EINA - Universidad de Zaragoza
Departamento de Informática e Ingeniería de Sistemas
3º Grado de Ingeniería Informática
30230 Procesadores de Lenguajes. **Curso 2019-2020**
Profesor: Javier Fabra – jfabra@unizar.es

==========

PRÁCTICA 3: Construcción de un analizador sintáctico para MiniLeng

==========

I. OBJETIVOS

En esta práctica vas a construir un analizador sintáctico de MiniLeng con JavaCC.

II. CONTENIDO

Para realizar esta práctica debes haber completado la Práctica 1 y la Práctica 2. Debes considerar los programas de ejemplo y los detalles de MiniLeng que se te proporcionaron en la práctica anterior para definir la gramática del lenguaje.

En el enunciado de la Práctica 2 se introdujo cómo se implementa un analizador sintáctico en JavaCC para un ejemplo sencillo (ver la sección titulada “Un ejemplo sencillo” en dicho enunciado). Nos basaremos en ese ejemplo para construir el analizador sintáctico de MiniLeng.

En primer lugar, para implementar el analizador, necesitaremos la gramática del lenguaje en formato BNF. A continuación se te proporciona un extracto de la gramática de MiniLeng en formato BNF:

```
Programa ::= <tPROGRAMA> <tIDENTIFICADOR> ";"
     declaracion_variables declaracion_acciones
     bloque_sentencias
declaracion_variables ::= ( declaracion ";" )*
declaracion ::= tipo_variables identificadores
tipo_variables ::= <tENTERO> | <tCARACTER> | <tBOOLEANO>
identificadores ::= <tIDENTIFICADOR> ( "," <tIDENTIFICADOR> )*
declaracion_acciones ::= ( declaracion_accion )*
declaracion_accion ::= cabecera_accion ";"
     declaracion_variables declaracion_acciones bloque_sentencias
cabecera_accion ::= <tACCION> <tIDENTIFICADOR> parametros_formales
parametros_formales ::= ...
lista_parametros      ::= ...
parametros ::= clase_parametros tipo_variables lista_parametros
clase_parametros ::= <tVAL> | <tREF>
bloque_sentencias ::= <tPRINCIPIO> lista_sentencias <tFIN>
lista_sentencias      ::= ...
sentencia ::= leer ";"
     | escribir ";"
     | asignacion
     | invocacion_accion
     | seleccion
     | mientras_que
```

```
leer ::= <tLEER> "(" lista_asignables ")"
lista_asignables ::= ...
escribir ::= <tESCRIBIR> "(" lista_escribibles ")"
lista_escribibles ::= ...
asignación ::= <tIDENTIFICADOR> <tOPAS> expresion ";"
invocacion_accion ::= <tIDENTIFICADOR> argumentos ";"
mientras_que ::= <tMQ> expresion lista_sentencias <tFMQ>
seleccion ::= ...
argumentos ::= "(" ( lista_expresiones )? ")"
lista_expresiones ::= ...
expresion ::= ...
operador_relacional ::= ...
expresion_simple ::= ...
operador_aditivo ::= ...
termino ::= ...
operador_multiplicativo ::= ...
factor ::= "-" factor
      | <tNOT> factor
      | "(" expresion ")"
      | <tENTACAR> "(" expresion ")"
      | <tCARAENT> "(" expresion ")"
      | <tIDENTIFICADOR>
      | <tCONSTENTERA>
      | <tCONSTCHAR>
      | <tCONSTCAD>
      | <tTRUE>
      | <tFALSE>
```

Como verás, se han dejado varios huecos en blanco (con puntos suspensivos). Debes completar la gramática anterior, considerando que:

- Cuando una acción no tiene parámetros, pueden aparecer los paréntesis o no, es decir, cualquiera de las variantes es válida (tanto `()` como no poner nada). Esto se da tanto en su declaración como en su invocación.
- La acción leer **debe** recibir, al menos, una variable para que se pueda realizar la asignación.
- La acción escribir debe recibir, al menos, una expresión para que se pueda realizar la escritura.
- Hay que completar la gramática para que soporte las expresiones de la forma más común, utilizando los paréntesis para evitar conflictos si los operadores están a mismo nivel. A este respecto, los operadores multiplicativos `"*"`, `<tDIV>`, `<tMOD>` y `<tAND>` tienen preferencia sobre los operadores aditivos `"+"`, `"-"` y `<tOR>`.

Una vez definida la gramática en BNF, no hay más que portarla a JavaCC, escribiendo una función recursiva por cada no terminal. El proceso suele resultar bastante sencillo.

ATENCIÓN: la gramática anterior puede dar un conflicto. Si te ocurre, identifica el problema y arréglalo sin aumentar el LOOK AHEAD. **El número máximo de tokens de preanálisis es igual a 1.**

III: RESULTADOS

En primer lugar, debes definir la gramática del lenguaje MiniLeng en formato BNF.

A partir de esta gramática, debes implementar un analizador sintáctico que incluya el analizador léxico que realizaste en la práctica anterior y determine si el programa está bien formado o no. **El analizador debe utilizar un LOOK_AHEAD = 1**

Como resultado, debes completar las definiciones de patrones y la definición de tu gramática en JavaCC para el reconocimiento del lenguaje y que, al compilarlo, genere un analizador **que admita un fichero por línea de comandos** y como salida muestre la siguiente información:

- Si ha habido un error léxico, mostrará la línea y columna, junto con el error léxico localizado, y **el compilador terminará su ejecución**:

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

- Si ha habido un error sintáctico, mostrará la línea y columna, junto con un mensaje de error, y **el compilador seguirá su ejecución**:

  `"ERROR SINTÁCTICO(<línea, columna>): <mensaje de error>"`

- Si no hay errores léxicos ni sintácticos, no mostrará nada como salida.

El detalle de la información de salida se deja a tu elección. **Se valorará muy positivamente** la capacidad de mostrar el mayor detalle posible en la salida de tu compilador.

NOTA: No se puede utilizar la función de error y el mensaje que saca JavaCC por defecto. Debes construir el mensaje de error tú mismo.

Utiliza la batería de programas de prueba (extensión *.ml*) de la práctica anterior para determinar el correcto funcionamiento del analizador sintáctico que has realizado. Además, deberás preparar **cinco** programas de prueba (con extensión .ml) que hayas realizado para comprobar el correcto funcionamiento de la práctica. No es necesario que sean muy complejos, pero sí que demuestren la corrección de la práctica de forma clara y concisa.

Finalmente, escribe la gramática que has definido en formato BNF y un fichero LEEME.txt donde describas las características principales de tu compilador ("permite definir funciones anidadas de la siguiente forma: ..", por ejemplo), así como las limitaciones que hayas podido identificar.

NOTA: Utiliza jjdoc para generar la documentación de la gramática en formato BNF: https://javacc.org/jjdoc

MEJORA OPCIONAL: Añade un modo de recuperación (PANIC MODE) cuando se encuentre un error típico como que falte el ";" al final de una sentencia.

Debes enviar el código fuente que hayas generado (.jj y clases adicionales Java si las hubieses implementado), la documentación generada con jjdoc y los programas de prueba del analizador léxico y sintáctico que hayas generado, así como un fichero LEEME.txt donde detalles las características de tu compilador y si has implementado alguna de las mejoras. Para ello tendrás que subir **antes del 5 de Abril** un fichero .zip que contenga los ficheros anteriores. Se realizará una evaluación del código entregado y, si tuvieras que realizar alguna modificación para garantizar que lo estás realizando correctamente, se te notificará a través del correo electrónico.