HITO 2 TAC

GRUPO E05

Javier Loro Carrasco

Diego Cordero Contreras

Mohamed Essalhi

Índice:

1. JFLEX

Lo primero que se nos pide es reconocer los elementos léxicos que nos encontramos en el lenguaje DOT e identificarlos. Además, debemos definir un token para cada elemento e implementar cada una de estas expresiones regulares con JFLEX para que tenga una acción asociada.

1.1. Cambios respecto al Hito 1

Cambio en la acción de las ER:
La primera modificación ha sido el cambio en la acción de las expresiones regulares, inicialmente lo que se hacía era imprimir el nombre del token en mayúsculas, ahora se hace un return de la siguiente manera:
{return new Symbol(sym.<TOKEN>, yytext());}

Arreglos con respecto al hito 1:

Comentarios: Se han cambiado los nombres a los comentarios, ahora todos están definidos con el mismo token COMENTARIO. Además, se ha arreglado un error que hacía que los comentarios que empiezan con '//' no se detectaban. Por último, se ha añadido la definición de comentarios que empiezan con el carácter '#'

Delimitadores: Se ha añadido una regla para ignorar los delimitadores, ya que esto producía problemas en la implementación de la Gramática de CUP.

Etiquetas: Se ha añadido la posibilidad para todas las cadenas de estar representadas entre comillas y sin comillas

Errores en las ER: Se ha añadido una regla, en caso de que alguna cadena no sea reconocida por ninguna ER, se lanza un error.

1.2. Tabla Resultante

La tabla resumen con los tokens reconocidos es la siguiente:

Expresión Regular	Token	Expresión Regular	Token
arbol	ARBOL	“bold” | bold	FT_BOLD
node	NODO	“dashed” | dashed	FT_DASHED
shape	SH	“solid” | solid	FT_SOLID
label	ETIQUETA	“none” | none	DIR_NONE
color	COLOR	“forward” | forward	DIR_FORWARD
fontcolor	COLOR_FUENTE	“back” | back	DIR_BACK
style	FUENTE	“both” | both	DIR_BOTH
edge	ARISTA	{	LLAVE_A
dir	DIRECCION	[	CORCHETE_A
hijos	HIJOS	}	LLAVE_C
“square” | square	SH_CUADRADO	]	CORCHETE_C
“circle” | circle	SH_CIRCULO	=	IGUAL
“doublecircle” | doublecircle	SH_DOBLECIRCULO	,	COMA
“rectangle” | rectangle	SH_RECTANGULO	;	PUNTO_COMA
“blue” | blue	C_AZUL		
“green” | green	C_VERDE		
“red” | red	C_ROJO		
“yellow” | yellow	C_AMARILLO		

Expresión Regular	Descripción	Token	Ejemplo
\"([^(\")]*)\"	Reconoce cualquier cadena comprendida entre dos comillas (“”) sin incluir la propia comilla	ETIQUETA_VALOR1	“¿Cómo?”
[^(" "| \t | \f | \n | \r |\{ |\[|\} |\] |= |, |;)]*	Reconoce cualquier palabra formada por cualquier carácter sin incluir: {,}, [,],=,’,’,;	ETIQUETA_VALOR2	¿Cómo?
[a-zA-Z]+[^(" "| \t | \f | \n | \r |\{ |\[|\} |\] |= |, |;)]*	Reconoce cualquier palabra que empiece con un carácter de a-z y seguido por cualquier carácter que no incluya: {,}, [,],=,’,’,;	IDENTIFICADOR	Diccionario
"//"+([^("\n")])*+\n #+([^("\n")])*+\n "/*"+ ([^("*/")])* +"*/"	Reconoce cualquier cadena que empiece con // o # y termina con un salto de línea Reconoce cualquier cadena que empiece con /* y acabe con */, siempre que en los caracteres intermedios no aparezca */	COMENTARIO	//Temporal \n /*Funcionalidad del método*/ #Temporal\n

2. CUP

En este segundo hito se pide la implementación de la gramática de CUP, la traducción al lenguaje DOT y como parte opcional añadir reglas semánticas para la comprobación que la estructura es correcta.

