



Carátula para entrega de prácticas

Facultad de Ingeniería

Laboratorios de docencia

Laboratorio de Computación Salas A y B

Profesor(a): René Adrián Dávila Pérez

Asignatura: Compiladores

Grupo: 04

No de Práctica(s): Analizador Sintáctico

Integrante(s): 318307921

318035327

318051363

318045351

*No. de lista o
brigada:* Equipo 11

Semestre: 2024-1

Fecha de entrega: 15 de Noviembre de 2023

Observaciones:

CALIFICACIÓN: _____

Índice

Índice	1
1. Introducción.....	2
1.1. Hipótesis	2
2. Desarrollo.....	2
2.1. Revisión Documental.....	2
2.1.1. ¿Qué es un analizador sintáctico?.....	2
2.1.2. Analizador sintáctico descendente recursivo	3
2.1.3. BNF (Backus-Naur Form)	3
2.1.4. Gramática Utilizada	5
2.2. Bibliotecas Utilizadas.....	6
2.2.1. Biblioteca enum.....	6
2.2.2. Biblioteca sys	7
2.3. Fragmentos Importantes.....	8
3. Conclusión.....	12
4. Repositorio de Github	12
5. Referencias	12

1. Introducción

El análisis sintáctico es una parte fundamental en el proceso de interpretación de lenguajes formales, ya que permite comprender la estructura gramatical de un texto y determinar si cumple con las reglas definidas por la gramática. En este contexto un parser juega un papel crucial al descomponer el texto en sus componentes léxicos y sintácticos, facilitando la comprensión y el procesamiento de la información.

El desarrollo de un parser es un proceso complejo que involucra la implementación de algoritmos y estructuras de datos eficientes para reconocer y analizar la estructura sintáctica de un lenguaje. Este documento tiene como objetivo explorar algunas técnicas utilizadas en el desarrollo de parser, así como las herramientas más comúnmente empleadas en el ámbito.

A lo largo de este documento, abordaremos algunos conceptos fundamentales y describiremos el tipo de parser utilizado.

1.1. Hipótesis

Se espera que este nuevo analizador pueda generar el parser de un archivo que previamente fue analizado léxicamente; este nuevo programa debe asignar un tipo de token a cada valor de la entrada, los tokens fueron definidos por el equipo y agregados en una tabla de símbolos que será comparada por el analizador para que pueda hacer la comparación y asignación respectiva en cada caso.

2. Desarrollo

2.1. Revisión Documental

2.1.1. ¿Qué es un analizador sintáctico?

Un analizador sintáctico, también conocido como parser, es una herramienta que se utiliza para analizar y comprender la estructura de alguna frase o un texto. Su

objetivo es determinar la relación entre las palabras, las frases de un texto y su función gramatical.[1]

Un analizador sintáctico es parte de un compilador de código. Se encarga de revisar que todo el código esté escrito correctamente traduciéndolo y asegurándose de que el código sea ejecutable.[2]

2.1.2. Analizador sintáctico descendente recursivo

Un analizador sintáctico descendente recursivo comienza a construir su árbol de parsing desde la raíz, es decir del símbolo inicial y desciende a través de las reglas de producción de izquierda a derecha y de esta forma genera el análisis de una cadena de entrada.

Su funcionamiento recursivo se basa en la asociación de una regla de producción a cada elemento no terminal y se busca la coincidencia de la entrada con la regla correspondiente, si se encuentra se avanza en la entrada. En caso de encontrar otro elemento no terminal se busca la regla de producción asociada a dicho elemento hasta encontrar una coincidencia.

La entrada es aceptada como válida si se llega al final de la misma por medio de las reglas de producción. Si no se cumple con las reglas la entrada es marcada como inválida, presentando así, un error sintáctico. [3]

2.1.3. BNF (Backus-Naur Form)

BNF (Backus-Naur Form) es una gramática libre de contexto comúnmente utilizada por los desarrolladores de lenguajes de programación para especificar las reglas de sintaxis de un lenguaje. John Backus fue un diseñador de lenguajes de programas que ideó una notación para documentar IAL (una implementación temprana de Algol). Peter Naur trabajó más tarde en los hallazgos de Backus, y la notación se atribuyó conjuntamente a ambos informáticos. BNF utiliza una variedad de símbolos

y expresiones para crear reglas de producción. Una regla de producción BNF simple podría verse así:

<dígito> ::= 0|1|2|3|4|5|6|7|8|9

Esto se interpretaría como: un dígito se puede definir como 0, 1, 2, 3, 4, 5, 6, 7, 8 o 9

Los galones (< >) se utilizan para indicar un símbolo no terminal. Si aparece un símbolo no terminal en el lado derecho de las reglas de producción, significa que habrá otra regla de producción (o conjunto de reglas) para definir su reemplazo.

