

Laboratorio

Procesadores del Lenguaje

uc3m

2019-2020



Presentación

Profesores Laboratorio.

Jesús García Herrero
(jgherrer@inf.uc3m.es)



Objetivos

- **Plantear una gramática para un lenguaje**
 - **Verificar si la gramática cumple propiedades para su traducción eficiente**
- **Generar un analizador para un lenguaje dado**
- **Construir los controles semánticos necesarios para verificar y traducir un lenguaje**
- **Conocer los principios de generación de código**



Desarrollo y Herramientas

Las Prácticas de Laboratorio:

- Se realizarán por parejas

- Se programará en

 - lenguaje Java

 - entorno Eclipse + plug-in específico

- Herramientas

 - JFlex - The Fast Scanner Generator for Java
(<http://jflex.de/>)

 - CUP - Construction of Useful Parsers
(<http://www2.cs.tum.edu/projekte/cup/index.php>)



Evaluación

Las Prácticas de Laboratorio:

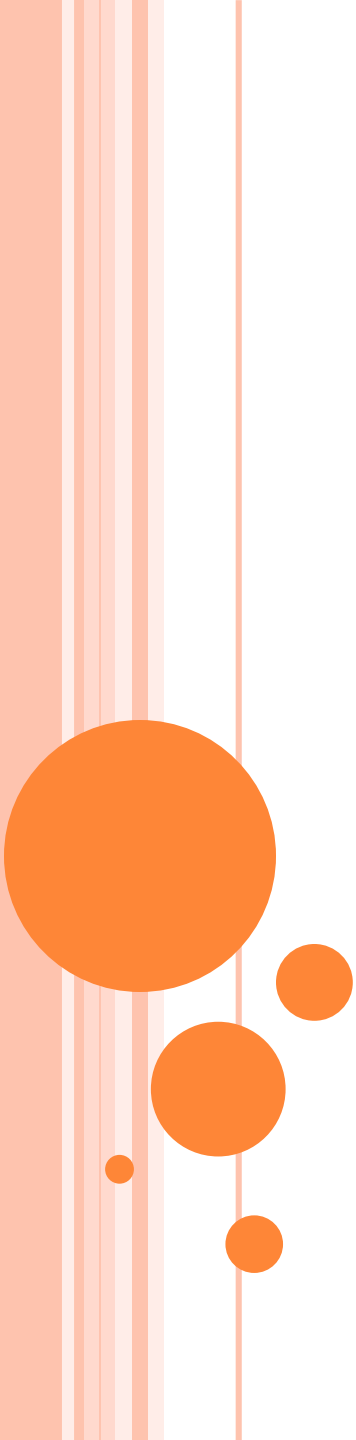
▪ **Supondrán un 40% de la nota de la asignatura:**

▪ **1ª Práctica – Introducción (0.75)**

▪ **2ª Práctica – Análisis lex/sint (0.75)**

▪ **3ª Práctica – Análisis sintáctico
y semántico (2.5)**





INTRODUCCIÓN A LAS HERRAMIENTAS DE COMPILADORES

Procesadores del lenguaje
(JCup y JFLex)

JFLEX

- Analizador léxico (también conocido **Scanner**), escrito en Java.
- Diseñado para trabajar junto CUP, analizador sintáctico para gramáticas LALR.
- <http://jflex.de/manual.html>



CUP

- Sirve para generar analizadores sintácticos (o ***parsers***) a partir de la especificación de una gramática independiente del contexto.
- CUP es la alternativa Java a las herramientas YACC y Bison para lenguaje C.
- <http://www.cs.princeton.edu/~appel/modern/java/CUP/manual.html>



INSTALACIÓN DE JFLEX Y CUP

- Descargar e instalar Eclipse
- Instalar el plugin CUP Eclipse. En la versión 3.7 los pasos son:
 - Help → Install new software...
 - Añadir una nueva localización: Add... → pegar la URL del sitio de actualizaciones del plug-in CUP y ponerle un nombre → Aceptar
- En la ventana anterior, seleccionar el repositorio recién creado y checkear el plug-in CUP.