2.1. Gramáticas y Reglas Semánticas

Inicialmente se han definido los símbolos terminales, coincidiendo estos con los tokens creados en el apartado de JFLEX.

A continuación, se han definido una lista de símbolos no terminales que explicaremos en distintos apartados:

DefEtiquetas:

Símbolo no terminal que sirve para definir los valores de las etiquetas, es decir, el valor de un label.

Así, las reglas semánticas serían las 2 siguientes:

1. DefEtiquetas: ETIQUETA_VALOR1 (con comillas). Devolvemos la propia ETIQUETA_VALOR1
2. DefEtiquetas: ETIQUETA_VALOR2 (sin comillas). Devolvemos ETIQUETA_VALOR2 encapsulada entre comillas.
3. DefEtiquetas: IDENTIFICADOR (sin comillas). Devolvemos IDENTIFICADOR encapsulada entre comillas

DefComentarios:

Símbolo no terminal que sirve para definir los comentarios de manera que la llamada es recursiva hasta que no encuentra un comentario (en teoría se definiría como λ). La regla de producción devuelve el comentario encontrado y por último un salto de línea para mejor visualización.

1. DefComentarios: comentario DefComentarios: resto de comentarios. Devolvemos la cocatenación de comentario + resto de comentarios.
2. DefComentarios: λ. Devolvemos un salto de línea

Tipos de Atributos

Símbolos no terminales que constan de la definición de los distintos tipos de etiquetas que se definen en DOT. Su regla semántica devuelve la propia cadena encontrada, que indica el tipo de atributo que se ha generado.

1. TiposSH: tipos de SH, puede ser:
    a. SH_CUADRADO
    b. SH_CIRCULO
    c. SH_DOBLECIRCULO
    d. SH_RECTANGULO

2. TiposColor: tipos de COLOR, puede ser:
    a. C_AZUL
    b. C_VERDE
    c. C_ROJO
    d. C_AMARILLO
3. TIposFuente: tipos de FUENTE, puede ser:
    a. FT_BOLD
    b. FT_DASHED
    c. FT_SOLID
4. TiposDireccion: tipos de DIRECCION, puede ser:
    a. DIR_NONE
    b. DIR_FORWARD
    c. DIR_BACK
    d. DIR_BOTH

Definición de los Atributos Generales de los Nodos:

Cuando queremos declarar los atributos predeterminados de los nodos. La gramática se define como:

- DefAN → node “{“ DefANAux
- DefANAux →
    - SH = TiposSH DefANTerminaciones
    - ETIQUETA = DefEtiquetas DefANTerminaciones
    - COLOR = TiposColor DefANTerminaciones
    - COLOR_FUENTE = TiposColor DefAATerminaciones
    - FUENTE = TiposFuente DefANTerminaciones
- DefANTerminaciones →
    - “,” DefANAux
    - “}”

De manera que empieza con: “node {“ seguido de una serie de atributos separados por comas hasta que encuentras el corchete de cierre “}”

La regla semántica que genera es una cadena de texto con la declaración de los atributos de los nodos

Definición de los Atributos Generales de las Aristas:

Al igual que con los nodos, es una gramática que utilizada cuando queremos declarar los atributos predeterminados de las aristas. La gramática se define como:

- DefAA →edge "{" DefAAAux
- DefAAAux →
    - DIRECCION = TiposDireccion DefAATerminaciones
    - ETIQUETA =DefEtiquetas DefAATerminaciones
    - COLOR = TiposColor DefAATerminaciones
    - COLOR_FUENTE = TiposColor DefAATerminaciones
    - FUENTE = TiposFuente DefAATerminaciones
- DefAATerminaciones →
    - "," DefAAAux
    - "}"

De manera que empieza con: "edge {" seguido de una serie de atributos hasta que encuentras el corchete de cierre "}"

La regla semántica que genera esta parte es una cadena de texto con la declaración de los atributos de las aristas.

