EINA - Universidad de Zaragoza
Departamento de Informática e Ingeniería de Sistemas
3º Grado de Ingeniería Informática
30230 Procesadores de Lenguajes. **Curso 2018-2019**
Profesor: Javier Fabra – jfabra@unizar.es

==========

PRÁCTICA 2: Construcción de un analizador léxico para *MiniLeng*

==========

I. OBJETIVOS

En esta práctica trabajaremos primero el Token Manager de JavaCC. Después construiremos un analizador léxico de MiniLeng con el metacompilador.

II. El Token Manager

A continuación, se resume la estructura de una especificación escrita en JavaCC. Se trata de una versión simplificada. El generador JavaCC admite especificaciones con otras muchas posibilidades no mencionadas aquí.

Una especificación para el generador JavaCC puede considerarse dividida en cuatro secciones:

- **Sección de opciones**. En esta sección, cuya presencia es optativa, se pueden asignar valores a diversos parámetros (llamados opciones) que sirven para configurar ciertas características del funcionamiento del generador o del analizador generado. Cada parámetro (opción) tiene un valor por defecto, que es el que toma cuando no se le asigna explícitamente un valor. Los valores de las opciones también se pueden fijar en la línea de comandos cuando se ejecuta el generador (lo indicado en la línea de comandos tiene prioridad sobre lo especificado en esta sección de opciones).
- **Sección de ejecución**. En esta sección se pone el código Java que contiene la llamada al analizador generado para que se realice el análisis de un determinado texto de entrada. También se establece aquí el nombre de la especificación, que es el nombre que se toma para formar los nombres de parte de los ficheros (clases) generados.
- **Sección de sintaxis**. En esta sección se describe la sintaxis del lenguaje para el que se desea generar el analizador, usándose para ello una notación parecida a la BNF.
- **Sección de lexicografía**. En esta sección se indica el léxico del lenguaje para el que se va a generar el analizador. La notación de JavaCC que representa la forma de cada una de las piezas sintácticas es una variante de las *expresiones regulares*.

El contenido de esta parte se describe básicamente en el "***Token Manager MiniTutorial***" incluido en la documentación de JavaCC: https://javacc.org/tutorials/tokenmanager

Declaraciones léxicas

La sintaxis básica de una declaración léxica en JavaCC es la siguiente:

```
TIPO_DECLARACIÓN : {
      expresión_regular
    | expresión_regular
    | ...
}
```

JavaCC ofrece cuatro tipos de declaraciones léxicas:

- **TOKEN**: permite definir un conjunto de categorías léxicas (*tokens*) que serán devueltas al analizador sintáctico.
- **SKIP**: define un conjunto de categorías léxicas que serán filtradas por el analizador léxico, es decir, que no serán enviadas al analizador sintáctico.
- **SPECIAL_TOKEN**: define un conjunto de categorías léxicas que no serán enviadas directamente al analizador sintáctico, sino que se añaden en el campo *specialToken* de la siguiente categoría léxica a enviar.
- **MORE**: define un conjunto de expresiones regulares que no generan una categoría léxica, sino que son añadidas como prefijo en el lexema de la siguiente categoría léxica reconocida.

Expresiones regulares

JavaCC ofrece **cuatro formas** de describir una expresión regular:

```
expresión_regular ::= "cadena"
expresión_regular ::= < identificador : expresión_regular_compleja >
expresión_regular ::= < identificador >
expresión_regular ::= < EOF >
```

La primera forma permite definir patrones constantes, por ejemplo "`/*`" para el comienzo de un comentario o "`\n`" para indicar un salto de línea. Este tipo de expresiones no asigna un identificador a la expresión regular, por lo que se suele utilizar en declaraciones de tipo `SKIP` o `MORE`.

La segunda forma permite describir expresiones regulares complejas y asignarles un identificador. Este formato es el utilizado habitualmente en las declaraciones de tipo `TOKEN` y `SPECIAL_TOKEN`. Si el identificador comienza con el símbolo `#`, entonces la expresión regular no define una categoría léxica (un *token*), sino que se considera una definición auxiliar que puede ser utilizada en otras expresiones regulares.

La dos últimas formas de describir una expresión regular representan una referencia a otras expresiones regulares. La tercera forma representa una referencia a una expresión regular definida en otro punto, mientras que la última forma se refiere al símbolo especial de fin de entrada.