CREACIÓN DEL PROYECTO

- Una vez instalado, debemos crear un nuevo proyecto Java de la forma usual en Eclipse.
- Añadimos un archivo JFlex al proyecto. Para ello, nos situaremos en la ventana de Eclipse y más concretamente en el explorador de proyectos:
 - Click derecho en la carpeta del proyecto recién creado → New → Other...
 - En la ventana que nos sale seleccionamos CUP Java Project
- En este punto ya podemos empezar a rellenar el archivo de acuerdo a lo especificado en el manual (<http://jflex.de/manual.html>).



EJEMPLO PROYECTO (JFLEX)

```
InputStream dataStream = System.in;
```

```
if( args.length >= 1 ) {  
    System.out.println( "Leyendo entrada de fichero... " );  
    dataStream = new FileInputStream(args[0]);  
}else{  
    System.out.println( "Inserta expresiones a reconocer, pulsando <ENTER> entre ellas"  );  
}
```

```
// Creamos el objeto scanner
```

```
IntroLex scanner = new IntroLex( dataStream );  
ArrayList<Symbol> symbols = new ArrayList<Symbol>();  
boolean end = false;
```



EJEMPLO PROYECTO (JFLEX)

```
while(!end){  
    // Mientras no alcancemos el fin de la entrada  
    try{  
        Symbol token = scanner.next_token();  
        symbols.add(token);  
        end = (token==null);  
        if( !end )  
            System.out.println("Encontrado: {" + token.sym+ "} >> " + token.value );  
    } catch (Exception x){  
        System.out.println("Ups... algo ha ido mal");  
        x.printStackTrace();  
    }  
}  
  
symbols.trimToSize();  
System.out.println("\n\n -- Bye-bye -- ");
```



DEFINIR LENGUAJE LÉXICO: PLANTILLA JFLEX

- Realizar los ajustes que sean convenientes mediante las directivas JFlex (comienzan por el símbolo de porcentaje ‘%’)
- Reconocer los patrones indicados:
 - Palabras reservadas: **lang**, **es**, **eng**, ...
 - Números
 - Identificadores
 - Comentarios...



