atomo= tokens[n].atomo;
                if(next_atomo=='i'||next_atomo=='n'||next_atomo=='r'||next_atomo=='s'){
                    arg();
                    if(next_atomo==')'){
                        n++;
                        next_atomo=tokens[n].atomo;
                        if(next_atomo==']'){
                            n++;
                            next_atomo=tokens[n].atomo;
                        }
                    }
```

```
            }else if(next_atomo==')'){
                n++;
                next_atomo=tokens[n].atomo;
                if(next_atomo==']'){
                    n++;
                    next_atomo=tokens[n].atomo;
                }else{
                    printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba un ]\n",next_atomo,n);
                    exit(0);
                }
            }else{
                printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba una constante cadena, entera, real o un identificador\n",next_atomo,n);
                exit(0);
            }
        }else{
            printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba una (\n",next_atomo,n);
            exit(0);
        }
    }else{
        printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba un identificador\n",next_atomo,n);
        exit(0);
    }
}else{
    printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba una [\n",next_atomo,n);
    exit(0);
}
}
```

Para “arg()”

```
void arg(){
    if(next_atomo=='i'||next_atomo=='n'||next_atomo=='r'||next_atomo=='s'){
        V();
        if(next_atomo==','){
            otroArg();
        }else{
            printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba una ,\n",next_atomo,n);
            exit(0);
        }
    }else{
        printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba una constante cadena, entera, real o un identificador\n",next_atomo,n);
        exit(0);
    }
}
```

Para “otroArg()”

```
void otroArg(){
    if(next_atomo==','){
        n++;
        next_atomo=tokens[n].atomo;
        if(next_atomo=='i'||next_atomo=='n'||next_atomo=='r'||next_atomo=='s'){
            V();
            if(next_atomo==','){
                otroArg();
            }
        }else{
            printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba una constante cadena, entera, real o un identificador\n",next_atomo,n);
            exit(0);
        }
    }else{
        printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba una ,\n",next_atomo,n);
        exit(0);
    }
}
```

Para “V()”

```
void V(){
    if(tokens[n].atomo=='i'||tokens[n].atomo=='n'||tokens[n].atomo=='r'||tokens[n].atomo=='s'){
        n++;
        next_atomo=tokens[n].atomo;
    }else{
        printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba una constante cadena, entera, real o un identificador\n",next_atomo,n);
        exit(0);
    }
}
```

Para “TP()”

```
void TP(){
    if(next_atomo=='*' || next_atomo=='/'||next_atomo=='\\'||next_atomo=='%'||next_atomo=='^'||next_atomo=='+'||next_atomo=='-'){
        n++;
        next_atomo= tokens[n].atomo;
        printf("TP()_aF(): next_atomo %c\n",next_atomo);
        F();
        printf("TP()_aTP(): next_atomo %c\n",next_atomo);
        if(next_atomo=='*' || next_atomo=='/'||next_atomo=='\\'||next_atomo=='%'||next_atomo=='^'||next_atomo=='+'||next_atomo=='-'){
            TP();
            printf("T()_dTP(): next_atomo %c\n",next_atomo);
        }else if(next_atomo==';'){

        }else{
            printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba un operador aritmetico o ; \n",next_atomo,n);
            exit(0);
        }
    }else if(next_atomo==';'){

    }else{
        printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba un operador aritmetico o ; \n",next_atomo,n);
        exit(0);
    }
}
```

Para “I()”

```
void I(){
    if(next_atomo=='f'){
        n++;
        next_atomo= tokens[n].atomo;
        if(next_atomo=='('){
            n++;
            next_atomo= tokens[n].atomo;
            R();
            if(next_atomo==')'){
                n++;
                next_atomo= tokens[n].atomo;
                if(next_atomo=='i'||next_atomo=='f'||next_atomo=='h'||next_atomo=='w'||next_atomo=='j'||next_atomo=='['||next_atomo=='z'||next_atomo=='c'){
                    listaP();
                    if(next_atomo=='t'){
                        IP();
                        if(next_atomo==':'){
                            n++;
                            next_atomo= tokens[n].atomo;
                        }else{
                            printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba :\n",next_atomo,n);
                            exit(0);
                        }
                    }else{
                        printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba un else \n",next_atomo,n);
                        exit(0);
                    }
                }else if(next_atomo==':'){
```

```
                }else{
                    printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba la declaracion de un parametro\n",next_atomo,n);
                    exit(0);
                }
            }else{
                printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba un )\n",next_atomo,n);
                exit(0);
            }
        }else{
            printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba un (\n",next_atomo,n);
            exit(0);
        }
    }else{
        printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba un if \n",next_atomo,n);
        exit(0);
    }
}
```

Para “R()”

```
void R(){
    if(next_atomo=='i'||next_atomo=='n'||next_atomo=='r'||next_atomo=='s'){
        n++;
        next_atomo= tokens[n].atomo;
        if(next_atomo=='>'||next_atomo=='<'||next_atomo=='l'||next_atomo=='e'||next_atomo=='d'||next_atomo=='u'){
            RP();
            if(tokens[n].atomo=='i'||tokens[n].atomo=='n'||tokens[n].atomo=='r'||tokens[n].atomo=='s'){
                V();
            }else{
                printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba una constante cadena, entera, real o un identificador\n",next_atomo,n);
                exit(0);
            }
        }else{
            printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba un operador relacional\n",next_atomo,n);
            exit(0);
        }
    }else{
        printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba una constante cadena, entera, real o un identificador\n",next_atomo,n);
        exit(0);
    }
}
```

Para “RP()”

```
void RP(){
    if(next_atomo=='>'||next_atomo=='<'||next_atomo=='l'||next_atomo=='e'||next_atomo=='d'||next_atomo=='u'){
        n++;
        next_atomo= tokens[n].atomo;
    }else{
        printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba un operador relacional\n",next_atomo,n);
        exit(0);
    }
}
```

Para “IP()”

```
void IP(){
    if(next_atomo=='t'){
        n++;
        next_atomo= tokens[n].atomo;
        listaP();
    }else{
        printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba un else\n",next_atomo,n);
        exit(0);
    }
}
```

Para “H()”

