Diseño y desarrollo de un lenguaje de programación

Parcial: primera parte

Nota

Estudiante	Escuela	Asignatura
Rodrigo Infanzón Acosta rinfanzona@ulasalle.edu.pe	Carrera Profesional de Ingeniería de Software	Compiladores

Informe	Tema	Duración
02	Parcial: primera parte	04 horas

Semestre académico	Fecha de inicio	Fecha de entrega
2024 - B	30/09/24	22/10/24

Índice

1. Introducción

En un mundo donde la conectividad y la comunicación son esenciales para el funcionamiento de casi todas nuestras actividades cotidianas de hoy en dia, las telecomunicaciones y la conectividad con el internet se han convertido en un campo de vital importancia. La necesidad de herramientas específicas para abordar los desafíos técnicos de este sector ha impulsado al desarrollo de **NetCode** en una primera instancia, en la cual esta desarrollado en un enfoque particular y amigable a las **telecomunicaciones y al mundo WISP**. Este informe se enfoca principalmente en presentar **NetCode**, describiendo su relevancia en el contexto actual, sus objetivos y la propuesta que ofrece para mejorar la programación en este campo.

1.1. Justificación

La rápida evolución tecnológica en **telecomunicaciones** ha incrementado significativamente la complejidad de los sistemas de comunicación. Las herramientas de programación actuales frecuentemente no satisfacen plenamente las demandas específicas del sector, especialmente en áreas críticas como la manipulación de señales, la gestión de paquetes de datos y la simulación de redes.

Además, **los proveedores de servicios de Internet inalámbrico (WISP)** enfrentan desafíos únicos. Estos proveedores deben gestionar la transmisión de datos en entornos variables, lidiar con interferencias de señal y optimizar el uso del espectro radioeléctrico. La naturaleza dinámica y a menudo impredecible de las redes inalámbricas exige soluciones especializadas que permitan una gestión eficiente y efectiva.

1.2. Objetivo

En este contexto, **NetCode** tiene como objetivo brindar una sintaxis específicamente adaptada para abordar las necesidades del sector de **telecomunicaciones** y, en particular, para el mundo **WISP**, **NetCode** ofrece una sintaxis elaborada para este ambito, que a futuro tambien pueda ofrecer herramientas de gestión, seguridad informatica, transmición de datos, optimización de redes inalámbricas y la gestión de sistemas de comunicación avanzados.

2. Propuesta

La propuesta actual de **NetCode** se basa principalmente en la integración de funcionalidades específicas para las telecomunicaciones, y se fundamenta en las características de los lenguajes **C++ y Go (Golang)**. **NetCode** aprovechará la robustez y el control de bajo nivel de **C++** para la manipulación

eficiente de señales y la gestión de datos, mientras que se beneficiará de la simplicidad y las capacidades de concurrencia de **Go** para la simulación detallada de redes y la gestión de paquetes de datos.

3. Especificación lexica

En esta sección, se detallará la especificación léxica de **NetCode**, abordando los aspectos fundamentales que definen el conjunto de símbolos y las reglas de formación de los elementos del lenguaje. A continuación, se detalla cada punto:

3.1. Definición de comentarios

La sintaxis de uno o varios comentarios en **NetCode** (realizados de manera lineal) será de la siguiente manera:

Listing 1: Comentarios en NetCode

```
// programa NetCode, hola mundo

program main interger [] {
   log["hola mundo"]
   echo 0
}
```

Los comentarios en **NetCode** seran escritos después de poner //

3.2. Definición de identificadores

Los identificadores (id) en **NetCode** tendran las siguientes características:

- Los identificadores deben iniciar siempre con una letra, ya sea minuscula o mayúscula.

- Después, pueden incluir o: números (0-9), letras reconocidas en el abecedario (A-Z) o (a-z) o sub guiones (_).

- No pueden existir espacios en blanco en los identificadores.

- Los identificadores **no** pueden ser palabras reservadas de **NetCode**.

Ejemplo:

Listing 2: Identificadores en NetCode

```
// programa NetCode, creacion y asignacion de ids

program main interger [] {
    var numero_01 interger
    var Nsa1_ texto
    var a01 decimal = 10.90
   echo
}
```

3.3. Definición de palabras reservadas

Las palabras clave en **NetCode** serán las siguientes:

- **function** : representa la definición de una función en **NetCode**.

- **main** : representa el punto de entrada principal del programa en **NetCode**.

- **void** : representa que una función no retorna ningún valor en especifico en **NetCode**.

- **log** : representa el poder imprimir un ente en **NetCode**, ya sea variables, números, funciones, etc.

- **echo** : representa (en algunos casos) el retorno de un valor especifico.

- **stop** : representa (en bucles) de **NetCode** el detenimiento del mismo.

- **for** : representa el inicio de una iteración definida en en **NetCode**.

- **if** : representa la evaluación de una condición en **NetCode**, si se llega a cumplir se ejecutará un bloque de código.

- **or** : se utiliza para realizar una operación lógica de disyunción en **NetCode**.

- **true** : representa un valor lógico de afirmación o veracidad. Indica que una condición o expresión lógica es verdadera en **NetCode**.

- **false** : representa un valor lógico de negación o falsedad. Indica que una condición o expresión lógica no es verdadera en **NetCode**.

- **and** : se utiliza para realizar una operación lógica de conjunción en **NetCode**.

- **else** : se utiliza en estructuras condicionales para especificar un bloque de código que se ejecuta si la condición del **if** no se cumple.

- **elif** : se usa como una condición alternativa en una estructura **if-else** en **NetCode**.

- **while** : inicia un bucle que se ejecuta mientras se cumpla una condición en **NetCode**.

- **integer** : representa un tipo de dato para números enteros en **NetCode**.

- **decimal** : representa un tipo de dato para números de punto flotante en **NetCode**.

- **boolean** : representa un tipo de dato para valores lógicos, es decir, **true** o **false** en **NetCode**.

- **text** : representa un tipo de dato para cadenas de caracteres en **NetCode**.

- **program** :

- **var** :

Esta lista define las palabras clave del lenguaje **NetCode** y sus funciones específicas.

Ejemplo:

Listing 3: Algunas palabras clave en NetCode

```
// programa NetCode, algunas palabras clave
function factorial[n interger] interger {
   result interger = 1
   for[i interger;i<=n;i++] {
      result = result * i
   }
   echo result
}
function main [] {
   numero interger = 5
   log["El factorial de " $ numero " es: " $ factorial[numero]]
   echo
}
```

3.4. Definición de literales

Los literales en **NetCode** son valores constantes utilizados directamente en el código fuente y representan datos específicos:

- **Literales numéricos** : representan valores numéricos y pueden ser enteros o de punto flotante. Por ejemplo, `42` o `3.14`.

- **Literales de cadenas** : representan secuencias de caracteres y se encierran entre comillas dobles. Por ejemplo, `"Hello, World!"` y `"1234"`.

- **Literales booleanos** : representan valores lógicos, específicamente `true` o `false`.

3.5. Definición de operadores

En **NetCode**, los operadores son símbolos que realizan operaciones sobre uno o más operandos. Estos operadores permiten la manipulación de datos y la implementación de lógica en el código. A continuación, se describen los principales operadores disponibles en **NetCode**:

3.5.1. Operadores aritméticos

Se utilizan para realizar operaciones matemáticas básicas. Tales como:

`+` : suma dos operandos. Ejemplo: `a + b`.