Considere la siguiente regla de producción:

<nombre completo> :: =<título><nombre><nombre>

Esto muestra que el nombre completo comprende un título, un nombre y otro nombre. Sin embargo, los tres componentes no son terminales. Por lo tanto, se requieren reglas de producción adicionales.

Por ejemplo, una regla de producción puede definir el título de la siguiente manera:

<título> :: =Señor|Señora|Señorita|Doctorr

En esta regla, Señor, Señora, Señorita y Doctor son símbolos terminales. No están encerrados entre galones, por lo que son los valores reales permitidos para el título. El símbolo de la tubería | es un meta carácter que se utiliza para indicar alternativas. Cada norma de producción se aplicará estrictamente; Es posible que sepa que existen otros títulos, por ejemplo, Lord, pero la regla de producción por sí sola determina las opciones válidas de uso en este lenguaje formal en particular.

Las reglas de producción para algo tan complejo como la sintaxis de un lenguaje vendrán como un conjunto muy grande de declaraciones BNF que especifican cómo se define cada aspecto del lenguaje. Siempre que encuentre un símbolo no terminal en el lado derecho de una regla de producción, debería haber otra regla que tenga

el símbolo en el lado izquierdo. Esto continúa hasta que se pueda especificar todo en relación con los símbolos de los terminales.

Aquí hay un conjunto completo de reglas (para un pequeño subconjunto de un lenguaje de programación):

<adición> ::= <número>+<número>

<número> ::= <signo><entero>|<entero>

<entero> ::= <dígito>|<dígito><integer>

<dígito> ::= 0|1|2|3|4|5|6|7|8|9

<signo> ::= +|-

Los símbolos de terminal son los dígitos del 0 al 9 y los signos más y menos. Tenga en cuenta que el signo más aparece dos veces, una como operador y otra como signo de un número. Una declaración de suma válida podría tener un signo más doble, p. 23 + +6. [4]

2.1.4. Gramática Utilizada

A continuación, se presenta una representación de la gramática para un lenguaje de programación básico, utilizando la notación Backus-Naur Form (BNF):

```

1  programa      ::= sentencia*
2  sentencia     ::= 'SI' comparacion 'ENTONCES' nl sentencia* 'FIN_SI'
3                | 'IMPRIMIR' ('expr' | 'CADENA')
4                | 'MIENTRAS' comparacion 'REPETIR' nl sentencia* 'FIN_MIENTRAS'
5                | 'ETIQUETA' ID
6                | 'GOTO' ID
7                | 'ENTERO' ID '=' expr
8                | 'ENTRADA' ID
9
10 comparacion   ::= expr (opComp expr)+
11 expr          ::= termino (('+' | '-') termino)*
12 termino       ::= unario (('*' | '/') unario)*
13 unario        ::= ('+' | '-')? primario
14 primario      ::= NUMERO | ID
15 opComp        ::= '==' | '!=' | '>' | '>=' | '<' | '<='
16 nl            ::= '\n'+
17

```

2.2. Bibliotecas Utilizadas

2.2.1. Biblioteca enum

La clase Enum de la biblioteca enum se utiliza para definir enumeraciones. Estas enumeraciones son una forma de crear un conjunto de nombres constantes que pueden ser más significativos y legibles que los valores numéricos.

TipoToken Enum:

En nuestro código, TipoToken es una enumeración que representa los diferentes tipos de tokens que pueden ser identificados durante el análisis léxico. Cada elemento de esta enumeración tiene un nombre (por ejemplo, FIN_LINEA, SALTO_LINEA, NUMERO, etc.) y un valor asociado. Los valores están asignados

automáticamente por la biblioteca enum, pero se puede acceder a ellos utilizando el atributo value.

Algunos ejemplos de elementos en la enumeración TipoToken y sus valores asociados son:

- FIN_LINEA con valor -1
- SALTO_LINEA con valor 0
- NUMERO con valor 1
- ID con valor 2
- ...