```
void H(){
    if(next_atomo=='h'){
        n++;
        next_atomo= tokens[n].atomo;
        if(next_atomo=='('){
            n++;
            next_atomo= tokens[n].atomo;
            if(next_atomo=='i'){
                n++;
                next_atomo= tokens[n].atomo;
                if(next_atomo==')'){
                    n++;
                    next_atomo= tokens[n].atomo;
                    if(next_atomo=='{'){
                        n++;
                        next_atomo= tokens[n].atomo;
                        if(next_atomo=='a'){
                            CP();
                            if(next_atomo=='o'){
                                OP();
                            }
                        }else if(next_atomo=='o'){
                            OP();
                        }else if(next_atomo=='}'){
                            n++;
                            next_atomo= tokens[n].atomo;
                        }else{
                            printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba un case o default\n",next_atomo,n);
                            exit(0);
                        }
```

```
                    }else{
                        printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba un {\n",next_atomo,n);
                        exit(0);
                    }
                }else{
                    printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba un ) \n",next_atomo,n);
                    exit(0);
                }
            }else{
                printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba un identificador \n",next_atomo,n);
                exit(0);
            }
        }else{
            printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba un ( \n",next_atomo,n);
            exit(0);
        }
    }else{
        printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba un switch\n",next_atomo,n);
        exit(0);
    }
}
```

Para “CP()”

```
void CP(){
    if(next_atomo=='a'){
        n++;
        next_atomo= tokens[n].atomo;
        if(next_atomo=='n'){
            n++;
            next_atomo= tokens[n].atomo;
            if(next_atomo==':'){
                n++;
                next_atomo= tokens[n].atomo;
                if(next_atomo=='i'||next_atomo=='f'||next_atomo=='h'||next_atomo=='w'||next_atomo=='j'||next_atomo=='['||next_atomo=='z'||next_atomo=='c'){
                    listaP();
                    if(next_atomo=='q'){
                        U();
                        if(next_atomo=='a'){
                            CP();
                        }
                    }else if(next_atomo=='a'){
                        CP();
                    }else{
                        printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba un break o un case\n",next_atomo,n);
                        exit(0);
                    }
                }else{
                    printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba un nuevo parametro\n",next_atomo,n);
                    exit(0);
                }
            }else{
                printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba un :\n",next_atomo,n);
                exit(0);
            }
        }else{
            printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba una constante numerica\n",next_atomo,n);
            exit(0);
        }
    }else{
        printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba un case\n",next_atomo,n);
        exit(0);
    }
}
```

Para “U()”

```
void U(){
    if(next_atomo=='q'){
        n++;
        next_atomo= tokens[n].atomo;
    }else{
        printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba un break\n",next_atomo,n);
        exit(0);
    }
}
```

Para “OP()”

```
void OP(){
    if(next_atomo=='o'){
        n++;
        next_atomo= tokens[n].atomo;
        if(next_atomo==':'){
            n++;
            next_atomo= tokens[n].atomo;
            if(next_atomo=='i'||next_atomo=='f'||next_atomo=='h'||next_atomo=='w'||next_atomo=='j'||next_atomo=='['||next_atomo=='z'||next_atomo=='c'){
                listaP();
            }else{
                printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba una declaracion de parametros\n",next_atomo,n);
                exit(0);
            }
        }else{
            printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba un :\n",next_atomo,n);
            exit(0);
        }
    }else{
        printf("Error de sintaxis en el atomo %c en la posicion %d. Se esperaba un default\n",next_atomo,n);
        exit(0);
    }
}
```

Para “W()”

```
void W(){
    if(next_atomo=='w'){
        n++;
        next_atomo= tokens[n].atomo;
        if(next_atomo=='('){
            n++;
            next_atomo= tokens[n].atomo;
            if(next_atomo=='i'||next_atomo=='n'||next_atomo=='r'||next_atomo=='s'){
                R();
                if(next_atomo==')'){
                    n++;
                    next_atomo= tokens[n].atomo;
                    if(next_atomo=='m'){
                        n++;
                        next_atomo= tokens[n].atomo;
                        if(next_atomo=='{'){
                            n++;
                            next_atomo= tokens[n].atomo;
                            if(next_atomo=='i'||next_atomo=='f'||next_atomo=='h'||next_atomo=='w'||next_atomo=='j'||next_atomo=='['||next_atomo=='z'||next_atomo=='c'){
                                listaP();
                                if(next_atomo=='}'){
                                    n++;
                                    next_atomo= tokens[n].atomo;
                                }
                            }else if(next_atomo=='}'){
                                n++;
                                next_atomo= tokens[n].atomo;
                            }
                        }
                    }
                }
            }
        }
    }
}
```

Para “J()”

```
void J(){
    if(next_atomo=='j'){
        n++;
        next_atomo= tokens[n].atomo;
        if(next_atomo=='('){
            n++;
            next_atomo= tokens[n].atomo;
            if(next_atomo=='i'||next_atomo==';'){
                Y();
                if(next_atomo=='i'||next_atomo=='n'||next_atomo=='r'||next_atomo=='s'||next_atomo==';'){
                    X();
                    if(next_atomo=='i'||next_atomo==')'){
                        Z();
                        if(next_atomo=='{'){
                            n++;
                            next_atomo= tokens[n].atomo;
                            if(next_atomo=='i'||next_atomo=='f'||next_atomo=='h'||next_atomo=='w'||next_atomo=='j'||next_atomo=='['||next_atomo=='z'||next_atomo=='c'){
                                listaP();
                                if(next_atomo=='}'){
                                    n++;
                                    next_atomo= tokens[n].atomo;
                                }
                            }else if(next_atomo=='}'){
                                n++;
                                next_atomo= tokens[n].atomo;
                            }
                        }
                    }
                }
            }
        }
    }
}
```

Para “Y()”

```
void Y(){
    if(next_atomo=='i'){
        n++;
        next_atomo= tokens[n].atomo;
        if(next_atomo=='='){
            n++;
            next_atomo= tokens[n].atomo;
            if(next_atomo=='('||next_atomo=='i'||next_atomo=='n'||next_atomo=='r'){
                E();
                if(next_atomo==';'){
                    n++;
                    next_atomo= tokens[n].atomo;
                }
            }
        }
    }else if(next_atomo==';'){
        n++;
        next_atomo= tokens[n].atomo;
    }
}
```

Para “X()”