`-` : resta el segundo operando del primero. Ejemplo: `a - b`.

`*` : multiplica dos operandos. Ejemplo: `a * b`.

`/` : divide el primer operando por el segundo. Ejemplo: `a / b`.

`%` : devuelve el residuo de la división entre dos operandos. Ejemplo: `a% b`.

3.5.2. Operadores de comparación

Se utilizan para comparar dos operandos y retornan un valor booleano. Tales como:

`==` : verifica si dos operandos son iguales. Ejemplo: `a == b`.

`!=` : verifica si dos operandos son diferentes. Ejemplo: `a != b`.

`>` : verifica si el primer operando es mayor que el segundo. Ejemplo: `a >b`.

`<` : verifica si el primer operando es menor que el segundo. Ejemplo: `a <b`.

`>=` : verifica si el primer operando es mayor o igual que el segundo. Ejemplo: `a >= b`.

`<=` : verifica si el primer operando es menor o igual que el segundo. Ejemplo: `a <= b`.

3.5.3. Operadores lógicos

Se utilizan para combinar expresiones booleanas. Tales como:

`and` : realiza una operación lógica de conjunción. Ejemplo: `a and b`.

`or` : realiza una operación lógica de disyunción. Ejemplo: `a or b`.

3.5.4. Operadores de asignación

Se utilizan para asignar valores a variables. Tales como:

`=` : asigna el valor del operando derecho al operando izquierdo. Ejemplo: `a = b`.

3.5.5. Operadores de incremento y decremento

Se utilizan para aumentar o disminuir el valor de una variable en solo el bucle for. Tales como:

`++` : incrementa el valor de una variable en 1. Ejemplo: `a++`.

`--` : decrementa el valor de una variable en 1. Ejemplo: `a--`.

Ejemplos:

Listing 4: Algunos operadores en NetCode

```
function verificar_numero [var n interger] text {
    if [n > 0] {
        echo "El nmero es positivo"
    } elif [n < 0] {
        echo "El nmero es negativo"
    } else {
        echo "El nmero es cero"
    }
}
program main interger [] {
    var numero interger = -3
    log["El nmero ingresado es: " $ numero $ ". Resultado: " $ verificar_numero[numero]]
    echo 0
}
```

Listing 5: Mezclas en NetCode

```
function proceso_anidado [] interger {
    var suma interger = 0
    var i interger = 0
    var j interger = 0
    var k interger = 0
    for [i = 1; i <= 3; i = i + 1] {
        var count_i interger = i
        while [j < 2] {
            j = j + 1
            var count_j interger = j
            while [k < 2] {
                k = k + 1
                var count_k interger = k
                if [i == 2 and j == 1 and k == 2] {
                    log["Condicion especial alcanzada: i=" $ i $ ", j=" $ j $ ", k=" $ k]
                } elif [i == 3 and j == 2 and k == 1] {
                    log["Otra condicion especial alcanzada: i=" $ i $ ", j=" $ j $ ", k=" $ k]
                } else {
                    log["Sumando valores: i=" $ i $ ", j=" $ j $ ", k=" $ k]
                    suma = suma + i + j + k
                }
            }
            k = 0
        }
        j = 0
    }
    echo suma
}
program main interger [] {
```

```
    var resultado interger = proceso_anidado[]
    log["El resultado de la suma es: " $ resultado]
    echo 0
}
```

Listing 6: Algunos operadores en NetCode

```
function es_primo [var n interger] boolean {
    if [n <= 1] { echo false }
    for [var i interger = 2; i * i <= n; i = i + 1] {
        if [n % i == 0] { echo false }
    }
    echo true
}
program main interger [] {
    var numero interger = 11
    log[es_primo[numero]]
    echo 0
}
```

3.6. Expresiones regulares

Token	Expresión regular
FUNCION	function
PROGRAMA	program
PRINCIPAL	main
CORCHETEABI	[
CORCHETECERR	]
IMPRIMIR	log
COMA	,
ID	[A-Z-a-z]$^+$([$A-Z$-$a-z$] | [$0-9$] | _)*
VARIABLE	var
FINALSENTENCIA	;
RETORNAR	echo
DETENER	stop
LLAVEABI	{
LLAVECERR	}
TIPOENTERO	interger
TIPOCADENA	text
TIPODECIMAL	decimal
TIPOBOOLEANO	boolean
TIPOVACIO	void
SI	if
Y	and
O	or
NO	not
SINO	elif
ENTONCES	else
MIENTRAS	while
PARA	for
SUMA	+
RESTA	-
MULTIPLICACION	*
DIVISION	/
RESIDUO	%
MENORQUE	<
MAYORQUE	>
MENORIGUALQUE	<=
MAYORIGUALQUE	>=
IGUAL	=
IGUALBOOLEANO	==
DIFERENTEDE	!=
AUMENTAR	++
DISMINUIR	--
CONCATENAR	$
NENTERO	[0-9]$^+$ | $-$[$0-9$]$^+$
NDECIMAL	[0-9]$^+$.[$0-9$]$^+$ | $-$[$0-9$]$^+$.[$0-9$]$^+$
NCADENA	" (^") "
NBOOLEANO	true | $false$

Los comentarios en expresión regular es `(//.*)`, sin embargo este se omite en el análisis del código.

3.7. Implementación léxica

La implementación léxica es una etapa clave en el proceso de compilación, donde se identifican y clasifican los distintos componentes del código fuente en unidades básicas llamadas tokens. Estos tokens representan palabras clave, operadores, identificadores, literales y otros elementos fundamentales del lenguaje de programación, permitiendo así la construcción de una estructura sintáctica más compleja en etapas posteriores. A continuación, se muestran partes importantes del código de la implementación lexica:

La clase Token define la estructura de un token con atributos para su tipo, valor, línea y columna, mientras que la función generate_data lee el contenido de un archivo especificado y maneja la excepción si el archivo no se encuentra:

Listing 7: Clase Token y función para generar datos desde un archivo

```
# Clase Token
class Token:
    def __init__(self, type, value, line, column):
        self.type = type
        self.value = value
        self.line = line
        self.column = column

# Funcion que genera una data a partir de un boceto de lenguaje
def generate_data(name_pathfile):
    try:
        with open(name_pathfile, 'r') as file:
            data = file.read()
    except FileNotFoundError:
        print(f"Error: El archivo '{name_pathfile}' no se encontr.")
        data = ''
    return data
```

Por otro lado, este código utiliza un lexer para generar tokens desde un archivo de entrada, creando objetos de tipo Token que son almacenados en una lista para su posterior procesamiento en el analizador sintáctico:

Listing 8: Generación de tokens utilizando un lexer

```
lexer = lex.lex()
lexer.input(generate_data(pathfile))
listtokens = []

def generate_tokens(list_tokens):
    while True:
        tok = lexer.token()
        if not tok: break
        token_obj = Token(tok.type, tok.value, tok.lineno, tok.lexpos)
        list_tokens.append(token_obj)
    print("Tokens generados correctamente.\n")

generate_tokens(listtokens)
```

4. Especificación sintáctica

4.1. Definición de la gramática

La gramática presentada define la sintaxis de Netcode, un lenguaje de programación estructurado que permite la creación de programas que incluyen funciones, bucles, condicionales y operaciones básicas. Este lenguaje está diseñado para facilitar la definición de tipos de datos enteros, decimales, cadenas y booleanos, así como la inclusión de estructuras de control de flujo.