Las expresiones regulares complejas pueden contener cadenas de caracteres (por ejemplo, `import`), listas de caracteres entre corchetes (por ejemplo, `["a"-"z","A"-"Z","0"-"9"]`), referencias a otras expresiones regulares (por ejemplo, `<DIGITO>`), así como los operadores de clausura (por ejemplo, `(<DIGITO>)*`), clausura positiva (por ejemplo, `(<DIGITO>)+`), disyunciones (por ejemplo `("import"|"IMPORT")`) y opcionalidad (por ejemplo, `(<DIGITO>)?`).

Para representar a cualquier carácter excepto algunos se utiliza "~" (por ejemplo, `~["\\"]`, representa cualquier carácter que no sea una barra).

Veamos un ejemplo de especificación léxica:

```
TOKEN:  {
      <MAS: "+">
  |   <MENOS: "-">
  |   <POR: "*">
  |   <DIV: "/">
  |   <PARAB: "(">
  |   <PARCE: ")">
  |   <NUM: (["0"-"9"])+ ( "." (["0"-"9"])* )? (<EXPONENT>)? >
  |   < #EXPONENT: ["e","E"] (["+","-"])? (["0"-"9"])+ >
}

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

Acciones asociadas al reconocimiento de expresiones regulares

La sintaxis de la especificación léxica en JavaCC permite incluir código Java que será ejecutado al reconocer una determinada expresión regular. Este código se incluye entre llaves detrás de la definición de la expresión regular. Por ejemplo, con la siguiente declaración léxica se escribe un mensaje cada vez que se reconoce la palabra `class` en el flujo de entrada.

```
TOKEN : {
     <CLASS: "class">   { System.out.println("Palabra class reconocida"); }
  |  <INTERFACE: "interface">
}
```

Dentro del código Java que describe las acciones asociadas a las expresiones regulares se puede acceder a las siguientes variables y métodos del analizador léxico:

- `image(StringBuffer)`: contiene la cadena de caracteres reconocida por medio de la expresión regular. Se puede modificar, aunque generalmente no produce efectos sobre el lexema entregado al analizador sintáctico ya que éste ya ha sido creado en el momento de ejecutar la acción. Para modificar el lexema es necesario acceder a `matchedToken.image`.

- `lengthOfMatch(int)`: longitud de la cadena reconocida. No tiene en cuenta los caracteres que puedan ser añadidos por una declaración `MORE`.

- `curLexState(int)`: código del contexto léxico actual. Veremos los contextos léxicos más adelante.

- `inputStream(InputStream)`: flujo de datos de entrada del analizador. En el momento de ejecutar la acción se encuentra posicionado en el último carácter reconocido, de manera que el método `read()` nos devuelve el siguiente carácter.

- `matchedToken(Token)`: categoría léxica que se va a devolver al analizador sintáctico, es decir, valor que va a devolver el método `getNextToken()` del analizador léxico.

- `void SwitchTo(int código)`: modifica el contexto léxico actual.

Código auxiliar incluido en el analizador léxico

La herramienta JavaCC permite definir un código que se introduce directamente en el analizador sintáctico. Este código es el que aparece entre las cláusulas `PARSER_BEGIN` y `PARSER_END`. También es posible definir código que se incorpora directamente en el analizador léxico, aunque en este caso el código no incluye la definición de la clase. Para ello se utiliza la siguiente expresión:

```
TOKEN_MGR_DECLS : {
     código_JAVA
}
```

Este código permite definir variables de instancia y métodos que pueden ser utilizados en las acciones asociadas a las expresiones regulares.

Contextos léxicos

Los contextos léxicos (o *lexical states* en la documentación de la herramienta) permiten simplificar los autómatas generados por JavaCC. Cada contexto léxico define sus propias expresiones regulares, de manera que al modificar el contexto del analizador se modifica su comportamiento.

Inicialmente el analizador léxico se encuentra en el contexto `DEFAULT`. Para definir las declaraciones léxicas de un cierto contexto, se incluye el identificador del contexto antes del tipo de declaración (entre signos < y >). Para realizar un cambio de contexto después de reconocer una expresión regular, se incluye el identificador del contexto detrás del a descripción de la expresión (separado por el signo ":"). El siguiente ejemplo muestra la descripción de los comentarios de C por medio de contextos.