```
void X(){
    if(next_atomo=='i'||next_atomo=='n'||next_atomo=='r'||next_atomo=='s'){
        R();
        if(next_atomo==';'){
            n++;
            next_atomo= tokens[n].atomo;
        }
    }else if(next_atomo==';'){
        n++;
        next_atomo= tokens[n].atomo;
    }
}
```

Para “Z()”

```
void Z(){
    if(next_atomo=='i'){
        n++;
        next_atomo= tokens[n].atomo;
        if(next_atomo=='='){
            n++;
            next_atomo= tokens[n].atomo;
            if(next_atomo=='('||next_atomo=='i'||next_atomo=='n'||next_atomo=='r'){
                E();
                if(next_atomo==')'){
                    n++;
                    next_atomo= tokens[n].atomo;
                }
            }
        }
    }else if(next_atomo==')'){
        n++;
        next_atomo= tokens[n].atomo;
    }
}
```

Para “Devuelve()”

```
void Devuelve(){
    if(next_atomo=='z'){
        n++;
        next_atomo= tokens[n].atomo;
        if(next_atomo=='('){
            n++;
            next_atomo= tokens[n].atomo;
            if(tokens[n].atomo=='i'||tokens[n].atomo=='n'||tokens[n].atomo=='r'||tokens[n].atomo=='s'){
                V();
                if(next_atomo==')'){
                    n++;
                    next_atomo= tokens[n].atomo;
                    if(next_atomo==';'){
                        n++;
                        next_atomo= tokens[n].atomo;
                    }
                }
            }else if(next_atomo==')'){
                n++;
                next_atomo= tokens[n].atomo;
                if(next_atomo==';'){
                    n++;
                    next_atomo= tokens[n].atomo;
                }
            }
        }
    }
}
```

Gracias a estas funciones es que nuestro analizador sintáctico empezará a reconocer nuestras cadenas de átomos. En caso de que nuestra cadena sea válida, y no encuentre ningún error en lo que analiza la cadena de entrada, el analizador seguirá verificando la validez de la cadena, pero si no reconoce un átomo, imprime el átomo que debería de seguir en nuestra cadena de átomos.

Por ejemplo, de la cadena de átomos presentada en este documento, su analisis quedaría de la siguiente manera:

```
Program
Func:  next_atomo #Cuerpo:  next_atomo #
D_antTipo:  next_atomo #
D_desTipo:  next_atomo i
K:  next_atomo ;
D_antTipo:  next_atomo x
D_desTipo:  next_atomo i
K:  next_atomo ;
D_antTipo:  next_atomo #
D_desTipo:  next_atomo i
K:  next_atomo ;
listaP:  next_atomo i
P:  next_atomo i
A:  next_atomo i
A_if2:  next_atomo n
AP_aif1:  next_atomo n
F_if2:  next_atomo ;
T()_dF(): next_atomo ;
P:  next_atomo i
A:  next_atomo i
A_if2:  next_atomo r
AP_aif1:  next_atomo r
F_if2:  next_atomo ;
T()_dF(): next_atomo ;
P:  next_atomo i
A:  next_atomo i
A_if2:  next_atomo i
AP_aif1:  next_atomo i
F_if2:  next_atomo *
T()_dF(): next_atomo *
T()_aTP(): next_atomo *
TP()_aF(): next_atomo i
F_if2:  next_atomo +
TP()_aTP(): next_atomo +
TP()_aF(): next_atomo n
F_if2:  next_atomo ;
TP()_aTP(): next_atomo ;
T()_dTP(): next_atomo ;
P:  next_atomo f
listaP:  next_atomo z
P:  next_atomo z
P()_dDevuelve: t
listaP:  next_atomo z
P:  next_atomo z
P()_dDevuelve: :
```

Al final, si se reconoce la cadena de átomos con éxito, se imprime el siguiente mensaje:

```
P()_dDevuelve: :
ANALISIS SINTACTICO REALIZADO CON EXITO
```

Pero, en el caso de que la cadena de entrada tenga algún error, se imprime el átomo que debería de seguir, por ejemplo, se modificó el programa de entrada para que diera error de la siguiente manera:

```
int main(int $numero){
    int $numero;
    float $pi;
    int $resultado;
```

Borrando el signo “$”, ya que este sirve para identificar los identificadores, y al momento de quitárselo, no se reconoce en la cadena como se muestra a continuación:

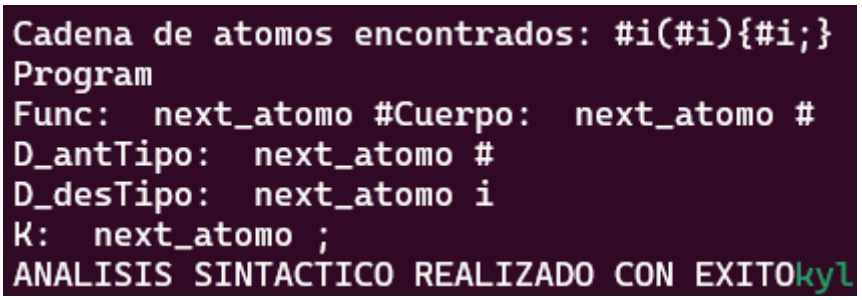
```
Cadena de atomos encontrados: #(#i){#i;xi;#i;i=n;i=r;i=i*i+n;f(i>n)z(n);tz(n);:}
Program
Func:  next_atomo #Error de sintaxis en el atomo ( en la posicion 1. Se esperaba un identificador
```

Donde, como se puede observar, a comparación de la cadena de átomos generada con anterioridad, al principio de esta no aparece #i, si no, solamente #, haciéndonos entender que el analizador no reconoció el identificador (Ya que como se planteó desde un principio, si no cuenta con este símbolo, no contará como un identificador).

Seguido de eso, al estar realizando el análisis sintáctico, se muestra el mensaje de error, ya que, al no haber átomo “i”, el átomo “(” está tomando su lugar, e imprimiendo el mensaje de que se esperaba un “i” en lugar de “(“. Aunque de igual manera, si de acuerdo a nuestra gramática planteada por medio de las funciones establecidas se puede dar una cadena, se dará, por ejemplo para nuestro programa de entrada:

```
int $main(int $numero){
    int $numero;
}
```

Su cadena de átomos y análisis sintáctico queda de la siguiente manera:



Que como se puede apreciar, también es una cadena válida.

CONJUNTOS SELECCIÓN DE LAS DIFERENTES ESTRUCTURAS DE LENGUAJE

SENTENCIA DECLARATIVA

Gramática:

D → <Tipo>K;	Q → =NC
<Tipo>→ b	Q → ,K
<Tipo>→ g	N → n
<Tipo>→ #	N → r
<Tipo>→ y	N → s
<Tipo>→ x	C → ξ
K → iQ	C → ,K
Q → ξ	

PRIMERO *Obtener el conjunto First de cada producción de cada no terminal*

$First(1) = \{b\ g\ \#\ y\ x\}$
$First(2) = \{b\}$
$First(3) = \{g\}$
$First(4) = \{\#\}$
$First(5) = \{y\}$
$First(6) = \{x\}$
$First(7) = \{i\}$
$First(8) = \{\ \}$
$First(9) = \{=\}$
$First(10) = \{,\}$
$First(11) = \{n\}$
$First(12) = \{s\}$
$First(13) = \{\ \}$
$First(14) = \{,\}$

$First(Q) = \{\ =\ ,\ \}$

$First(C) = \{\ ,\ \}$

SEGUNDO *Obtener el conjunto Follow de cada no terminal anulable*

- los no terminales anulables son (Q y C), calcular los Follow(Q) y Follow(C)

CALCULAR FOLLOW(Q)

- Q está en el lado derecho de la producción 7
  - se agrega fin de cadena

CALCULAR FOLLOW(C)

- C está en el lado derecho de la producción 9
  - como esta al final de dicha producción, incorporar los elementos del Follow(Q)

TERCERO *Revisar las producciones*

8: Q -> ε c.s (8) = { ┤ }
9: Q-> =NC c.s (9) = { = } ∴ Son disjuntos
10: Q -> ,K c.s (10) = { , }

11: N -> n c.s (11) = {n}
12: N -> s c.s (12) = {s} ∴ Son disjuntos

13: C -> ε c.s (13) = { ┤ }
14: C -> ,K c.s (14) = { , } ∴ Son disjuntos

∴ Si es una gramática LL(1) debido a que cumple con la condición

SENTENCIA DE ASIGNACIÓN

Gramática:

$A \rightarrow i=A';$

$A' \rightarrow s$

$A' \rightarrow E$

PRIMERO *Obtener el conjunto First de cada producción de cada no terminal*

$$First(1) = \{i\}$$
$$First(2) = \{s\}$$
$$First(3) = \{ \dashv \}$$

SEGUNDO *Obtener el conjunto Follow de cada no terminal anulable*

- los no terminales anulables son (A'), calcular los Follow(A')

CALCULAR FOLLOW(A')

- Q está en el lado derecho de la producción 1
  - se agrega fin de cadena

TERCERO *Revisar las producciones*

2: A' -> s c.s (2) = {s}
3: A' -> E c.s (3) = { ┤ } ∴ Son disjuntos

∴ Si es una gramática LL(1) debido a que cumple con la condición

EXPRESIÓN ARITMÉTICA

E → T E'	T' → %FT'
E' → + T E'	T' → ^FT'
E' → - T E'	T' → ξ
E' → ξ	F → (E)
T → F T'	F → i
T' → *F T'	F → n
T' → /FT'	F → r
T' → \FT'	F →<Llama>

PRIMERO *Obtener el conjunto First de cada producción de cada no terminal*

$First(1) = \{\}$
$First(2) = \{+\}$
$First(3) = \{-\}$
$First(4) = \{\}$
$First(5) = \{\}$
$First(6) = \{*\}$
$First(7) = \{/\}$
$First(8) = \{\backslash\}$
$First(9) = \{\%\}$
$First(10) = \{\hat{}\}$
$First(11) = \{\}$
$First(12) = \{(\}$
$First(13) = \{i\}$
$First(14) = \{n\}$
$First(15) = \{r\}$
$First(16) = \{[\}$

$First(E') = \{ \ + \ \ - \ \}$

$First(T') = \{* \ \ / \backslash \ \% \ \hat{} \ \}$

CALCULAR FIRST (1)

- inicia con el no terminal T, incluir los elementos de First(T)

  CALCULAR FIRST (T)

  - solo tiene la producción 5: T -> FT'
  - inicia con el no terminal F, se agregan los First(F)

$$First(1) = \{ (\ i\ n\ r\ [\}$$

CALCULAR FIRST(4)

- solo tiene la producción 5: T -> FT'

$$First(4) = \{ (\ i\ n\ r\ [\}$$

SEGUNDO *Obtener el conjunto Follow de cada no terminal anulable*

- los no terminales anulables son (E' y T'), calcular los Follow(E') y Follow(T')

CALCULAR FOLLOW(E')

- Q está en el lado derecho de la producción 1,2 y 3
  - De la 1
    - se agrega fin de cadena
    - los elementos del Follow (E)
  - De la 2
    - como es una producción del mismo no terminal y este esta al final, se descarta
  - De la 3
    - como es una producción del mismo no terminal y este esta al final, se descarta

$$Follow(E') = \{ \dashv) \}$$

CALCULAR FOLLOW(T')

- C está en el lado derecho de la producción 5,6,7,8,9 y 10
  - De la 5
    - como T' está al final de la producción, se agregan los elementos de Follow (T)
  - De la 6
    - como es una producción del mismo no terminal y este esta al final, se descarta
  - De la 7
    - como es una producción del mismo no terminal y este esta al final, se descarta
  - De la 8
