Informe: Lenguaje FusionCod

Examinacion Parcial

Nota

Estudiante	Escuela	Asignatura
Iben Omar Flores Polanco ifloresp@ulasalle.edu.pe	Carrera Profesional de Ingeniería de Software	Compiladores
Joshua David Ortiz Rosas jortizr@ulasalle.edu.pe	Carrera Profesional de Ingeniería de Software	Compiladores
Andrea J. Ticona Mamani aticonam@ulasalle.edu.pe	Carrera Profesional de Ingeniería de Software	Compiladores
Jerson Ernesto Chura Pacci jchurap@ulasalle.edu.pe	Carrera Profesional de Ingeniería de Software	Compiladores

Informe	Tema	Duración
02	Examinacion Parcial	06 horas

Semestre académico	Fecha de inicio	Fecha de entrega
Semestre V	22/03/25	13/05/25

Índice

1. Introducción

1.1. Justificación

El desarrollo de software enfrenta el desafío de equilibrar rendimiento, seguridad y facilidad de uso. Lenguajes como C ofrecen control y eficiencia, pero su complejidad puede ser una barrera para principiantes. Por otro lado, lenguajes como Python priorizan la accesibilidad, sacrificando a menudo el rendimiento. En este contexto, surge **FusionCod**, un lenguaje de programación diseñado para combinar lo mejor de ambos mundos: una sintaxis clara y estructurada para estudiantes y nuevos desarrolladores, junto con un alto rendimiento y control para programadores experimentados.

1.2. Objetivo

La motivación detrás de **FusionCod** es facilitar la enseñanza de conceptos fundamentales de programación mientras se mantiene la capacidad de desarrollar aplicaciones eficientes. Este lenguaje busca cerrar la brecha entre simplicidad y potencia, ofreciendo una herramienta versátil para la educación y la industria.

Este informe describe **FusionCod**, incluyendo su especificación léxica, gramática, y la implementación de un analizador léxico y sintáctico. También se presentan ejemplos de código y se proporciona un enlace al repositorio del proyecto.

2. Especificación Léxica

2.1. Definición de comentarios

Los comentarios en **FusionCod** son secuencias de texto que el analizador léxico ignora. Se definen como cualquier texto que sigue al carácter # hasta el final de la línea, permitiendo a los desarrolladores añadir notas o deshabilitar código temporalmente.

2.2. Definición de identificadores

Los identificadores en **FusionCod** son nombres utilizados para variables, funciones u otros elementos definidos por el usuario. Se componen de una letra o guion bajo (_) seguido de cero o más letras, números o guiones bajos, respetando las reglas de la expresión regular asociada.

2.3. Definición de palabras reservadas

Las palabras reservadas son identificadores predefinidos con un significado especial en **FusionCod**. Estas se mapean a tipos de tokens específicos durante el análisis léxico. La lista completa está implementada en el siguiente código Python extraído de `lexico.py`:

Listing 1: Palabras reservadas de FusionCod

```
palabras_reservadas = {
    'fn': 'funcion', 'main': 'principal', 'show': 'imprimir', 'return': 'devolver',
    'stop': 'detener', 'int': 'tentero', 'float': 'tflotante', 'text': 'tcadena',
    'bool': 'tbooleano', 'void': 'tvacio', 'if': 'si', 'and': 'y', 'or': 'o',
    'elif': 'sino', 'else': 'entonces', 'while': 'mientras', 'for': 'para',
    'true': 'nbooleano', 'false': 'nbooleano', 'read': 'leer'
}
```

2.4. Definición de literales

Los literales en **FusionCod** representan valores constantes directamente en el código. Incluyen:

- `nentero`: Números enteros (e.g., `42`).
- `nflotante`: Números con decimales (e.g., `3.14`).
- `ncadena`: Cadenas de texto entre comillas (e.g., `"Hola"`).
- `nbooleano`: Valores booleanos (`true`, `false`).

2.5. Definición de operadores

Los operadores en **FusionCod** se dividen en varias categorías según su función.

2.5.1. Operadores aritméticos

Estos operadores realizan cálculos numéricos:

- `suma`: Suma (`+`).
- `resta`: Resta (`-`).
- `mul`: Multiplicación (`*`).
- `div`: División (`/`).
- `residuo`: Módulo (`%`).

2.5.2. Operadores de comparación

Estos operadores evalúan relaciones entre valores:

- `menorque`: Menor que (`<`).
- `mayorque`: Mayor que (`>`).
- `menorigualque`: Menor o igual que (`<=`).
- `mayorigualque`: Mayor o igual que (`>=`).
- `igualbool`: Igualdad (`==`).
- `diferentede`: Desigualdad (`<>`).

2.5.3. Operadores lógicos

Estos operadores combinan condiciones booleanas:

- `y`: Conjunción lógica (`and`).
- `o`: Disyunción lógica (`or`).

2.5.4. Operadores de asignación

Este operador asigna valores a variables:

- `igual`: Asignación (`=`).

2.5.5. Operadores de incremento y decremento

Actualmente, **FusionCod** no incluye operadores específicos de incremento (`++`) ni decremento (`--`). Esta funcionalidad podría considerarse en futuras versiones del lenguaje.

2.6. Expresiones regulares

Las expresiones regulares definen cómo se reconocen los tokens en el código fuente. La siguiente tabla resume las expresiones regulares implementadas en el analizador léxico:

Token	Expresión regular
funcion	`fn`
principal	`main`
pabierto	`\(`
pcerrado	`\)`
imprimir	`show`
coma	`,`
id	`[a-zA-Z_][a-zA-Z0-9_]*`
fsentencia	`;`
devolver	`return`
detener	`stop`
llaveabi	`\{`
llavecerr	`\}`
tentero	`int`
tflotante	`float`
tcadena	`text`
tbooleano	`bool`
tvacio	`void`
si	`if`
y	`and`
o	`or`
sino	`elif`
entonces	`else`
mientras	`while`
para	`for`
suma	`+`
resta	`-`
mul	`*`
div	`/`
residuo	`%`
menorque	`<`
mayorque	`>`
menorigualque	`<=`
mayorigualque	`>=`
igual	`=`
igualbool	`==`
differentede	`<>`
nentero	`-?[0-9]+`
nflotante	`-?[0-9]+\.[0-9]+`
ncadena	`"[^"]*"`
nbooleano	`true`false—
leer	`read`
comentario	`#.*`
ignorar	`[ \t]+`
newline	`\n+`

Los comentarios (`#.*`) se ignoran durante el análisis léxico. Los espacios y tabulaciones (`[ \t]+`) también se descartan. Las nuevas líneas (`\n+`) incrementan el contador de líneas en el analizador.

2.7. Implementación léxica

El analizador léxico de **FusionCod** está implementado en Python utilizando la biblioteca PLY. Lee un archivo de código fuente (e.g., `buclesanidados.txt`) y genera una lista de objetos `Token` con atributos como tipo, valor, línea y columna. Los errores léxicos se detectan y almacenan en `lista_errores_lexicos`. A continuación, se muestra la implementación principal extraída de `lexico.py`:

Listing 2: Implementación del analizador léxico en lexico.py

```
import ply.lex as lex

# Lista para almacenar errores léxicos
lista_errores_lexicos = []

# Definición de tokens
tokens = (
    'funcion', 'principal', 'pabierto', 'pcerrado', 'imprimir', 'coma', 'id',
    'fsentencia', 'devolver', 'detener', 'llaveabi', 'llavecerr', 'tentero',
    'tflotante', 'tcadena', 'tbooleano', 'tvacio', 'si', 'y', 'o', 'sino',
    'entonces', 'mientras', 'para', 'suma', 'resta', 'mul', 'div', 'residuo',
    'menorque', 'mayorque', 'menorigualque', 'mayorigualque', 'igual', 'igualbool',
    'diferentede', 'nentero', 'nflotante', 'ncadena', 'nbooleano', 'leer'
)

# Palabras reservadas
palabras_reservadas = {
    'fn': 'funcion', 'main': 'principal', 'show': 'imprimir', 'return': 'devolver',
    'stop': 'detener', 'int': 'tentero', 'float': 'tflotante', 'text': 'tcadena',
    'bool': 'tbooleano', 'void': 'tvacio', 'if': 'si', 'and': 'y', 'or': 'o',
    'elif': 'sino', 'else': 'entonces', 'while': 'mientras', 'for': 'para',
    'true': 'nbooleano', 'false': 'nbooleano', 'read': 'leer'
}

# Aniadir palabras reservadas a los tokens
tokens = tokens + tuple(palabras_reservadas.values())

# Expresiones regulares para tokens simples
t_pabierto = r'\('
t_pcerrado = r'\)'
t_coma = r','
t_fsentencia = r';'
t_llaveabi = r'\{'
t_llavecerr = r'\}'
t_suma = r'\+'
t_resta = r'-'
t_mul = r'\*'
t_div = r'/'
t_residuo = r'%'
t_menorque = r'<'
t_mayorque = r'>'
t_menorigualque = r'<='
t_mayorigualque = r'>='
t_igual = r'='
t_igualbool = r'=='
t_diferentede = r'<>'

# Regla para identificadores
```

```
def t_id(t):
    r'[a-zA-Z_][a-zA-Z0-9_]*'
    t.type = palabras_reservadas.get(t.value, 'id')
    return t

# Regla para números enteros
def t_nentero(t):
    r'-?[0-9]+'
    t.value = int(t.value)
    return t

# Regla para números flotantes
def t_nflotante(t):
    r'-?[0-9]+\.[0-9]+'
    t.value = float(t.value)
    return t

# Regla para cadenas
def t_ncadena(t):
    r'"[^"]*"'
    t.value = t.value[1:-1] # Eliminar comillas
    return t

# Regla para comentarios
def t_comentario(t):
    r'#.*'
    pass  # Ignorar comentarios

# Regla para ignorar espacios y tabulaciones
t_ignore = ' \t'

# Contador de líneas
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

# Manejo de errores léxicos
def t_error(t):
    lista_errores_lexicos.append(f"Error léxico en línea {t.lineno}, posición {t.lexpos}:
        Carácter no reconocido '{t.value[0]}'")
    t.lexer.skip(1)

# Construir el lexer
lexer = lex.lex()

# Función para generar tokens
def generar_tokens():
    lista_de_tokens = []
    while True:
        tok = lexer.token()
        if not tok: break
        token_obj = Token(tok.type, tok.value, tok.lineno, tok.lexpos)
        lista_de_tokens.append(token_obj)
    return lista_de_tokens
```

Ejemplos de código analizados incluyen "Hola mundo", "Bucles anidados", y "Fibonacci recursivo",

como se detalla en la sección de ejemplos.

3. Gramatica

3.1. Definición de la gramática

La gramática de **FusionCod** define las reglas sintácticas que estructuran el lenguaje, permitiendo la generación de programas válidos mediante un analizador LL(1). A continuación, se detalla su estructura, componentes clave y las producciones completas.

3.1.1. Estructura general

La gramática sigue una estructura jerárquica que comienza con el símbolo inicial `programaprincipal`, el cual representa un programa completo. Este se descompone en una o más funciones, incluyendo una función principal (`main`) y otras funciones definidas por el usuario. Cada función contiene un bloque de instrucciones que pueden incluir asignaciones, bucles, condicionales, entrada/salida y expresiones.

3.1.2. Componentes clave

Los principales no terminales que estructuran el lenguaje son:

- `programaprincipal`: Representa el programa completo, compuesto por funciones.
- `restomain`: Define la estructura de la función principal (`main`).
- `restofuncn`: Define funciones adicionales con parámetros y tipos de retorno.
- `instruccion`: Representa las instrucciones posibles (e.g., asignaciones, bucles, condicionales).
- `expresion`: Define expresiones aritméticas, lógicas y de comparación.
- `valordato`: Representa literales como enteros, flotantes, cadenas y booleanos.

3.1.3. Gramática de Netcode

A continuación, se presentan las producciones de la gramática de **FusionCod**, definidas en el archivo `gramatica.txt`. Se utiliza una notación BNF (Backus-Naur Form) adaptada, donde `e` representa la cadena vacía (ϵ):

Listing 3: Gramática de FusionCod (gramatica.txt)

```
programaprincipal -> funcion opcionprincipal masfuncn
opcionprincipal -> restomain
opcionprincipal -> restofuncn
masfuncn -> programaprincipal
masfuncn -> e
restomain -> principal pabierto pcerrado tentero llaveabi masinstrucciones llavecerr
restofuncn -> id pabierto parametrosf pcerrado opciondato llaveabi masinstrucciones llavecerr
parametrosf -> id tipodato masparametrosf
parametrosf -> e
masparametrosf -> coma parametrosf
masparametrosf -> e
opciondato -> tvacio
opciondato -> tipodato
instruccion -> asignaciones fsentencia
instruccion -> mostrar fsentencia
instruccion -> buclepara
```

```
instruccion -> buclemientras
instruccion -> condicional
instruccion -> detener fsentencia
instruccion -> leer pabierto id pcerrado fsentencia
instruccion -> devolver expresion fsentencia
masinstrucciones -> instruccion masinstrucciones
masinstrucciones -> e
buclepara -> para pabierto asignaciones fsentencia expresion fsentencia asignaciones
    pcerrado llaveabi masinstrucciones llavecerr
buclemientras -> mientras pabierto expresion pcerrado llaveabi masinstrucciones llavecerr
condicional -> si pabierto expresion pcerrado llaveabi masinstrucciones llavecerr posibilidad
posibilidad -> sino pabierto expresion pcerrado llaveabi masinstrucciones llavecerr
    posibilidad
posibilidad -> entonces llaveabi masinstrucciones llavecerr
posibilidad -> e
mostrar -> imprimir pabierto comandos pcerrado
comandos -> expresion mascomandos
comandos -> e
mascomandos -> suma expresion mascomandos
mascomandos -> e
asignaciones -> id ext
ext -> tipodato opcionesasig
ext -> extension
extension -> igual expresion
extension -> pabierto parametros pcerrado
opcionesasig -> igual expresion
opcionesasig -> e
expresion -> pabierto expresion pcerrado masexpresiones
expresion -> id opciones masexpresiones
expresion -> valordato masexpresiones
masexpresiones -> e
masexpresiones -> operacion expresion
opciones -> pabierto parametros pcerrado
opciones -> e
parametros -> expresion restoparametros
parametros -> e
restoparametros -> coma expresion restoparametros
restoparametros -> e
operacion -> suma
operacion -> resta
operacion -> mul
operacion -> div
operacion -> residuo
operacion -> igualbool
operacion -> menorque
operacion -> mayorque
operacion -> menorigualque
operacion -> mayorigualque
operacion -> diferentede
operacion -> y
operacion -> o
valordato -> ncadena
valordato -> nflotante
valordato -> nentero
valordato -> nbooleano
tipodato -> tentero
```

```
tipodato -> tcadena
tipodato -> tflotante
tipodato -> tbooleano
```

3.2. Implementación de la tabla sintáctica

3.2.1. Descripción general

La tabla sintáctica de **FusionCod** es una tabla LL(1) que mapea combinaciones de no terminales y terminales a producciones específicas, permitiendo un análisis sintáctico determinista. Fue generada automáticamente a partir de la gramática usando el script `generador-ll1.py`, el cual calcula los conjuntos de primeros (FIRST) y siguientes (FOLLOW) para cada no terminal y construye la tabla. La tabla se almacena en `tabla_ll1.csv` y es utilizada por el analizador sintáctico para tomar decisiones de parsing.

A continuación, se muestra una versión simplificada de la tabla LL(1) extraída de `tabla_ll1.csv`. Debido a su tamaño, se presenta un fragmento representativo con algunos no terminales y terminales clave:

	`funcion`	`id`	
`programaprincipal`	funcion opcionprincipal masfuncn		
`instruccion`		asignaciones fsentencia	
`condicional`			si pabierto expresion pcerrad
`buclemientras`			
`buclepara`			
`masinstrucciones`		instruccion masinstrucciones	instr

La tabla completa incluye todos los no terminales y terminales definidos en la gramática. Para verificar su corrección, se puede usar una herramienta en línea como `https://jsmachines.sourceforge.net/machines/ll1.html`, ingresando la gramática en notación BNF y generando la tabla de parsing LL(1). Los ejemplos de código proporcionados (e.g., "Hola mundo", "Bucles anidados") se parsean correctamente, confirmando que la gramática es LL(1) y que la tabla no presenta conflictos.

El script `generador-ll1.py` implementa el algoritmo para generar la tabla, como se muestra en el siguiente fragmento:

Listing 4: Fragmento de generador-ll1.py para generar la tabla LL(1)

```python
# Rellenar la tabla LL(1) con las producciones correspondientes
for lhs in gramatica_general:
    for production in gramatica_general[lhs]:
        firsts = set()
        if production[0] == 'e':
            firsts.add('e')
        else:
            for sym in production:
                sym_first = compute_first(sym)
                firsts.update(sym_first - set(['e']))
                if 'e' not in sym_first:
                    break
            else:
                firsts.add('e')
        # Llenar la tabla para cada terminal en FIRST
        for terminal in firsts - set(['e']):
            ll1_table[lhs][terminal] = ' '.join(production)
```

```
        # Si 'e' está en FIRST, se rellena también para los terminales en FOLLOW
        if 'e' in firsts:
            for terminal in FOLLOW[lhs]:
                ll1_table[lhs][terminal] = 'e'
```

3.3. Implementación del analizador sintáctico

El analizador sintáctico de **FusionCod** utiliza la tabla LL(1) generada para procesar una lista de tokens y construir un árbol sintáctico. Está implementado en Python en el archivo `sintactico.py`, basado en un algoritmo de análisis descendente predictivo. El proceso comienza con el símbolo inicial (`programaprincipal`) y utiliza una pila para gestionar los no terminales y terminales, comparándolos con los tokens de entrada.

A continuación, se presenta una versión ampliada de la implementación del analizador sintáctico:

Listing 5: Implementación del analizador sintáctico en sintactico.py

```
def analizador_sintactico(lista_de_tokens, tabla_ll1):
    pila = []
    inicial = tabla_ll1.index[0] # Símbolo inicial (programaprincipal)
    contador = 0
    # Crear nodos iniciales para el símbolo de fin y el símbolo inicial
    nodo_dolar = Nodo(contador, "$", None, None, None, True)
    nodo_inicio = Nodo(contador + 1, inicial, None, None, None, False)
    pila.append(nodo_dolar)
    pila.append(nodo_inicio)

    lista\_de\_tokens.append(Token("\$", "\$", 0, 0))
    # Aniadir símbolo de fin
    lista_errores_sintacticos = []
    indice = 0
    arbol = []

    while pila:
        elemento_actual = pila.pop()
        if elemento_actual.es_terminal:
            if indice < len(lista_de_tokens):
                token_actual = lista_de_tokens[indice]
                if elemento_actual.nombre == token_actual.type:
                    arbol.append(elemento_actual)
                    indice += 1
                else:
                    lista_errores_sintacticos.append(
                        f"Error sintáctico en línea {token_actual.lineno}: se esperaba
                            {elemento_actual.nombre}, pero se encontró {token_actual.type}"
                    )
                    indice += 1
            else:
                lista_errores_sintacticos.append("Error: se esperaba un token, pero la entrada
                    terminó")
        else:
            if indice < len(lista_de_tokens):
                token_actual = lista_de_tokens[indice]
                if elemento_actual.nombre in tabla_ll1.index and token_actual.type in
                    tabla_ll1.columns:
                    produccion = tabla_ll1.loc[elemento_actual.nombre, token_actual.type]
```

```
                if produccion:
                    simbolos = produccion.split()
                    simbolos.reverse()
                    hijos = []
                    for simbolo in simbolos:
                        contador += 1
                        es_terminal = simbolo in lista_terminales
                        nodo = Nodo(contador, simbolo, None, None, None, es_terminal)
                        pila.append(nodo)
                        hijos.append(nodo)
                    elemento_actual.hijos = hijos
                    arbol.append(elemento_actual)
                else:
                    lista_errores_sintacticos.append(
                        f"Error sintáctico en línea {token_actual.lineno}: no se encontró
                            producción para {elemento_actual.nombre} con
                            {token_actual.type}"
                    )
                    indice += 1
            else:
                lista_errores_sintacticos.append(
                    f"Error sintáctico: no se encontró entrada en la tabla para
                        {elemento_actual.nombre} y {token_actual.type}"
                )
                indice += 1
        else:
            lista_errores_sintacticos.append("Error: entrada terminada antes de completar
                el análisis")

    return arbol, lista_errores_sintacticos
```

3.4. Implementación del árbol sintáctico

Durante el análisis sintáctico, se construye un árbol sintáctico que representa la estructura jerárquica del código fuente. Cada nodo del árbol es una instancia de la clase `Nodo`, que almacena el nombre del símbolo, sus hijos, y si es terminal o no terminal. El árbol se genera dinámicamente mientras se aplican las producciones de la tabla LL(1), y al final se exporta a un archivo en formato `.dot` para su visualización con herramientas como Graphviz.

El siguiente fragmento de `sintactico.py` muestra cómo se genera el archivo `.dot`:

Listing 6: Generación del árbol sintáctico en sintactico.py

```
def generar_dot(arbol, nombre_archivo):
    with open(f"{nombre_archivo}.dot", "w", encoding='utf-8') as f:
        f.write("digraph G {\n")
        for nodo in arbol:
            f.write(f'  "{nodo.id}_{nodo.nombre}" [label="{nodo.nombre}"];\n')
            if nodo.hijos:
                for hijo in nodo.hijos:
                    f.write(f'  "{nodo.id}_{nodo.nombre}" -> "{hijo.id}_{hijo.nombre}";\n')
        f.write("}\n")
```

Este archivo `.dot` puede visualizarse con Graphviz para obtener una representación gráfica del árbol sintáctico, facilitando la depuración y verificación del análisis.

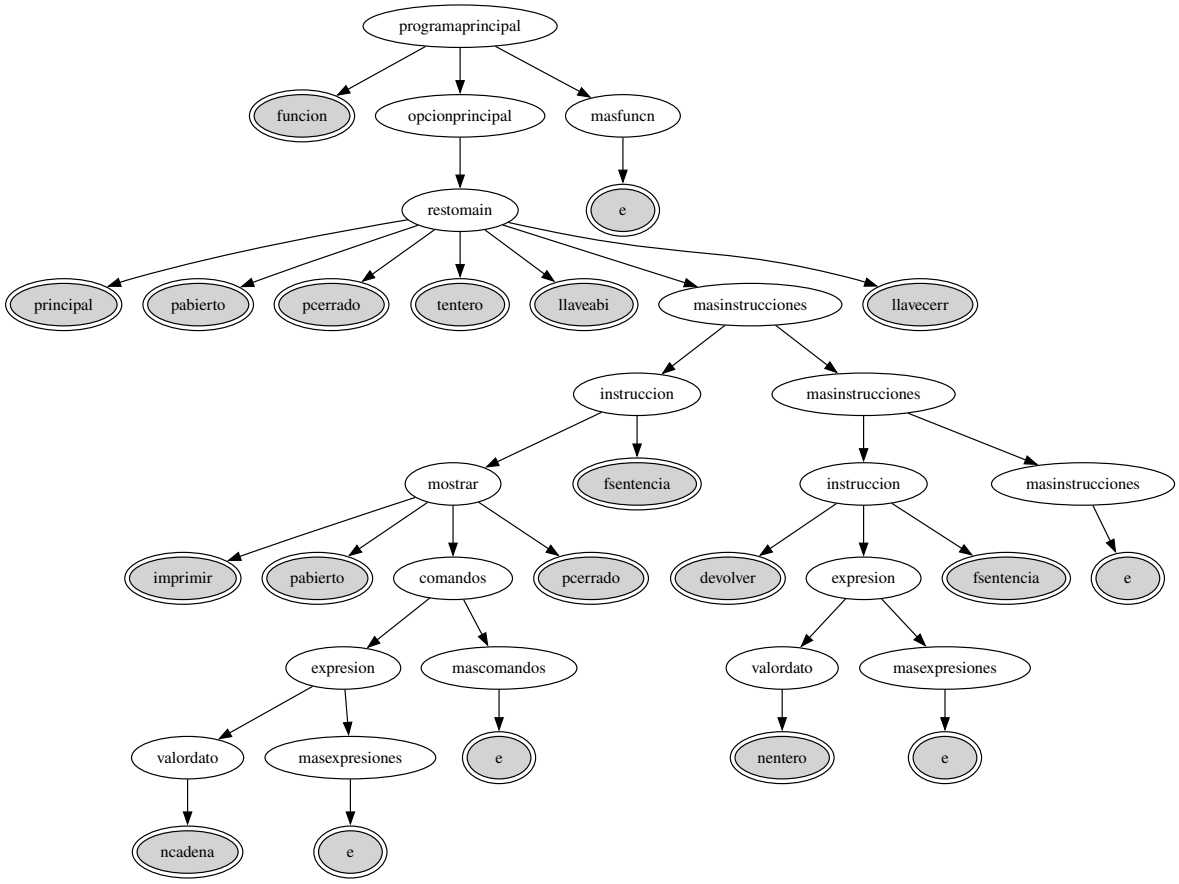
4. Ejemplos

A continuación, se presentan ejemplos de código en **FusionCod** que demuestran su funcionalidad y el arbol sintactico respectivo:

4.1. Hola Mundo

Listing 7: Hola mundo

```
fn main() int {
    show("Hola mundo");
    return 0;
}
```

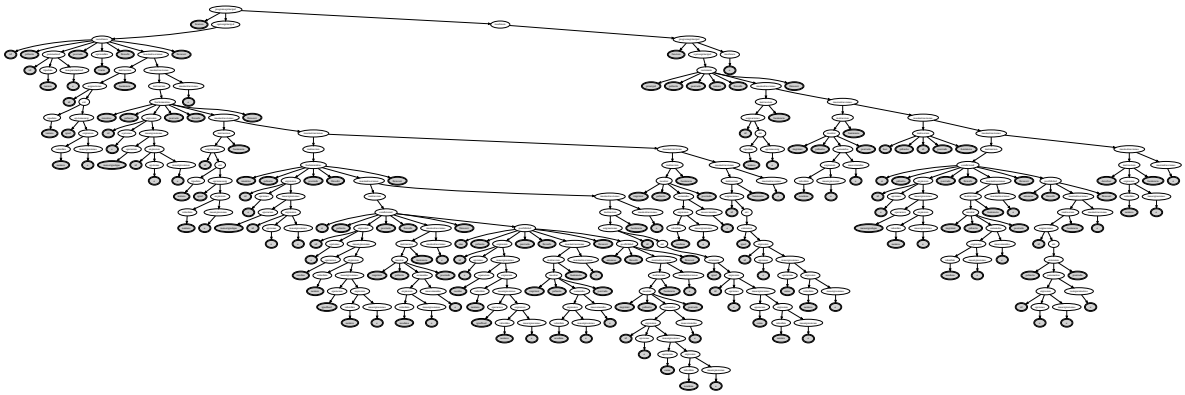


4.2. Bucles Anidados

Listing 8: Bucles anidados

```
fn generar_patron(n int) void {
    i int = 1;
    while (i <= n) {
        j int = 1;
        while (j <= i) {
            if (j % 2 == 0) {
```

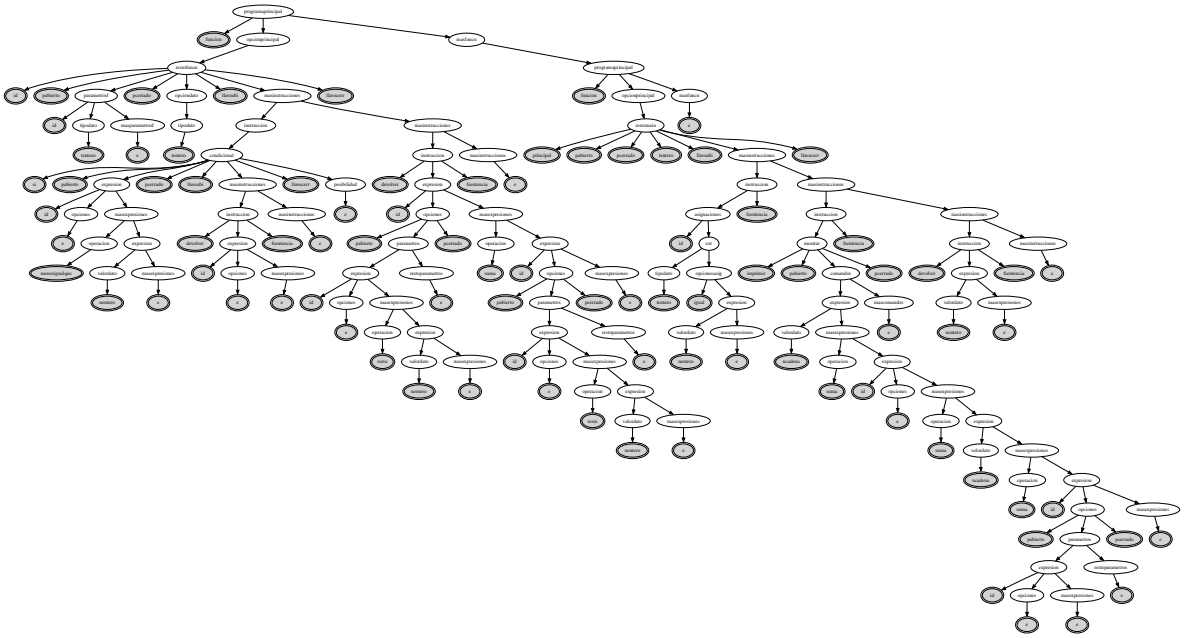
```
            show("* ");
        } elif (j % 3 == 0) {
            show("# ");
        } else {
            show(j + " ");
        }
        j = j + 1;
    }
    show("\n");
    i = i + 1;
  }
}
fn main() int {
  num int;
  show("Ingrese un número para generar el patrón: ");
  read(num);
  if (num <= 0) {
    show("El número debe ser mayor que 0.");
  } else {
    generar_patron(num);
  }
  return 0;
}
```



4.3. Fibonacci Recursivo

Listing 9: Fibonacci recursivo

```
fn fibonacci_recursivo(n int) int {
  if (n <= 1) {
    return n;
  }
  return fibonacci_recursivo(n - 1) + fibonacci_recursivo(n - 2);
}
fn main() int {
  numero int = 5;
  show("La secuencia fibonacci en " + numero + " es: " + fibonacci_recursivo(numero));
  return 0;
}
```



5. Repositorio

El código fuente del analizador léxico y sintáctico, junto con los ejemplos, está disponible en el siguiente repositorio:

- URL: `https://github.com/JersonCh1/compiladores-25-I.git`