Proyecto #1

Scanner para el lenguaje de programación MyPython

Integrantes:

Yordan Jimenez Hernandez

Jose Fernando Molina Chacon.

Instituto Tecnológico de Costa Rica

Compiladores e Intérpretes

Curso: IC-5701

Cartago

2016

Funcionamiento

Durante el desarrollo de esta tarea programada se utilizó el paradigma de orientación a objetos, específicamente con el lenguaje de programación Java con la ayuda del entorno de desarrollo libre Netbeans 8.1. Gracias a ello se pudo utilizar la herramienta Jflex para establecer las expresiones regulares con la que se basa nuestro compilador del lenguaje de programación 'ficticio', llamado MyPython.

Explicando más profundamente el funcionamiento de nuestro programa o tarea, podemos dividirla en dos secciones, las clases para la interacción de los datos y las reglas mediante de la sintáctica permitida por la librería JFlex para establecer las expresiones regulares. Empezaremos explicando las clases, estas se encuentran encapsuladas en tres distintos paquetes los cuales son scanner, Scanner_Jlex y FilesScanner.

El paquete scanner guarda en él, la abstracción de Token principalmente, que almacena su valor, cantidad de apariciones y la línea donde se encontra, además incluye funcionalidades de comparación entre los elementos de esta misma clase, dicha clase se pudo especializar en EOFToken, LiteralToken, SeparatorToken, OperatorToken, IdentifierToken, ReservedWordToken y ErrorToken, los cuales por obviedad se puede deducir su papel en nuestro programa con solo leer su nombre. Además es importante aludir que LiteralToken se especializó en 6 clases más, esto para facilitar el trabajo al parser a la hora de recibir los token, procedemos a explicar cada una:

- **BooleanToken**, se encarga de representar solamente los valores True o False, booleans dentro de este lenguaje de programación.
- **CharToken,** almacena solamente un valor caracter.
- **FloatToken**, identifica los valores numéricos que poseen valor punto flotante.
- **IntegerToken,** representa valores numéricos enteros.
- **NullToken,** establece un valor nulo en el programa.
- **StringToken**, el papel de este token es identificar los valores string entre comillas de este lenguaje de programación.

Es necesario aclarar que estos tokens se encuentran dentro de una jerarquía desde Token, por lo cual implementan las cualidades de este. La clase ViewToken, establece la lógica para sincronizar los Token generados por MyPythonScanner con la clase que imprime los token en pantalla ya se unitariamente o por medio de un hash que simplifica en mostrar la cantidad de apariciones por línea, si hay existencia de repetición de Tokens. Se omite la clase Scanner donde se establece el método main del programa.

El paquete Scanner_Jflex almacena la lógica para conectar de la herramienta Jflex con Java, existen tres archivos dentro de este, GeneratorLexer, MyPythonScanner y Rules.flex. La clase GeneratorLexer implementa código para crear la clase MyPythonScanner, por medio de la librería jflex y un path que establece la ubicación del archivo Rules.flex. Este archivo Rules.flex se explicará su comportamiento más adelante.

El paquete FilesScanner implementa dos clases para acciones generales, como abrir, obtener el reader de un archivo en uso, además posee la clase printer donde se desarrolla la lógica de agrupar repeticiones de token por línea utilizando la herramienta del hash.

La última sección del programa son las reglas de las expresiones regulares, que se desarrollaron en el archivo Rules.flex. Se crearon distintas reglas para identificar la variedad

de Tokens que acepta nuestro lenguaje de programación utilizando la sintaxis permitida por la librería jflex, entre las cuales podemos encontrar las más importantes:

- **Separator**, indica la existencia válida de un token separador.
- **Operator**, indica la existencia válida de un token operador.
- **BlockComment**, identifica la existencia válida de un comentario de bloque para ignorar.
- **EndOfLineComment**, identifica la existencia válida de un comentario de bloque para ignorar.
- **Identifier**, indica la existencia válida de un token identificador.
- **ErrorIdent,** identifica un Token identificador inválido, cuando inicia con un número.
- **DecIntegerLiteral,** indica la existencia válida de un token numérico entero.
- **HexIntegerLiteral**, indica la existencia válida de un token numérico entero con el formato de hexadecimal.
- **OctIntegerLiteral**, indica la existencia válida de un token numérico entero con el formato de Octal.
- **BinIntegerLiteral**, indica la existencia válida de un token numérico entero con el formato de binario.
- **FloatLiteral**, indica la existencia válida de un token numérico con la propiedad de punto flotante.
- Para los string y char se utilizó los estados <STRING> y <CHARLITERAL>, donde se valida la estructura de estos.
- Para las palabras reservadas se estableció una por una debido a que su aparición debe ser única en su formato o estructura.

Para los números enteros en otros formatos se utilizaron los estados <OCTOVAL>, <HEXAVAL> y <BINAVAL> para validar la estructura estrictamente e identificar errores en estos. Lo mismo sucede para los comentarios de bloque con el estado <BLOCKCOMMENT>. Para concluir es importante remarcar que las reglas utilizan otras reglas no incluidas pues no son las principales y son de ayuda auxiliar.

Manual de Usuario

Cómo compilar el proyecto:

```
public class Scanner {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        String pathPuntoFlex ="..\\Compilador-MyPython\\src\\Scanner_Jflex\\Rules.flex";
        String pathPuntoMyPython="..\\Compilador-MyPython\\docs\\Prueba 4.mypython";

//      GeneratorLexer generator = new GeneratorLexer(pathPuntoFlex);
//      generator.createlexer();
//
        ViewTokens.view(pathPuntoMyPython);
    }
}
```

1. Es necesario ubicarse en la clase llamada Scanner en el paquete llamado scanner, allí se encuentra el método main de nuestro proyecto.
    a. Existen dos path,**pathPuntoFlex** y el **pathPuntoMyPython,** que son respectivamente la ubicación del archivo Rules.flex (expresiones regulares) y la ubicación del archivo MyPython (donde se encuentra el código a prueba).
2. Se debe elegir entre generar la clase que establece las expresiones regulares o ejecutar un código MyPython:
    a. Si desea ejecutar el código del archivo .flex para actualizar las expresiones regulares, se debe eliminar de comentario, las líneas de instanciación de GeneratorLexer, líneas en amarillo.
    b. Si desea solo ver un resultado de un archivo prueba, ejecute el proyecto solamente llamado a la clase estática ViewTokens y su metodo view con el path del archivo MyPython.
3. Ejecutar el programa con un resultado similar al siguiente formato:

```
Valor: testMedian
        Tipo: Identifier
        Linea: 10
Valor: unittest
        Tipo: Identifier
        Linea: 1 9 13
Valor: ñ
        Tipo: Error
        Linea: 1
BUILD SUCCESSFUL (total time: 0 seconds)
```

Análisis de Resultados

Luego del desarrollo del proyecto se logró realizar un Scanner para el lenguaje descrito en clase llamado MyPython el cual divide los tokens en los tipos indicados en el apartado anterior. Se logra también imprimir dichos tokens en consola para su visualización. Los tokens son impresos en orden de acuerdo a su valor ASCII, y cada token es impreso solamente una vez junto con la lista de las líneas donde apareció. Para iniciar el proyecto, se inició con uno de los ejemplos provistos por JFlex, el del lenguaje de programación Java. Mirando el ejemplo, se empezó a modificar para hacerle frente a la especificación de MyPython.

Luego de una semana de trabajo, se tiene un Scanner que cumple con todos los requerimientos planteados por la profesora para el proyecto. En el desarrollo se encontraron varios errores que fueron resueltos satisfactoriamente, en el siguiente apartado se explica cada uno de ellos. Debido a la cantidad de pruebas que realizamos, se podría decir que es poco probable que haya algún error en el programa, pero no podemos asegurar esto. Un aspecto importante de recalcar es la manera en que fueron manejados los errores léxicos. Tratamos en la medida de lo posible de utilizar errores con un mensaje significativo para que el usuario pudiera saber dónde está su error y así pueda solucionarlo. Pensamos que esta primera parte del compilador es exitosa y esperamos que la evaluación de los resultados lo pruebe.

