Universidad Nacional de Asunción
Facultad Politécnica

Diseño de Compiladores
Trabajo Práctico 1
Traductor Dirigido por la Sintaxis

Integrantes:

- Juan Carlos Cañiza
- María José Melgarejo

Introducción

En el presente trabajo presentamos dos propuestas de solución con sus correspondientes reglas, acciones, BNF y código fuente. Optamos por realizar ambas soluciones debido a que en el proceso de realizar el BNF encontramos que ambas se veían prometedoras de dar una traducción correcta a nuestra gramática asignada.
De igual manera, presentamos primero la que nos resultó más adecuada debido a que no se requieren hacer modificaciones a su BNF y las reglas se aplican de manera más natural que la segunda alternativa.
Para el desarrollo del código fuente optamos por el lenguaje Python en su versión 3.6.9, debido a que ambos integrantes estamos más familiarizados con este que con otro lenguaje.

El código fuente se encuentra en el siguiente repositorio: Repositorio

Gramática Asignada

TDS que reciba cadenas separadas por espacio de 1's y 0's y retorna la suma de decimal de los números que representan.

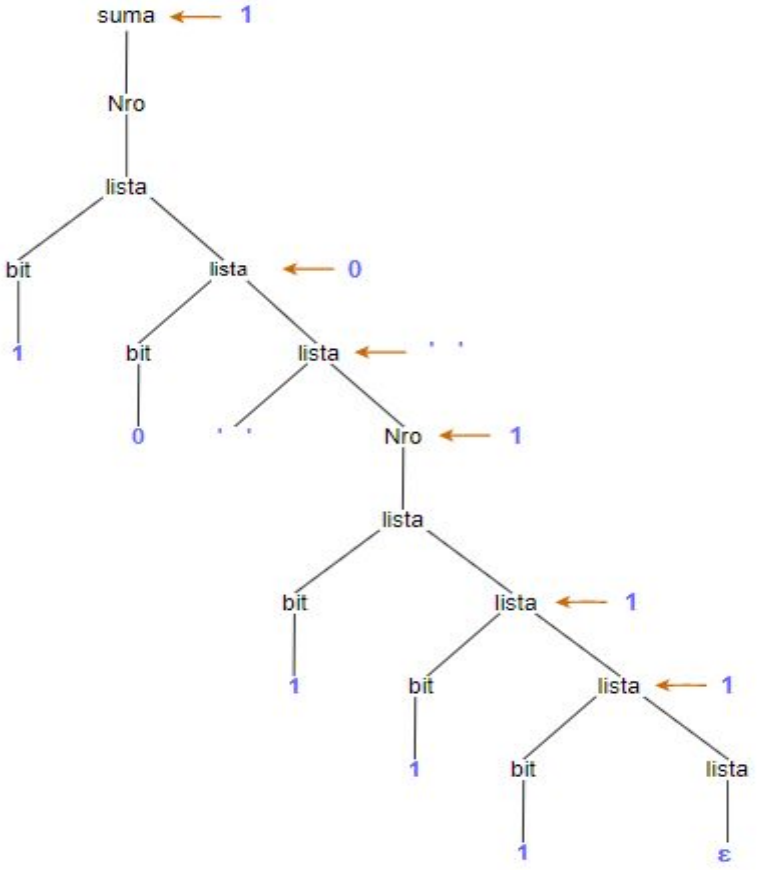
Ejemplo:

PROPUESTA N° 1

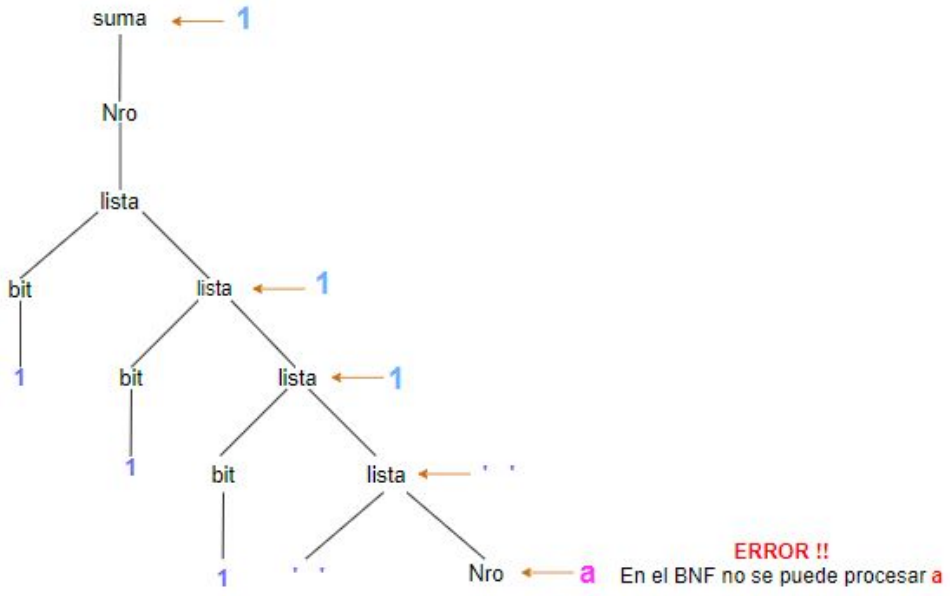
1. BNF que acepte la cadena de entrada:

suma → Nro
Nro → lista
lista → bit lista | ‘ ‘ Nro | ε
bit → 1 | 0

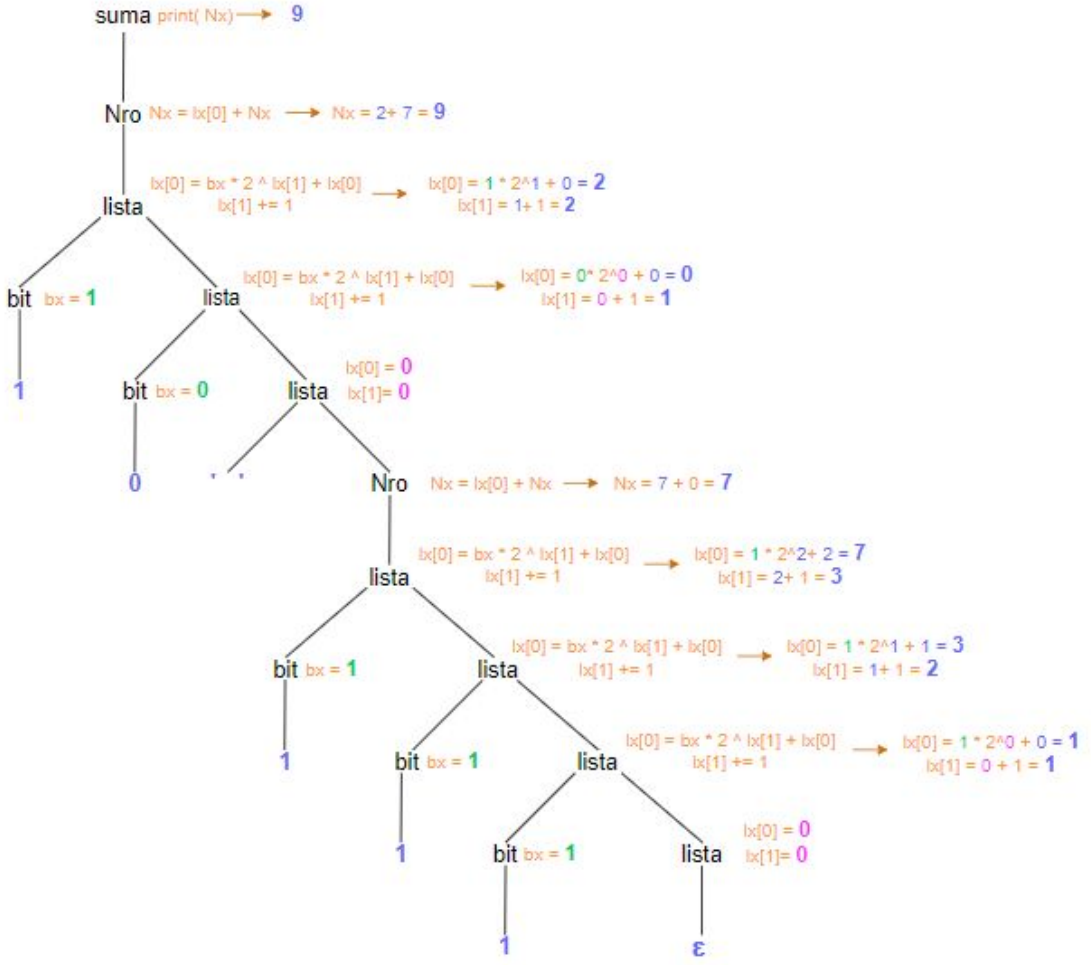
2. Árbol Sintáctico para la cadena de entrada: 10 111



