APÉNDICE 1
Instalación y uso de las herramientas.

1. Instalación de las herramientas:

- *Java:*

Lo primero que debemos instalar es el *JDK* (no basta con el JRE), que es el entorno necesario para compilar y ejecutar clases Java.

El SDK para *J2SE* (Java 2 Standard Edition) podrás descargarse de la página web de *Sun*[1] para los Sistemas Operativos Linux, Windows y Solaris. Los usuarios de FreeBSD podrán obtener la versión 1.3.1 de Java siguiendo las instrucciones de la Fundación FreeBSD[2]. Alternativamente, los usuarios de Linux podrán usar el JDK desarrollado por Blackdown[3].

Las instrucciones para la instalación difieren en cada sistema, y encontrarás las instrucciones apropiadas en la página web de la descarga o en la propia distribución de Java.

- *Jlex:*

Jlex es una herramienta desarrollada en Java que usaremos para realizar el analizador léxico de nuestro compilador. Las instrucciones de instalación son muy sencillas, y son las mismas para cualquier Sistema Operativo:

1. Crea un directorio nuevo (aquí lo llamaremos *jdir*) que esté en tu CLASSPATH (también puedes usar un directorio que ya exista). Crea un directorio llamado *jdir/JLex* (o *jdir\JLex* en Windows), y copia en el fichero Main.java que podrás descargar de la página web[4] de Jlex.
2. Compila el fichero Java con el siguiente comando:
   javac Main.java
3. Se habrán creado las clases Java que componen la distribución de Jlex.

Ahora ya podrás usar el *Jlex* mediante este comando:

java Jlex.Main fichero.jlex

donde *fichero.jlex* es el nombre del fichero con la especificación del análisis sintáctico para tu lenguaje.

Alternativamente a este proceso de instalación, los usuarios de Linux podrán instalar un paquete binario RPM o DEB con la distribución de Jlex, lista para usar. En concreto, los usuarios de Debian podrán obtener el paquete del FTP oficial del proyecto Debian y podrán obtenerlos mediante este comando:

apt-get install jlex

para usar esta distribución de Jlex se debe utilizar este comando:

jlex fichero.jlex

donde *fichero.jlex* tiene el mismo significado que antes. En este caso no será necesario modificar el

---

1http://java.sun.com/j2se/1.4.2/download.html
2http://www.freebsdfoundation.org/downloads/java.shtml
3http://www.blackdown.org
4http://www.cs.princeton.edu/~appel/modern/java/JLex/

CLASSPATH.

- *CUP:*

CUP es la herramienta que usaremos para generar el analizador sintáctico de nuestro lenguaje. Al igual que JLex está escrita en Java, y existe un proceso de instalación común para todas las plataformas:

1. Descarga el código fuente de CUP desde su página web[5] a un directorio de tu sistema que esté en el CLASSPATH. Descomprime el paquete una vez lo hayas descargado
2. Ahora compilaremos el código fuente de CUP. Para ello, desde el directorio donde descargamos la distribución, ejecutaremos el siguiente comando:
   javac java_cup/.java java_cup/runtime/*.java*

Ahora podrás ejecutar *CUP* mediante el siguiente comando:

java java_cup.Main < fichero.cup

donde *fichero.cup* es el fichero con la especificación del analizador sintáctico de nuestro lenguaje.

Y ya está hecho. Si eres usuario de Linux, de nuevo puedes saltarte estas instrucciones e instalarte el paquete binario para tu distribución. En este caso, podrás ejecutar *CUP* mediante este comando:

cup < fichero.cup

- *Jasmin:*

Jasmin es un ensamblador para la Máquina Virtual Java, el cual toma como entrada un fichero de texto con la descripción de una clase Java, y produce como resultado el *.class* de esa clase. Nosotros usaremos esta herramienta para obtener el programa ejecutable resultado del proceso de compilación.

El proceso de instalación de *Jasmin* se puede resumir en los siguientes pasos:

1. Descarga la distribución de su página web[6], y descomprimelo en /usr/local si eres usuario de UNIX, o C:\ si eres usuario de Windows. Se creará un directorio llamado *jasmin*.
2. Ahora debes añadir el directorio /usr/local/jasmin/bin (o C:\jasmin\bin) a tu variable de entorno $PATH. En UNIX bastará con hacer un enlace simbólico a /usr/local/bin, si es que este directorio está ya en el $PATH:

   ln -s /usr/local/jasmin/bin/jasmin /usr/local/bin/

Ahora podrás ejecutar *Jasmin* mediante el siguiente comando:

jasmin fichero.j

donde *fichero.j* es la descripción de una clase Java en ensamblador.