Pruebas

- Se prueba el siguiente código:

```
name = raw_input('Wololo')
print 'Hi, %s.' % name
```

    - Resultado esperado: El Scanner debería detectar el error en ‘Wololo’, debido a que es un char con más de un carácter.
    - Resultado Inesperado:

```
Valor: raw_input
        Tipo: Identifier
        Linea: 1
Valor: (
        Tipo: Separator
        Linea: 1
Valor: W
        Tipo: Error
        Linea: 1
Valor: o
        Tipo: Error
        Linea: 1
Valor: l
        Tipo: Error
        Linea: 1
Valor: o
        Tipo: Error
        Linea: 1
Valor: l
        Tipo: Error
        Linea: 1
Valor: 'o'
        Tipo: Literal String
        Linea: 1
```

- El scanner encontró el error, un char no puede tener más de un carácter entre comillas simples, pero separa lo subsecuente como si fueran errores separados y no uno concreto. Se procede a reparar el error.
- Se vuelve a probar el código, resultando:

```
Valor: (
        Tipo: Separator
        Linea: 1
Valor: 'Wololo'
        Tipo: Error
        Linea: 1
Valor: )
        Tipo: Separator
        Linea: 1
```

- Se prueba el siguiente código:

```
import unittest
def median(pool):
    copy = sorted(pool)
    size = len(copy)
    if size % 2 == 1:
       return copy[(size - 1) / 2]
    else:
       return (copy[size/2 - 1] + copy[size/2]) / 2
class TestMedian(unittest.TestCase):
    def testMedian(self):
      self.failUnlessEqual(median([2, 9, 9, 7, 9, 2, 4, 5, 8]), 7)
if __name__ == "__main__":
    unittest.main()
```

- Resultado esperado: No hay errores. Se espera no haya ningún error y tome en cuenta todos los Tokens aceptados.
- Resultado Inesperado:

```
Valor: (
        Tipo: Separator
        Linea: 2
Valor: pool
        Tipo: Identifier
        Linea: 2
Valor: )
        Tipo: Separator
        Linea: 2
Valor: :
        Tipo: Separator
        Linea: 2
Valor: copy
        Tipo: Identifier
        Linea: 3
```

- Se nota que falta tomar en cuenta las tabulaciones. Se agregan las tabulaciones a la lista de reglas, resultando en la solución del problema:

```
Valor: (
        Tipo: Separator
        Linea: 2
Valor: pool
        Tipo: Identifier
        Linea: 2
Valor: )
        Tipo: Separator
        Linea: 2
Valor: :
        Tipo: Separator
        Linea: 2
Valor: Tab
        Tipo: Separator
        Linea: 3
Valor: copy
        Tipo: Identifier
```

- Se prueba el siguiente código:

  - Resultado Esperado: Se reconoce como un literal de tipo entero.
  - Resultado inesperado:

```
Valor: 0001
        Tipo: Literal Float
        Linea: 1
```

  - Se hace match como si fuera un literal de tipo float, cuando en realidad es un entero. Se procede a revisar la regla de los float y se encuentra un error. Se repara y el error queda resuelto:

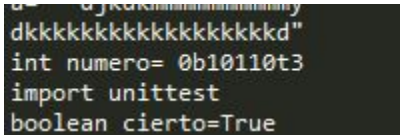
```
Valor: 0001
        Tipo: Literal Integer
        Linea: 1
```

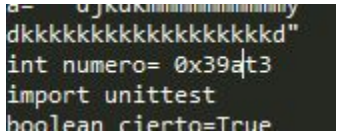
- Se prueba el siguiente código:

- ```
  23"hola"list
      int a4 = 1
  ```
- Resultado esperado: El scanner debería dividir lo tokens en 23, “hola”, list, Tab, int, a4, = y 1.
- Resultado inesperado:

```
Valor: list
        Tipo: Reserved Word
        Linea: 1
Valor:
        Tipo: Separator
        Linea: 2
Valor: int
        Tipo: Reserved Word
        Linea: 2
```

- El scanner reconoce la tabulación pero no la imprime. Se procede a reparar el error resultando:

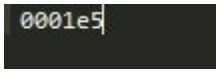
```
Valor: list
        Tipo: Reserved Word
        Linea: 1
Valor: Tab
        Tipo: Separator
        Linea: 2
Valor: int
        Tipo: Reserved Word
        Linea: 2
```
```

- Se prueba el siguiente código:
  - 
    ```
    dkkkkkkkkkkkkkkkkkd"
    int numero= 0b10110t3
    import unittest
    boolean cierto=True
    ```
  - Resultado esperado: El scanner debería reconocer el error 0b10110t3, no es un número binario bien formado.
  - Resultado inesperado:
    ```
    Valor: int
            Tipo: Reserved Word
            Linea: 4
    Valor: numero
            Tipo: Identifier
            Linea: 4
    Valor: =
            Tipo: Operator
            Linea: 4
    Valor: 0
            Tipo: Literal Integer
            Linea: 4
    Valor: 11010k
            Tipo: Error
            Linea: 4
    Valor: 3
            Tipo: Operator
            Linea: 4
    ```
  - El scanner separa incorrectamente el error en tres distintos tokens, los cuales solo deberían ser dos para el caso presentado, el resultado al reparar el error es:
    ```
    Valor: int
            Tipo: Reserved Word
            Linea: 5
    Valor: numero
            Tipo: Identifier
            Linea: 5
    Valor: =
            Tipo: Operator
            Linea: 5
    Valor: 10110t
            Tipo: Error
            Linea: 5
    Valor: 3
            Tipo: Literal Integer
            Linea: 5
    Valor: import
            Tipo: Reserved Word
            Linea: 6
    ```

- Se prueba el siguiente código:
  - ```
    a= " djkdkmmmmmmmmmy
    dkkkkkkkkkkkkkkkkkkd"
    int numero= 0o45t3
    ```
  - Resultado esperado El scanner debería reconocer el error 0o45t3, no es un número octal bien formado.
  - Resultado inesperado:

```
Valor: int
        Tipo: Reserved Word
        Linea: 4
Valor: numero
        Tipo: Identifier
        Linea: 4
Valor: =
        Tipo: Operator
        Linea: 4
Valor: 37
        Tipo: Literal Integer
        Linea: 4
Valor: t3
        Tipo: Identifier
        Linea: 4
```

  - El scanner separa el el token que se desea que sea número octal, convierte la primera parte un número octal correcto y la siguiente parte en un identificador, cuestión que es incorrecto, la solución mostró:

```
Valor: int
        Tipo: Reserved Word
        Linea: 5
Valor: numero
        Tipo: Identifier
        Linea: 5
Valor: =
        Tipo: Operator
        Linea: 5
Valor: 45t
        Tipo: Error
        Linea: 5
Valor: 3
        Tipo: Literal Integer
        Linea: 5
Valor: import
        Tipo: Reserved Word
        Linea: 6
```
```

- Se prueba el siguiente código:
  - 
    ```
    dkkkkkkkkkkkkkkkkkd"
    int numero= 0x39at3
    import unittest
    boolean cierto=True
    ```
  - Resultado esperado: El scanner debería reconocer el error 0x39at3, no es un número hexadecimal bien formado.
  - Con el resultado:

```
Valor: int
        Tipo: Reserved Word
        Linea: 4
Valor: numero
        Tipo: Identifier
        Linea: 4
Valor: =
        Tipo: Operator
        Linea: 4
Valor: 922
        Tipo: Literal Integer
        Linea: 4
Valor: t3
        Tipo: Identifier
        Linea: 4
Valor: import
        Tipo: Reserved Word
        Linea: 5
```

  - El scanner separa el el token que se desea que sea número hexadecimal, convierte la primera parte un número hexadecimal correcto y la siguiente parte en un identificador, cuestión que es incorrecto, la solución mostró:

```
Valor: int
        Tipo: Reserved Word
        Linea: 5
Valor: numero
        Tipo: Identifier
        Linea: 5
Valor: =
        Tipo: Operator
        Linea: 5
Valor: 39at
        Tipo: Error
        Linea: 5
Valor: 3
        Tipo: Literal Integer
        Linea: 5
Valor: import
        Tipo: Reserved Word
        Linea: 6
```

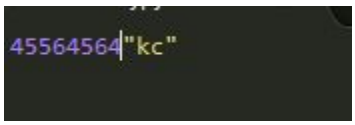
- Se prueba el siguiente código:
  - 

  - Resultado esperado: El scanner reconoce el numero como un numero en punto flotante con exponenete 5.
  - Brindando el resultado:

```
Valor: Error: Identificador invalido: 0001e
        Tipo: Error
        Linea: 1
Valor: 5
        Tipo: Literal Integer
        Linea: 1
BUILD SUCCESSFUL (total time: 0 seconds)
```

  - Es claro que el Scanner, tomo este token como un identificador, y no como un float que es el correcto, la solución fue:

```
Valor: 0001e5
        Tipo: Literal Float
        Linea: 1
```

- Se prueba el siguiente código:
  - 

  - Resultado esperado: El scanner debe separar los tokens en uno de tipo Integer y otro de tipo String.
  - Con el resultado:

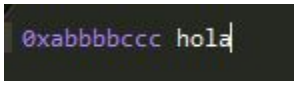
```
Exception in thread "main" java.lang.Error: Error: pushback value was too large
        at Scanner_Jflex.MyPythonScanner.zzScanError(MyPythonScanner.java:747)
        at Scanner_Jflex.MyPythonScanner.yypushback(MyPythonScanner.java:761)
        at Scanner_Jflex.MyPythonScanner.yylex(MyPythonScanner.java:912)
        at scanner.Scanner.main(Scanner.java:44)
C:\Users\jjime\AppData\Local\NetBeans\Cache\8.1\executor-snippets\run.xml:53: Java returned: 1
BUILD FAILED (total time: 0 seconds)
```

  - Presentamos un error en la pila de la clase generada por el jflex a la hora de trabajar con el final del archivo, se soluciono de esta forma:

```
Valor: Error: Identificador invalido: 45564564"kc
        Tipo: Error
        Linea: 1
Valor: Fin del archivo en "
        Tipo: Error
        Linea: 1
BUILD SUCCESSFUL (total time: 0 seconds)
```

- Pero esa no era la solución requerida, necesitábamos que los dividiera en dos Tokens un string y un integer, de la siguiente forma:

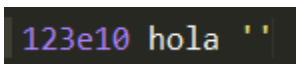
```
Valor: 45564564
        Tipo: Literal Integer
        Linea: 1
Valor: "kc"
        Tipo: Literal String
        Linea: 1
```

- Se prueba el siguiente código:
  - 
  - Resultado esperado: Separación en dos tokens: un número entero hexadecimal y un identificador.
  - Resultado inesperado:

```
run:
Valor: 0xabbbbccc h
        Tipo: Error
        Linea: 1
Valor: ola
        Tipo: Identifier
        Linea: 1
BUILD SUCCESSFUL (total time: 0 seconds)
```

  - Se presenta un error que aceptaba el carácter espacio como un elemento válido hexadecimal, por lo cual separa en dos creando un Token error y un identificador, solución:

```
Valor: 180075724
        Tipo: Literal Integer
        Linea: 1
Valor: hola
        Tipo: Identifier
        Linea: 1
BUILD SUCCESSFUL (total time: 0 seconds)
```

- Se prueba el siguiente código:
  - 
  - Resultado esperado: Separación un token de tipo Integer, otro identificador y un error de char.
  - Con el resultado:

```
Valor: 123e10
        Tipo: Literal Float
        Linea: 1
Valor: hola
        Tipo: Identifier
        Linea: 1
Valor: Char con mas de 1 elemento: ''
        Tipo: Error
        Linea: 1
BUILD SUCCESSFUL (total time: 0 seconds)
```

- El char, aunque está vacío, el scanner dice que tiene más de un elemento, lo cual es incorrecto. Se procede a solucionar el error:

```
Valor: Char con mas o menos de 1 elemento: ''
        Tipo: Error
        Linea: 13
```

- Se prueba el siguiente código:

```
123e10 hola '\n'
```

  - Resultado esperado: Separación en tres tokens: Integer, identificador y char.
  - Con el resultado:

```
Valor: Char con mas de 1 elemento: '\n'
        Tipo: Error
        Linea: 1
```

  - Los char no están aceptando las secuencias de escape o saltos de linea compuestos por un backslash (\n, \s...). Se procede a solucionar el problema, resultando:

```
Valor: '\n'
        Tipo: Literal Char
        Linea: 1
```