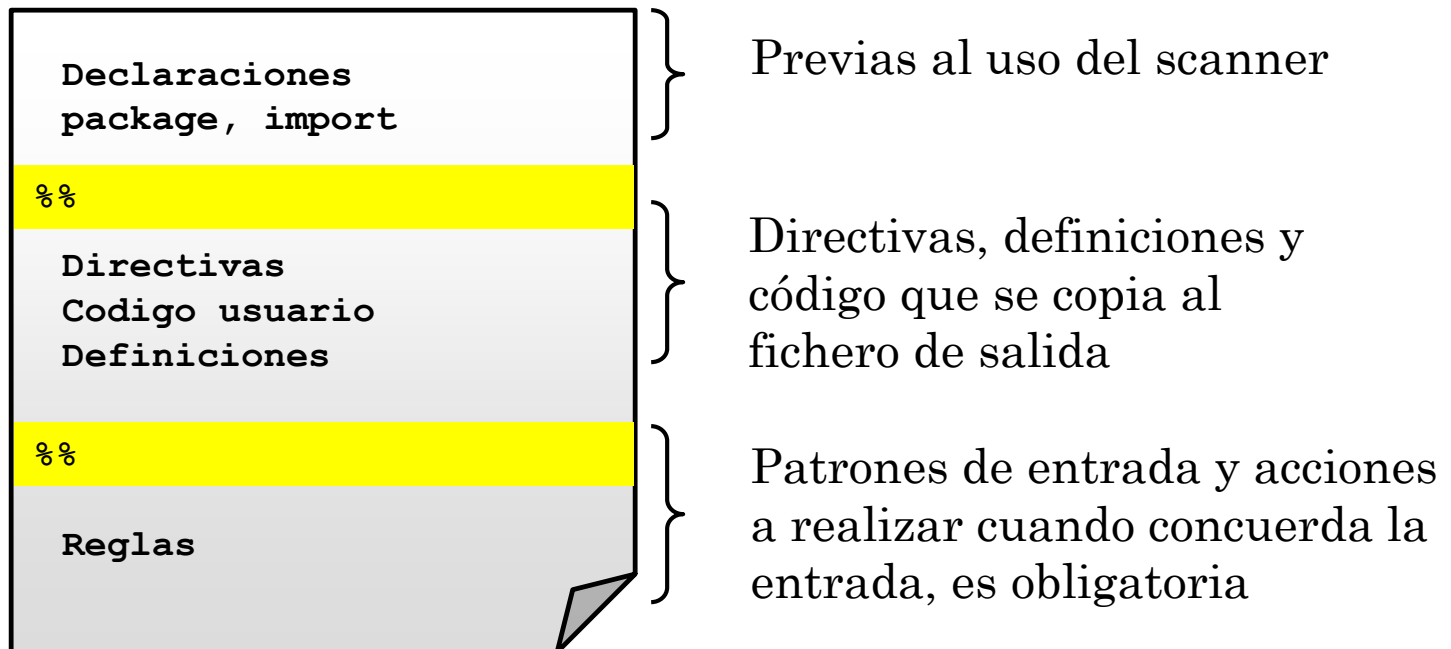


LEX

Procesadores del lenguaje
(JCup y JFLex)

FORMATO DE JFLEX

- Tres partes separadas por una línea con “%%”



FORMATO DE JFLEX

- Funcionamiento general: el analizador creado busca en la entrada ocurrencia de los patrones
 - Cuando encuentra un patrón, ejecuta sus acciones
 - Si no devuelven el control (return), continúa la búsqueda de nuevos patrones
 - Si varios patrones encajan, selecciona el más largo y el primero declarado
- Por defecto, el texto que no encaja con ningún patrón lo envía a la salida



SECCIÓN DE DECLARACIONES

Declaraciones
%%
<i>Directivas</i> %unicode ...
%{
<i>Código de usuario</i> private Clase1 obj;...
%}
<i>Definiciones</i> DIGITO =[0-9]
%%
<i>Condiciones de arranque</i> <YYINITIAL>{...}
Reglas



SECCIÓN DE DECLARACIONES

- Contiene cuatro tipos de declaraciones
 - **Directivas** que controlan el comportamiento de JFleX
 - **Código** del usuario que aparecerá copiado en el fichero de salida, delimitado por “%{” y “}%”
 - Incluir fichero de cabecera
 - Declarar variables globales
 - Declarar procedimientos que se describirán en la parte de código de usuario
 - Definición de **alias**, poner nombre a expresiones regulares
 - Aparece al principio de la línea
 - Se separa la expresión regular por un “=“
 - Definición de las **condiciones de arranque**



SECCIÓN DE DECLARACIONES

○ Definiciones

- Dar nombre a patrones complejos facilitando la legibilidad
- Se pueden anidar, otras definiciones se pueden utilizar entre llaves

```
entero  = [0-9]+  
real    = {entero}.{entero}  
real2   = {real}[eE][\+|-]?{entero}  
numero  = {entero}|{real}|{real2}
```



REPRESENTACIÓN DE PATRONES

Literal	"x"	La cadena x
Literal	*	El carácter literal: '*'
Definición	{nombre}	La expresión regular nombre (uso)
Selección	a ab	Selección de una alternativa: {a, ab}
Rango	[ad-gB-EG2-4]	{a,d,e,f,g,B,C,D,E,G,2,3,4}
Negación del Rango	[^a-z]	Cualquiera excepto minúsculas
Agrupar	(a-z) (0-9)	Para agrupar patrones [a-z0-9]
Numeración	r*	Ocurrencia de r >=0
	r+	Ocurrencia de r >0
	r?	Cero o una ocurrencia de r
	r{2,4}	De 2 a 4 ocurrencias de r
Cualquier carácter (no \n)	.	(.\n)* representa cualquier fichero
Localización	^r	r al principio de la línea
	r\$	r al final de la línea

SECCIÓN DE REGLAS

- Utilizan el formato PATRÓN ACCIÓN
- Los patrones pueden utilizar definiciones, expresiones regulares y condiciones de arranque
- Las acciones son código C, excepto la activación y desactivación de las condiciones de arranque, y otras (ECHO)
- El lexema que concuerda con el patrón está en la variable **yytext**, su longitud es **yytext**
 - Ej.: `[a-z] { System.out.println("%s",yytext());
return(1); }`