NOTA: en algunas sistemas Windows, tendrás que añadir el directorio C:\jasmin\classes a tu CLASSPATH para que funcione correctamente.

5http://www.cs.princeton.edu/~appel/modern/java/CUP/
6http://cat.nyu.edu/meyer/jasmin

2. Uso de las herramientas:

En esta sección vamos a ver cómo podemos usar las herramientas anteriores para construir un compilador para un lenguaje dado. Aunque ahora trataremos solamente el uso básico de las herramientas, el el *Apéndice 2* veremos un ejemplo de analizador sintáctico para un lenguaje sencillo.

La clase de traductor que implementaremos es un traductor dirigido por la sintaxis. Esto significa que la herramienta de mayor importancia en el desarrollo de la práctica será *CUP*, que es la que se encarga del análisis sintáctico. *Jlex* actuará proporcionándole tokens según los requiera, y *Jasmin* tan sólo transformará la salida del proceso de análisis sintáctico e una clase Java. Esto significa que la salida proporcionada por *CUP* será una especificación en lenguaje ensamblador de una clase Java, que luego pueda ser transformada a la clase final mediante *Jasmin.*

El primer paso que debemos dar es el de desarrollar el analizador léxico de nuestro lenguaje. Para ello crearemos un fichero de especificación de *Jlex* en el que se especifiquen los tokens permitidos por nuestro lenguaje. Este fichero de especificación tiene el siguiente formato:

```
Código de usuarios
%%
Directivas Jlex
%%
Expresiones regulares
```

Los caracteres %% se usan para dividir cada una de estas secciones. En la sección *Código de usuario* se importarán los paquetes necesarios y se crearán las clases necesarias para nuestro analizador léxico. Por ejemplo, se puede importar la clase *Symbol*, que más tarde nos será de utilidad:

```
import java_cup.runtime.Symbol;
```

En la sección *Directivas Jlex* se incluirán algunas órdenes propias de *Jlex*, como por ejemplo la de especificar el uso de *CUP:*

```
%cup
```

y también se pueden definir macros que resuman ciertas expresiones regulares útiles que nos serán útiles a la hora de identificar tokens en la siguiente sección. Por ejemplo, podría definirse una macro para reconocer números enteros:

```
NUMERO = [1-9][0-9]*
```

Por último, en la sección *Expresiones regulares* definiremos las expresiones regulares que indican los tokens de nuestro lenguaje, y que debemos hacer una vez detectados dichos tokens. Por ejemplo, dos líneas para reconocer identificadores limitados y enteros sería las siguientes:

```
"[a-b][a-b0-9]*"          { return new Symbol(sym.ID); }
{NUMERO}          { return new Symbol(sym.NUMERO, new Integer(yytext()); }
```

A la izquierda se especifica la expresión regular del patrón (entre llaves si se especifica una macro de la sección anterior), y a la derecha se especifica la acción que se llevará a cabo al detectar uno de esos tokens. En este caso, sólo válido si activamos la compatibilidad con *CUP*, devolvemos un

Symbol con información con el token encontrado, que luego será recogido por *CUP* (recuerda que el analizador léxico actúa por demanda). Así mismo, ten en cuenta que entre las llaves de la derecha puedes meter cualquier código Java que necesites.

El analizador sintáctico es la piedra angular de esta práctica. Él solicitará los tokens al analizador léxico según los necesite, y dará como resultado un especificación de la clase Java en ensamblador, que luego pueda ser transformada por *Jasmin*. Básicamente, en el fichero de especificación de *CUP* se deben definir tres partes:

- el código necesario para iniciar el análisis del texto de entrada,
- la lista de terminales y no terminales y
- las reglas que definen la gramática del lenguaje, expresadas como una gramática independiente del contexto.

Al igual que con el fichero de especificación léxica, en cada regla de producción pondremos el código Java que se ejecutará cuando se de alguna de esas producciones, ya que básicamente consistirá en ir escribiendo un fichero con la especificación de la clase Java para *Jasmin*. Las reglas de producción tiene este aspecto:

expr ::= expr MAS term {: Código Java :};

donde *expr* y *term* son no terminales, *MAS* es un terminal (devuelto por *Jlex* en la forma *return new Symbol(sym.MAS)*), y a la derecha está el código Java que se ejecutará al producirse esta producción.

Si todo va bien, el resultado de este proceso será el fichero de especificación en ensamblador de la clase Java, que podrá transformarse en una clase Java mediante *Jasmin*.

