	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorios de docencia

Reporte Analizador Parser

Profesor(a): Rene Adrian Davila Perez

Asignatura: Compiladores

Grupo: 04

Integrante(s): 315278473

318126218

315348079

Semestre: 2024-1

Fecha de entrega: 15 - Noviembre - 2023

Introducción

En el presente trabajo se mostrará la aplicación y el funcionamiento del Parser, un Parser es una parte esencial de un compilador, ya que se encarga de analizar la estructura gramatical de un lenguaje de programación y convertirlo en una representación que pueda ser procesada por el compilador. Su función principal es validar si el código fuente cumple con la gramática definida por el lenguaje.
El parser recibe como entrada el código fuente en forma de tokens generados por el analizador léxico. Utiliza reglas gramaticales, como la definición de la sintaxis del lenguaje a través de gramáticas contextuales o expresiones regulares, para construir un árbol de sintaxis abstracta (AST) que representa la estructura jerárquica del código.
El proceso de parsing puede realizarse mediante distintos métodos, como análisis sintáctico ascendente (bottom-up) o descendente (top-down), cada uno con sus ventajas y desventajas en términos de eficiencia y complejidad.

Hipótesis

Al ejecutar el programa principal (main) y en conjunto con un archivo con sentencias del propio lenguaje el parser podrá recibir la salida del analizador léxico y revisar la estructura comparándola con las reglas gramaticales propuestas.
La implementación se hará desde recibir los parámetros a revisar empezando desde la “raíz” que es nuestra oración (declaración) completa y la irá separando hacia los nodos (hojas), es decir que usará una metodología TOP-BOTTOM.
De cumplirse lo anterior el programa nos indicará si dicha entrada es aceptada o rechazada.

Desarrollo

En nuestro código anterior agregaremos las siguientes funciones para que podamos implementar nuestro parser, y podamos obtener el árbol.

se creó la clase parser ya que se analizara el léxico y solicita un nuevo token según sea necesario

```
class Parser:
    def __init__(self, lexer):
        self.lexer = lexer

        self.simbolos = set()
        self.etiquetasdeclarada = set()
        self.etiquetasGotoed = set()

        self.tokenActual = None
        self.tokenObservandose = None
        self.siguienteToken()
        self.siguienteToken()
```

Las siguientes funciones permitirán al analizador decidir qué regla gramatical aplicar a continuación dado el token actual o el siguiente.

```
def revisarToken(self, kind):
    return kind == self.tokenActual.kind


def analizarToken(self, kind):
    return kind == self.tokenObservandose.kind
```

En los casos en el que el analizador ya sabe que regla gramatical, usaremos la función ‘comparaToken’. Espera que el token actual sea algo específico o, de lo contrario producirá un error. En otros casos, como cuando se usa ‘revisarToken’, solo queremos saltar al siguiente token con ‘siguienteToken’.

```
def comparaToken(self, kind):
    if not self.revisarToken(kind):
        self.Abortar("Esperando " + kind.name + ", encontro " + self.tokenActual.kind.name)
    self.siguienteToken()


def siguienteToken(self):
    self.tokenActual = self.tokenObservandose
    self.tokenObservandose = self.lexer.omitirToken()
```

Implementaremos ‘programa’, Esta función inicia el analizador sintáctico y es la regla principal en nuestra gramática.

```
def programa(self):
    print("programa")


    while self.revisarToken(tipoToken.NUEVA_LINEA):
        self.siguienteToken()


    while not self.revisarToken(tipoToken.FIN_DE_LINEA):
        self.declaracion()


    for etiqueta in self.etiquetasGotoed:
        if etiqueta not in self.etiquetasdeclarada:
            self.Abortar("intentando ir a una etiqueta no declarada: " + etiqueta)
```

La siguiente declaración espera un token 'IMPRIMIR' primero. A continuación, puede seguir una de estas dos cosas, una expresión matemática o un token de cadena.

```
if self.revisarToken(tipoToken.IMPRIMIR):
    print("Imprimiendo declaración")
    self.siguienteToken()

    if self.revisarToken(tipoToken.STRING):

        self.siguienteToken()

    else:

        self.expresion()
```

Vamos a implementar la función 'nuevaLinea' que maneja las nuevas líneas. Llamaremos a esto al final de la declaración, ya que se aplica a todas las declaraciones ya que se aplica en cada una de ellas. Funciona al menos un carácter de nueva linea, pero esta permite más.

```
def nl(self):
    print("Nueva Linea")


    self.comparaToken(tipoToken.NUEVA_LINEA)

    while self.revisarToken(tipoToken.NUEVA_LINEA):
        self.siguienteToken()
```