Esta enumeración proporciona una forma clara y semánticamente significativa de referirse a los diferentes tipos de tokens en lugar de usar valores numéricos directamente. Por ejemplo, en lugar de comparar si el tipo de token es igual a 1 para verificar si es un número, puedes usar TipoToken.NUMERO. Esto hace que el código sea más legible y menos propenso a errores.

En resumen, la biblioteca enum y la enumeración TipoToken se utilizan para mejorar la claridad y mantenibilidad del análisis léxico al proporcionar nombres significativos para los diferentes tipos de tokens.

2.2.2. Biblioteca sys

La biblioteca sys en Python proporciona acceso a algunas variables utilizadas o mantenidas por el intérprete de Python y funciones que interactúan fuertemente con el intérprete. En nuestro código, se utiliza la función sys.exit(mensaje) para salir del programa si ocurre un error.

```
import sys  
sys.exit(mensaje):
```

- Esta función se utiliza para salir del programa de manera inmediata.

- "mensaje es opcional y se imprime en la salida estándar (por lo general, la consola) antes de finalizar el programa.
- Se utiliza comúnmente para indicar un error crítico y detener la ejecución del programa.

Ejemplo de uso en nuestro código:

```
def abortar(self, mensaje):  
    sys.exit("Error: " + mensaje)
```

En este caso, si la función abortar es llamada, el programa se cerrará inmediatamente con un mensaje de error indicando la razón de la terminación.

2.3. Fragmentos Importantes

```
1 def __init__(self, lexico):  
2     self.lexico = lexico  
3     self.tokenActual = None # Este es el token actual  
4     self.asomarToken = None # Este es el token que sigue (pero sin que se guarde)  
5     self.siguienteToken()  
6     # Se tiene que llamar dos veces para inicializar actual y asomar  
7     self.siguienteToken()
```

Esta parte del código es la que nos permite ir recorriendo lexema a lexema el archivo de salida del analizador léxico, de esta manera le será más fácil clasificar los tokens

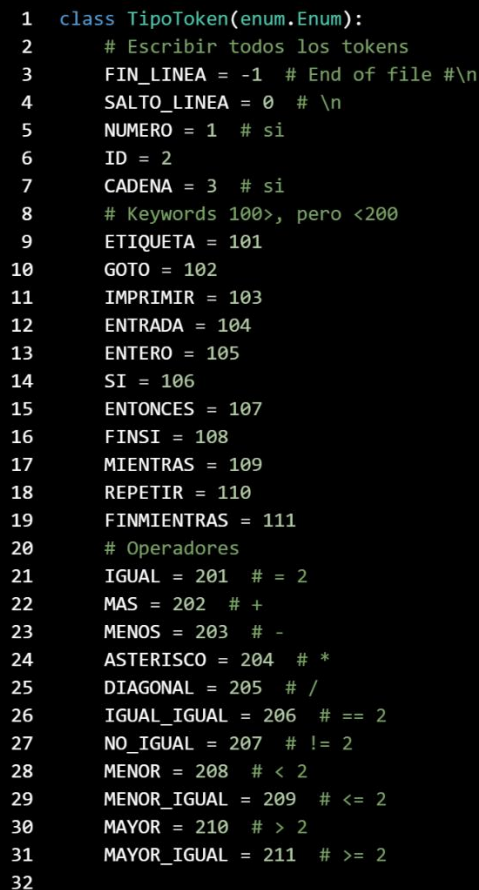
```

1 # Regresar true si el token actual es igual (del mismo tipo)
2 def revisarToken(self, tipo):
3     if (tipo == self.tokenActual.token):
4         return True
5     # return tipo == self.tokenActual.token
6
7 # Regresar true si el token siguiente es igual (del mismo tipo)
8 def revisarAsomar(self, tipo):
9     if (tipo == self.asomarToken.token):
10        return True
11
12 # Revisar si el tipo de token es el esperado
13 def match(self, tipo):
14     if not self.revisarToken(tipo): # Si no es el tipo que se estaba esperando
15         self.abortar("Se esperaba: " + tipo.name + ", se obtuvo " + self.tokenActual.token.name)
16     # Si es el tipo que se esperaba
17     self.siguienteToken()
18
19 # Pasar al siguiente token
20 def siguienteToken(self):
21     # Reemplazara el token actual por el siguiente
22     self.tokenActual = self.asomarToken
23     # Obtiene el token que sigue en el código
24     self.asomarToken = self.lexico.getToken()
25
26 def abortar(self, mensaje):
27     sys.exit("Error: " + mensaje)