```
SKIP : {
   "/*" : COMMENT
}

<COMMENT> SKIP : {
  "*/" : DEFAULT
}

<COMMENT> MORE : {
   <~[]>
}
```

Las declaraciones léxicas que no incluyen un identificador de contexto pertenecen al contexto `DEFAULT`. Es posible definir declaraciones léxicas que pertenezcan a varios contextos. Para hacer esto se introduce entre los signos "<" y ">" la lista de identificadores de contexto a los que pertenece la declaración.

Veamos un ejemplo:

```
<DEFAULT,COMMENT> TOKEN : { ... }
```

Para indicar que una declaración pertenece a todos los contextos se utiliza <*>

III. UN EJEMPLO SENCILLO

Vamos a ver un ejemplo de implementación con JavaCC. El lenguaje *L* está formado por las expresiones en las que pueden aparecer:

- Variables
- Constantes
- Operadores + y *

Las variables son nombres formados por una única letra (minúscula o mayúscula); las constantes son números enteros de una o más cifras. El espacio y el tabulador pueden estar presentes, pero no tienen ningún significado. Los finales de línea tampoco son significativos (una expresión puede codificarse ocupando una o más líneas).

La sintaxis de las expresiones se especifica mediante la siguiente gramática en BNF:

```
<Expresión> ::= <Término> { + <Término> }
<Término> ::= <Factor> { * <Factor> }
<Factor> ::= variable
   |  constante
   |  ( <Expresión> )
```

NOTA: Observad cómo hemos expresado la **precedencia y asociatividad** del operador * de forma explícita.

Especificación para el análisis léxico y sintáctico

Una manera de escribir la especificación (para la que se ha elegido el nombre *ExprMin*), de forma que sea aceptada por el generador JavaCC, es la siguiente:

```
/**********************************
 * Sección de opciones
 **********************************/
options {
   Ignore_Case = true;
}

/**********************************
 * Sección de ejecución
 **********************************/
PARSER_BEGIN (ExprMin)
   public class ExprMin {
      public static void main (String[] argum) throws ParseException {
         ExprMin anLexSint = new ExprMin (System.in);
         anLexSint.unaExpresion();
         System.out.println("Análisis terminado:");
         System.out.println("No se han encontrado errores.");
      }
   }
PARSER_END (ExprMin)
```

```
/**********************************
 * Sección de sintaxis
 **********************************/
void unaExpresion() :
   { }
   {
      expresion() <EOF>
   }

void expresion() :
   { }
   {
      termino() ( "+"  termino() )*
   }

void termino() :
   { }
   {
      factor() ( "*"  factor() )*
   }

void factor() :
   { }
   {
      <constante>
      | <variable>
      | "(" expresion() ")"
   }

/**********************************
 * Sección de patrones y tokens
 **********************************/
TOKEN:
   {
      <  variable : ["a"-"z"]  >
   }
TOKEN:
   {
      <  constante :  ( ["0"-"9"] ) +  >
   }
SKIP:
   { " " | "\t" | "\n" | "\r" }
```

Obtención del analizador

Si la especificación precedente se tiene grabada en un fichero de nombre `Ejemplo.jj`, para obtener el analizador debemos seguir estos pasos:

- ejecutamos el generador:

```
$ javacc Ejemplo.jj
```

- compilamos el analizador generado (JavaCC es un *metacompilador*):

```
$ javac ExprMin.java
```

Ejecución del analizador

Si se quiere analizar una expresión grabada en un fichero de nombre `PruebaExp.txt`, no tenemos más que ejecutar el analizador obtenido:

```
$ java ExprMin < PruebaExp.txt
```

Opcionalmente, podríamos implementar las instancias necesarias para abrir el fichero desde el propio código (podéis cablear el fichero).

IV. DESARROLLO DE UN ANALIZADOR LÉXICO Y SINTÁCTICO PARA EL LENGUAJE *MiniLeng*

En esta práctica deberás completar una serie de programas que implementan un analizador léxico ***MiniLeng***. MiniLeng es un lenguaje estructurado sencillo, con algunas características similares a Pascal, y otras a C, entre las cuales destacan:

- Se permiten comentarios de una línea, comenzados por % y terminados por el fin de línea.
- Todo programa debe contener, al menos, una instrucción.
- Se permite la declaración y uso de variables simples, tanto globales como locales, de tres tipos: **entero**, **carácter** y **booleano**. La declaración de variables sigue la sintaxis de C, y permite más de una variable en cada declaración:

```
entero i, j, k;
caracter c, d, e;
booleano v, f;
entero z;
```

- Ningún símbolo puede llamarse como las palabras reservadas. Es decir, no se permite declarar una variable o una función de nombre 'entero', por ejemplo.
- Los identificadores estarán compuestos por letras, números y el símbolo "_", con las siguientes restricciones:
  - Un identificador nunca puede comenzar por un número.
  - Un identificador nunca puede terminar en "_".
- El lenguaje **no es *case-sensitive*** (es decir, no distingue mayúsculas de minúsculas).
- Se permite el uso de cadenas de caracteres constantes, aunque solamente para escritura:

```
escribir("Escribe un numero: ");
```

NOTA: Las cadenas y los caracteres simples deben ir siempre entre comillas **dobles**, no se admite la utilización de comillas simples.

- Se permite la declaración y uso de acciones (procedimientos) anidadas con paso de parámetros simples tanto por valor como por referencia, con el significado habitual de estos términos. En el caso de paso de parámetros por valor, se precederá la signatura del parámetro con la palabra reservada *val*, mientras que en el paso por referencia se utilizará la palabra *ref*. Es obligatorio especificar si el parámetro se pasa por valor o por referencia.

```
accion procesar( val entero i, j, k; ref caracter c, d, e;
      val booleano f; val vector[3] de entero v);
```

- Se permite la escritura de variables simples. Como salida, la operación de escritura mostrará por pantalla el valor entero, y las cadenas “Verdadero” o “Falso” en el caso de booleanos (“Verdadero” si el valor del booleano es 1, “Falso” si es 0).

Este es un pequeño ejemplo de un programa escrito en MiniLeng:

```
Programa p;
   entero i, j, m;

%-----------------------------------------------------------
accion dato (ref entero d);
%-----------------------------------------------------------
Principio
   d:=0;
   Mq d <= 0
      escribir("Escribe un numero: ");
      leer(d);
      Si d <= 0 ent
         escribir("El numero debe ser positivo.");
         escribir (entacar(13), entacar(10));
      FSi
   FMq
Fin

%-----------------------------------------------------------
accion mcd(val entero a,b; ref entero m);
%-----------------------------------------------------------
   entero i, j, k;
Principio
   j := a;
   k := b;
   i := a mod b;

   Mq i <> 0
      j := k;
      k := i;
      i := j mod k;
   FMq
   m := i;
Fin

Principio
   dato(i);
   dato(j);
   mcd(i,j,m);
   escribir("El MCD es: ", m, entacar(13), entacar(10));
Fin
```

V. RESULTADOS

Debes implementar un analizador léxico que reconozca las palabras reservadas, identificadores, valores constantes (cadenas, caracteres, booleanos y números) e identificadores del lenguaje MiniLeng.

Como resultado, debes generar un único fichero *minilengcompiler.jj* que contenga las definiciones de patrones para el reconocimiento del lenguaje y que, al compilarlo, genere un analizador **que admita un fichero por línea de comandos** (es decir, que se especifique el nombre del fichero como argumento al invocar el compilador) y como salida muestre la siguiente información:

- Si ha habido un error, mostrará la línea y columna, junto con el error léxico localizado:

  `"ERROR LÉXICO (<línea, columna>): símbolo no reconocido: <símbolo>"`

- Si no hay errores léxicos, no mostrará nada como salida.
- Si se le pasa como argumento el *flag* "-v", mostrará al terminar una tabla con un resumen del número de veces que ha aparecido cada palabra reservada del lenguaje, así como el número de identificadores, de constantes enteras, de booleanos, etc. Se deja a tu elección el formato y el detalle de la información a mostrar.

Se os proporciona una serie de baterías de programas de prueba (extensión *.ml*), para que probéis el correcto funcionamiento del analizador léxico que habrás realizado.

Además, deberás generar **cinco** programas de prueba (con extensión .ml) para comprobar el correcto funcionamiento de la práctica (es decir, que generen errores léxicos, principalmente). No es necesario que sean muy complejos, pero sí que demuestren la corrección de la práctica de forma clara y concisa.

NOTA: Recuerda que en esta práctica sólo hay que hacer un análisis léxico, no importa si la estructura del programa es correcta o no. De eso se encargará el analizador sintáctico en la próxima práctica.

NOTA: De los programas de prueba que se os proporcionan, todos son correctos desde el punto de vista del analizador léxico. Sin embargo, **sintácticamente y semánticamente** el programa "mcd.ml" debería generar varios errores, aunque este es un tema que abordaremos más adelante (en esta práctica, el objetivo es el analizador léxico). Los demás programas son correctos desde la perspectiva del análisis tanto sintáctico como semántico.