Instituto Tecnológico de Costa Rica

Escuela de Computación

Ingeniería en computación

Compiladores e Intérpretes

Grupo 2

Profesor: Ericka Marín

Análisis léxico

Adrián López Quesada, 2014081634

Josué Arrieta Salas, 2014008153

Cartago

Lunes 18 de abril

Índice

Análisis de resultados

Se toman en cuenta las reglas de Python para la realización de este análisis de resultados.

Objetivo	Porcentaje de éxito (100%)
Programa recibe código fuente escrito en Python y analiza el archivo.	100%
Se listan errores léxicos encontrados (por línea y el error).	100%
El programa ante un error léxico ser recupera de este y no despliega errores en cascada.	100%
Se listan los tokens encontrados (por tipo, línea del código fuente donde se presentan y cantidad de ocurrencias de cada token en cada línea).	100%
La lista de tokens encontrados es ordenada en orden alfabético	100%
Tokens de operadores son procesados correctamente.	100%
Tokens de literales (strings, enteros, flotantes, char) son procesados correctamente.	100%
Tokens de identificadores son procesados correctamente.	100%
Tokens de palabras reservadas son procesados correctamente.	100%
El programa identifica comentarios (de bloque o de línea) y omite los tokens dentro de ellos.	100%: se ha de mencionar que si un comentario de bloque no se cierra, y se alcanza el EOF deberá notificar error.
Números aceptados en cualquier formato de Python (binario, hexadecimal y octal).	100%: este objetivo es importante ya que se estará realizando un scanner lo más parecido posible al lenguaje Python real.
Se lograron características en un buen scanner.	100%: este objetivo es lo que diferencia un buen scanner de uno malo. Casos como mañana, 999hola o ho+as son tomados como incorrectos.

Casos de pruebas

Prueba 1 : Tokens de operadores

Objetivo: cada operador permitido en el lenguaje es analizado. Se agregan operadores que no existen para verificar errores.

Resultados esperados: se espera la lista de errores para los operadores incorrectos y el listado de tokens de operadores correctos (agrupados por familia de operadores). Se ha de mencionar que anteriormente se tenía un Token por cada operador, sin embargo ahora es por familia de operadores. Esta es la primera prueba

Entrada: En el archivo Prueba.mypy se tiene el siguiente texto:

```
1  + - * / // % ** +
2  = += -= *= /= **= //=
3  == != <> < > <= >=
4  ( ) , . :
5  >> << | & ~ ^
6  and or not
7
8  $ @
9
```

- En la línea 8 se agregan operadores no existentes. Deben ser errores.
- En la línea 1 se agregan 2 '+' que deben Salir repetidos.

Resultados Obtenidos (correctos):

```
Analizador Creado. Se procede a scannear:
Error : $. En la línea: 8
Error : @. En la línea: 8
         de tipo: opDelimitadores en las lineas: 4.
!= de tipo: opComparadores en las lineas: 3.
% de tipo: opAritmeticos en las lineas: 1.
& de tipo: opBits en las lineas: 5.
( de tipo: opDelimitadores en las lineas: 4.
) de tipo: opDelimitadores en las lineas: 4.
* de tipo: opAritmeticos en las lineas: 1.
** de tipo: opAritmeticos en las lineas: 1.
**= de tipo: opAsingaciones en las lineas: 2.
*= de tipo: opAsingaciones en las lineas: 2.
+ de tipo: opAritmeticos en las lineas: 1(2).
+= de tipo: opAsingaciones en las lineas: 2.
, de tipo: opDelimitadores en las lineas: 4.
- de tipo: opAritmeticos en las lineas: 1.
-= de tipo: opAsingaciones en las lineas: 2.
. de tipo: opDelimitadores en las lineas: 4.
/ de tipo: opAritmeticos en las lineas: 1.
// de tipo: opAritmeticos en las lineas: 1.
//= de tipo: opAsingaciones en las lineas: 2.
/= de tipo: opAsingaciones en las lineas: 2.
: de tipo: opDelimitadores en las lineas: 4.
< de tipo: opComparadores en las lineas: 3.
<< de tipo: opBits en las lineas: 5.
<= de tipo: opComparadores en las lineas: 3.
<> de tipo: opComparadores en las lineas: 3.
= de tipo: opAsingaciones en las lineas: 2.
== de tipo: opComparadores en las lineas: 3.
> de tipo: opComparadores en las lineas: 3.
>= de tipo: opComparadores en las lineas: 3.
>> de tipo: opBits en las lineas: 5.
^ de tipo: opBits en las lineas: 5.
and de tipo: opLogicos en las lineas: 6.
not de tipo: opLogicos en las lineas: 6.
or de tipo: opLogicos en las lineas: 6.
| de tipo: opBits en las lineas: 5.
~ de tipo: opBits en las lineas: 5.
Análisis léxico terminado.
```

Prueba 2 : Tokens de palabras reservadas

Objetivo: cada palabra reservada en el lenguaje es analizado.