➔ **Árbol Sintáctico para una entrada no válida: 111 a**



3. En el árbol sintáctico definimos las reglas semánticas.



Divide y Vencerás	Reglas Semánticas	Acciones Semánticas
suma → Nro	suma.x = Nro.x[0]	print(suma.x)
Nro → lista	Nro.x[0]=Nro.x[0] + lista.x[0]	
lista → bit lista	lista.x[0] = bx*2^by + lista.x[0] lista.x[1] = lista.x[1] + 1	
lista → ‘ ‘ Nro	lista.x[0]=0 lista.x[1] = 0	
lista → ε	lista.x[0]=0 lista.x[1] = 0	
bit → 1	b.x = 1	
bit → 0	b.x = 0	

Para verificar si la gramática es predictiva, realizamos los siguientes pasos:

1. **Verificamos que no exista Recursividad por Izquierda:** No se observa recursividad por izquierda en el BNF, por lo que no habría problemas de backtracking.

2. **Verificamos que no exista Factor Común:** No se observa factor común en el BNF, por lo que no va haber problemas de ambigüedad sintáctica.

3. **Realizamos el Conjunto Primero:**

   P (suma) = { P(Nro) } = { 1, 0, ‘ ‘, ε }
   P(Nro) = { P(lista) } = { 1, 0, ‘ ‘, ε }
   P(lista) = { P(bit) } U { ‘ ‘ } U { ε }= { 1, 0, ‘ ‘, ε }
   P(bit) = { 1 } U { 0 } = { 1, 0 }

   ➔ El ε podría causar problemas en cuanto a la ambigüedad semántica, pero esto lo resolvemos en el código fuente.

4. **Finalmente, la gramática queda definida de la siguiente manera:**

BNF	Reglas Semánticas	Acciones Semánticas
suma → Nro	suma.x = Nro.x[0]	print(suma.x)
Nro → lista	Nro.x[0]=Nro.x[0] + lista.x[0]	
lista → bit lista	lista.x[0] = bx*2^by + lista.x[0] lista.x[1] = lista.x[1] + 1	
lista → ‘ ‘ Nro	lista.x[0]=0 lista.x[1] = 0	
lista → ε	lista.x[0]=0 lista.x[1] = 0	
bit → 1	b.x = 1	
bit → 0	b.x = 0	

Aclaraciones sobre el Código Fuente

Se utilizan 3 funciones: *Nro, lista y bit.*
Además de 1 funciones adicionales: *sgte_caracter.*

```
entrada = input("Introduzca una cadena: ")
band=0
tam=len(entrada)
```

- La función "**sgte_caracter**" se encarga de obtener los caracteres de la cadena de entrada y realizar las validaciones correspondientes.
  Con cada llamada a la función, obtenemos los caracteres de la cadena de entrada y verificamos que este pertenezca al alfabeto A del lenguaje o que la cadena de entrada no sea una incompleta. Consideramos cadenas incompletas aquellas que empiecen o terminen con un espacio ' '.
  A = { 1, 0, ' ' }, siendo ' ' un espacio.

```
def sgte_caracter():
    global entrada, band
    if(entrada!=""):
        if( (entrada[0]==' ' and len(entrada)==1)
         or (entrada[0]==' ' and len(entrada)==tam)):
            print("Error. Cadena incompleta")
            entrada=""
            band=1
        elif(entrada[0]== '1' or entrada[0]== '0'
             or entrada[0]==' '):
            num = entrada[0]
            entrada = entrada[1:]
            return(num)
        else:
            print("Error. No pertenece al alfabeto de entrada")
            entrada=""
            band=1
```

- La función **bit** retorna un valor entero 1 o 0 en caso de que el bit sea '1' o '0' respectivamente.

```
def bit(b):
    if(b=='1')
        return(1)
    else:
        return(0)
```

- La función **lista** toma un vector “lista_x” como entrada, donde, lista_x[0] corresponde a la conversión de binario a decimal y lista_x[1] corresponde al exponente al que debe ser elevado el 2 para realizar dicha conversión.
  Para realizar la conversión primeramente llama a la función “sgte_caracter” para obtener el valor que está en la entrada, dependiendo de si es 1, 0, ‘ ‘ o vacío realiza distintas operaciones.
  - ➔ Si es 1 o 0: llama a la función “bit”, se llama a sí misma, eleva el 2 al exponente guardado en el vector, lo multiplica por el valor obtenido en bit. El resultado lo suma al valor acumulado en lista_x[0]. Retorna el vector lista_x.
  - ➔ Si no, se asume que es un espacio o el vacío: retorna un vector con valor 0.

```
def lista(lista_x):
    num = sgte_caracter()
    if(num == '1' or num == '0'):
        bx = bit(num)
        lista_x = lista(lista_x)
        lista_x[0] = lista_x[0] + bx*(2**lista_x[1])
        lista_x[1]+=1
        return(lista_x)
    else:
        lista_x=[0,0]
        return(lista_x)
```