Definición de los Nodos:

Para definir los nodos debemos tener en cuenta que existen distintas gramáticas para su definición:

1. "id del nodo" "hijos = {..(lista de los hijos)..}" "[..(lista de atributos)..]" ";"
2. "id del nodo" "hijos = {..(lista de los hijos)..}" ";"
3. "id del nodo" "[..(lista de atributos)..]" ";"
4. "id del nodo" ";"

Además, para facilitar la tarea se ha definido una clase NodoDelArbol con 3 atributos, un identificador, una lista con las aristas y una lista de los atributos del nodo. Las aristas se definen mediante la clase AristaDelArbol, que tiene 2 atributos, el identificador de destino de la arista y una lista de los atributos de la arista.

Comenzaremos con la definición del no terminal DefDefiniciónNodo que expresará los 4 casos mencionados anteriormente, lo que devolverá será una instancia de la clase NodoDelArbol con los datos del nodo obtenido. Gramática de DefDN:

1. DefDN: ID DefHDN "[DefANAux" ;""
2. DefDN: ID DefHDN ";"
3. DefDN: ID "[DefANAux" ";"
4. DefDN: ID ";"

DefHijosDelNodo (DefHDN): Símbolo no terminal que sirve para definir los hijos de un nodo, su gramática es la siguiente:

1. DefHDN: hijos = [DefHDNA
2. DefHDNA: →
    a. [id DefAAAux (Definición auxiliar de los atributos de una arista) DefNAT
    b. [id DefNAT
3. DefNAT: →
    a. "," DefHDNA
    b. "]"

De manera que empieza con: "hijos = {" seguido de una serie de hijos hasta que encuentras la llave de cierre "}"

La regla semántica resultante de DefHijosDelNodo es un ArrayList de AristaDelArbol, que expresan los hijos del nodo.

Definición del Árbol:

Es el último apartado en el que declararemos la creación de un árbol. Será el símbolo no terminal con el que se empiece el programa. Primero explicaremos su método auxiliar DefArbolAux.

DefArbolAux: símbolo no terminal que tiene 4 posibles gramáticas:

- DAA: DefComentarios DefAN DAA
    - Definición de los atributos predeterminados de un nodo
    - La regla semántica devuelve una cadena con los comentarios si los hubiera, los atributos de los nodos y el resto del arbol
- DAA: DefComentarios DefAA
    - Definición de los atributos predeterminados de una arista
    - La regla semántica devuelve una cadena con los comentarios si los hubiera, los atributos de las aristas y el resto del arbol
- DAA: DefComentarios DefDN DAA
    - Definición de un nodo
    - La regla semántica devuelve una cadena con los comentarios si los hubiera, el nodo formateado con la sintaxis de DOT y el resto del árbol
- DAA: DefComentarios "}" DefComentarios
    - Cierre del árbol
    - La regla semántica devuelve una cadena con los comentarios si los hubiera, la llave de cierre y comentarios si los hubiera

Con esta definición auxiliar, podemos definir el árbol con el símbolo no terminal DefArbol, que solo tiene una posible Gramática:

- DA: DefComentarios "arbol" IDENTIFICADOR "{" DAA

La regla semántica resultante debe comprobar que la estructura de árbol es la correcta (Versión 3 de CUP), en caso de que se cumplan se creará un fichero en la carpeta resultado con el nombre resultado_analisis.dot con la traducción de LEA a DOT y en caso de que tengamos instalado DOT en nuestro dispositivo, se generará la imagen y se abrirá.

3. Apartados extra realizados

Este trabajo trataba de la traducción sencilla del lenguaje LEA a DOT, si bien es cierto que estos apartados se han realizado conforme se ha explicado en este mismo documento, se han realizado apartados extra.

1. Versión 3 de CUP: En esta versión de la práctica se nos pedía que la estructura del árbol es correcta, para ello:

1. Todos los nodos declarados como hijos deberían tener una definición como nodo.
2. Todos los nodos definidos deberían estar, al menos una vez, en una definición de un hijo de un nodo (excepto la raíz).

Para ello, se ha utilizado un HashMap en java definido de la siguiente manera: “Identificador”:{“Nodo definido”:True/False “Nodo como hijo”: True/False}

Conforme reconocíamos que un nodo ha sido definido se añade a la lista y su atributo “Nodo definido” pasaría a ser True y si se le ha añadido en la lista de hijos de un nodo, “Nodo como hijo” pasaría también a ser True, se hace sencillamente mediante el método ActualizarMap.
Finalmente, en la regla semántica de DefArbol se comprueba que la estructura sea correcta y en tal caso se genera el archivo. En caso de que no lo hagan, te muestra los nodos que causan dicho error.

2. Errores en DOT para cadenas sin comillas

Esta necesidad nace ya que en DOT no existe manera de representar ciertos caracteres sin poner comillas. Por ejemplo no podríamos hacer una combinación de números y caracteres, por ejemplo: ‘11abc’ sería una cadena no válida en DOT ya que no tiene comillas.

Para solucionarlo en el símbolo no terminal DefEtiquetas a las cadenas que no tienen comillas le añadimos las comillas.

3. Generar Imagen automáticamente

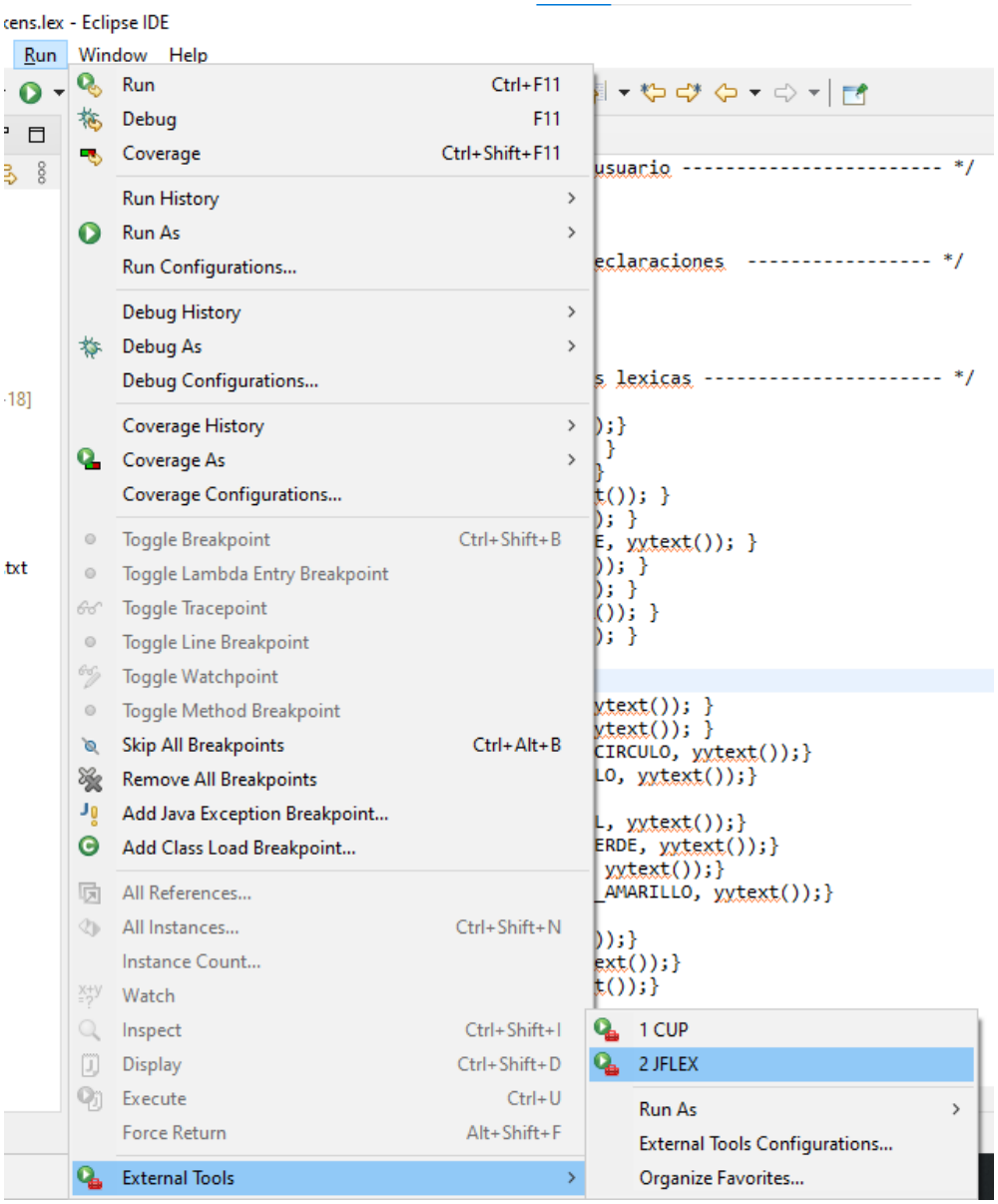
Para automatizar el paso desde LEA hasta DOT, hemos implementado un sencillo script bash, que en el caso de que tengamos instalado DOT en nuestro dispositivo, genere la imagen resultante en la carpeta de resultado como resultado_analisis.png, en caso contrario, abrirá automáticamente el archivo resultado_analisis.dot.

4. Manual de usuario

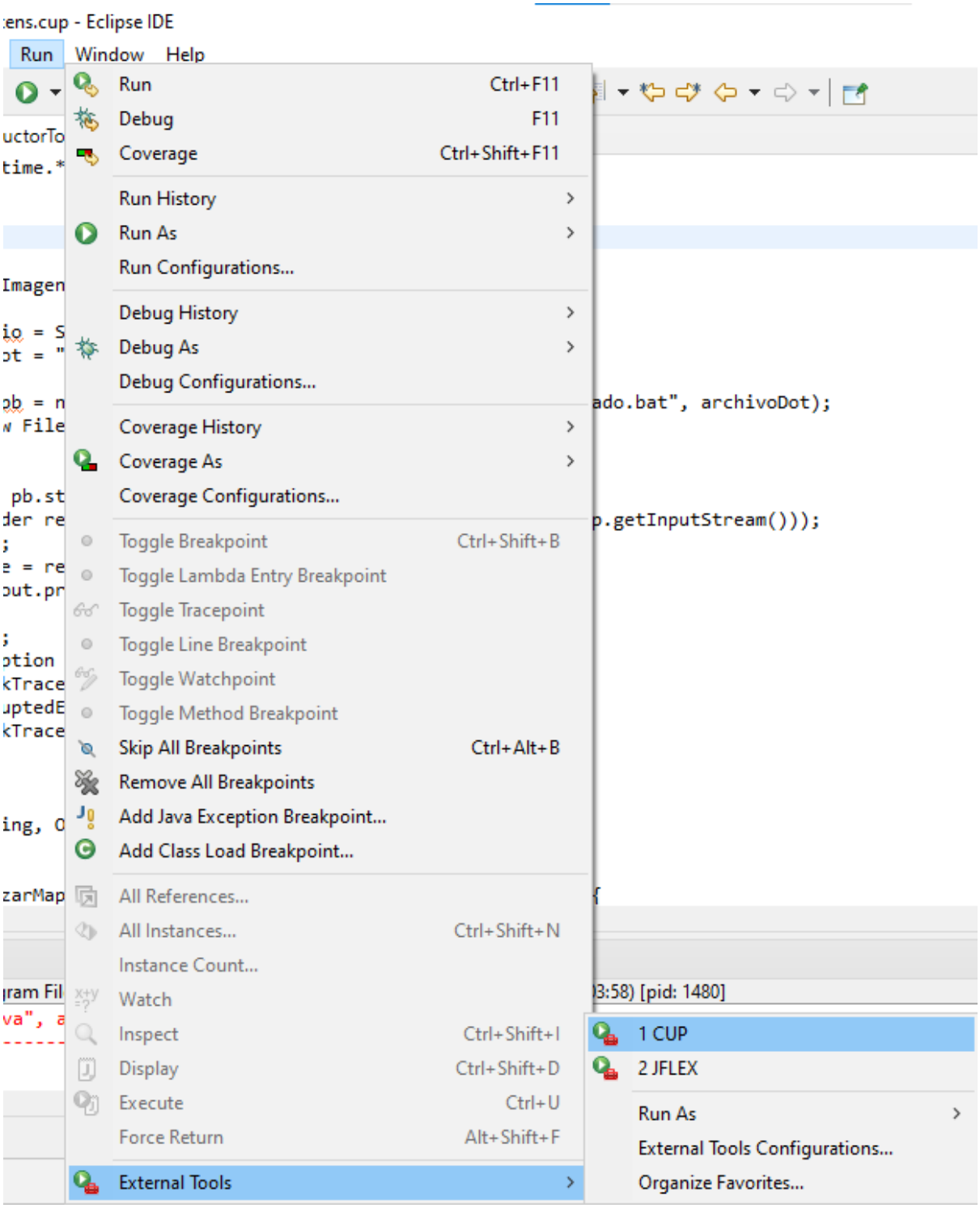
Para ejecutar el programa debemos seguir la siguiente lista de pasos:

Paso 1: Definir las configuraciones JFLEX y CUP en eclipse. https://campusvirtual.uclm.es/pluginfile.php/5768319/mod_resource/content/10/Instalacion_Jflex_CUP_INTEGRADO_EN_ECLIPSE_.pdf

Paso 2: Abrimos el fichero TraductorTokens.lex → Run → External Tools → JFLEX y se nos generará el archivo Yylex.java en la carpeta src, este archivo por sí solo dará error pero no hay que preocuparse por ello

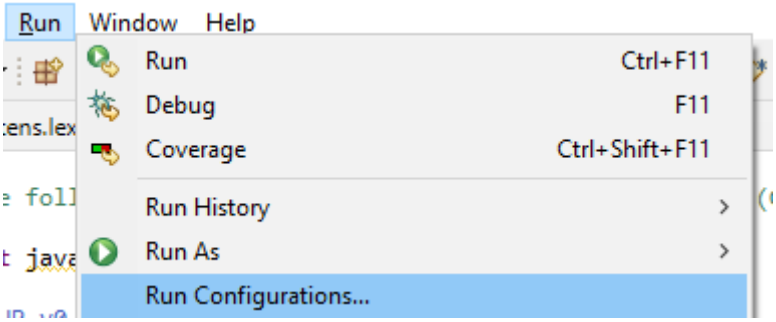


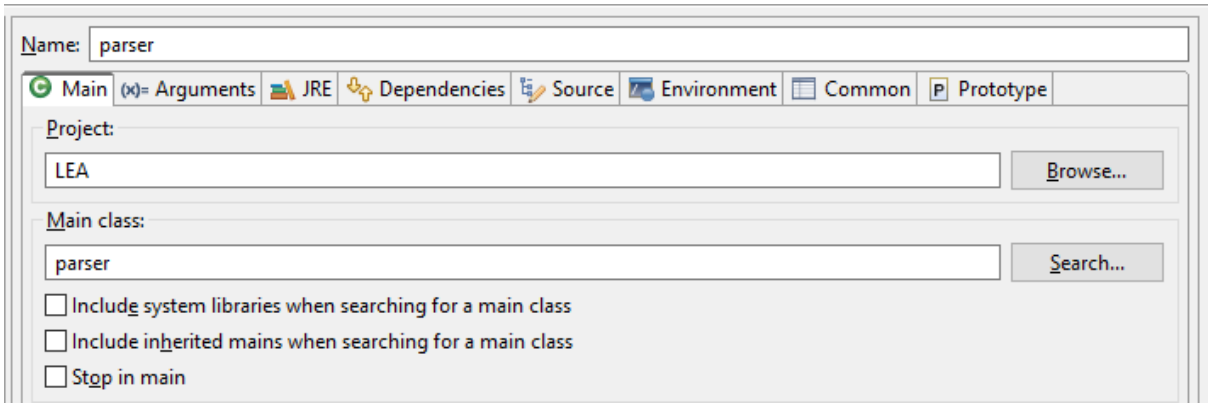
Paso 3: Abrimos el fichero TraductorTokens.cup → Run → External Tools → CUP y se nos generarán los archivos parser.java y sym.java en la carpeta src, se nos quitará el error del archivo Yylex.java

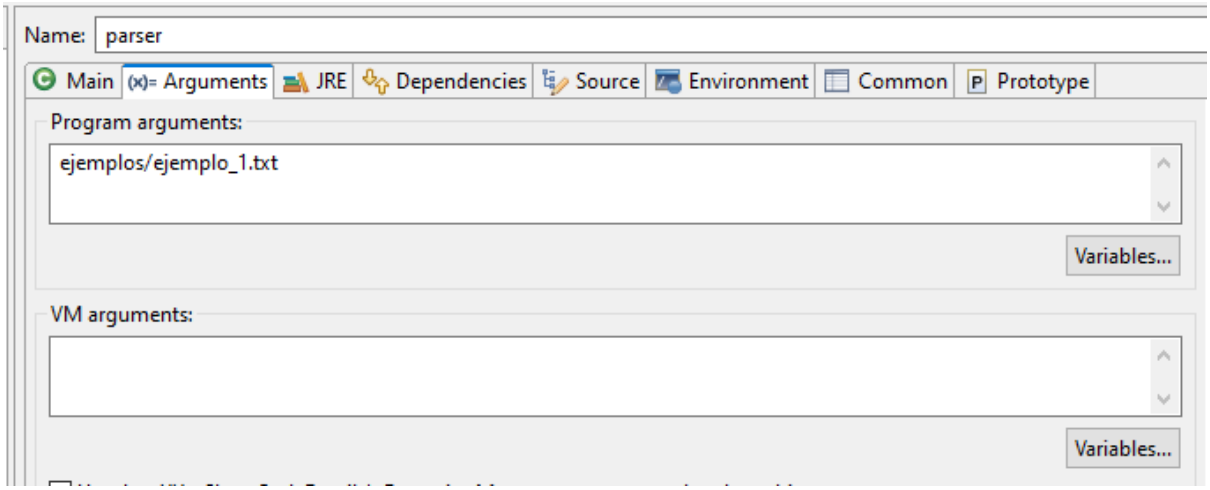


Paso 4: Abriremos el archivo parser.java → Run → Run Configurations → Definimos una configuración → Pestaña Arguments → En “Program arguments” escribiremos el nombre de alguno de los archivos que aparecen en la carpeta ejemplos:

1. ejemplos/ejemplo_1.txt
2. ejemplos/ejemplo_2.txt
3. ejemplos/ejemplo_3.txt
4. ejemplos/ejemplo_4.txt
5. ejemplos/ejemplo_5.txt