Resultados esperados: se espera el listado de tokens de operadores correctos Se ha de mencionar que anteriormente se tenía un Token por cada palabra reservada, sin embargo ahora se agrupan en un solo token. Esta es la primera prueba.

Entrada: En el archivo Prueba.mypy se tiene el siguiente texto:

```
1  assert break class continue def del elif else except exec finally for from
2  global if import in is lambda pass print raise return try while
3  break break
4  int string list float bool
```

En la línea 3 se puso 2 palabras reservadas repetidas.

Resultados obtenidos (correctos):

```
Analizador Creado. Se procede a scannear:
assert de tipo: PalabraReservada en las lineas: 1.
bool de tipo: PalabraReservada en las lineas: 4.
break de tipo: PalabraReservada en las lineas: 1, 3(2).
class de tipo: PalabraReservada en las lineas: 1.
continue de tipo: PalabraReservada en las lineas: 1.
def de tipo: PalabraReservada en las lineas: 1.
del de tipo: PalabraReservada en las lineas: 1.
elif de tipo: PalabraReservada en las lineas: 1.
else de tipo: PalabraReservada en las lineas: 1.
except de tipo: PalabraReservada en las lineas: 1.
exec de tipo: PalabraReservada en las lineas: 1.
finally de tipo: PalabraReservada en las lineas: 1.
float de tipo: PalabraReservada en las lineas: 4.
for de tipo: PalabraReservada en las lineas: 1.
from de tipo: PalabraReservada en las lineas: 1.
global de tipo: PalabraReservada en las lineas: 2.
if de tipo: PalabraReservada en las lineas: 2.
import de tipo: PalabraReservada en las lineas: 2.
in de tipo: PalabraReservada en las lineas: 2.
int de tipo: PalabraReservada en las lineas: 4.
is de tipo: PalabraReservada en las lineas: 2.
lambda de tipo: PalabraReservada en las lineas: 2.
list de tipo: PalabraReservada en las lineas: 4.
pass de tipo: PalabraReservada en las lineas: 2.
print de tipo: PalabraReservada en las lineas: 2.
raise de tipo: PalabraReservada en las lineas: 2.
return de tipo: PalabraReservada en las lineas: 2.
string de tipo: PalabraReservada en las lineas: 4.
try de tipo: PalabraReservada en las lineas: 2.
while de tipo: PalabraReservada en las lineas: 2.
Análisis léxico terminado.
```

Prueba 3 : Comentarios de línea y de bloque

Objetivo: comentarios son prbados..

Resultados esperados: Se deben ignorar. Si un comentario de bloque no se cierra y se llega al final del archivo, es considerado un error. Se deben listar los tokens y errores correctamente. Esta es la primera prueba.

Entrada: En el archivo Prueba.mypy se tiene el siguiente texto:

Prueba.mypy | Prueba.mypy

```
1  #este es un comentario de línea
2  expresion = 4 + 5
3  """este comentario esta correctamente cerrado"""
4  """hola
5  como !"#$% 12341234          " " "" "
6  estas
7      este comentario no esta correctamente cerrado
8  ""
```

Resultados obtenidos (correctos):

```
Analizador Creado. Se procede a scannear:
Error : Comentario de bloque sin terminar: """hola
como !"#$% 12341234              " " "" "
estas

"". En la línea: 8
+ de tipo: opAritmeticos en las lineas: 2.
4 de tipo: INT en las lineas: 2.
5 de tipo: INT en las lineas: 2.
= de tipo: opAsignaciones en las lineas: 2.
expresion de tipo: Identificador en las lineas: 2.
Análisis léxico terminado.
```

Los comentarios correctos son ignorados.

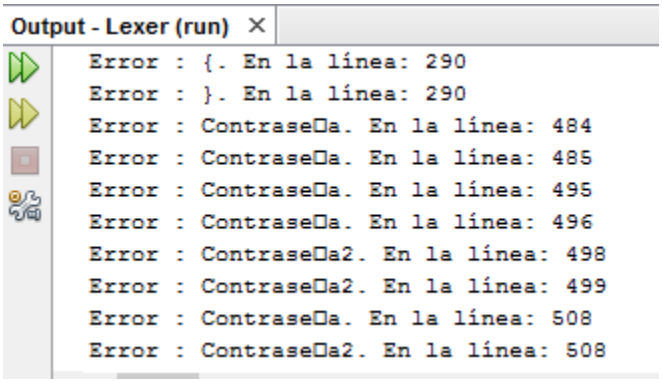
Prueba 4: Código exhaustivo

Objetivo: probar el lexer en un código verdadero

Se prueba código de 592 líneas, trozo relevante:

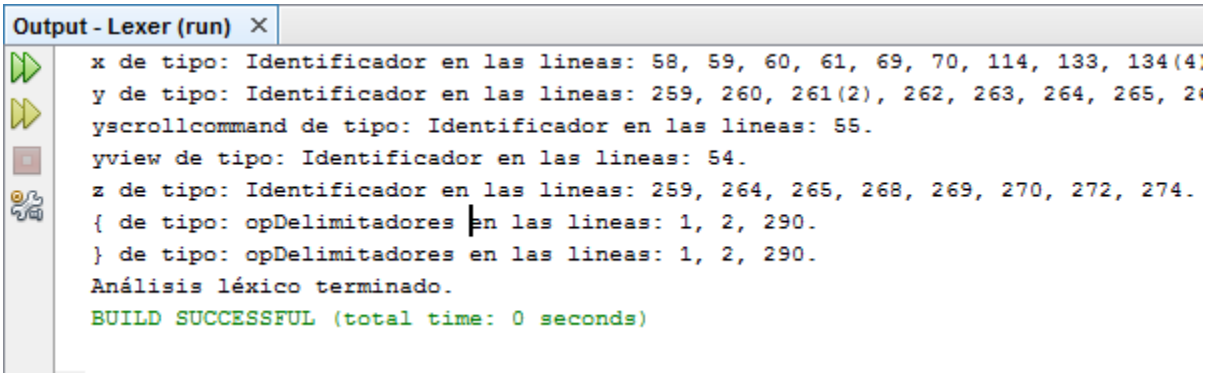
```
            imagen=PhotoImage(file=paths[x+ind])
            self.N_postales[x].configure(image=imagen)
            self.N_postales[x].configure(compound=NONE)
            self.N_postales[x].image=imagen
            self.N_postales[x].configure(command=self.hol)
    users=pickle.load(open("usuarios.p","rb"))
    for x in users:
        r=pickle.load(open(x+".p","rb"))
        if r[0].count(1)==self.todo:
            self.screen2=Toplevel(v0)
            Label(self.screen2,text="Felicidades").pack()
            dicc={}
            for x in users:
                r=pickle.load(open(x+".p","rb"))
                dicc[x]=r[0].count(1)
            dic(list(dicc.values()),list(dicc.items()),0,self.screen2)
            self.screen.withdraw()
            self.anterior.withdraw()
            f=codecs.open("admi.txt","w","utf-8")
            f.write("12")
            f.close
            r=[]
            pickle.dump(r,open("subasta2.p","wb"))
```

Resultados obtenidos (Incorrectos):



Los errores en la línea 290 no deberían dar error, los demás son problemas de ñ's y todo lo demás sale correcto.

Resultados obtenidos tras la correcion:



Prueba 5: Strings y Chars

Objetivo: probar errores y reconocimiento de strings y chars

Se prueba código de 592 líneas, trozo relevante:

```
"hola2+"
"hela"
"he+l
a"
'h'
"       hola mundo \t yes"
'h s'

'h
'
```

Resultados obtenidos (Incorrectos):

```
Error : "he+la". En la línea: 4
Error :          . En la línea: 6
Error :   . En la línea: 6
Error :   . En la línea: 6
Error :   . En la línea: 6
Error : 'h s'. En la línea: 7
Error : 'h
'. En la línea: 10
"hela" de tipo: STRING en las lineas: 2.
"hola2+" de tipo: STRING en las lineas: 1.
"holamundo      yes" de tipo: STRING en las lineas: 6.
'h' de tipo: CHAR en las lineas: 5.
```

No reconce tabs normales y espacios en strings, los errores de la línea 6 no deberían aparecer, los otros si tienen sentido ya que están malos. Acepto todos los otros strings y chars correctamente.

Resultados obtenidos tras la corrección:

```
Error : "he+l
a". En la línea: 4
Error : 'h s'. En la línea: 7
Error : 'h
'. En la línea: 10
"       hola mundo       yes" de tipo: STRING en las lineas: 6.
"hela" de tipo: STRING en las lineas: 2.
"hola2+" de tipo: STRING en las lineas: 1.
'h' de tipo: CHAR en las lineas: 5.
Análisis léxico terminado.
```

Todo caso del string y char es cubierto.

Prueba 6: Números y flotantes

Objetivo: se prueba todos los posibles literales de números permitidos (flotantes y enteros). También se agregan números no posibles para probar errores.

Resultados esperados: se espera la lista de errores para los operadores incorrectos y el listado de tokens de operadores correctos. Se espera algunos resultados incorrectos.

Entrada: En el archivo Prueba.mypy se tiene el siguiente texto:

```
4 1234 -2 0
0.0 0.03 1233.41234 0.124312301
0b1010101 0b123
0o123123 0o123456789
0x123abcd 0x12hola
```

Resultados obtenidos (correctos):

```
Error : 0b123. En la línea: 3
Error : 0o123456789. En la línea: 4
Error : 0x12hola. En la línea: 5
- de tipo: opAritmeticos en las lineas: 1.
0 de tipo: INT en las lineas: 1.
0.0 de tipo: FLOAT en las lineas: 2.
0.03 de tipo: FLOAT en las lineas: 2.
0.124312301 de tipo: FLOAT en las lineas: 2.
0b1010101 de tipo: INT en las lineas: 3.
0o123123 de tipo: INT en las lineas: 4.
0x123abcd de tipo: INT en las lineas: 5.
1233.41234 de tipo: FLOAT en las lineas: 2.
1234 de tipo: INT en las lineas: 1.
2 de tipo: INT en las lineas: 1.
4 de tipo: INT en las lineas: 1.
Análisis léxico terminado.
```

Prueba numero 7: identificadores

Objetivo: se prueban todos los posibles identificadores permitidos, además de errores de identificadores.

Resultados esperados: se espera la lista de errores con todo el token incorrecto, además de un listado de que incluya los identificadores correctos.

Entrada: En el archivo Prueba.mypy se tiene el siguiente texto:

```
hola mundo
ho
la
cuatro+cinco
variable=6
def funcion()
999hola
hola99
__variable__
mañana
manána
mañana
_varia_ble
```

Resultados obtenidos (correctos):

```
Analizador Creado. Se procede a scannear:
Error : 999hola. En la línea: 7
Error : ma□ana. En la línea: 10
Error : man□na. En la línea: 11
Error : ma□ana. En la línea: 12
( de tipo: opDelimitadores en las lineas: 6.
) de tipo: opDelimitadores en las lineas: 6.
+ de tipo: opAritmeticos en las lineas: 4.
6 de tipo: INT en las lineas: 5.
: de tipo: opDelimitadores en las lineas: 6.
= de tipo: opAsignaciones en las lineas: 5.
__variable__ de tipo: Identificador en las line
_varia_ble de tipo: Identificador en las lineas
cinco de tipo: Identificador en las lineas: 4.
cuatro de tipo: Identificador en las lineas: 4.
def de tipo: PalabraReservada en las lineas: 6.
funcion de tipo: Identificador en las lineas: 6
ho de tipo: Identificador en las lineas: 2.
hola de tipo: Identificador en las lineas: 1.
hola99 de tipo: Identificador en las lineas: 8.
la de tipo: Identificador en las lineas: 3.
mundo de tipo: Identificador en las lineas: 1.
variable de tipo: Identificador en las lineas:
Análisis léxico terminado.
```

Se puede apreciar como contiene los errores de identificadores completos y no separa el 999 como int del hola como identificador. Pasa lo mismo con caracteres inválidos en los nombres de identificadores, pero acepta todo identificador que empieza en carácter o “_” y cualquier identificador que tenga caracteres válidos, números y “_” en su interior.

¿Cómo compilar y correr el Scanner?

La parte esencial para compilar el proyecto es agregar el archivo “jflex-1.6.1.jar” como JAR a las librerías que tiene acceso. De esta manera se pueden utilizar todas las funcionalidades del Jflex. En caso de duda, por favor seguir las instrucciones del video siguiente: https://www.youtube.com/watch?v=w-KfjJdRas8 cerca del minuto 10, tomando en cuenta que se importa como JAR, no como librería.

Aunque no es de relevancia, el proyecto se desarrolló bajo el IDE de Netbeans. Para correr el scanner solo se ejecuta el proyecto y este leerá el archivo Prueba.mypy que se encuentra dentro de la carpeta Lexer. Se debe usar esta misma localización y este archivo (es posible modificarlo) para que el scanner pueda hacer su análisis.