SECCIÓN DE REGLAS

○ Acciones


- Código entre llaves
- Si la acción es “|” entonces significa que se ejecuta la misma acción que para el siguiente patrón
- La instrucción **BEGIN<condición_de_arranque>** activa la condición de arranque especificada
- La instrucción **END<condición_de_arranque>** finaliza la activación de la condición de arranque



SECCIÓN DE REGLAS

- Reglas para la identificación de patrones
 - Siempre que para la entrada puedan aplicarse varias reglas:
 1. Se aplica el patrón que concuerda con el mayor número de caracteres en la entrada
 - Con la entrada **abc**:

```
a    {return (1) ;}  
ab   {return (2) ;}  
c    {return (3) ;}  
abc  {return (4) ;}
```


 2. Si hay dos patrones que concuerdan con el mismo número de caracteres en la entrada, entonces se aplica el que esté definido primero



VARIABLES, FUNCIONES, PROCEDIMIENTOS, MACROS

- LeX incorpora facilidades, las más comunes son:

Variable	Tipo	Descripción
yytext	char * o char []	Contiene la cadena de texto del fichero de entrada que ha encajado con la expresión regular descrita en la regla.
yyleng	int	Longitud de yytext yylength = strlen (yytext)
yyin	FILE*	Referencia al fichero de entrada.
yyval yylval	struct	Contienen la estructura de datos de la pila con la que trabaja YACC. Sirve para el intercambio de información entre ambas herramientas.



VARIABLES, FUNCIONES, PROCEDIMIENTOS, MACROS

Método	Descripción
yylex()	Invoca al Analizador Léxico, el comienzo del procesamiento.
yymore()	Añade el yytext actual al siguiente reconocido.
yylless(n)	Devuelve los n últimos caracteres de la cadena yytext a la entrada.
yyerror()	Se invoca automáticamente cuando no se puede aplicar ninguna regla.
yywrap()	Se invoca automáticamente al encontrar el final del fichero de entrada.

- yyerror e yywrap pueden ser reescritos

Nombre	Descripción
ECHO	Escribe yytext en la salida estandar. ECHO = printf ("%s", yytext)
REJECT	Rechaza la aplicación de la regla. Pone yytext de nuevo en la entrada y busca otra regla donde encajar la entrada. REJECT = ylless (yylength) + 'Otra regla'
BEGIN	Fija nuevas condiciones para las reglas. (Ver Condiciones sobre Reglas).
END	Elimina condiciones para las reglas. (Ver Condiciones sobre Reglas).



CUP

Procesadores del lenguaje
(JCup y JFLex)

CREACIÓN DE PROYECTO CUP

- Importar las clases de java.
- Copiar el método main en `parser code{: ... :}`
- Definir los terminales y no terminales.
- Crear la gramática LARL.