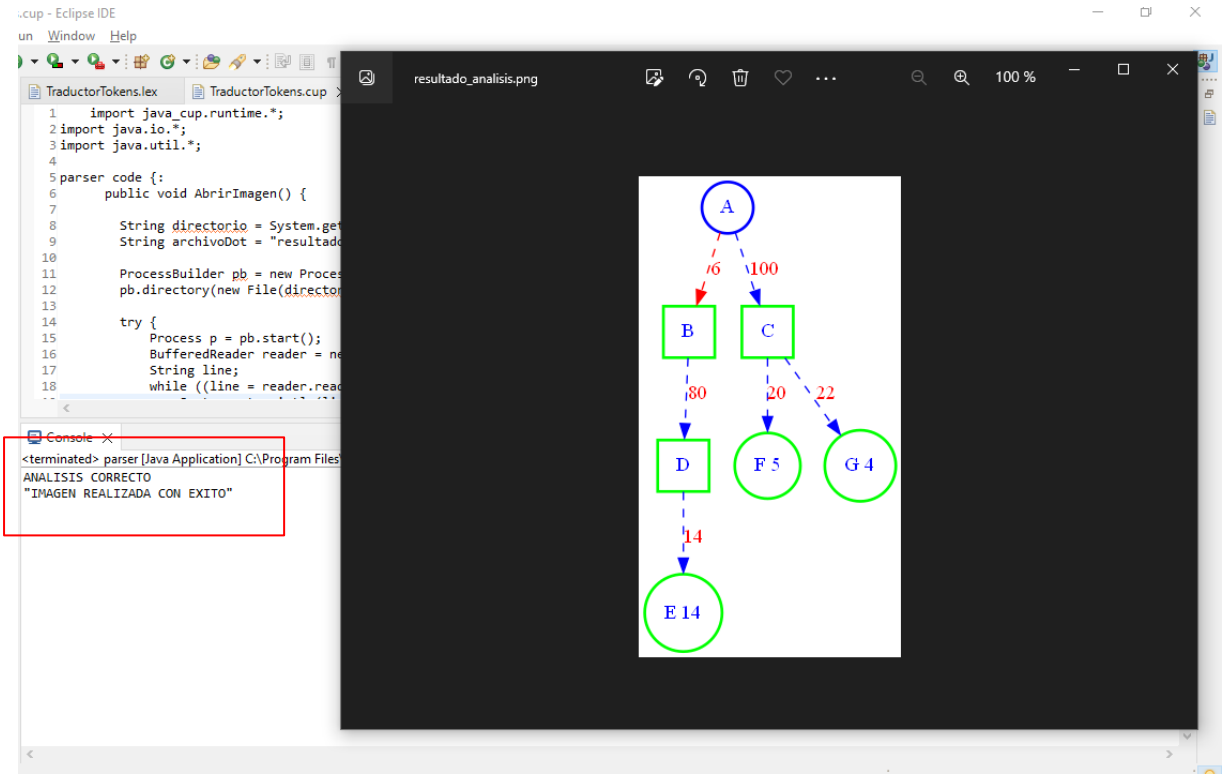


Finalmente, en la carpeta resultado, se generará en la carpeta resultado un archivo llamado resultado_analisis.dot con la traducción a DOT. Además, si tenemos instalado Graphviz en nuestro dispositivo (https://graphviz.org/download/), se generará la imagen resultante y se abrirá automáticamente.

Si no lo tenemos instalado o no se puede generar la imagen, se abrirá el archivo resultado_analisis.dot creado.

Ejemplo:

Para el ejemplo: ejemplos/ejemplo_1, al ejecutar el programa, obtendríamos la siguiente salida, se nos abre la imagen y nos indica por consola de comandos que la salida es correcta



5. Conclusión

Una vez realizado el trabajo y concluido el desarrollo al completo del traductor, tenemos un programa java que es capaz de traducir un archivo del lenguaje LEA (con las especificaciones dadas) a DOT de manera satisfactoria.

Hemos aprendido conceptos vistos en la asignatura de manera práctica, viendo la importancia y la complicación que tiene realizar un compilador en cualquier lenguaje de programación. Al habernos estado informando tanto sobre cómo es la realización de un compilador convencional, que usamos día tras día, le hemos dado valor a cosas que hasta ahora ni contemplábamos al programar.

Con esto queremos remarcar que, no solo hemos concluido el trabajo funcional, sino que, además, nos hemos llevado una gran experiencia que nos ha permitido formarnos ampliamente en conceptos que, de otra manera, nos habría sido imposible entender tan bien.