Las siguiente partes del código es para implementar cada una de las palabras reservadas para que cuando se lea no marque error y pueda continuar con la ejecución

```
elif self.revisarToken(tipoToken.IF):
    print("declarando IF")
    self.siguienteToken()
    self.comparacionn()

    self.comparaToken(tipoToken.THEN)
    self.nl()


    while not self.revisarToken(tipoToken.ENDIF):
        self.declaracion()

    self.comparaToken(tipoToken.ENDIF)


elif self.revisarToken(tipoToken.WHILE):
    print("declarando WHILE")
    self.siguienteToken()
    self.comparacionn()

    self.comparaToken(tipoToken.REPETIR)
    self.nl()


    while not self.revisarToken(tipoToken.ENDWHILE):
        self.declaracion()

    self.comparaToken(tipoToken.ENDWHILE)

elif self.revisarToken(tipoToken.LABEL):
    print("declarando etiqueta")
    self.siguienteToken()


    if self.tokenActual.text in self.etiquetasdeclarada:
        self.Abortar("etiqueta ya existente: " + self.tokenActual.text)
    self.etiquetasdeclarada.add(self.tokenActual.text)

    self.comparaToken(tipoToken.VARIABLE)
```

```
elif self.revisarToken(tipoToken.GOTO):
    print("declaracion-GOTO")
    self.siguienteToken()
    self.etiquetasGotoed.add(self.tokenActual.text)
    self.comparaToken(tipoToken.VARIABLE)


elif self.revisarToken(tipoToken.LET):
    print("declaracion-LET")
    self.siguienteToken()


    if self.tokenActual.text not in self.simbolos:
        self.simbolos.add(self.tokenActual.text)

    self.comparaToken(tipoToken.VARIABLE)
    self.comparaToken(tipoToken.IGUAL)

    self.expresion()


elif self.revisarToken(tipoToken.ENTRADA):
    print("declaracion-Entrada")
    self.siguienteToken()


    if self.tokenActual.text not in self.simbolos:
        self.simbolos.add(self.tokenActual.text)

    self.comparaToken(tipoToken.VARIABLE)


else:
    self.Abortar("declaracion invalida " + self.toker


self.nl()
```

```
def comparacionn(self):
    print("comparacion")

    self.expresion()

    if self.revisaOperadorComparacion():
        self.siguienteToken()
        self.expresion()
    else:
        self.Abortar("Operador de comparación esperado en: " + self.tokenActual.text)


    while self.revisaOperadorComparacion():
        self.siguienteToken()
        self.expresion()



def expresion(self):
    print("expresion")

    self.terminoMat()

    while self.revisarToken(tipoToken.SUMA) or self.revisarToken(tipoToken.RESTA):
        self.siguienteToken()
        self.terminoMat()



def terminoMat(self):
    print("terminoMat")

    self.unario()

    while self.revisarToken(tipoToken.ASTERISCO) or self.revisarToken(tipoToken.DIAGONAL):
        self.siguienteToken()
        self.unario()
```

```
def unario(self):
    print("unario")

    if self.revisarToken(tipoToken.SUMA) or self.revisarToken(tipoToken.RESTA):
        self.siguienteToken()
    self.primario()



def primario(self):
    print("primario (" + self.tokenActual.text + ")")

    if self.revisarToken(tipoToken.NUMERO):
        self.siguienteToken()
    elif self.revisarToken(tipoToken.VARIABLE):

        if self.tokenActual.text not in self.simbolos:
            self.Abortar("Variable de referencia antes de la asignación: " + self.tokenActual.text)

        self.siguienteToken()
    else:

        self.Abortar("Token inesperado en " + self.tokenActual.text)


def nl(self):
    print("Nueva Linea")


    self.comparaToken(tipoToken.NUEVA_LINEA)

    while self.revisarToken(tipoToken.NUEVA_LINEA):
        self.siguienteToken()
```

Al final al meter el programa junto con tu código fuente mostrará lo siguiente:
Código fuente
-IMPRIMIR "Prueba"

ENTRADA ABC

LET a = 1230
LET b = 1456

IMPRIMIR a-

```
                Parser . Proyecto 2 compiladores
programa
Imprimiendo declaración
Nueva Linea
declaracion-Entrada
Nueva Linea
declaracion-LET
expresion
terminoMat
unario
primario (1230)
Nueva Linea
declaracion-LET
expresion
terminoMat
unario
primario (1456)
Nueva Linea
Imprimiendo declaración
expresion
terminoMat
unario
primario (a)
Nueva Linea
```

Conclusiones

El parser nos ayuda a conocer la estructura después de analizarlo léxicamente, esto nos permite, de ser correcto, avanzar hacia la siguiente etapa de un compilador, que sería el análisis semántico donde buscaremos que lo que se introduce tenga significado.

Al realizar este programa (que fue un Top-Bottom) logramos entender como se va estructurando nuestro árbol de términos y como, una vez clasificados por el analizador léxico, buscamos que la entrada pueda ser producida por alguna de nuestras gramáticas definidas.

Nuestra hipótesis no fue del todo correcta, ya que realmente hicimos que el programa muestre cuáles sentencias han sido aceptadas y cuando una es rechazada imprime el error y el programa se termina.

Nos damos cuenta que, este parser aunque se basa mucho en el analizador léxico es el cimiento para el analizador semântico ya que nos dice si una producción es correcta o incorrecta y después podamos identificar que, lo que se le asigna a cada declaración o variable es correcto a lo que se espera.