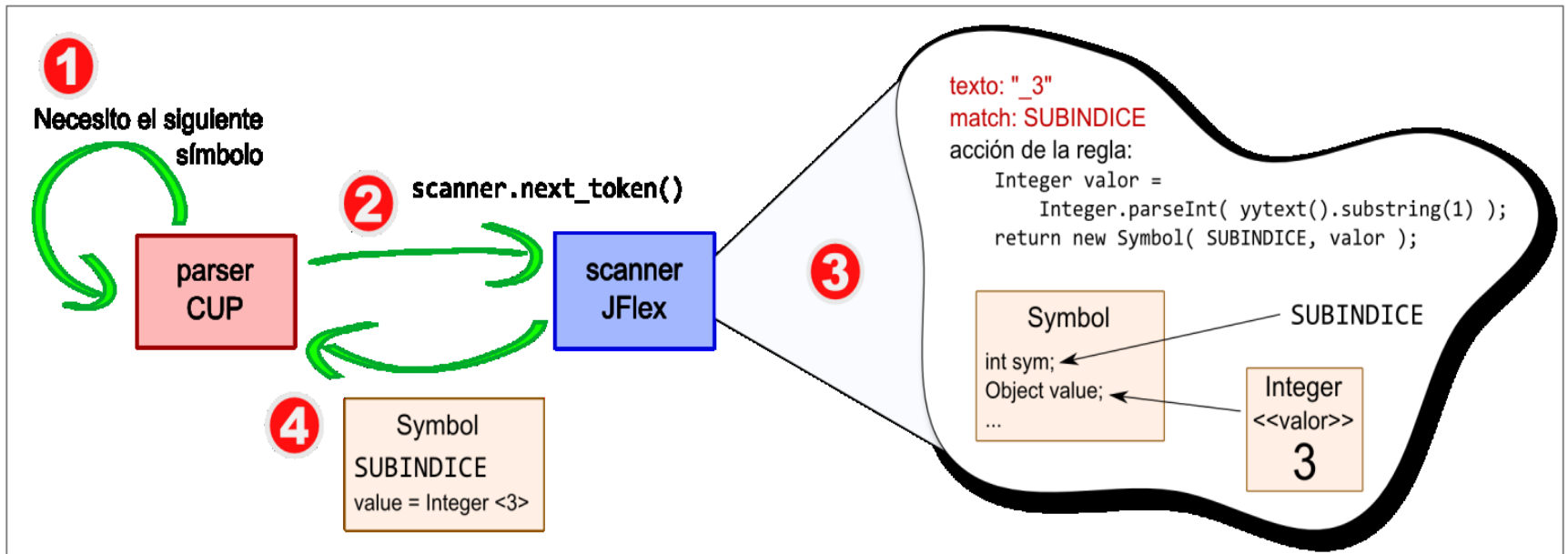


PRINCIPIO DE CUP/YACC

- Análisis previo de la gramática para construir el analizador LALR: notifica incidencias
- Funcionamiento general del analizador: pide tokens definidos al analizador léxico
 - Como es LALR(1), pide un token de pre-análisis
 - Cuando realiza desplazamiento pide el siguiente token
 - Cuando realiza reducciones de producciones, ejecuta el código en sus acciones



COMUNICACIÓN JFLEX-CUP



ELEMENTOS DE PLANTILLA CUP

- `parser code{: ... :}`
 - El código que se usará para comenzar a parsear la entrada, método `main`.
- `action code {: ... :}`
 - Permite incluir código dentro de la clase `CUP$acciones`. Se podrán llamar desde la gramática.
- `init with {: ... :}`
 - Proporciona código que será ejecutado por el analizador antes de que se solicite el primer token, inicialización de variables.
- `scan with {: ... :}`
 - Indica cómo el analizador debe pedir el siguiente token del Scanner (Jflex). El tipo de retorno del código debe ser el mismo que `java_cup.runtime.token`.



CREACIÓN DEL PROYECTO (CUP)

```
public static void main(String[] args) throws FileNotFoundException {
    InputStream dataStream = System.in;
    if( args.length >= 1 ) {
        System.out.println( "Leyendo entrada de fichero... " );
        dataStream = new FileInputStream(args[0]);
    }else{
        System.out.println( "Inserta expresiones a reconocer, pulsando <ENTER> entre ellas" );
    }
    IntroLex scanner = new IntroLex( dataStream );
    // NUEVO
    IntroCup parser = new IntroCup( scanner );
    try{
        Symbol parse tree = parser.parse();
    }catch (Exception x){
        System.out.println("Ups... algo ha ido mal");
        x.printStackTrace();
    }
    System.out.println( "\n\n -- Bye-bye -- " );
}
```



CREACIÓN DEL PROYECTO (CUP)

Terminales

No Terminales

terminal ES, ENG, LANG, SALTO;

terminal String TEXTO;

non terminal S, idioma, texto;

start with S;

Gramática

```
S::=LANG idioma texto SALTO | LANG idioma texto  
SALTO S
```

```
;
```

```
texto::= TEXTO:ex1
```

```
{:
```

```
System.out.println("El texto introducido es " + ex1);
```

```
;} 
```

```
| texto TEXTO
```

```
;
```

```
idioma::=ES
```

```
{:
```

```
System.out.println("El Idioma elegido es  
Castellano");
```

```
;} 
```

```
| ENG
```

```
{:
```

```
System.out.println("El Idioma elegido es Ingles");
```

```
;} 
```

```
;
```