En el **primer cuatrimestre** sólo será necesario generar hasta el analizador sintáctico, pero sin la generación de código. Esto significa que que habrá que aprender *Jlex* y *CUP*, per no Jasmin. El resultado de esta primera parte debe ser un analizador sintáctico capaz de verificar la corrección de un programa escrito en el lenguaje *Jo99*.

En el **segundo cuatrimestre** debemos aprender la sintaxis de los ficheros de especificación de *Jasmin*, y hacer que *CUP* genere como salida la especificación correspondiente al programa escrito en *Jo99* que hemos pasado como entrada al compilador. Después transformaremos esa especificación en una clase Java mediante *Jasmin*, y esa clase debe ser correcta.

3. Enlaces de interés:

A continuación podrás ver enlaces con más información sobre *Jlex*, *CUP* y *Jasmin*:

- Página web de *Jlex*:
  http://www.cs.princeton.edu/~appel/modern/java/JLex
- Página web de *CUP*:
  http://www.cs.princeton.edu/~appel/modern/java/CUP
- Página web de *Jasmin*:
  http://mrl.nyu.edu/~meyer/jasmin
- Página web de *Jo99*:
  http://bmrc.berkeley.edu/courseware/cs164/fall99/assignments/jo99.html
- Breve introducción a *Jlex* en castellano:
  http://www.it.uc3m.es/luis/fo1/JLex.html

APÉNDICE 2
Un ejemplo de analizador sintáctico.

En este apéndice veremos cómo construir un analizador léxico y sintáctico para un lenguaje sencillo, con el cual se podrán hacer operaciones aritméticas como sumar, restar, multiplicar y dividir. En objetivo es que veas cómo se desarrolla un proyecto con las herramientas de la práctica y te sirva de apoyo para la misma. Así mismo, te servirá para empezar a trabajar con dichas herramientas, aunque para conocer toda su funcionalidad debes leer los manuales de cada una de ellas.

1. El lenguaje:

El lenguaje que describiremos será el de una pequeña calculadora con la que realizar operaciones de sumar, restar, multiplicar y dividir, junto con el uso de paréntesis. Aunque este es un lenguaje muy sencillo se aplican los mismos pasos que para otros más complicados, cómo el que tienes que desarrollar en tu práctica.

El primer paso es escribir la gramática del lenguaje. Este paso es muy importante, y es recomendable que lo escribas en papel y lo revises antes d empezar a escribir código. En nuestro caso, la gramática es muy sencilla y puede ser extraída del libro de teoría de la asignatura (para la gramática de *Jo99* fíjate en su documento de especificaciones[7]):

```
E <- E + T | T
T <- T – F | F
F <- F * G | G
G <- G / H | H
H <- (E) | n
```

donde *E, F, T, G* y *H* son no terminales, y *n* es un terminal. La gramática ha sido escrita directamente sin recursividad por la izquierda, y aunque podríamos modificarla para que fuese LL(1) no será necesario.

2. El analizador léxico:

Para realizar el analizador léxico usaremos la herramienta *Jlex*. Para realizar esta parte de la práctica sólo necesitaremos conocer dos cosas:

- el conjunto de tokens de nuestro lenguaje y
- las expresiones regulares que los definen.

Los tokens que nos atañen en este lenguaje son muy pocos, y hay que dividirlos en terminales y no terminales:

- *Terminales:* +, -, *, /, (,) y n (donde *n* es un número entero).
- *No terminales:* E, F, T, G y H.

Para el analizador léxico sólo nos interesa tratar los terminales, que son los tokens que leerá desde el fichero con el código de nuestro lenguaje. Debemos escribir en nuestro papel un nombre (en mayúsculas estaría bien) que represente cada uno de esos terminales, por ejemplo:

7http://bmrc.berkeley.edu/courseware/cs164/fall99/assignments/jo99.html

```
'+' -> MAS
'-' -> MENOS
'*' -> POR
'/' -> DIV
'(' -> PARI
')' -> PARD
'n' -> NUMERO
```

Así el analizador léxico, que escribiremos en un fichero llamado *calculadora.jlex*, queda como sigue:

```
/* calculadora.jlex*/
import java_cup.runtime.Symbol;

%%

%eofval{
     { System.exit(0); }
%eofval}

%cup

NUMBER  = [1-9][0-9]*

%%

{NUMBER}        { return new Symbol(sym.NUMERO, new
Integer(yytext())); }
"+"             { return new Symbol(sym.MAS); }
"-"             { return new Symbol(sym.MENOS); }
"*"             { return new Symbol(sym.POR); }
"/"             { return new Symbol(sym.DIV); }
"("             { return new Symbol(sym.PARI); }
")"             { return new Symbol(sym.PARD); }
";"             { return new Symbol(sym.FIN); }
.|\n            { }
```