4.1.1. Estructura general

La gramática se divide en varias producciones que especifican las reglas para la formación de distintas construcciones del lenguaje:

- **NETCODE**: representa el código neto del programa, que puede contener múltiples funciones o una función principal (MAIN).
- **FUNC**: define la estructura de una función, incluyendo su nombre, parámetros y el cuerpo de la función.
- **MAIN**: Es la entrada del programa, que siempre debe ser de tipo entero y define el cuerpo principal donde se ejecutan las instrucciones.
- **Instrucciones y Control de Flujo**: se incluyen construcciones para condicionales (CONDICIONAL), bucles (BUCLEWHILE, BUCLEFOR) y asignaciones (ASIG).
- **Expresiones**: la gramática permite la definición de expresiones matemáticas y lógicas a través de una serie de operaciones y funciones.

4.1.2. Componentes clave

La gramática de Netcode incluye varios componentes esenciales que permiten a los desarrolladores construir programas complejos:

- **Tipos de Datos**: se definen varios tipos de datos, como TIPOENTERO, TIPOCADENA, TIPODECIMAL y TIPOBOOLEANO, lo que permite a los programadores manejar diferentes tipos de información de manera eficiente.
- **Operaciones y Expresiones**: Netcode permite una amplia gama de operaciones, incluyendo aritméticas (como SUMA, RESTA, MULTIPLICACION, y DIVISION) y lógicas (como Y, O y comparaciones). Estas operaciones se pueden combinar para formar expresiones complejas que son evaluadas durante la ejecución del programa.
- **Control de Flujo**: la gramática incluye construcciones de control de flujo que permiten a los programadores dirigir la ejecución del código mediante condicionales (SI ... SINO) y bucles (MIENTRAS, PARA). Esto permite crear programas dinámicos y reactivos.
- **Funciones**: Netcode permite la definición de funciones (FUNC) que pueden recibir parámetros y devolver resultados, facilitando la modularidad y la reutilización de código. Las funciones pueden ser anidadas, lo que permite una mayor flexibilidad en la organización del código.

4.1.3. Gramática de Netcode

La gramática de Netcode se define de la siguiente manera:

Listing 9: Gramatica

```
NETCODE -> FUNC MASFUNCIONES MAIN
NETCODE -> MAIN

MASFUNCIONES -> FUNC MASFUNCIONES
MASFUNCIONES -> ''

FUNC -> FUNCION ID CORCHETEABI PARAMETROS CORCHETECERR OPDATO LLAVEABI INS LLAVECERR

MAIN -> PROGRAMA PRINCIPAL TIPOENTERO CORCHETEABI CORCHETECERR LLAVEABI INS LLAVECERR

OPDATO -> TIPOVACIO
OPDATO -> TIPODATO

PARAMETROS -> ''
PARAMETROS -> VARIABLE ID TIPODATO MASPARAMETROS

MASPARAMETROS -> COMA PARAMETROS
MASPARAMETROS -> ''

CONDICIONAL -> SI CORCHETEABI EXPRESION CORCHETECERR LLAVEABI INS LLAVECERR POSIBILIDAD
POSIBILIDAD -> SINO CORCHETEABI EXPRESION CORCHETECERR LLAVEABI INS LLAVECERR POSIBILIDAD
POSIBILIDAD -> ENTONCES LLAVEABI INS LLAVECERR
POSIBILIDAD -> ''

BUCLEWHILE -> MIENTRAS CORCHETEABI EXPRESION CORCHETECERR LLAVEABI INS LLAVECERR

BUCLEFOR -> PARA CORCHETEABI ASIG PUNTOYCOMA EXPRESION PUNTOYCOMA ID DERIVA2 CORCHETECERR
    LLAVEABI INS LLAVECERR

INS -> INSTRUCCION MASINSTRUCCION
INS -> ''

DERIVA2 -> UNARIOS
DERIVA2 -> IGUAL EXPRESION

INSTRUCCION -> ASIG
INSTRUCCION -> IMP
INSTRUCCION -> BUCLEFOR
INSTRUCCION -> BUCLEWHILE
INSTRUCCION -> CONDICIONAL
INSTRUCCION -> DETENER
INSTRUCCION -> RETORNAR EXPRESION

MASINSTRUCCION -> INSTRUCCION MASINSTRUCCION
MASINSTRUCCION -> ''

IMP -> IMPRIMIR CORCHETEABI CMDS CORCHETECERR

CMDS -> ''
CMDS -> EXPRESION MASCOMANDOS

```

```
MASCOMANDOS -> CONCATENAR EXPRESION MASCOMANDOS
MASCOMANDOS -> ''

ASIG -> VARIABLE ID TIPODATO OPC
ASIG -> ID OG
OG -> IGUAL EXPRESION
OG -> CORCHETEABI PAR CORCHETECERR

OPC -> IGUAL EXPRESION
OPC -> ''

EXPRESION -> CORCHETEABI EXPRESION CORCHETECERR MASEXPRESION
EXPRESION -> ID OPCION MASEXPRESION
EXPRESION -> DATO MASEXPRESION

MASEXPRESION -> ''
MASEXPRESION -> OPERACION EXPRESION

OPCION -> CORCHETEABI PAR CORCHETECERR
OPCION -> ''

PAR -> TF RESTO_PARAMETROS
PAR -> ''

RESTO_PARAMETROS -> COMA TF RESTO_PARAMETROS
RESTO_PARAMETROS -> ''

TF -> DATO MASEXPRESION
TF -> ID OPCION MASEXPRESION

OPERACION -> SUMA
OPERACION -> RESTA
OPERACION -> MULTIPLICACION
OPERACION -> DIVISION
OPERACION -> RESIDUO
OPERACION -> IGUALBOOLEANO
OPERACION -> MENORQUE
OPERACION -> MAYORQUE
OPERACION -> MENORIGUALQUE
OPERACION -> MAYORIGUALQUE
OPERACION -> DIFERENTEDE
OPERACION -> Y
OPERACION -> O

DATO -> NCADENA
DATO -> NDECIMAL
DATO -> NENTERO
DATO -> NBOOLEANO

TIPODATO -> TIPOENTERO
TIPODATO -> TIPOCADENA
TIPODATO -> TIPODECIMAL
TIPODATO -> TIPOBOOLEANO

UNARIOS -> AUMENTAR
UNARIOS -> DISMINUIR
```

4.2. Implementación de la tabla sintáctica

Esta tabla se construye a partir de una gramática LL(1), elaborada y probada utilizando una herramienta proporcionada por el profesor para verificar que cumpla con las condiciones necesarias. Posteriormente, esta tabla se utiliza en el proceso de análisis sintáctico dentro de un compilador. La tabla LL(1) permite predecir la producción correcta de una gramática durante el análisis de las entradas, garantizando que el proceso sea eficiente y sencillo. **Cabe aclarar que mediante el uso de los tokens de lexer, podremos ayudar a identificar los simbolos terminales:**

Listing 10: Uso de los tokens de lexer

```
import os
import csv
from collections import defaultdict
from lexic import tokens
from functions_sintactic import generate_syntax_table_png
```

Defaultdict es una versión mejorada del diccionario (dict) que facilita el manejo de claves que no existen previamente en el diccionario. En un diccionario normal, si intentas acceder a una clave que no existe, obtendrás un error. Sin embargo, con defaultdict, puedes especificar un valor predeterminado que se asignará automáticamente a las nuevas claves cuando se intente acceder a ellas.