```

En esta sección del código nos encontramos con 5 funciones que nos ayudan a lo largo de todo el análisis: la función `revisarToken` en conjunto con la función `revisarAsomar` sirven para leer el carácter actual y si coincide con el buscado entonces usamos la función `siguienteToken` la cuál es la encargada de pasar el carácter revisado al siguiente. La función `match` nos permite hacer la comparación entre `revisarToken` y el mismo token en sí, por último, la función `abortar` nos ayuda para el manejo de errores y poderle indicar al usuario qué es lo que ocurrió.

En esta parte del código podemos observar el funcionamiento de las funciones que nos ayudan a reconocer los tokens, esto lo hace con ayuda de las funciones mencionadas en la imagen anterior (match, siguienteToken, revisarToken, etc) de esta manera el analizador recorre lexema a lexema para después definir un token en específico.



```
1 class TipoToken(enum.Enum):
2     # Escribir todos los tokens
3     FIN_LINEA = -1 # End of file #\n
4     SALTO_LINEA = 0 # \n
5     NUMERO = 1 # si
6     ID = 2
7     CADENA = 3 # si
8     # Keywords 100>, pero <200
9     ETIQUETA = 101
10    GOTO = 102
11    IMPRIMIR = 103
12    ENTRADA = 104
13    ENTERO = 105
14    SI = 106
15    ENTONCES = 107
16    FINSI = 108
17    MIENTRAS = 109
18    REPETIR = 110
19    FINMIENTRAS = 111
20    # Operadores
21    IGUAL = 201 # = 2
22    MAS = 202 # +
23    MENOS = 203 # -
24    ASTERISCO = 204 # *
25    DIAGONAL = 205 # /
26    IGUAL_IGUAL = 206 # == 2
27    NO_IGUAL = 207 # != 2
28    MENOR = 208 # < 2
29    MENOR_IGUAL = 209 # <= 2
30    MAYOR = 210 # > 2
31    MAYOR_IGUAL = 211 # >= 2
32
```

Finalmente, esta sección del código nos define la tabla de símbolos utilizada tanto para el analizador léxico como para este parser, es la tabla a la que se está consultando cada vez para lograr hacer las comparaciones con efectividad.

3. Conclusión

Cómo conclusión de este proyecto el equipo llegó a la conclusión de si haber logrado cumplir con lo que la hipótesis menciona, esto debido a que en la misma y en los requerimientos del proyecto se pide un analizador sintáctico que nos permita generar el parser de un archivo de texto en el cual se leen instrucciones en un lenguaje de programación creado por nosotros, para ello fue importante crear la gramática del mismo y además, haber hecho un analizador léxico previo el cuál arrojará una salida que será la entrada de este parser para así poder seguir con el manejo de errores pero aún más importante, poder clasificar cada lexema en tokens específicos. Este analizador realiza dichos procedimientos y si es que se presenta el caso de encontrar algún error detiene todo el proceso e indica en dónde se presentó el error, en caso de no presentar errores entonces devuelve una salida similar al archivo, pero indicando que token es cada parte del mismo.

4. Repositorio de Github

https://github.com/Dumm312/Equipo11_Compiladores_Grupo4.git

5. Referencias

[1]“Qué es un analizador sintáctico o parser | Definición y tipos”. Arimetrics. Accedido el 16 de noviembre de 2023. [En línea]. Disponible: <https://www.arimetrics.com/glosario-digital/analizador-sintactico-parser>

[2]J. Jancso. “¿Qué es un Analizador Sintáctico y para qué sirve?” Accedido el 15 de noviembre de 2023. [En línea]. Disponible: <https://juliusjancso.com/wiki/analizador-sintactico/>

[3] “Tema 4. Análisis Sintáctico Descendente”, notas de clase de 1764, Departamento de Ingeniería eléctrica, Universidad Nacional Autónoma de México, otoño 2023

[4] “Backus-Naur Form”. Isaac Computer Science. Accedido el 16 de noviembre de 2023. [En línea]. Disponible:

https://isaacomputerscience.org/concepts/dsa_toc_bnf?examBoard=aqa&stage=a_level