DECLARACIONES DE TERMINALES

- Ejemplo definición terminales
 - Lenguaje de expresiones aritméticas con enteros

```
%{  
#include <stdio.h>  
%}  
  
terminal NUMERO, MAS, MENOS, POR, DIV, PAR_I, PAR_D  
non terminal Integer    expr;  
start expr /* simbolo axioma sentencial */  
...
```

Declaración en CUP de terminales y no terminales



DECLARACIONES DE TERMINALES

- Ejemplo definición terminales

```
%{  
#include "expresiones_tab.h"  
%}  
digito [0-9]  
  
%%  
[ \t]+ ;  
{digito}+ {yylval=atoi(yytext); return NUMERO;}  
"+" return MAS;  
"-" return MENOS;  
"*" return POR;  
"/" return DIV;  
"(" return PAR_I;  
")" return PAR_D;  
. {printf("token erroneo\n");}
```

Definición en JFlex de patrones

REGLAS EN YACC

- Formato BNF simplificado

LI: LD *acción*;

- **LI:** es un símbolo no-terminal del lenguaje
- **LD:** secuencia de símbolos no-terminales y terminales

- Agrupar varias reglas del mismo no terminal :

```
expr: expr '+' expr {....}  
      | expr '-' expr {....}
```

;

- Si se deja vacía es el regla de la palabra vacía

```
sentencias : sentencias ';' sentencia {....}  
            |
```

;

- ***acción*:** { sentencias en código } (puede ser vacío)



DECLARACIONES DE REGLAS

```
terminal NUMERO, MAS, MENOS, POR, DIV, PAR_I, PAR_D
non terminal Integer      expr;
start expr /* simbolo axioma sentencial */
```

```
%%
```

```
expr:  expr MAS term    {System.out.println("expr --> expr MAS term\n");}
      |expr MENOS term  {System.out.println("expr --> expr MENOS term\n");}
      |term              {System.out.println("expr --> term\n");}
;
term:  term POR factor   {System.out.println("term --> term POR factor\n");}
      |term DIV factor   {System.out.println("term --> term DIV factor\n");}
      |factor            {System.out.println("term --> factor\n");}
;
factor: NUMERO           {System.out.println("factor--> NUMERO(%d)\n", $1);}
      |PAR_I expr PAR_D {System.out.println("factor--> ( expr )\n");}
;
```

```
%%
```

```
. . .
```

EJEMPLO DE ANÁLISIS

○ Gramática

$\text{expr} \rightarrow \text{expr} + \text{term}$

$\text{expr} \rightarrow \text{term}$

$\text{term} \rightarrow \text{term} * \text{factor}$

$\text{term} \rightarrow \text{factor}$

$\text{factor} \rightarrow \text{NUMERO}$

$\text{factor} \rightarrow (\text{expr})$

○ Ejemplo de ejecución

- $5*(6+1)+3$



factor --> **NUMERO (5)**
term --> **factor**
factor --> **NUMERO (6)**
term --> **factor**
expr --> **term**
factor --> **NUMERO (1)**
term --> **factor**
expr --> **expr MAS term**
factor --> **(expr)**
term --> **term POR factor**
expr --> **term**
factor --> **NUMERO (3)**
term --> **factor**
expr --> **expr MAS term**

RESOLUCIÓN DE AMBIGÜEDAD

- Por defecto, después de avisar, yacc resuelve
 - D/R: Desplazar prioridad sobre reducir
 - R/R: Reducir por la producción primera
- Preferible resolver los conflictos explícitamente
 - Criterios de prioridad



PRECEDENCIA

○ Especificación de precedencia

- Asociatividad izquierda
 - **%left op: x op y op z → (x op y) op z**
- Asociatividad derecha
 - **%right op: x op y op z → x op (y op z)**
- No asociatividad
 - **%nonassoc op: x op y op z INCORRECTO**
- Con otros operadores: los declarados en líneas posteriores más precedencia

○ Ejemplo de asociación:

`% left '+' '-'`

`% left '*' '/'`

- El último declarado es el que tiene más precedencia



GRAMