

Práctica 2

Pulido Reséndiz Aarón Isai

Lenguajes y Autómatas II

Introducción

En esta práctica se verán y aplicarán conceptos vistos anteriormente en la materia. Se deberá de desarrollar un pequeño programa que sea capaz de identificar de manera clara los diferentes componentes del lenguaje *Swift 5.3*; basándonos en la documentación que se encuentra en la página:

<https://docs.swift.org/swift-book/ReferenceManual/LexicalStructure.html>

Así mismo, se deberán de efectuar varias pruebas para verificar la funcionalidad del programa.

Desarrollo

Para explicar esta práctica se pidió que se realizara una captura de pantalla del programa junto con comentarios de las diferentes partes del programa (estas partes se explicarán en breve), la cual es la siguiente:

```
options { Ignore_Case = false; } //Apartado option
PARSER_BEGIN (Practica2) //Inicio del Parser
public class Practica2 {
    public static void main (String[] argum) throws ParseException {
        Practica2 anLexSint = new Practica2(System.in);
        anLexSint.unaExpresion();
        //Unidad de procesamiento
        System.out.println("Análisis terminado:");
    }
}
PARSER_END (Practica2) //Fin del Parser
```

Figura 1. Captura de pantalla de la parte principal del programa de la práctica 2.

En esta sección se encuentra el *main* y las instrucciones de nuestro programa; pero debido a que esto ya se explicó en la anterior práctica, no se profundizará de nuevo en el tema.

En la siguiente sección se declararon los tokens del programa. En la sección de comentarios se tenía anteriormente los caracteres de espacio en blanco, salto de line, retorno de carro y distintas combinaciones de estos, pero debido a que esto llenaría de mucho texto repetido durante las pruebas, se decidió omitir; además de que no era necesario agrupar dichos caracteres en un solo token.

```
TOKEN:
{
    //ESPACIOS

    //COMENTARIOS
    <COMENT: "//" (~["\n", "\r"])* ("\"|\"\\n|\"\\r|\"\\n\\r") | "/" (~["/", "*"])* "*" /> |
    <LETRA: ["A"-"Z"]> |

    //IDENTIFICADORES
    <IDENTIFI: (<IDHEAD><IDCHA> | ("")( <IDHEAD><IDCHA> ("") | ("$( <DECI> ) | ("$( <IDCHA> |
    ("")( <PALABRAR> ) ) )> |
    <IDHEAD: (<LETRA> | "_" | "\u00a8" | "\u00aa" | "\u00ad" | "\u00af" | ["\u00b2"-" \u00b5"] |
    ["\u00b7"-" \u00ba" | ["\u00bc"-" \u00ce" | ["\u00c0"-" \u00d6" | ["\u00d8"-" \u00f6" |
    ["\u00f8"-" \u00ff" | ["\u0100"-" \u02ff" | ["\u0370"-" \u167f" | ["\u1681"-" \u180d" |
    ["\u180f"-" \u1dbf" | ["\u1e00"-" \u1fff" | ["\u200b"-" \u200d" | ["\u202a"-" \u202e" |
    ["\u203f"-" \u2040" | "\u2054" | ["\u2060"-" \u206f" | ["\u2070"-" \u20cf" | ["\u2100"-"
    "\u218f" | ["\u2460"-" \u24ff" | ["\u2776"-" \u2793" | ["\u2c00"-" \u2dff" | ["\u2e80"-"
    "\u2fff" | ["\u3004"-" \u3007" | ["\u3021"-" \u302f" | ["\u3031"-" \u303f" | ["\u3040"-"
    "\ud7ff" | ["\uf900"-" \ufd3d" | ["\ufd40"-" \ufdcf" | ["\ufdf0"-" \ufelf" | ["\ufe30"-"
    "\ufe44" | ["\ufe47"-" \ufffd"]> |
    <IDCHA: ((["0"-"9"])+ | ["\u0300"-" \u036f" | ["\u1dc0"-" \u1dff" | ["\u20d0"-" \u20ff" |
    ["\ufe20"-" \ufe2f"]> |
```

Figura 2. Inicio de los tokens. Tokens de comentario e identificadores.

En el proceso de desarrollo del programa se tuvo un problema en específico. Este era que, al momento de compilar todas las clases java, el CMD arrojaba una leyenda que decía que el programa generado era demasiado largo como para trabajarlo dentro de Java:

```
$~
Exception in thread "main" TokenMgrError: Lexical error at line 7, column 2. Encountered: "~" (126), after : "$"
    at Practica2TokenManager.getNextToken(Practica2TokenManager.java:6628)
    at Practica2.jj_ntk(Practica2.java:445)
    at Practica2.unaExpresion(Practica2.java:242)
    at Practica2.main(Practica2.java:6)

C:\Clases\Lenguajes II\Proyectos\Practica2>java Practica2
$~
Exception in thread "main" TokenMgrError: Lexical error at line 1, column 2. Encountered: "?" (63), after : "$"
    at Practica2TokenManager.getNextToken(Practica2TokenManager.java:6628)
    at Practica2.jj_ntk(Practica2.java:445)
    at Practica2.unaExpresion(Practica2.java:13)
    at Practica2.main(Practica2.java:6)
```

Figura 3. Captura de pantalla del error.

Debido a este problema, y al no encontrar una solución en páginas o blogs de ayuda sobre este tema, se determinó que la mejor opción era recortar una parte “prescindible” del programa, lo cual se mostrara a continuación:

```
//PALABRAS RESERVADAS
<PALABRAR: (<KEYD> | <KEYST> | <KEYET>)> |
<KEYD: "associatedtype" | "class" | "deinit" | "enum" | "extension" | "fileprivate" |
"func" | "import" | "init" | "inout" | "internal" | "let" | "open" | "operator" | "private"
| "protocol" | "public" | "rethrows" | "static" | "struct" | "subscript" | " typealias" |
"var"> |
<KEYST: ("break" | "case" | "continue" | "default" | "defer" | "do" | "else" |
"fallthrough" | "for" | "guard" | "if" | "in" | "repeat" | "return" | "switch" | "where" |
"while")> |
<KEYET: ("as" | "Any" | "catch" | "is" | "super" | "self" | "Self" | "throw" | "throws" |
"try")> |

//PALABRAS RESERVADAS NO UTILIZADAS
//<KEYP: (" ")> |
//<KEYNS: ("#available" | "#colorLiteral" | "#column" | "#else" | "#elseif" | "#endif" |
"#error" | "#file" | "#fileID" | "#fileLiteral" | "#filePath" | "#function" | "#if" |
"#imageLiteral" | "#line" | "#selector" | "#sourceLocation" | "#warning")> |
//<KEYRC: ("associativity" | "convenience" | "dynamic" | "didSet" | "final" | "get" |
"infix" | "indirect" | "lazy" | "left" | "mutating" | "mutating" | "nonmutating" |
"optional" | "override" | "postfix" | "precedence" | "prefix" | "Protocol" | "required" |
"right" | "set" | "Type" | "unowned" | "weak" | "willSet")> |
```

Figura 4. Captura de pantalla de las palabras reservadas que fueron utilizadas para el funcionamiento del programa y las que no se utilizaron.

Se tomo esta decisión ya que se consideró que las palabras que ya se habían declarado en el programa eran más que suficientes para demostrar su correcto funcionamiento y lo aprendido durante clases pasadas.

Después, se tienen los caracteres y cadenas que tienen que ver con números (binarios,

octales, decimales, hexadecimales y flotantes). Solo se desea hacer dos sencillas aclaraciones en cuanto a esta sección:

1. Los decimales que se ingresan dentro del programa siempre los toma como identificadores, ya que estos comparten la misma estructura.
2. Los números pueden ir con un signo negativo (-), pero si se usaba el carácter para indicar opcionalidad (?) siempre se reconocían como <LITERAL>, por lo cual se decidió no hacerlo opcional. Si no se hace opcional el signo negativo, se logra que los distintos números se puedan identificar, y si se hace uso del signo negativo se pueden identificar como <LITERAL>.

```
//NÚMEROS
<BINL: ("0b"((("0" | "1")+((("_")?("0" | "1")+)*)+)> |
<OCTL: ("0o"((("[0"- "7"])+((("_")?("[0"- "7"])+)*)+)> |
<DECI: ("0"- "9"])> |
<HEXL: ("0d"((("[0"- "9", "A"- "F"])+((("_")?("[0"- "9", "A"- "F"])+)*)+)> |

//NUML
<NUML: ((("-")<BINL> | ("")<OCTL> | ("")<DECI> | ("")<HEXL> | ("")<FLOL>)> |
<BOOL: ("true" | "false")> |

//FLOTANTES
<FLOL: (<DECI>(<DECF>)(<DECE>)?)> |
<DECF: ("."(<DECI>)+)> |
<DECE: (<FLOE>((<DECI>)+)?)> |
<HEXF: ("."(<HEXL>)+)> |
<HEXE: (<FLOP>((<HEXL>)+)?)> |
<FLOE: ("e" | "E")> |
<FLOP: ("p" | "P")> |
<SIG: ("+" | "-")> |
```

Figura 5. Tokens numéricos.

A continuación, se muestran los tokens de cadenas literales. En este caso, se comentó durante la clase que el CMD de Windows o Java (una disculpa por no recordarlo) no era capaz de diferenciar los caracteres de doble comilla (") y de diagonal invertida (\), lo cual imposibilitaba que se pudieran reconocer esta clase de cadenas. Pero pudo lograr un funcionamiento bastante parecido utilizando dos apostrofes juntas (') para emular la doble comilla. También cabe mencionar que se omitió un token llamado *expression* ya que este tipo de cadenas o caracteres redirigían a otra documentación más extensa, pero que eran parte de este tipo de tokens.

```
//CADENAS LITERALES
<STRIL: (<STASTRL> | <INTSL>)>> |
<STRILO: ((<EXSTRD>)?"'")> |
<STRILC: ("'"(<EXSTRD>)?)> |
<STASTRL: (<STRILO>(<QUOTE>)??<STRILC> | <MULSTLO>(<MQQUOTE>)??<MULSTLC>)> |
<MULSTLO: (<EXSTRD>"'""'""'""'")> |
<MULSTLC: ("'""'""'""'""'#")> |
<EXSTRD: ("#"("'#")+)?> |
<QUOTE: (<QUOTEI>)+> |
<QUOTEI: (<ESCC>)> |
<MQQUOTE: (<MQQUOTEI>)+> |
<MQQUOTEI: (<ESCC>)> |
<INTSL: (<STRILO>(<INTEX>)??<STRILC> | <MULSTLO>(<MINTEX>)??<MULSTLC>)>> |
<INTEX: (<QUOTEI>)+> |
<MINTEX: (<MQQUOTEI>)+> |
<ESCS: (<EXSTRD>)> |
<ESCC: (<ESCS>"0" | <ESCS>"\r" | <ESCS>"t" | <ESCS>"n" | <ESCS>"r" | <ESCS>"'" | <ESCS>"'"
| <ESCS>"u{"<UNICO>"}")> |
<UNICO: (((["0"- "9", "A"- "F"]){4,8})>> |
```

Figura 6. Tokens de cadenas literales.

En seguida se encuentran los operadores y los signos de puntuación.

```
//OPERADORES
<OPERADOR: (<OPERHEAD>(<OPERCH>)? | "."<DOTOPERC>)> |
<OPERHEAD: ("/" | "-" | "=" | "+" | "!" | "*" | "%" | "<" | ">" | "&" | "|" | "&" | "~" |
"?" | ["\u00a1"-" \u00a7"] | ["\u00a9" | "\u00ab" | "\u00ac" | ["\u00ae" | ["\u00b0"-" \u00b1" |
"\u00b6" | "\u00bb" | "\u00bf" | "\u00d7" | "\u00f7" | ["\u0016"-" \u0017" | ["\u0020"-"
\u0027" | ["\u0030"-" \u003e" | ["\u0041"-" \u0053" | ["\u0055"-" \u005e" | ["\u00190"-"
\u0023ff" | ["\u002500"-" \u002775" | ["\u002794"-" \u002bff" | ["\u002e00"-" \u002e7f" | ["\u003001"-"
\u003003" | ["\u003008"-" \u003020" | "\u003030"]> |
<OPERCH: (<OPERHEAD> | ["\u003000"-" \u00306f" | ["\u001dc0"-" \u001dff" | ["\u0020d0"-" \u0020ff" |
["\ufe00"-" \ufe0f" | ["\ufe20"-" \ufe2f"]> |
<DOTOPERC: ( "." | <OPERCH>)+ |

<PUNT: ("(" | ")" | "{" | "}" | "[" | "]" | "." | "," | ":" | ";" | "=" | "@" | "#" | "&" |
"->" | "!" | "?" | "!">
```

Figura 7. Tokens de signos de puntuación y operadores.

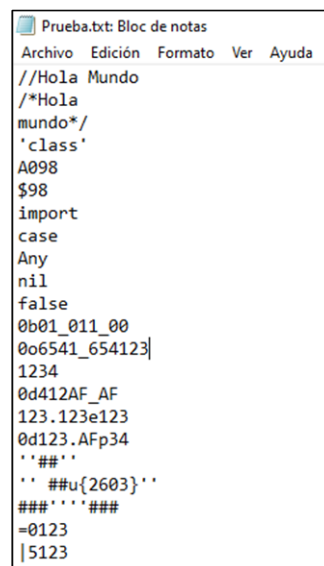
Y, por último, dentro del programa, se tiene el método que contiene todos los tokens que puede recibir el programa. Para esta parte se decidió agrupar los tokens en varios grupos para poder identificarlos de manera más fácil a la hora de realizar la prueba.

```
SKIP: { " " | "\t" | "\n" | "\r" }

void unaExpresion() : {}
{
    (<COMENT> {System.out.println("<COMENTARIO>");} | <DECI> {System.out.println("<DECIMAL>");} |
    | <LETRA> | (<IDHEAD> | <IDCHA> | <IDENTIFI>) {System.out.println("<IDENTIFICADOR>");} |
    | <KEYD> | <KEYST> | <KEYET> | <PALABRAR>) {System.out.println("<PALABRA RESERVADA>");} |
    <PUNT> {System.out.println("<PUNTUACION>");} | <BINL> {System.out.println("<BINARIO>");} |
    <OCTL> {System.out.println("<OCTAL>");} | <HEXL> {System.out.println("<HEXADECIMAL>");} |
    | <FLOL> | <DECF> | <DECE> | <HEXF> | <HEXE>) {System.out.println("<LITERAL FLOTANTE>");} |
    <FLOE> | <FLOP> | <SIG> | (<NUML> | <BOOL> | "nil") {System.out.println("<LITERAL>");} |
    | <STRILO> | <STRILC> | <STASTRL> | <MULTSTLO> | <MULTSTLC> | <STRIL> | <EXSTRD> | <QUOTE> |
    <QUOTEI> | <MQQUOTE> | <MQQUOTEI> | <INTSL> | <INTEX> | <MINTEX> | <ESCS> | <ESCC>)
    {System.out.println("<CADENA LITERAL>");} | <UNICO> | (<OPERADOR> | <OPERHEAD> | <OPERCH> |
    <DOTOPERC>) {System.out.println("<OPERADOR>");}+<EOF>
}
```

Figura 8. Captura de pantalla del método del programa.

Para las pruebas se utilizó un .txt llamado Prueba.txt como método de entrada, en el cual se definieron distintos caracteres y cadenas:



```
Prueba.txt: Bloc de notas
Archivo Edición Formato Ver Ayuda
//Hola Mundo
/*Hola
mundo*/
'class'
A098
$98
import
case
Any
nil
false
0b01_011_00
0o6541_654123|
1234
0d412AF_AF
123.123e123
0d123.AFp34
'##'
' ##u{2603}'
###'###
=0123
|5123
```

Figura 9. Archivo de texto para las pruebas.

Para las pruebas, se procedió a compilar el programa y darle un archivo de entrada, el cual arrojó los siguientes resultados:

```
C:\Windows\System32\cmd.exe
Microsoft Windows [Versión 10.0.18362.1882]
(c) 2019 Microsoft Corporation. Todos los derechos reservados.

C:\Clases\Lenguajes II\Proyectos\Practica2>javacc Practica2.jj
Java Compiler Compiler Version 5.0 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file Practica2.jj . . .
Warning: Line 17, Column 5: Non-ASCII characters used in regular expression.
Please make sure you use the correct Reader when you create the parser, one that can handle your character set.
File "TokenMgrError.java" is being rebuilt.
File "ParseException.java" is being rebuilt.
File "Token.java" is being rebuilt.
File "SimpleCharStream.java" is being rebuilt.
Parser generated with 0 errors and 1 warnings.

C:\Clases\Lenguajes II\Proyectos\Practica2>javac *.java

C:\Clases\Lenguajes II\Proyectos\Practica2>java Practica2 < Prueba.txt
<COMENTARIO>
<COMENTARIO>
<IDENTIFICADOR>
<IDENTIFICADOR>
<IDENTIFICADOR>
<PALABRA RESERVADA>
<PALABRA RESERVADA>
<PALABRA RESERVADA>
<LITERAL>
<LITERAL>
<BINARIO>
<OCTAL>
<IDENTIFICADOR>
<HEXADECIMAL>
<IDENTIFICADOR>
<LITERAL FLOTANTE>
<LITERAL FLOTANTE>
<HEXADECIMAL>
<OPERADOR>
<LITERAL FLOTANTE>
<IDENTIFICADOR>
<CADENA LITERAL>
<CADENA LITERAL>
<CADENA LITERAL>
<CADENA LITERAL>
<CADENA LITERAL>
<CADENA LITERAL>
<OPERADOR>
<IDENTIFICADOR>
<OPERADOR>
<IDENTIFICADOR>
Análisis terminado:

C:\Clases\Lenguajes II\Proyectos\Practica2>
```

Figura 10. Resultados de la prueba.

Conclusiones

Durante el desarrollo de la práctica se pusieron a prueba los distintos conocimientos que fueron adquiridos con las clases y prácticas que hemos tenido. La programación de esta práctica fue bastante similar a la de la practica 1, pero más extensa y con muchas más variables a tomar en cuenta.

Fue interesante el aprender cómo se pueden utilizar y referenciar los distintos caracteres que hay pro medio de códigos Unicode y como no es necesario agregarlos de manera individual, teniendo la posibilidad de declararlos como rangos; algo que ahorro mucho tiempo en esta práctica. A pesar de haber tenido algunos inconvenientes en el desarrollo de la práctica, se pudieron solventar de manera óptima sin comprometer el correcto funcionamiento del programa.

Como conclusión final, esta práctica fue un tanto larga pero interesante para recordar ciertos conceptos y métodos de trabajo que se manejaban en otras materias de programación con Java.