Ahora explicaremos brevemente cada parte del fichero:

- *import java_cup.runtime.Symbol;*
  Importamos la clase Symbol del paquete *cup.jar*, ya que vamos a trabajar colaborativamente con la herramienta *CUP*, y el analizador léxico debe devolverle los tokens dentro de la clase *Symbol*. Tendremos que tener el fichero *cup.jar* para que esto funcione.
- *%eofval*
  Es una directiva de *Jlex* (todas las directivas empiezan con el carácter '%' y van en esta sección del fichero). Indica el código que se debe ejecutar al alcanzar el final de fichero. En este caso ejecutamos un *System.exit(0)* para salir del programa.
- *%cup*
  En una directiva de *Jlex* Indica que *Jlex* trabajará conjuntamente con *CUP*, y le servirá los tokens por demanda. Esto significa que este fichero no puede ejecutarse sólo.
- *NUMBER = [1-9][0-9]**
  Es una macro que indica la expresión regular que corresponde a un número. Aunque

podríamos usar la expresión regular directamente en la siguiente sección sin usar la macro, conviene hacerlo así por claridad.

- *{NUMBER} { return new Symbol(sym.NUMERO, new Integer(yytext())); }*
  Indica que cuando se encuentre un numero en la entrada (que encaje con expresión regular de la macro definida anteriormente), debe devolverse un *Symbol* al analizador sintáctico que contenga el tipo de token (NUMERO) y su valor (new Integer(yytext())). La función *yytext()* está implícita en *Jlex*, y devuelve el texto leído de la entrada y que corresponde con el patrón. Fíjate que en el caso de los otros terminales sólo se devuelve el *Symbol* con un argumento (el tipo de token), porque en este caso su valor en la entrada no es relevante o es implícito al tipo de token.
- *.|\n { }*
  Indica que cuando se reciba un token que no corresponda con ninguno de los anteriores no se ejecutará ninguna acción. En la práctica, aquí se debe lanzar algún tipo de excepción, ya que hemos encontrado un token no reconocido en nuestro lenguaje.

Por último, destacar que en la tercera sección de este fichero sólo se tratan las expresiones regulares de los no terminales de nuestro lenguaje (recuerda que los tienes escritos en un papel...).

3. El analizador sintáctico:

Debemos escribir un fichero que represente todas las reglas de producción de nuestro lenguaje, de modo que se pueda realizar el análisis sintáctico de forma automática. Estas reglas de producción son las que definimos en nuestra gramática en el paso anterior, y que tenemos ya escritas y revisadas (aunque es posible que haya que cambiarlas un poco, si fuese necesario).

En este momento ya no nos interesan las expresiones regulares de nuestros tokens, ya que ahora trabajamos en un nivel de abstracción superior: ahora sólo nos interesan lo tokens en si, según el nombre que les dimos en el apartado anterior, y que devolvíamos en un objeto *Symbol* (MAS, MENOS, POR...).

Luego lo que debemos hacer ahora es expresar de una manera formal y que pueda entender *CUP* la gramática de nuestro lenguaje. Como adelanto veremos como se escribiría la regla de producción E <- E+T | T:

```
expr_e ::= expr_e:l MAS expr_t:r    {: RESULT=new Integer(l.intValue() +
                                       r.intValue()); :}
         | expr_t:e                 {: RESULT=e; :}
         ;
```

Donde cada componente de la regla significa lo siguiente:

- *expr_e* y *expr_t* representan a los no terminales E y T respectivamente.
- Las letras *l*, *r* y *e* que siguen a los no terminales se indican para poder trabajar con su valor real a continuación.
- *MAS:* en un terminal que *CUP* recoge de *Jlex*. Recuerda que lo devolvíamos dentro del objeto *Symbol*.
- *RESULT=new Integer(l.intValue() + r.intValue());* Esto es código java que indica que el resultado de la producción es la suma de sus dos miembros. Fíjate que *l* y *r* son de la clase Integer, y para sumarlos hay que aplicarles el método *intValue()*. Fíjate también que el resultado de la suma debe pasarse otra vez a Integer ya que el objeto RESULT es de ese tipo.
- *| expr_t:e {: RESULT=e :};* El símbolo '|' es un OR que permite separar las dos posibles producciones de *expr_e*. En este caso se trata de la producción E <- T, y el resultado de E es simplemente el valor de T.

Y ahora, pasemos a escribir el código completo del analizador sintáctico, con algunos comentarios para entenderlo mejor:

```
/* Importamos las clases necesarias del paquete cup.jar */
import java_cup.runtime.*;

/**
 * Aquí ponemos el código que se usará para comenzar a parsear la entrada.
 */
parser code {:
        public static void main(String args[]) throws Exception {
                // La clase Yylex es creada por el analizador léxico
                // Jlex (ver sección siguiente).
                new parser(new Yylex(System.in)).parse();
        }
:}

/* Aquí especificamos los terminales del lenguaje. */
terminal MAS, MENOS, POR, DIV, PARI, PARD, FIN;
/**
 * Este terminal tiene un valor entero. Recuerda que le dábamos el valor
 * en el código del analizador léxico, al darle como parámetro un valor
 * entero al objeto Symbol.
 */
terminal Integer NUMERO;

/* Lista de no terminales. */
non terminal expr_list, expr_part;
/**
 * Aquí están los no terminales con valor entero, que son con los que
 * podemos hacer cálculos, y podemos escribirlos de la forma expr_e:l
 * (por ejemplo, no se podría hacer expr_list:l, ya que a ese no
 * terminal no le damos valor.
 */
non terminal Integer expr_e;
non terminal Integer expr_t;
non terminal Integer expr_f;
non terminal Integer expr_g;
non terminal Integer expr_h;

/* Aquí especificamos la precedencia de los operadores. */
precedence left MAS;
precedence left MENOS;
precedence left POR;
precedence left DIV;

/**
 * Ahora comenzamos con las reglas de producción.
 */

/**
 * Estas dos reglas son nuevas. Nos sirven para encadenar varias
 * expresiones separadas por un ';'
 */
expr_list ::= expr_list expr_part | expr_part;

expr_part ::= expr_e:e {: System.out.println("= "+e); :} FIN;

/* E <- E + T | T */
expr_e ::= expr_e:l MAS expr_t:r {: RESULT=new Integer(l.intValue() +
                                    r.intValue()); :}
         | expr_t:e {: RESULT=e; :}
         ;
```

```
/* T <- T – F | F */
expr_t ::= expr_t:l MENOS expr_f:r {: RESULT=new Integer(l.intValue() -
                                      r.intValue()); :}
         | expr_f:e {: RESULT=e; :}
         ;

/* F <- F * G | G */
expr_f ::= expr_f:l POR expr_g:r {: RESULT=new Integer(l.intValue() *
                                      r.intValue()); :}
         | expr_g:e {: RESULT=e; :}
         ;

/* G <- G / H | H */
expr_g ::= expr_g:l DIV expr_h:r {: RESULT=new Integer(l.intValue() /
                                      r.intValue()); :}
         | expr_h:e {: RESULT=e; :}
         ;

/* H <- (E) | n */
expr_h ::= PARI expr_e:e PARD {: RESULT=e; :}
         | NUMERO:n {: RESULT=n; :}
         ;
```

Para terminar, fíjate en que los valores enteros se recogen en la última regla, ya que el terminal 'n' es el único que lleva su valor cuando sale del analizador léxico. Después ese valor se va operando con otros según se mueve por las reglas. Fíjate también que el objeto RESULT es el que se mueve entre las reglas, y que cuando un regla evalúa *expr_g:l / expr_h:r*, *l* y *r* contiene los RESULT de las expresiones *expr_g* y *expr_h* respectivamente.

4. Cómo ejecutar la aplicación:

Los pasos para compilar y ejecutar nuestro analizador sintáctico son los siguientes:

- Generamos el fichero Java correspondiente del analizador léxico:
  JLex calculadora.jlex
- Renombramos el resultado a *Yylex.java* (recuerda que lo usábamos con ese nombre en el analizador sintáctico, dentro del código que indica como se analizará el fichero):
  mv calculadora.jlex.java Yylex.java
- Generamos los ficheros Java del analizador sintáctico:
  cup < calculadora.cup
- Compilados todos los ficheros para generar el resultado:
  javac -d . parser.java sym.java Yylex.java

Con esto ya podemos ejecutar nuestro analizador sintáctico. Podemos probarlos con un fichero de prueba, llamado *file*, que contenga esto:

```
4+3*(2-7)+20;
```

el resultado será:

```
$ java parser < file
= 9
```