En el contexto de la implementación de la tabla LL(1), defaultdict se utiliza para simplificar la creación de las entradas de la tabla y los conjuntos FIRST y FOLLOW, evitando errores cuando se intenta acceder a claves que aún no han sido definidas.

4.2.1. Descripción general

El código se estructura en varias secciones principales, cada una con un papel clave en la generación de la tabla LL(1). Estas secciones incluyen:

Lectura y procesamiento de la gramática desde un archivo de texto: el código cuenta con dos funciones principales para la lectura de gramáticas: read_grammar_word_file y read_grammar_comun_file. Ambas funciones leen la gramática desde un archivo de texto y procesan las producciones especificadas, permitiendo el uso de diferentes formatos (`->` o `::=`):

Listing 11: Funciones de lectura y procesamiento de gramaticas

```
def read_grammar_word_file(filename):
    grammar = {}
    with open(filename, 'r', encoding='utf-8') as file:
        for line in file:
            # Eliminar comentarios y espacios en blanco
            line = line.strip()
            if not line or line.startswith('#'):
                continue # Saltar lneas  vacas  y comentarios
            # Manejar diferentes operadores de produccion (-> o ::=)
            if '->' in line:
                lhs, rhs = line.split('->', 1)
            elif '::=' in line:
                lhs, rhs = line.split('::=', 1)
            else:
                continue # Saltar lneas no  vlidas
            lhs = lhs.strip()
            rhs = rhs.strip()
            # Dividir la produccion en smbolos
            symbols = rhs.split()
```

```
            # Reemplazar '' por 'e' en producciones vacias
            symbols = ['e' if symbol == "''" else symbol for symbol in symbols]
            # Agregar cada produccion de manera separada
            if lhs not in grammar:
                grammar[lhs] = []
            grammar[lhs].append(symbols)
    return grammar

def read_grammar_comun_file(filename):
    grammar = {}
    with open(filename, 'r', encoding='utf-8') as file:
        for line in file:
            # Eliminar comentarios y espacios en blanco
            line = line.strip()
            if not line or line.startswith('#'):
                continue # Saltar lineas vacias y comentarios
            if '->' not in line:
                continue # Saltar lineas no validas
            lhs, rhs = line.split('->', 1)
            lhs = lhs.strip()
            rhs = rhs.strip()
            productions = rhs.split('|')
            grammar.setdefault(lhs, [])
            for production in productions:
                # Dividir la produccion en simbolos
                symbols = production.strip().split()
                grammar[lhs].append(symbols)
    return grammar
```

Cálculo de los conjuntos FIRST y FOLLOW de cada símbolo no terminal de la gramática:

Conjunto FIRST: Se calcula utilizando la función compute_first, la cual determina los símbolos terminales que pueden aparecer al inicio de una producción. Para cada no terminal, la función recursivamente determina el conjunto de símbolos terminales que pueden derivarse. El código relevante se muestra a continuación:

Listing 12: FIRST

```
FIRST = defaultdict(set)
FOLLOW = defaultdict(set)

# Funciones para calcular FIRST y FOLLOW
def compute_first(symbol):
    if symbol in FIRST and FIRST[symbol]:
        return FIRST[symbol]
    if symbol in tokens:
        FIRST[symbol] = set([symbol])
        return FIRST[symbol]
    first = set()
    for production in grammar[symbol]:
        if production[0] == 'e':
            first.add('e')
        else:
            for sym in production:
                sym_first = compute_first(sym)
                first.update(sym_first - set(['e']))
                if 'e' not in sym_first:
```

```
                break
            else:
                first.add('e')
    FIRST[symbol] = first
    return first
```

Conjunto FOLLOW: La función compute_follow se encarga de calcular los símbolos terminales que pueden seguir a un no terminal en una derivación. Se asegura que el símbolo inicial de la gramática tenga el símbolo de fin de entrada ($) en su conjunto FOLLOW. El cálculo se realiza de la siguiente manera:

Listing 13: FOLLOW

```
def compute_follow(symbol):
    if symbol == list(grammar.keys())[0]: # FOLLOW del smbolo inicial
        FOLLOW[symbol].add('$')
    for lhs in grammar:
        for production in grammar[lhs]:
            for i, sym in enumerate(production):
                if sym == symbol:
                    if i + 1 < len(production):
                        next_sym = production[i + 1]
                        next_first = compute_first(next_sym)
                        FOLLOW[symbol].update(next_first - set(['e']))
                        if 'e' in next_first:
                            FOLLOW[symbol].update(FOLLOW[lhs])
                    else:
                        if lhs != symbol:
                            FOLLOW[symbol].update(FOLLOW[lhs])

# Calcular FIRST para todos los simbolos
for non_terminal in non_terminals:
    compute_first(non_terminal)

# Calcular FOLLOW para todos los simbolos
for non_terminal in non_terminals:
    FOLLOW[non_terminal] = set()
```

El código utiliza un bucle iterativo para calcular los conjuntos FOLLOW, hasta que estos no cambien más, asegurando que todos los símbolos tengan un conjunto FOLLOW consistente y correcto:

Listing 14: Ciclo para Calcular FOLLOW Completo

```
changed = True
while changed:
    changed = False
    for non_terminal in non_terminals:
        before = len(FOLLOW[non_terminal])
        compute_follow(non_terminal)
        after = len(FOLLOW[non_terminal])
        if before != after:
            changed = True
```

Esto se hace porque al agregar elementos a un conjunto FOLLOW, estos pueden afectar otros símbolos, por lo que se debe recalcular hasta que todos los FOLLOW se estabilicen.

Construcción de la tabla LL(1): para cada producción de la gramática, se determina el conjunto FIRST de la producción, y se rellenan las celdas correspondientes de la tabla. Si una producción contiene la producción vacía (e), se utilizan los símbolos del conjunto FOLLOW para rellenar la tabla. A continuación se muestra la lógica principal utilizada para construir la tabla LL(1): conjunto FOLLOW consistente y correcto:

Listing 15: Construcción de la Tabla LL(1)

```
ll1_table = defaultdict(dict)

for lhs in grammar:
    for production in grammar[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')

        for terminal in firsts - set(['e']):
            ll1_table[lhs][terminal] = ' '.join(production)
        if 'e' in firsts:
            for terminal in FOLLOW[lhs]:
                ll1_table[lhs][terminal] = 'e'
```

Este fragmento de código construye la tabla LL(1) asegurándose de llenar correctamente cada celda según las reglas de la gramática y utilizando los conjuntos FIRST y FOLLOW para definir las entradas. Esto garantiza que, al realizar el análisis sintáctico, se puedan tomar decisiones basadas en los símbolos actuales de la entrada y el estado del análisis.

Exportación de la tabla a un archivo CSV: la tabla LL(1) se exporta a un archivo CSV para facilitar su visualización y análisis posterior. El código incluye una funcionalidad para crear un directorio de salida y almacenar el archivo en él:

Listing 16: Exportacion a .csv

```
with open(archivo_salida, 'w', newline='', encoding='utf-8') as csvfile:
    csvwriter = csv.writer(csvfile)
    # Escribir encabezados
    headers = [''] + tokens
    csvwriter.writerow(headers)
    # Escribir filas de la tabla
    for idx, nt in enumerate(non_terminals):
        row = [nt]
        for t in tokens:
            action = ll1_table[nt].get(t, '')
            row.append(action)
        csvwriter.writerow(row)

# Reabrir el archivo en modo lectura/escritura y eliminar la ultima linea si esta vacia
with open(archivo_salida, 'rb+') as csvfile:
    csvfile.seek(-2, os.SEEK_END)
```

```
    while csvfile.tell() > 0 and csvfile.read(1) in [b'\n', b'\r']:
        csvfile.seek(-2, os.SEEK_CUR)
    if csvfile.tell() > 0:
        csvfile.truncate()
```

Inicialización:

Listing 17: Inicialización

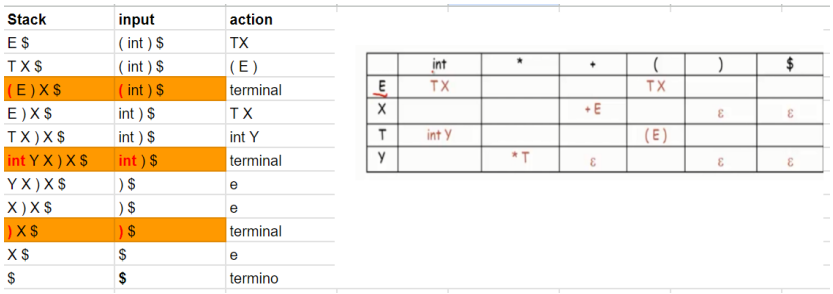
```
directory = os.path.dirname(__file__)
sketchfile = 'grammar_js_machines.txt'
pathfile = os.path.join(directory, '..', 'grammar', sketchfile)

grammar = read_grammar_word_file(pathfile)
tokens.append('$')
.....
# Crear la tabla LL(1) en formato CSV
csv_filename = 'table_ll1_parent2.csv'
#csv_png = 'table_ll1_parent3.png'
carpeta_salida = 'table_ll1'

if not os.path.exists(carpeta_salida):
    os.makedirs(carpeta_salida)

archivo_salida = os.path.join(carpeta_salida, csv_filename)
#archivo_salida2 = os.path.join(carpeta_salida, csv_png)
```

Tabla sintáctica generada:

```
table_ll1 > table_ll1_netcode.csv > data
,FUNCION,ID,CORCHETEABI,CORCHETECERR,LLAVEABI,LLAVECERR,PROGRAMA,PRINCIPAL,TIPOENTERO,TIPOVACIO,VARIABLE,COMA,SI,SINO,ENTONCES,MIENTRAS,PARA,PUNTOYCOMA,IGUAL,DETENER,RETORNAR,I
NETCODE,FUNC MASFUNCIONES MAIN,,,,,,MAIN,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
MASFUNCIONES,FUNC MASFUNCIONES,,,,,,e,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
FUNC,FUNCION ID CORCHETEABI PARAMETROS CORCHETECERR OPDATO LLAVEABI INS LLAVECERR,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
MAIN,,,,,,,PROGRAMA PRINCIPAL TIPOENTERO CORCHETEABI CORCHETECERR LLAVEABI INS LLAVECERR,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
OPDATO,,,,,,,,,TIPODATO,TIPOVACIO,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,TIPODATO,TIPODATO,TIPODATO,,,
PARAMETROS,,,,e,,,,,,,VARIABLE ID TIPODATO MASPARAMETROS,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
MASPARAMETROS,,,,e,,,,,,,,COMA PARAMETROS,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
CONDICIONAL,,,,,,,,,,,,,SI CORCHETEABI EXPRESION CORCHETECERR LLAVEABI INS LLAVECERR POSIBILIDAD,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
POSIBILIDAD,,e,,,,e,,,,,e,,e,SINO CORCHETEABI EXPRESION CORCHETECERR LLAVEABI INS LLAVECERR POSIBILIDAD,ENTONCES LLAVEABI INS LLAVECERR,e,e,,,e,e,e,,,,,,,,,,,,,,,,,,,,,,,,
BUCLEWHILE,,,,,,,,,,,,,,,,MIENTRAS CORCHETEABI EXPRESION CORCHETECERR LLAVEABI INS LLAVECERR,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
BUCLEFOR,,,,,,,,,,,,,,,,,PARA CORCHETEABI ASIG PUNTOYCOMA EXPRESION PUNTOYCOMA ID DERIVA2 CORCHETECERR LLAVEABI INS LLAVECERR,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
INS,,INSTRUCCION MASINSTRUCCION,,,,e,,,,,INSTRUCCION MASINSTRUCCION,,INSTRUCCION MASINSTRUCCION,,,INSTRUCCION MASINSTRUCCION,INSTRUCCION MASINSTRUCCION,,,INSTRUCCION MASINSTRUC
DERIVA2,,,,,,,,,,,,,,,,,,,IGUAL EXPRESION,,,,,,,,,,,,,,,,,,,,,,,,,UNARIOS,UNARIOS,
INSTRUCCION,,ASIG,,,,,,,,,ASIG,,CONDICIONAL,,,BUCLEWHILE,BUCLEFOR,,,DETENER,RETORNAR EXPRESION,IMP,,,,,,,,,,,,,,,,,,,,,,,,
MASINSTRUCCION,,INSTRUCCION MASINSTRUCCION,,,,e,,,,,INSTRUCCION MASINSTRUCCION,,INSTRUCCION MASINSTRUCCION,,,INSTRUCCION MASINSTRUCCION,INSTRUCCION MASINSTRUCCION,,,INSTRUCCION
IMP,,,,,,,,,,,,,,,,,,,,,,IMPRIMIR CORCHETEABI CMDS CORCHETECERR,,,,,,,,,,,,,,,,,,,,,,,,
CMDS,,EXPRESION MASCOMANDOS,EXPRESION MASCOMANDOS,e,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,EXPRESION MASCOMANDOS,EXPRESION MASCOMANDOS,EXPRESION MASCOMANDOS,EXPRESION MASCOMANDOS,,,,
MASCOMANDOS,,,,e,,,,,,,,,,,,,,,,,,,CONCATENAR EXPRESION MASCOMANDOS,,,,,,,,,,,,,,,,,,,,,,,
ASIG,,ID OG,,,,,,,,,VARIABLE ID TIPODATO OPC,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
OG,,,CORCHETEABI PAR CORCHETECERR,,,,,,,,,,,,,,,,IGUAL EXPRESION,,,,,,,,,,,,,,,,,,,,,,,,,,,
OPC,,e,,,,e,,,,,e,,e,,,e,e,e,IGUAL EXPRESION,e,e,e,,,,,,,,,,,,,,,,,,,,,,,,
EXPRESION,,ID OPCION MASEXPRESION,CORCHETEABI EXPRESION CORCHETECERR MASEXPRESION,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,DATO MASEXPRESION,DATO MASEXPRESION,DATO MASEXPRESION,DATO MA
MASEXPRESION,,e,,e,,e,,,,,e,e,e,,,e,e,e,,e,e,e,e,OPERACION EXPRESION,OPERACION EXPRESION,OPERACION EXPRESION,OPERACION EXPRESION,OPERACION EXPRESION,OPERACION EXPRESION,OPERACI
OPCION,,e,CORCHETEABI PAR CORCHETECERR,e,,e,,,,,e,e,e,,,e,e,e,,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,,,,,,,,,,
PAR,,TF RESTO_PARAMETROS,,e,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,TF RESTO_PARAMETROS,TF RESTO_PARAMETROS,TF RESTO_PARAMETROS,TF RESTO_PARAMETROS,,,,,,
RESTO_PARAMETROS,,,,e,,,,,,,,COMA TF RESTO_PARAMETROS,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
TF,,ID OPCION MASEXPRESION,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,DATO MASEXPRESION,DATO MASEXPRESION,DATO MASEXPRESION,DATO MASEXPRESION,,,,,,
OPERACION,,,,,,,,,,,,,,,,,,,,,,,,SUMA,RESTA,MULTIPLICACION,DIVISION,RESIDUO,IGUALBOOLEANO,MENORQUE,MAYORQUE,MENORIGUALQUE,MAYORIGUALQUE,DIFERENTEDE,Y,O,,,,,,,,,,
DATO,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,NCADENA,NDECIMAL,NENTERO,NBOOLEANO,,,,,,
TIPODATO,,,,,,,,,TIPOENTERO,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,TIPOCADENA,TIPODECIMAL,TIPOBOOLEANO,,,
UNARIOS,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,AUMENTAR,DISMINUIR,
```

4.3. Implementación del analizador sintáctico

En esta sección se hace la implementacion del algoritmo, explicado por el profesor de la siguiente manera:



Stack	input	action
E $	(int) $	TX
T X $	(int) $	(E)
(E) X $	(int) $	terminal
E) X $	int) $	T X
T X) X $	int) $	int Y
int Y X) X $	int) $	terminal
Y X) X $	) $	e
X) X $	) $	e
) X $	) $	terminal
X $	$	e
$	$	termino

	int	*	+	(	)	$
E	TX			TX		
X			+E		ε	ε
T	int Y			(E)		
Y		*T	ε		ε	ε

En el caso del desarrollo de mi gramática NetCode, desarrollé la lógica del analizador ll1 de la siguiente manera:

Importaciones:

Listing 18: Inicialización

```
import pandas as pd
import matplotlib.pyplot as plt
from graphviz import Digraph
import math
from functions_lexic import Token
```

Inicialización:

- Se inicializa una pila con el símbolo inicial de la gramática (inicial) y un símbolo de fin de entrada $.
- También se mantiene una copia de la lista de tokens para facilitar el análisis. Con el fin de no perjudicar la lista de tokens original

Listing 19: Inicialización

```
node_PROGRAMA = Node(inicial, inicial, None, None, False, count)
node_dolar = Node("$", "$", None, None, True, count + 1)
stack = [node_PROGRAMA, node_dolar]
nodoPadre = node_PROGRAMA
listtokens_copy = list(listtokens) # Copia de tokens para no modificar la lista original
count += 2
```

Bucle principal:

- El bucle continúa hasta que la pila o la lista de tokens se vacíen, lo cual podría ser un error.

Listing 20: Bucle principal

```
while True:
    print(f"Estado de la pila: {[s.simbolo_lexer for s in stack]}")
    print(f"Tokens restantes: {[token.type for token in listtokens]}")
    if len(stack) == 0 or len(listtokens) == 0:
        error = True
```

```
        print("Error: Pila vacia o lista de tokens vacia antes de tiempo.")
        break
```

Coincidencia de terminales:

- Si el símbolo en la parte superior de la pila es un terminal y coincide con el token actual, significa que el token ha sido reconocido correctamente.
- Se elimina de la pila y de la lista de tokens.

Listing 21: Coincidencia de terminales

```
if stack[0].simbolo_lexer == "$" and listtokens[0].type == "$":
    print(" ")
    print("Analisis exitoso: Se alcanz el smbolo de fin de entrada.")
    break
elif stack[0].es_terminal and stack[0].simbolo_lexer == listtokens[0].type:
    print(f"Coincidencia encontrada: {stack[0].simbolo_lexer}")
    stack.pop(0)
    listtokens.pop(0)
    print(" ")
```

Error por no coincidencia:

- Si el símbolo en la parte superior de la pila es un terminal, pero no coincide con el token actual, se marca un error.

Listing 22: Error por no coincidencia

```
elif stack[0].es_terminal and stack[0].simbolo_lexer != listtokens[0].type:
    error = True
    print(f"Error: Se esperaba '{stack[0].simbolo_lexer}', pero se encontro
        '{listtokens[0].type}'.")
    break
```

Aplicación de producciones:

- Si el símbolo en la parte superior de la pila es un no terminal, se consulta la tabla LL(1) para determinar la producción a aplicar.
- Luego, se reemplaza el símbolo no terminal con los símbolos de la producción.

Listing 23: Aplicacion de producciones

```
else:
    try:
        production = table_ll1.loc[stack[0].simbolo_lexer][listtokens[0].type]
        print(f"Produccion encontrada para {stack[0].simbolo_lexer}: {production}")
        print(" ")
    except KeyError:
        error = True
        print(f"Error: No hay produccion valida para el no terminal
            '{stack[0].simbolo_lexer}' con el token '{listtokens[0].type}'.")
        break
    if production == "e":
```

```
        print(f"Produccion vacia aplicada para {stack[0].simbolo_lexer}")
        padre_stack = stack.pop(0)
        padre = buscar(nodoPadre, padre_stack.id)
        nodo_e = Node("e", "e", None, None, True, count)
        nodo_e.padre = padre
        padre.add_child(nodo_e)
        count += 1
        print(" ")
```

Error de producción:

- Si no se encuentra una producción válida en la tabla LL(1), se marca un error y el análisis se detiene.

Listing 24: Error de producción

```
except KeyError:
    error = True
    print(f"Error: No hay produccion valida para el no terminal '{stack[0].simbolo_lexer}'
        con el token '{listtokens[0].type}'.")
    break
```

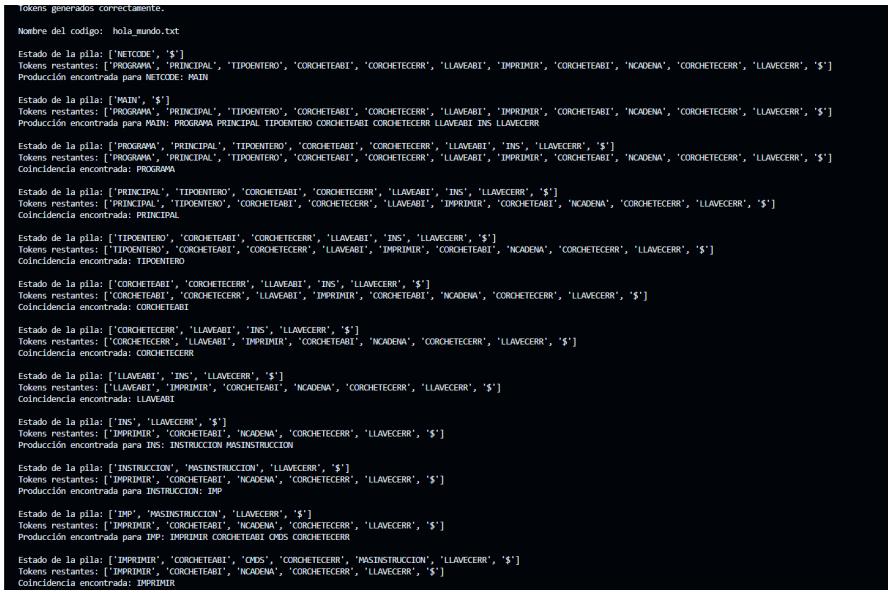
Finalización del análisis:

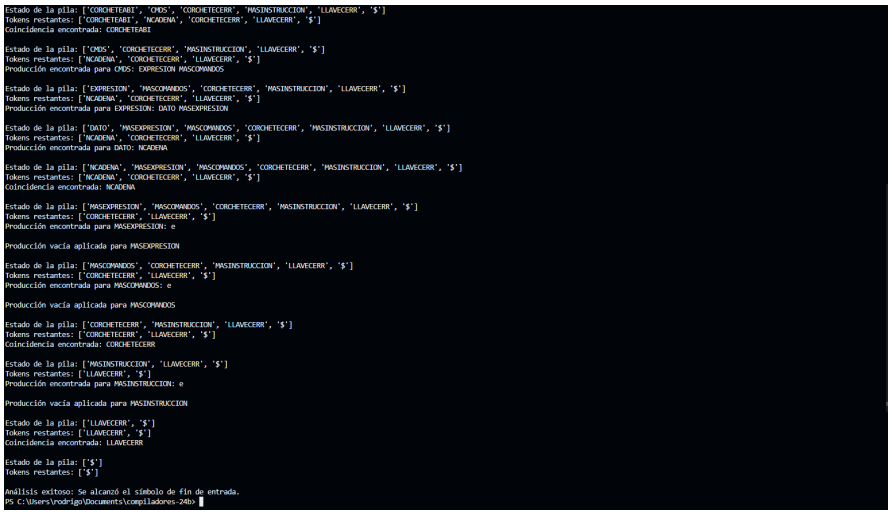
- El análisis termina exitosamente cuando tanto la pila como la lista de tokens contienen únicamente el símbolo de fin de entrada $, indicando que toda la entrada fue reconocida correctamente.

Listing 25: Error de produccion

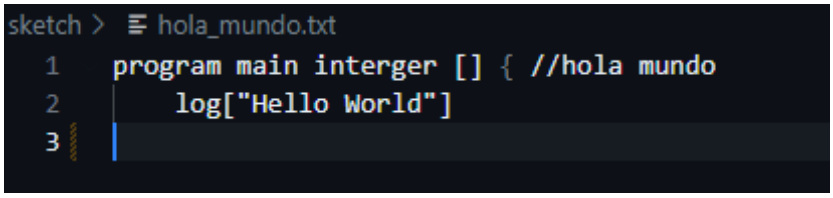
```
if stack[0].simbolo_lexer == "$" and listtokens[0].type == "$":
    print(" ")
    print("Analisis exitoso: Se alcanzo el simbolo de fin de entrada.")
    break
```

Ejemplo del análisis ll1, archivo boceto (hola mundo):

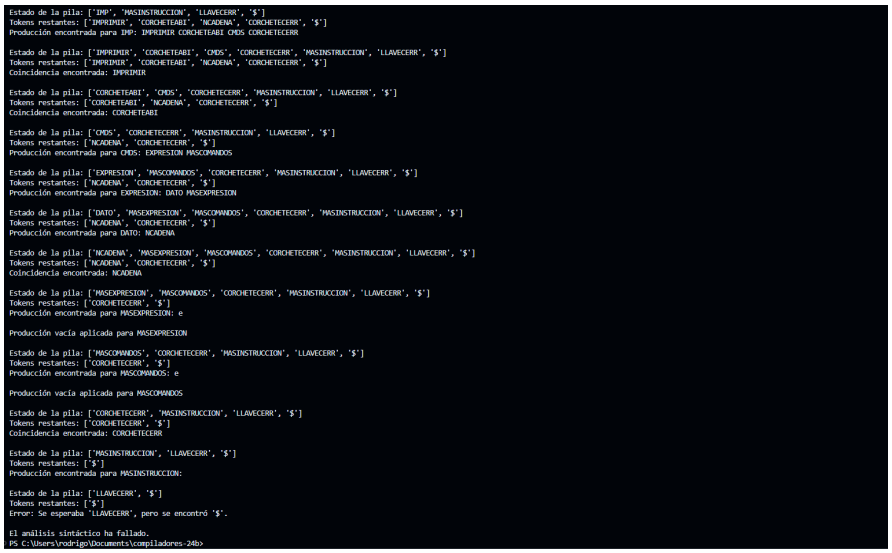




Quitamos un corchete para ver el resultado en fallo:



Ejemplo del análisis ll1, archivo boceto (hola mundo) con error sintáctico:



4.4. Implementación del árbol sintáctico

El árbol sintáctico se construye a medida que se realiza el análisis de la entrada usando la tabla LL(1). A continuación, explico cada parte del proceso y cómo el código contribuye a la construcción del árbol:

Definición de clases para el arbol sintáctico:

Clase Simbolo:

- Se utiliza para representar un símbolo que tiene información sobre el tipo de dato, el nombre, la función (contexto de definición) y el valor, si lo hay. Esto es útil para el análisis semántico.

Listing 26: Clase simbolo

```
class Simbolo:
    def __init__(self, tipo_data, nom_sym, function, valor=None):
        self.tipo_data = tipo_data # Tipo de dato del simbolo (e.g., int, float, string)
        self.nom_sym = nom_sym    # Nombre del simbolo
        self.function = function # Contexto o funcin donde se define el smbolo
        self.valor = valor       # Valor asignado al simbolo (si lo tiene)
```

Clase Node:

- Representa un nodo del árbol sintáctico. Cada nodo contiene información sobre el símbolo léxico, su lexema, línea y columna del código fuente, y si es terminal o no.
- Cada nodo tiene referencias a sus hijos (children) y su padre (padre), formando la jerarquía del árbol.

Listing 27: Clase Node

```
class Node:
    def __init__(self, simbolo_lexer, lexema, line, column, es_terminal, id, tipo_data=None,
        valor=None):
        self.id = id                    # Identificador unico del nodo
```

```
        self.lexema = lexema            # Lexema asociado al nodo
        self.simbolo_lexer = simbolo_lexer # Tipo de simbolo del analizador lexico (e.g.,
            identificador, palabra clave)
        self.es_terminal = es_terminal  # Booleano que indica si es un nodo terminal
        self.line = line                # Linea en la que se encuentra el simbolo en el
            codigo fuente
        self.column = column            # Columna donde comienza el simbolo en el cdigo fuente
        self.children = []              # Lista de nodos hijos
        self.padre = None               # Nodo padre (para referencia en el arbol)
        self.simbolos = []              # Lista de smbolos definidos en este nodo
        self.tipo_data = tipo_data      # Tipo de dato asociado (para analisis semantico)
        self.valor = valor              # Valor asociado al nodo (si lo tiene)

    def add_child(self, child):
        self.children.append(child)
        child.padre = self # Establecer al nodo actual como padre del hijo
```

Construcción del árbol sintáctico durante el análisis:

Durante el análisis LL(1), el árbol sintáctico se construye a medida que se seleccionan producciones y se manejan los símbolos léxicos:

Búsqueda de nodos:

- La función buscar(node, id) permite buscar un nodo específico en el árbol según su identificador (id). Esta función es útil para encontrar el nodo padre al que se deben agregar los hijos durante la construcción del árbol.

Listing 28: Busqueda de nodos

```
def buscar(node, id):
    if node.id == id:
        return node
    for c in node.children:
        found_node = buscar(c, id)
        if found_node is not None:
            return found_node
    return None
```

Añadir producción a la pila y crear nodos:

- Cuando se selecciona una producción de la tabla LL(1), se crean nodos para cada uno de los símbolos en la producción y se añaden a la pila.
- Además, se establece la relación jerárquica con el nodo padre. Cada símbolo se convierte en un nodo hijo del nodo que lo produjo.