- La función **nro()** inicializa un Nro_x en 0 para guardar el valor de la conversión. Mientras que la cadena no esté vacía, inicializa un vector “cadena” en 0, llama a la función lista pasando como parámetro el vector inicializado y guarda el valor retornado por la función en el vector lista_x, toma el primer valor del vector lista_x y lo suma al valor guardado en la variable Nro_x. Finalmente retorna el valor guardado en Nro_x.

```
def nro():
    Nro_x=0
    while(entrada!=""):
        cadena=[0,0]
        lista_x = lista(cadena)
        Nro_x= Nro_x + lista_x[0]
```

```
    return(Nro_x)
```

- La variable **suma** llama a la función nro() y guarda el valor retornado, si no hubo errores en la cadena de entrada (indicado por la variable band) imprime el valor de la conversión.

```
suma=nro()
if(band!=1):
    print("La suma decimal es: ",suma)
```

Entrada y Salida en el código fuente

- **Entrada Válida: 10 111**

  Entrada:

```
>>>
== RESTART: C:\Users\MaríaJosé\Desktop\Compiladores\TDS-TrabajoPractico.py ==
Introduzca una cadena: 10 111
```

  Salida:

```
Introduzca una cadena: 10 111
La suma decimal es:  9
```

- **Entrada no válida: 111 a**

  Entrada:

```
>>>
== RESTART: C:\Users\MaríaJosé\Desktop\Compiladores\TDS-TrabajoPractico.py ==
Introduzca una cadena: 111 a
```

  Salida:

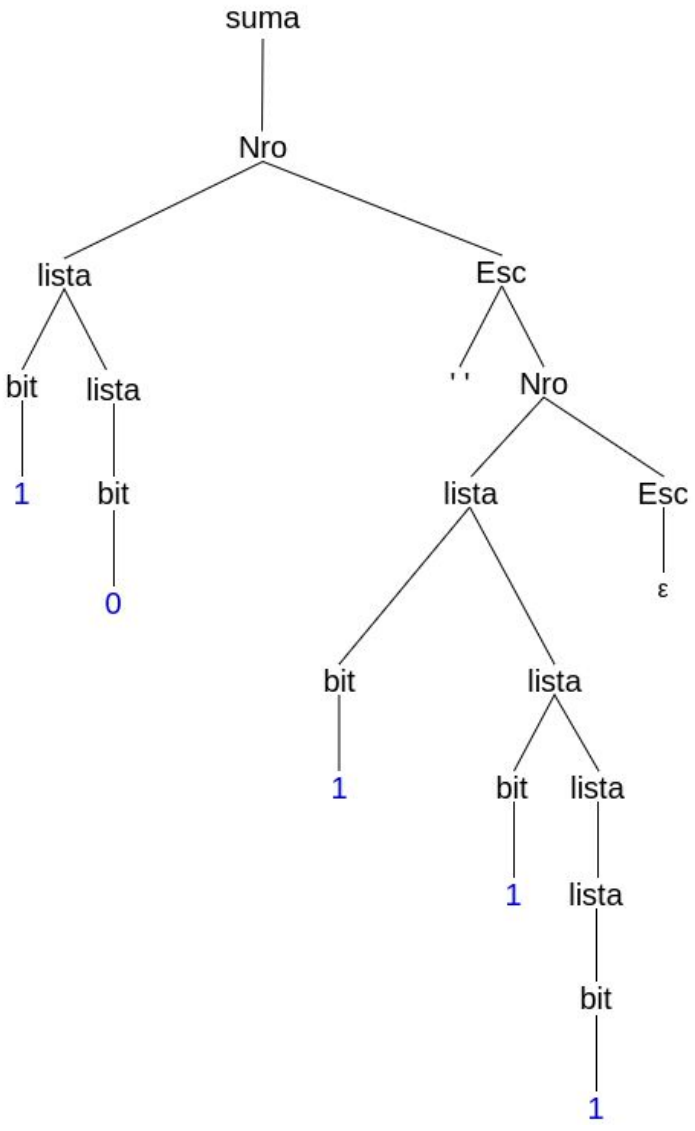
```
Introduzca una cadena: 111 a
Error. No pertenece al alfabeto de entrada
```

PROPUESTA N° 2

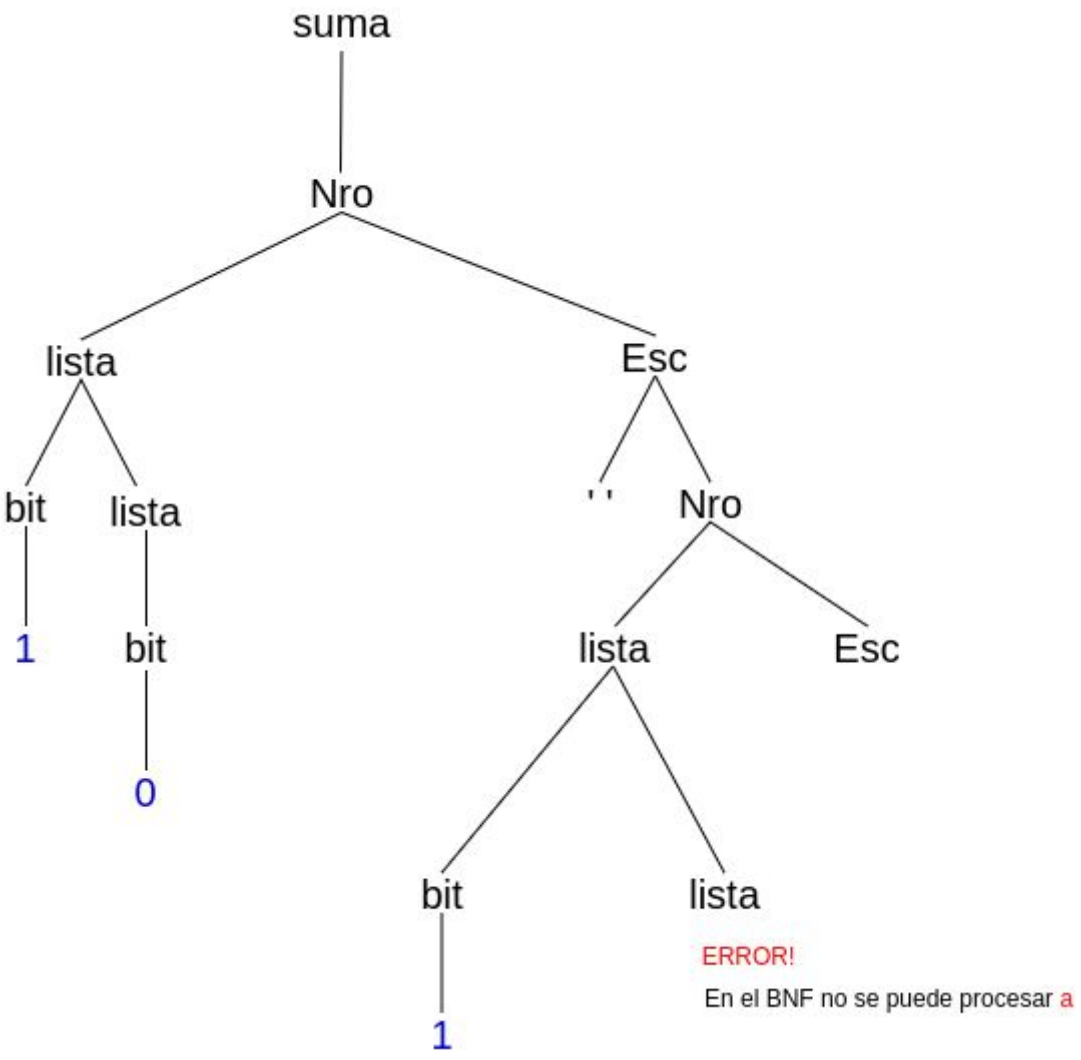
1. BNF que acepte la cadena de entrada:

suma → Nro
Nro → lista Esc
Esc → ‘ ‘ Nro | ε
lista → bit lista | bit
bit → 1 | 0

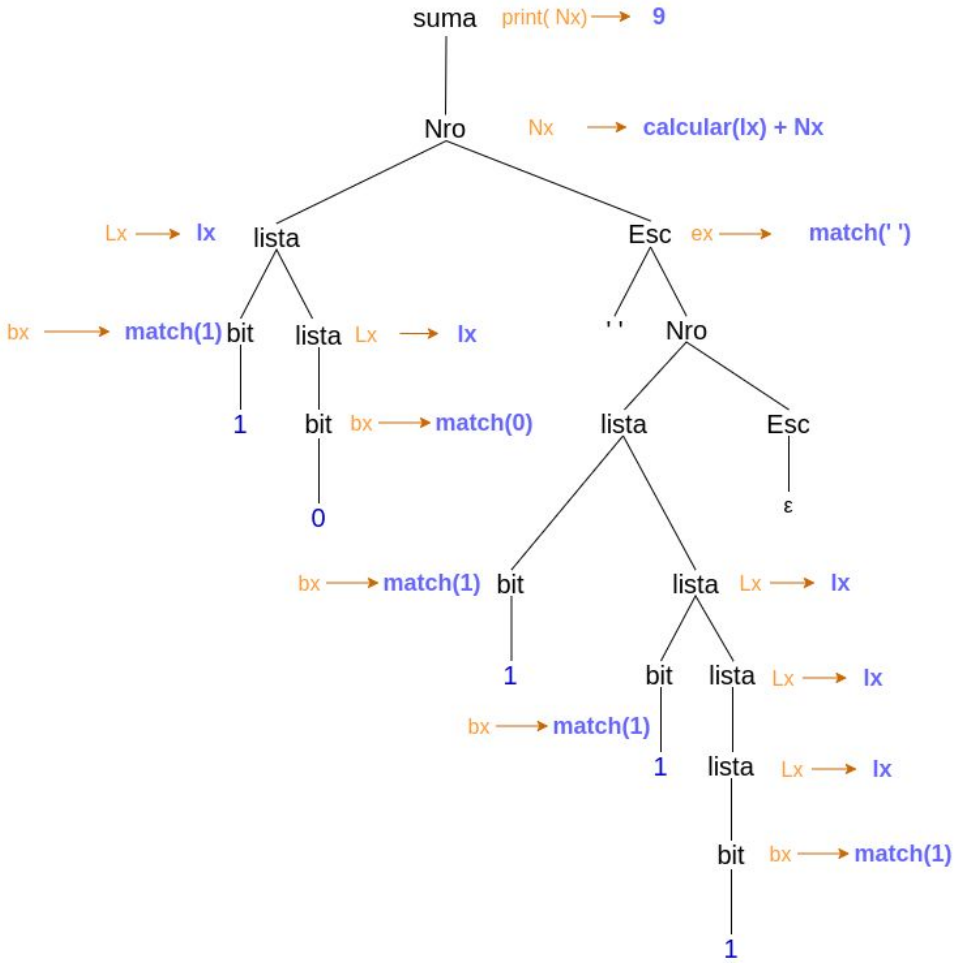
2. Árbol Sintáctico para la cadena de entrada: 10 111



➔ **Árbol Sintáctico para una entrada no válida: 10 1a**



3. En el árbol sintáctico definimos las reglas semánticas.



Divide y Vencerás	Reglas Semánticas	Acciones Semánticas
suma → Nro		print(suma.x)
Nro → lista Esc	Nx = calcular(Lx) + Nx	
Esc → ‘ ‘ Nro	Esc_x = match(‘ ‘)	
Esc → ε	----------	
lista → bit lista	lista_x = lista_x	
lista → bit	lista_x = lista_x	
bit → 1	bx = match(1)	
bit → 0	bx = match (0)	

Para verificar si la gramática es predictiva, realizamos los siguientes pasos:

1. **Verificamos que no exista Recursividad por Izquierda:** No se observa recursividad por izquierda en el BNF, por lo que no habría problemas de backtracking.
2. **Verificamos que no exista Factor Común:** Se observa que existe factor común en la producción “lista” en el BNF, por lo que hay ambigüedad sintáctica.

- Se procede a eliminar la ambigüedad sintáctica mediante el algoritmo estudiado en clase:

  lista → bit lista | bit

  *Se elimina el factor común y se obtiene el siguiente sistema de producción que reemplaza al anterior:

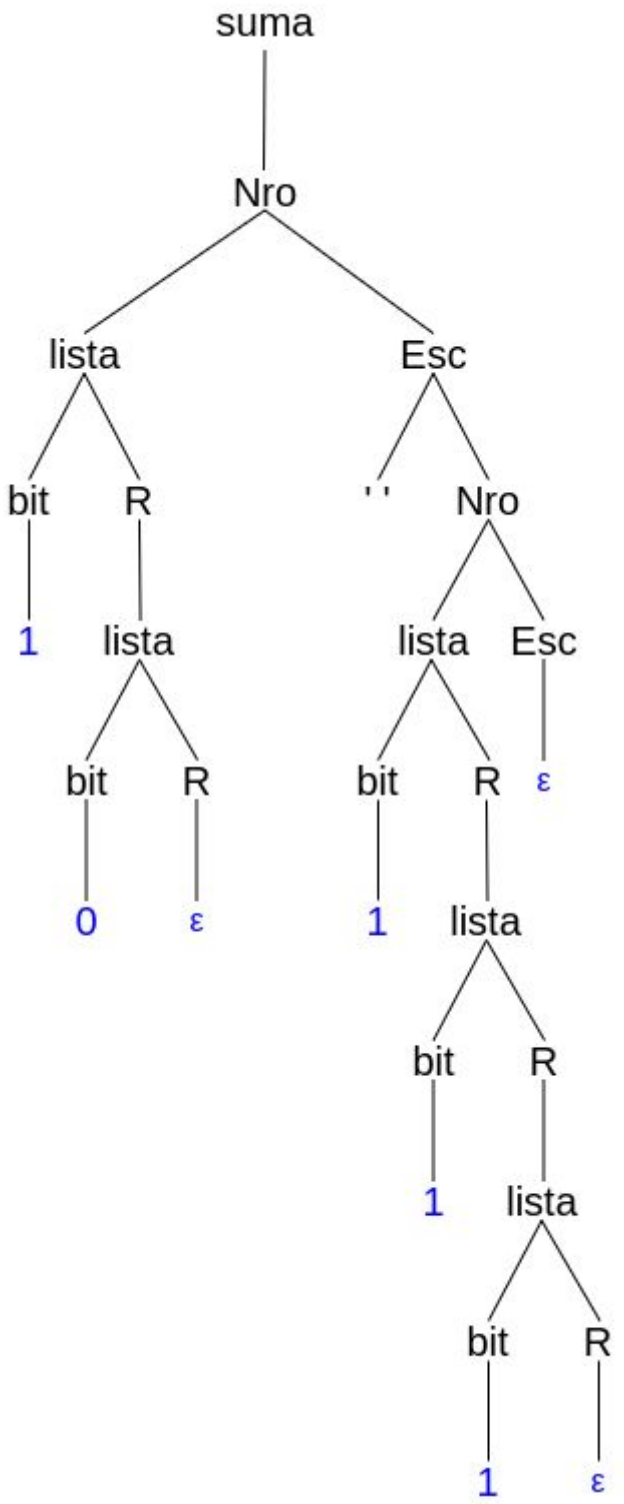
  lista → bit R
  R → lista | ε

3. **Realizamos el Conjunto Primero:**

   P (suma) = { P(Nro) } = { 1, 0, ‘ ‘, ε }
   P(Nro) = { P(lista) } U {P(Esc)} = { 1, 0 } U { ‘ ‘, ε } = { 1, 0, ‘ ‘, ε }
   P (Esc) = { ‘ ‘ } U { ε } = { ‘ ‘, ε }
   P(lista) = { P(bit R) } = { 1, 0 }
   P(R) = { P (lista) } U { ε } = {1, 0, ε }
   P(bit) = { 1 } U { 0 } = { 1, 0 }

   ➔ El ε podría causar problemas en cuanto a la ambigüedad semántica, pero esto lo resolvemos en el código fuente.

➔ **Luego de las modificaciones el Árbol Sintáctico para una entrada válida: 10 111 quedaría:**



4. **Finalmente, la gramática queda definida de la siguiente manera:**

BNF	Reglas Semánticas	Acciones Semánticas
suma → Nro		print(suma.x)
Nro → lista Esc	Nx = calcular(Lx) + Nx	
Esc → ‘ ‘ Nro	Esc_x = match(‘ ‘)	
Esc → ε	----------	
lista → bit R	lista_x = lista_x	
R → lista	lista_x = lista_x	
R → ε	----------	
bit → 1	bx = match(1)	
bit → 0	bx = match (0)	

Aclaraciones sobre el Código Fuente

Se utilizan 6 funciones: *suma, nro, escape, lista, R y bit.*
Además de 2 funciones adicionales: *match y calcular.*