    - como es una producción del mismo no terminal y este esta al final, se descarta
  - De la 9
    - como es una producción del mismo no terminal y este esta al final, se descarta
  - De la 10
    - como es una producción del mismo no terminal y este esta al final, se descarta

$$Follow(T') = \{ + \dashv) \}$$

TERCERO *Revisar las producciones*

2: E' -> +TE' c.s (2) = {+}
3: E' -> - TE' c.s (3) = { - } ∴ Son disjuntos
4: E' -> ε c.s (4) = { ┤) }

6: T' -> *FT' c.s (6) = {*}
7: T' -> /FT' c.s (7) = {/} ∴ Son disjuntos
8: T' -> \FT' c.s (8) = {\}
9: T' -> %FT' c.s (9) = { %}
10: T' -> ^FT' c.s (10) = {^}
11: T' -> ε c.s (11) = { + ┤) }

12: F -> (E) c.s (12) = { (}
13: F -> i c.s (13) = { i } ∴ Son disjuntos
14: F -> n c.s (14) = { n }
15: F -> r c.s (15) = { r }
16: F -> <Llama> c.s (16) = { [}

∴ Si es una gramática LL(1) debido a que cumple con la condición

EXPRESIÓN RELACIONAL

Gramática:

$R \rightarrow iR'V$	$V \rightarrow i$
$R \rightarrow nR'V'$	$V \rightarrow n$
$R \rightarrow rR'V''$	$V \rightarrow r$
$R \rightarrow sR'V'''$	$V \rightarrow s$
$R' \rightarrow >$	$V' \rightarrow n$
$R' \rightarrow <$	$V' \rightarrow i$
$R' \rightarrow l$	$V'' \rightarrow r$
$R' \rightarrow e$	$V'' \rightarrow i$
$R' \rightarrow d$	$V''' \rightarrow s$
$R' \rightarrow u$	$V''' \rightarrow i$

PRIMERO *Obtener el conjunto First de cada producción de cada no terminal*

$First(1) = \{ i \}$
$First(2) = \{ n \}$
$First(3) = \{ r \}$
$First(4) = \{ s \}$
$First(5) = \{ > \}$
$First(6) = \{ < \}$
$First(7) = \{ l \}$
$First(8) = \{ e \}$
$First(9) = \{ d \}$
$First(10) = \{ u \}$
$First(11) = \{ i \}$
$First(12) = \{ n \}$
$First(13) = \{ r \}$
$First(14) = \{ s \}$
$First(15) = \{ n \}$
$First(16) = \{ i \}$
$First(17) = \{ r \}$
$First(18) = \{ i \}$
$First(19) = \{ s \}$
$First(20) = \{ i \}$

SEGUNDO *Revisar las producciones*

1: R -> iR'V c.s(1) = { i }
2: R -> nR'V' c.s (2) = { n }
3: R -> rR'V'' c.s (3) = { r } ∴ Son disjuntos
4: R -> sR'V''' c.s (4) = { s }

5: R' -> > c.s (5) = { > }
6: R' -> < c.s (6) = { < } ∴ Son disjuntos
7: R' -> l c.s (7) = { l }
8: R' -> e c.s (8) = { e }
9: R' -> d c.s (9) = { d }
10: R' -> u c.s (10) = { u }

11: V -> i c.s (11) = { i }
12: V -> n c.s (12) = { n } ∴ Son disjuntos
13: V -> r c.s (13) = { r }
14: V -> s c.s (14) = { s }

15: V' -> n c.s (15) = { n }
16: V' -> i c.s (16) = { i } ∴ Son disjuntos

17: V'' -> r c.s (17) = { r }
18: V'' -> i c.s (18) = { i } ∴ Son disjuntos

17: V''' -> s c.s (19) = { s }
18: V''' -> i c.s (20) = { i } ∴ Son disjuntos

∴ Si es una gramática LL(1) debido a que cumple con la condición

PROPOSICIONES

P → A
P → I
P → H
P → W
P → J
P → <Llama>
P → <Devuelve>
P → c;

27:	A → i=A';
77:	I → f(R)<listaP>I':
87:	H→ h(i){C'O'}
76:	W → w(R)m{<listaP>}
80:	J → j(YXZ{<listaP>}
97:	<Llama> → [i(<arg>)]
94:	<Devuelve> → z(<valor>);

PRIMERO *Obtener el conjunto First de cada producción de cada no terminal*

$First(1) = \{ i \}$
$First(2) = \{ f \}$
$First(3) = \{ h \}$
$First(4) = \{ w \}$
$First(5) = \{ j \}$
$First(6) = \{ [\}$
$First(7) = \{ z \}$
$First(8) = \{ c \}$

SEGUNDO *Revisar las producciones*

1: P -> A c.s(1) = { i }
2: P -> I c.s (2) = { f }
3: P -> H c.s (3) = { h }
4: P -> W c.s (4) = { w } ∴ Son disjuntos
5: P -> J c.s (5) = { j }
6: P -> <Llama> c.s (6) = { [}
7: P -> <Devuelve> c.s (7) = { z }
8: P -> c; c.s (8) = { c }

∴ Si es una gramática LL(1) debido a que cumple con la condición

LISTA DE 0 O MÁS PROPOSICIONES

<listaP> → ξ
<listaP> → P<listaP>

PRIMERO *Obtener el conjunto First de cada producción de cada no terminal*

$First(2) = \{ i \}$
$First(2) = \{ f \}$
$First(2) = \{ h \}$
$First(2) = \{ w \}$
$First(2) = \{ j \}$
$First(2) = \{ [\}$
$First(2) = \{ z \}$
$First(2) = \{ c \}$

SEGUNDO *Obtener el conjunto Follow de cada no terminal anulable*

- los no terminales anulables son (<listaP>), calcular los Follow(<listaP>)

CALCULAR FOLLOW(<listaP>)

- <listaP> está en el lado derecho de la producción 1 y 2
  - De la 1
    - se agrega fin de cadena
  - De la 2
    - como es una producción del mismo no terminal y este esta al final, se descarta

TERCERO *Revisar las producciones*

1: <listaP> -> ε c.s (1) = { ┤ }
2: <listaP> -> P<listaP> c.s (2) = { i f h w j [z c } **∴** Son disjuntos

∴ Si es una gramática LL(1) debido a que cumple con la condición

SENTENCIA EVALUATE

Gramática:

I → f(R)<listaP>I':
I' → t<listaP>
I' → ξ

PRIMERO *Obtener el conjunto First de cada producción de cada no terminal*

$First(1) = \{ f \}$
$First(2) = \{ t \}$
$First(3) = \{ \}$

SEGUNDO *Obtener el conjunto Follow de cada no terminal anulable*

- los no terminales anulables son (I'), calcular los Follow(I')

CALCULAR FOLLOW(I')

- I' está en el lado derecho de la producción 1
  - De la 1
    - se agrega fin de cadena

TERCERO *Revisar las producciones*

2: I' -> t<listaP> c.s (2) = { t }
3: I' -> ε c.s (3) = { ⊣ } ∴ Son disjuntos

∴ Si es una gramática LL(1) debido a que cumple con la condición

SENTENCIA REPEAT

Gramática
J → j(YXZ{<listaP>}
Y → i=E;
Y → ;
X → R;
X → ;
Z → i=E)
Z →)

46:	R → iR'V
47:	R → nR'V'
48:	R → rR'V''
49:	R → sR'V'''

PRIMERO *Obtener el conjunto First de cada producción de cada no terminal*

$First(1) = \{ j \}$
$First(2) = \{ i \}$
$First(3) = \{ ; \}$
$First(4) = \{ i\ n\ r\ s \}$
$First(5) = \{ ; \}$
$First(6) = \{ i \}$
$First(7) = \{) \}$

SEGUNDO *Revisar las producciones*

2: Y -> i=E; c.s(2) = { i }
3: Y -> ; c.s (3) = { ; } ∴ Son disjuntos

4: X -> R; c.s (4) = { i n r s }
5: X -> ; c.s (5) = { ; } ∴ Son disjuntos

6: Z -> i=E) c.s (6) = { i }
7: Z ->) c.s (7) = {) } ∴ Son disjuntos

∴ Si es una gramática LL(1) debido a que cumple con la condición

SENTENCIA SELECT

Gramática:

H→ h(i){C'O'}
C' → an:<listaP>UC'
C' → ξ

O' → o:<listaP>
O' → ξ
U → q
U → ξ

PRIMERO *Obtener el conjunto First de cada producción de cada no terminal*

$First(1) = \{ h \}$
$First(2) = \{ a \}$
$First(3) = \{ \}$
$First(4) = \{ o \}$
$First(5) = \{ \}$
$First(6) = \{ q \}$
$First(7) = \{ \}$

SEGUNDO *Obtener el conjunto Follow de cada no terminal anulable*

- los no terminales anulables son (C', O' y U), calcular los Follow(C', O' y U)

CALCULAR FOLLOW(C')

- C' está en el lado derecho de la producción 1 y 2
  - De la 1
    - se agregan los elementos del Follow(O')
  - De la 2
    - como es una producción del mismo no terminal y este esta al final, se descarta

$$Follow(C') = \{ o \}$$

CALCULAR FOLLOW(C')

- C' está en el lado derecho de la producción 1
  - De la 1
    - se agregan los elementos del Follow(H) y el fin de cadena

$$Follow(O') = \{ \dashv \}\}$$

CALCULAR FOLLOW(U)

- U está en el lado derecho de la producción 2
  - De la 2
    - se agregan los elementos del Follow(C')

$$Follow(U) = \{ o \}$$

TERCERO *Revisar las producciones*

2: C' -> an:<listaP>UC' c.s (2) = { a }
3: C' -> ε c.s (3) = { o } ∴ Son disjuntos

4: O' -> o:<listaP> c.s (4) = { o }
5: O' -> ε c.s (5) = { ┤ } } ∴ Son disjuntos

6: U -> q c.s (6) = { q }
7: U -> ε c.s (7) = { o } ∴ Son disjuntos

∴ Si es una gramática LL(1) debido a que cumple con la condición

SENTENCIA THROW

Gramática:
<Devuelve> → z(<valor>);
<valor> → V
<valor> → ξ

56:	V → i
57:	V → n
58:	V → r
59:	V → s

PRIMERO *Obtener el conjunto First de cada producción de cada no terminal*

$$First(1) = \{ z \}$$
$$First(2) = \{ i\ n\ r\ s \}$$
$$First(3) = \{\ \}$$

SEGUNDO *Obtener el conjunto Follow de cada no terminal anulable*

CALCULAR FOLLOW(<valor>)

- <valor> está en el lado derecho de la producción 1
  - De la 1
    - se agregan los elementos del Follow(<Devuelve>) y el fin de cadena

$$Follow(O') = \{ \dashv) ; \}$$

TERCERO *Revisar las producciones*

2: <valor> -> V c.s (2) = { i n r s }
3: <valor> -> ε c.s (3) = { ⊣) ; } ∴ Son disjuntos

∴ Si es una gramática LL(1) debido a que cumple con la condición

LLAMADA A UNA FUNCIÓN

Gramática:

<Llama> → [i(<arg>)]
<arg> → ξ
<arg> → V<otroArg>
<otroArg> → ,V<otroArg>
<otroArg> → ξ

56:	V → i
57:	V → n
58:	V → r
59:	V → s

PRIMERO *Obtener el conjunto First de cada producción de cada no terminal*

$$First(1) = \{ [\}$$
$$First(2) = \{ \}$$
$$First(3) = \{ i\ n\ r\ s \}$$
$$First(4) = \{ , \}$$
$$First(5) = \{ \}$$

SEGUNDO *Obtener el conjunto Follow de cada no terminal anulable*

CALCULAR FOLLOW(<arg>)

- <arg> está en el lado derecho de la producción 1
  - De la 1
    - se agregan los elementos del Follow(<Llama>) y el fin de cadena

$$Follow(< arg >) = \{ \dashv)] \}$$

CALCULAR FOLLOW(<otroArg>)

- <otroArg> está en el lado derecho de la producción 3 y 4
  - De la 3
    - como <otroArg> se encuentra al final de la producción se agregan los elementos de Follow <arg>
  - De la 4
    - como es una producción del mismo no terminal, y está al final, se descarta

$$Follow(< otroArg >) = \{ \dashv)] \}$$

TERCERO *Revisar las producciones*

2: <arg> -> ε c.s (2) = { ⊣)] }
3: <arg> -> V<otroArg> c.s (3) = { $i\ n\ r\ s$ } **∴** Son disjuntos

4: <otroArg> -> ,V<otroArg> c.s (4) = { , }
5: <otroArg> -> ε c.s (5) = { ⊣)] } **∴** Son disjuntos

∴ Si es una gramática LL(1) debido a que cumple con la condición

FUNCIONES

Gramática:
<Func> → <Tipo>i(<Param>){<Cuerpo >}
<Param> → <Tipo>i<otroParam>
<Param> → ξ
<otroParam>→,<Tipo>i<otroParam>
<otroParam> → ξ
<Cuerpo> → <Decl><listaP>
<Decl> → ξ
<Decl> → D<Decl>

13:	<Tipo>→ b
14:	<Tipo>→ g
15:	<Tipo>→ #
16:	<Tipo>→ y
17:	<Tipo>→ x

12:	D → <Tipo>K;

PRIMERO *Obtener el conjunto First de cada producción de cada no terminal*

$First(1) = \{ b\ g\ \#\ y\ x \}$
$First(2) = \{ b\ g\ \#\ y\ x \}$
$First(3) = \{ \}$
$First(4) = \{ , \}$
$First(5) = \{ \}$
$First(6) = \{ b\ g\ \#\ y\ x \}$
$First(7) = \{ \}$
$First(8) = \{ b\ g\ \#\ y\ x \}$

SEGUNDO *Obtener el conjunto Follow de cada no terminal anulable*

CALCULAR FOLLOW(<Param>)

- <Param> está en el lado derecho de la producción 1
  - De la 1
    - se agregan los elementos del First(<Cuerpo>)

$$Follow(< Param >) = \{ i\ f\ h\ w\ j\ [\ z\ c \}$$

CALCULAR FOLLOW(<otroParam>)

- <otroParam> está en el lado derecho de la producción 2 y 3
  - De la 2
    - como <otroParam> se encuentra al final de la producción se agregan los elementos de Follow <Param>
  - De la 3
    - como es una producción del mismo no terminal, y está al final, se descarta

$$Follow(< otroParam >) = \{ b\ g\ \#\ y\ x \}$$

CALCULAR FOLLOW(<Decl>)

- <Decl> está en el lado derecho de la producción 6 y 8
  - De la 6
    - como <listaP> es no terminal, agregar los First(<listaP>
  - De la 8
    - como es una producción del mismo no terminal y está al final, se descarta.

$$Follow(< Decl >) = \{ i\ f\ h\ w\ j\ [\ z\ c \}$$

TERCERO *Revisar las producciones*

2: <Param> -> <Tipo>i<otroParam> c.s (2) = { $i\ f\ h\ w\ j\ [\ z\ c\}$
3: <Param> -> ε c.s (3) = { $b\ g\ \#\ y\ x$ } ∴ Son disjuntos

4: <otroParam> -> ,<Tipo>i<otroParam> c.s (4) = { , }
5: <otroParam> -> ε c.s (5) = { $b\ g\ \#\ y\ x$ } ∴ Son disjuntos

7: <Decl> -> ε c.s (7) = { $i\ f\ h\ w\ j\ [\ z\ c$ }
8: <Decl> -> D<Decl> c.s (8) = {$b\ g\ \#\ y\ x$ } ∴ Son disjuntos

∴ Si es una gramática LL(1) debido a que cumple con la condición

ESTRUCTURA DEL PROGRAMA

4:	<Func> → <Tipo>i(<Param>){<Cuerpo >}

Gramática:

<Program> →<Func><otraFunc>
<otraFunc> →<Func><otraFunc>
<otraFunc> → ξ

13:	<Tipo>→ b
14:	<Tipo>→ g
15:	<Tipo>→ #
16:	<Tipo>→ y
17:	<Tipo>→ x

PRIMERO *Obtener el conjunto First de cada producción de cada no terminal*

$First(1) = \{ b\ g\ \#\ y\ x \}$ $First(3) = \{\ \}$
$First(2) = \{ b\ g\ \#\ y\ x \}$

SEGUNDO *Obtener el conjunto Follow de cada no terminal anulable*

CALCULAR FOLLOW(<otraFunc>)

- <otraFunc> está en el lado derecho de la producción 1 y 2
  - De la 1
    - por estar al final, se debe incorporar los elementos de Follow(<Program>) y fin de cadena

$$Follow(< otraFunc >) = \{ \dashv \}$$

  - De la 2
    - como es una producción del mismo no terminal y este esta al final, se descarta

TERCERO *Revisar las producciones*

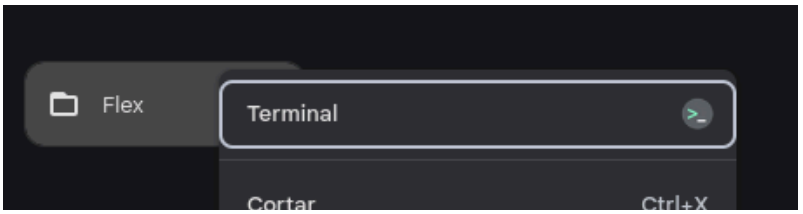
2: <otraFunc> -> <Func><otroFunc> c.s (2) = $\{ b\ g\ \#\ y\ x \}$
3: <otraFunc> -> ε c.s (3) = $\{ \dashv \}$ ∴ Son disjuntos

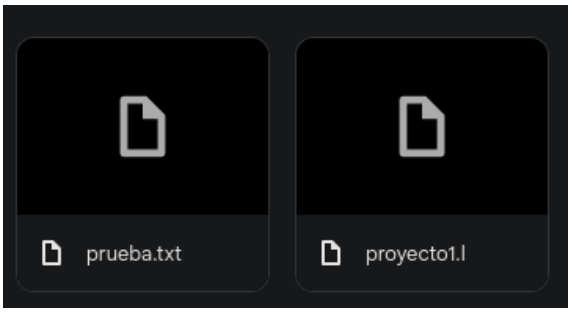
∴ Si es una gramática LL(1) debido a que cumple con la condición

INDICACIONES PARA EJECUTAR EL PROGRAMA

DESDE TERMINAL DE LINUX

- Abrir la carpeta en donde se encuentran los documentos con la Terminal

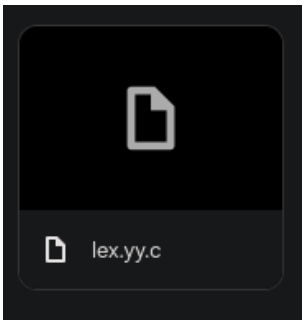


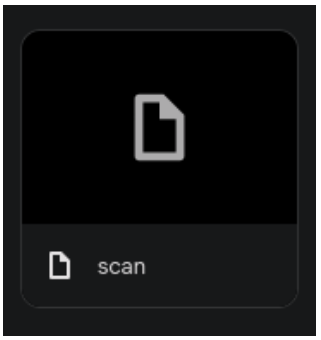


- Ejecutar los siguientes comandos

```
-/Flex$ flex proyecto2.l
-/Flex$ gcc lex.yy.c -lfl -o scan
-/Flex$ ./scan prueba.txt
```

← Sirve para compilar el programa creado
← Sirve para crear lex.yy.c
← Sirve para utilizar el documento .txt





- Finalmente en pantalla se muestra la ejecución del programa

```
<--> BIENVENIDO AL PROGRAMA ANALIZADOR LEXICO Y SINTACTICO <-->
  <-->                    DESCENDENTE RECURSIVO           <-->

A continuacion se empezara a analizar el archivo que ha introducido al arrancar el programa
empezando con la definicion de las clases de cada sentencia del programa que se esta evaluando
```

CONCLUSIONES

Barragán Pilar Diana

Con la realización de este proyecto me fue posible comprender los múltiples conceptos que vimos durante la clase, entre ellos fundamentalmente lo que es un analizador sintáctico, puesto que como ya sabemos este es el que se encarga de revisar la estructura sintáctica de un programa con la utilización de gramáticas, después de que el programa haya sido procesado por el analizador léxico y haber obtenido los tokens. Esto fue lo que realizamos en nuestro proyecto de construir un analizador sintáctico descendente recursivo, ya que a base del primer proyecto que fue un analizador léxico, con los respectivos tokens que se nos generaron y los átomos que agregamos en este proyecto fue que lo realizamos revisando cada que cada gramática cumpliera con ser una gramática LL(1). Creamos para cada no terminal de la gramática una función en la cual por cada terminal validamos con un if si el átomo en la lista de átomos era igual al terminal y en caso de no serlo mandaba un error y así sucesivamente fuimos creando las funciones a base de las reglas de producción.

De igual forma para cada gramática validamos que se tratará de una gramática LL(1) la cual significa Lectura y Analisis Izquierdo y el uno es el número de tokens que consulta para saber qué regla gramatical se aplicó, para validar que las gramáticas sean LL(1) revisamos que los conjuntos selección de un mismo terminal sean disjuntos, es decir, que los elementos que conforman estos conjuntos sean todos diferentes, para ello calculamos el first de cada una de las reglas de producción y después en caso de que las producciones fueran vacías calculamos el Follow de esa producción.

Finalmente el realizar este proyecto representó un reto en el que tuvimos que implementar los temas vistos en clase, como ya se mencionó anteriormente hicimos uso tanto del analizador léxico, como las gramáticas LL(1) y el analizador sintáctico descendente recursivo, lo que representó diversos problemas a la hora de realizar el código ya que eran muchas funciones para los respectivos no terminales de las gramáticas por lo que tuvimos diversos errores que fuimos arreglando.

Ramírez Martínez Valeria

En conclusión, el desarrollo de un analizador sintáctico descendente recursivo implementado en Flex ha demostrado ser un reto grande, sin embargo, al obtener resultados favorables, se evidencia su valor como herramienta analítica enriquecedora.
A lo largo de este proyecto, hemos explorado las complejidades del análisis sintáctico, comprendiendo en profundidad cómo interpretar y estructurar el flujo de tokens de un lenguaje determinado. La integración de varias funciones adicionales al programa realizado con anterioridad, enriqueció la versatilidad y utilidad del analizador, permitiendo una mayor flexibilidad y adaptabilidad a diferentes contextos de uso.

Mediante la implementación de técnicas de análisis sintáctico descendente recursivo, hemos logrado construir un sistema robusto y eficiente capaz de procesar una amplia gama de gramáticas, desde las más simples hasta las más complejas. La recursividad ha demostrado ser una herramienta poderosa para abordar la estructura jerárquica de los lenguajes, permitiendo una interpretación precisa y eficiente de la sintaxis.

Además, la utilización de Flex como herramienta de generación de analizadores léxicos ha facilitado significativamente la implementación del analizador sintáctico proporcionando una manera elegante y eficiente de manejar el flujo de entrada de tokens. La combinación de Flex con el enfoque descendente recursivo ha resultado es un sistema altamente modular y fácilmente mantenible, lo que facilita su expansión y personalización en futuros proyectos.

En resumen, este proyecto ha representado un paso significativo en nuestra comprensión y dominio de las técnicas de análisis sintáctico. La implementación de un analizador sintáctico descendente recursivo en Flex, con la adición de varias funciones, ha demostrado ser una herramienta invaluable para el desarrollo de compiladores, intérpretes y otras aplicaciones que requieren un procesamiento eficiente del lenguaje formal.

Silverio Martínez Andrés

Gracias a la elaboración de este proyecto, pude aterrizar de mejor manera el cómo es que, en conjunto, funciona un analizador léxico, como un analizador sintáctico en conjunto. Gracias a la generación de la cadena de átomos, y a la gramática dada por la profesora es que pudimos realizar este proyecto, además de las clases donde se abarcó el análisis sintáctico descendente recursivo. Además, se verificó a detalle que cada una de las gramáticas cumplieran con una gramática LL(1), y si fuese en caso contrario, modificar dichas gramáticas para que si sean LL(1).

Principalmente lo que se implementó en nuestro programa, y que fue lo que más tiempo nos llevó en realizar, fue la implementación de cada una de las funciones que son capaces de verificar si la cadena de átomos cumple con la gramática, ya que se agregaron muchas sentencias if - else, dependiendo de cada no terminal y sus funciones que se implementaron, haciendo que nuestro código se extendiera casi hasta las 2000 líneas, pero de ahí en fuera, estuvo más fácil implementar la creación de la cadena de átomos.

Por último, el proyecto me gustó, ya que me ayudó a comprender de mejor forma el cómo implementar y el cómo se realiza el análisis sintáctico descendente recursivo, que, a pesar de ser de difícil implementación, fue entretenido el idear una solución, que en conjunto como equipo pudimos idearla.

REFERENCIAS

- Analizador léxico. (s/f). Ecured.cu. Recuperado el 30 de abril de 2024, de https://www.ecured.cu/Analizador_l%C3%A9xico

- Álvarez, M. (s/f). *Analizador Sintáctico.* Programación II. Recuperado el 30 de abril de 2024, de https://hopelchen.tecnm.mx/principal/sylabus/fpdb/recursos/r135307.PDF

- Celis, J. (2015). *Aplicaciones de un Analizador Léxico*. https://www.geocities.ws/itmina_web/lya/Auto_Archivos/proy/compil2.gif

- Universidad de Huelva. (s/f). *Análisis Léxico*. Recuperado el 30 de abril de 2024, de https://www.uhu.es/francisco.moreno/gii_pl/docs/Tema_2.pdf