Listing 29: Añadir producción a la pila y crear nodos

```
symbols = production.split(" ")
padre_stack = stack.pop(0)
padre = buscar(nodoPadre, padre_stack.id)
for Symlexer in reversed(symbols):
    if Symlexer: # Evitar agregar simbolos vacios
        is_terminal = Symlexer in table_ll1.columns
        node = Node(Symlexer, Symlexer, None, None, is_terminal, count)
        count += 1
```

```
        stack.insert(0, node)
        padre.children.insert(0, node)
        node.padre = padre
```

Manejo de producción vacía (e):

- Cuando una producción vacía (e) es seleccionada, se crea un nodo especial que representa esta producción vacía.
- Se establece la relación de este nodo con el nodo padre en el árbol, permitiendo que el árbol refleje la derivación vacía.

Listing 30: Manejo de producción vacía (e)

```
if production == "e":
    print(f"Produccin  vaca  aplicada para {stack[0].simbolo_lexer}")
    padre_stack = stack.pop(0)
    padre = buscar(nodoPadre, padre_stack.id)
    nodo_e = Node("e", "e", None, None, True, count)
    nodo_e.padre = padre
    padre.add_child(nodo_e)
    count += 1
    print(" ")
```

Visualización del árbol sintáctico:

- Se utiliza graphviz para crear una representación gráfica del árbol sintáctico. La función toma el nodo raíz del árbol como parámetro y genera un grafo (graph) que se puede visualizar para entender la estructura del análisis.
- Dentro de la función, se utiliza generar_nodos(node) para recorrer el árbol en una especie de recorrido en preorden, generando nodos gráficos y las conexiones entre ellos.

Funcionamiento de arbolSintactico:

- La función recorre el árbol sintáctico desde la raíz y genera un nodo gráfico para cada nodo del árbol sintáctico.
- El label del nodo gráfico incluye información sobre el símbolo (simbolo_lexer), su línea, columna, y valor, si están disponibles.
- Cada nodo gráfico también se conecta con su padre utilizando graph.edge(), lo que ayuda a visualizar la relación jerárquica entre los nodos.

Listing 31: Funcionamiento de arbolSintactico

```
def arbolSintactico(raiz):
    graph = Digraph()
    def generar_nodos(node):
        label = f"{node.simbolo_lexer}"
        if node.line is not None:
            label += f"\nline: {node.line}"
        if node.column is not None:
            label += f"\ncol: {node.column}"
        if node.valor is not None:
            label += f"\nvalor: {node.valor}"
```

```
        graph.node(str(node.id), label, style="filled", fillcolor='white')
        if node.padre:
            graph.edge(str(node.padre.id), str(node.id))
        for child in node.children:
            generar_nodos(child)
    generar_nodos(raiz)
    return graph
```

Main sintatic:

Listing 32: sintactic.py

```
from lexic import listtokens
import os
from functions_sintactic import generate_table_ll1
from functions_sintactic import arbolSintactico
from functions_sintactic import parser_sintactico_ll1

directory2 = os.path.dirname(__file__)
sketchfile = 'table_ll1_netcode.csv'
pathfile2 = os.path.join(directory2, '..', 'table_ll1', sketchfile)

table_ll1 = generate_table_ll1(pathfile2)

simboloinicial = "NETCODE"

success, parse_tree_root = parser_sintactico_ll1(listtokens, table_ll1, simboloinicial)

if success:
    output_folder = 'tree'
    if not os.path.exists(output_folder): os.makedirs(output_folder)
    graph = arbolSintactico(parse_tree_root)
    output_pdf_path = os.path.join(output_folder, 'tree_sintactic_hola_mundo')
    graph.render(output_pdf_path, format='png', view=True)
else:
    print(" ")
    print("El anlisis  sintctico  ha fallado.")
```

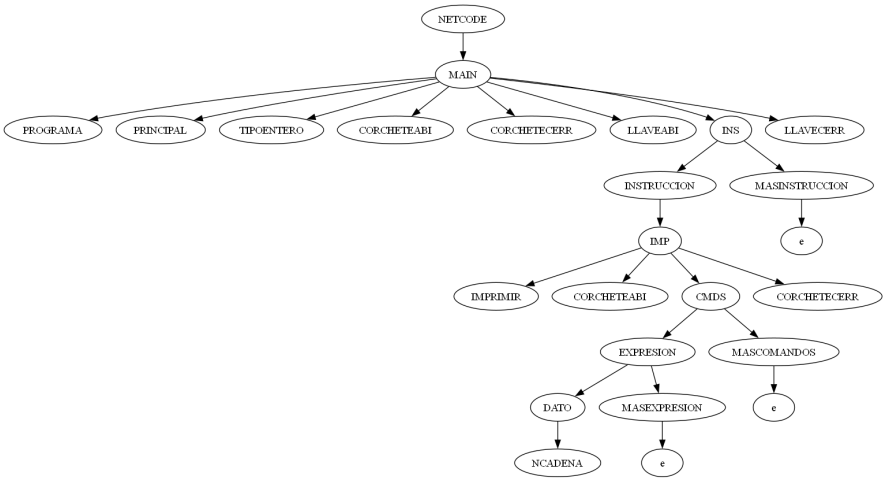
generate_table_ll1:

La función generate_table_ll1 carga un archivo .csv que contiene una tabla LL(1), lo convierte en un DataFrame de pandas usando la primera columna del archivo como índice, y luego reemplaza cualquier valor nulo en la tabla con una cadena vacía. Finalmente, devuelve este DataFrame listo para ser usado en análisis o procesamiento sintáctico:

Listing 33: csv usando pandas

```
def generate_table_ll1(pathfile):
    df = pd.read_csv(pathfile, index_col = 0)
    df = df.fillna('')
    return df
```

Árbol ll1, archivo boceto (hola mundo):



5. Entregables en repositorio GitHub:

- **lexic.py:** `https://github.com/RodrigoStranger/compiladores-24b/blob/main/netcode/lexic.py`
- **functions_lexic.py:** `https://github.com/RodrigoStranger/compiladores-24b/blob/main/netcode/functions_lexic.py`
- **sintactic.py:** `https://github.com/RodrigoStranger/compiladores-24b/blob/main/netcode/sintactic.py`
- **functions_sintactic.py:** `https://github.com/RodrigoStranger/compiladores-24b/blob/main/netcode/functions_sintactic.py`
- **table_sintactic_ll1.py:** `https://github.com/RodrigoStranger/compiladores-24b/blob/main/netcode/table_sintactic_ll1.py`

6. Ruta de carpetas NetCode en repositorio GitHub:

- **Ruta principal:** `https://github.com/RodrigoStranger/compiladores-24b`
- **grammar:** `https://github.com/RodrigoStranger/compiladores-24b/tree/main/grammar`
- **listtokens:** `https://github.com/RodrigoStranger/compiladores-24b/tree/main/listtokens`
- **netcode:** `https://github.com/RodrigoStranger/compiladores-24b/tree/main/netcode`
- **sketch:** `https://github.com/RodrigoStranger/compiladores-24b/tree/main/sketch`
- **table_ll1:** `https://github.com/RodrigoStranger/compiladores-24b/tree/main/table_ll1`
- **tree:** `https://github.com/RodrigoStranger/compiladores-24b/tree/main/tree`