Proyecto: Entrega 1

Compiladores e Intérpretes

Germán Alejandro Gómez

28/08/2018

INDICE

Instrucciones para ejecutar.

Para poder utilizar el programa hace falta seguir las siguientes instrucciones:

1. Abrir una consola en Windows, por ejemplo cmd.
2. Usando cmd, ir al directorio donde se encuentra Principal.java.
3. Utilizar el comando 'javac Principal.java' para compilar el código.
4. Para ejecutarlo hay que ingresar la siguiente sentencia: 'java Principal <IN_FILE> [<OUT_FILE>]'. IN_FILE es el archivo a compilar, y OUT_FILE, es una alternativa para mostrar el resultado en un archivo. Aclaración: cualquier error se mostrará por consola, por lo cual el archivo solo es para cadenas de Tokens válidas.

Información del Analizador léxico

Alfabeto

El alfabeto reconocido por el analizador léxico, es el código ASCII exceptuando al 0, por lo cual son validos los caracteres entre 1 a 255 inclusive.

Tokens

Palabras claves

- ‘p_class’: class
- ‘p_string’: string
- ‘p_public’: public
- ‘p_if’: if
- ‘p_this’: this
- ‘p_extends’: extends
- ‘p_boolean’: boolean
- ‘p_private’: prívate
- ‘p_else’: else
- ‘p_new’: new
- ‘p_static’: static
- ‘p_char’: char
- ‘p_void’: void
- ‘p_while’: while
- ‘p_true’: true
- ‘p_false’: false
- ‘p_dynamic’: dynamic

- ‘p_int’: int
- ‘p_null’: null
- ‘p_return’: return

Expresión regular = class | String | public | if |this | extends |boolean |private | else | new | static | dynamic | void | while |true |int | null | return | false.

Identificadores:

- IdClase: identificador de clase. Es una letra mayúscula seguida de cero o más letras (mayúsculas o minúsculas), dígitos y underscores. Expresión regular = [a..z]([a..z] | [A..Z] | [0..9] | ‘_’)*
- IdMetVar: identificador de método y variable. Es una letra minúscula seguida de cero o más letras (mayúsculas o minúsculas), dígitos y underscores. Expresión regular = [A..Z]([a..z] | [A..Z] | [0..9] | ‘_’)*

Comentarios

- '/* comentario */’: Comentarios multi-línea. Todos los caracteres desde /* hasta */ son ignorados. No son Tokens necesarios para el analizador, una vez que se reconoce uno de ellos, no lo guarda. Expresión regular = /*(carac_ascii)**/
- ‘// comentario’: Comentario simple. Todos los caracteres de desde // hasta el final de la línea son ignorados. No son Tokens necesarios para el analizador, una vez que se reconoce uno de ellos, no lo guarda. Expresión regular = //(carac_ascii)*’/n’

Literales enteros

- ‘entero’: secuencia de uno o más dígitos. El valor de un literal entero corresponde a su interpretación estándar en base 10. Expresión regular = $([0..9])^{+}$

Literales caracteres:

- ‘caracter’: es un carácter encerrado entre comillas simples. Por ejemplo: ‘a’, ‘/n’, ‘/t’. Expresión regular = ‘(carac_ascii | (/carac_ascii))’

Strings:

- ‘string’: Un literal string se representa mediante una comilla doble (") seguida de una secuencia de caracteres y finaliza con otra comilla doble ("). El valor del literal corresponde a la cadena de caracteres entre las comillas. Se permite el uso de ‘/n’ para salto de línea y ‘/t’ para tab. Expresión regular = “(carac_ascii) $^{+}$ “

Puntuación:

- ‘pa_a’: paréntesis que abre (‘(’).
- ‘pa_c’: paréntesis que cierra (‘)’).
- ‘ll_a’: llave que abre (‘{’).
- ‘ll_c’: llave que cierra (‘}’).
- ‘punto_coma’: punto y coma (‘;’).
- ‘punto’: punto (‘.’).
- ‘coma’: coma (‘,’).
- ‘co_a’: corchete que abre (‘[’).

- ‘co_c’: corchete que cierra (‘]’).

Expresión regular = (|) | { | } | ; | . | , | [|]

Operadores:

- ‘op_mayor’: mayor (‘>’).
- ‘op_menor’ : menor (‘<’).
- ‘op_neg’: negación (‘!’).
- ‘op_igual’: igualdad(‘==’).
- ‘op_mai’: mayor o igual (‘>=’).
- ‘op_mei’: menor o igual (‘<=’).
- ‘op_dis’: distinto (‘!=’).
- ‘op_mas’: más (‘+’).
- ‘op_menos’: menos (‘-’).
- ‘op_mult’: multiplicación (‘*’).
- ‘op_div’: división (‘/’).
- ‘op_and’: and (‘&&’).
- ‘op_or’: or (‘||’).
- ‘asignacion’: asignación (‘=’).

Expresión regular = > | < | ¡ | == | >= | <= | ¡= | + | - | * | / | && | || | =

Errores:

- CharAscii: se ingreso un carácter que no pertenece al alfabeto.
- CharFormat: el formato del char no es válido.
- CharInesperado: se encuentra al empezar a construir un token, un char que no pertenece al alfabeto.
- Comentario abierto: comentario multi-linea abierto.
- IntFormato: se encuetra un carácter invalido después de un numero.

- OperadorAnd: se encuentra solo un & en vez de &&.
- OperadorOR: se encuentra solo un | en vez de ||.
- StringAbierto: se encuentra un string sin cerrar.

Autómata finito

Se adjunta una imagen con el autómata finito q se uso de guía.

Casos de prueba

Se adjunta una carpeta con el nombre test. Cada archivo corresponde a un caso de prueba, y el resultado esperado, se especifica en forma de comentario dentro de cada uno.

Consideraciones de diseño

La cátedra propuso dos formas de realizar el Analizador Léxico: subprogramas o switch case. Dado la eficiencia que tiene switch case sobre subprogramas, se decidió usar switch con la mayor cantidad de comentarios posibles para mejorar la legibilidad.

El programa elimina las comillas dobles que contiene a un String, por ejemplo: “hola” se guarda como hola. De forma contraria, se decidió guarda las comillas simples en los caracteres, por lo cual ‘a’ se guarda como ‘a’.

En la clase Archivo se utiliza la función read() que nos facilita la clase BufferedReader. Si bien la documentación especifica que cuando no hay más caracteres devuelve el entero -1, en muchas ocasiones devuelve 65535. Se tomo la convención que en el caso que se reciba -1 o 65535, archivo devolverá un 0 como próximo carácter y el analizador léxico lo interpretará con fin de archivo.

Otro dato interesante es que en la clase Archivo se había utilizado la función readLine()que facilitaba BufferedReader, pero se cambio por read() ya que es más eficiente y se lee menos veces el archivo de entrada a compilar.

Al momento de hacer el logro de Strings multi-linea, también se generó la posibilidad de agregar tab ingresando ‘\t’.

Se agrego el comando –h para pedir ayuda.

Clases utilizadas

Se utilizaron 13 clases en total, las cuales se pueden dividir en 2 grandes grupos: analizador léxico y excepciones.

La primera corresponde a:

- Analizador Léxico: El cual recibe la ubicación de un archivo y mediante la función getNextToken(), se le puede pedir Token que se encuentran en el archivo.
- Archivo: recibe la ubicación de un archivo y mediante la función getNext() se obtiene el próximo carácter del archivo.
- Token: tiene la estructura general de un Token valido (tipo, lexema, columna y fila).
- Principal: obtiene todos los Token de Archivo usando el analizador léxico y lo imprime por pantalla o en un archivo.

La segunda corresponde a las excepciones que pueden ocurrir al momento de querer usar el Analizador Léxico para encontrar los Tokens.

Logros

Se intentan cumplir los logros de:

- Entrega anticipada.
- Columnas.
- String Multi-linea.