La función suma es llamada luego de leer la entrada y hace una llamada a ***nro*** con la entrada y luego de calcular el último valor que fue leído (última secuencia de bits) imprime el resultado de la suma.

```
def suma(entrada):
      nro(entrada)
      calcular(lista_aux)
      print("El resultado es:", resultado)
```

La función ***nro*** llama a ***lista*** con el índice actual (el índice inicial es cero), luego de verificar que existan aún valores por leer, calcula el valor decimal de la secuencia de bits que se proceso en ***lista*** y finalmente llama a ***escape*** para continuar el procesamiento.

```
def nro(entrada):
      global lista_aux, mayor

      lista(entrada[indice])
      if (indice < len(entrada)):
            calcular(lista_aux)
            lista_aux = []
            escape(entrada[indice])
```

La función escape recibe el siguiente carácter y si es un espacio llama ***match*** para avanzar el índice y luego llama a ***nro*** para volver a repetir la secuencia para el próximo valor.

```
def escape(e):
      if (e == ' '):
            match(e)
            nro(entrada)
```

La función lista llama a ***bit*** con el valor que recibe como parámetro y luego de verificar que existan aún valores por leer, llama a ***R*** para continuar el procesamiento.

```
def lista(e):
      bit(e)
      if (indice < len(entrada)):
            R(entrada[indice])
```

La función R verifica si el valor que recibe como parámetro es un '1' o un '0', si esto no sucede lanza una excepción debido a que se ingresó un carácter no permitido, sino llama a ***lista*** para continuar el procesamiento.

```
def R(e):
      if (e == '1' or e == '0'):
            lista(e)
      elif(e != ' '):
            print ("Valor de char:", e)
            raise NameError(msg_error)
```

La función bit verifica si el valor que recibe como parámetro es un '1' o un '0', si esto no sucede lanza una excepción debido a que se ingresó un carácter no permitido, sino llama a ***match*** para continuar el procesamiento.

```
def bit(e):
      if (e == '1' or e == '0'):
            match(int(e))
      else:
            print ("Valor de char:", e)
            raise NameError(msg_error)
```

La función ***match*** es una función auxiliar que es la que procesa los terminales. Si su entrada es **'1'** o **'0'** agrega el valor correspondiente en la **lista auxiliar** para calcular el valor decimal de la secuencia de bits leída. Si es un espacio **(' ')** simplemente avanza y en caso que no ocurra ninguna de las anteriores lanza una excepción que se ingresó un carácter no permitido.

```
def match(valor):
	global indice, lista_aux, inicio

	if (valor == 1):
		lista_aux.append(1)
	elif (valor == 0):
		lista_aux.append(0)
	elif(valor == ' '):
		inicio = indice + 1
	else:
		raise NameError(msg_error)
	indice += 1
```

La función ***calcular*** es una función auxiliar que como su nombre lo define calcula el valor en decimal de la secuencia de bits (almacenada en el vector auxiliar). Esto lo realiza iterando la lista auxiliar y aumentando el **resultado** en su valor anterior más el valor del número (1 o 0) multiplicado por 2 elevado a la potencia de la posición correspondiente del bit.

```
def calcular(lista):
	global resultado

	mayor = len(lista)-1
	i=0
	for numero in lista:
		resultado += numero * 2**(mayor-i)
		i+=1
```

Entrada y Salida en el código fuente

- **Entrada Válida: 10 111**

  Entrada:

```
juancanhiza@juanca-pc:~/Escritorio/Facultad/Compiladores/TP$ python3 --version
Python 3.6.9
juancanhiza@juanca-pc:~/Escritorio/Facultad/Compiladores/TP$ python3 TDS2.py
Ingrese la cadena a traducir: 10 111
```

  Salida:

```
El resultado es: 9
```

- **Entrada no válida: 111 a**

  Entrada:

```
juancanhiza@juanca-pc:~/Escritorio/Facultad/Compiladores/TP$ python3 TDS2.py
Ingrese la cadena a traducir: 111 a
```

  Salida:

```
Traceback (most recent call last):
  File "TDS2.py", line 78, in <module>
    suma(entrada)
  File "TDS2.py", line 20, in suma
    nro(entrada)
  File "TDS2.py", line 31, in nro
    escape(entrada[indice])
  File "TDS2.py", line 36, in escape
    nro(entrada)
  File "TDS2.py", line 27, in nro
    lista(entrada[indice])
  File "TDS2.py", line 39, in lista
    bit(e)
  File "TDS2.py", line 53, in bit
    raise NameError(msg_error)
NameError: Se ingreso caracter no valido
```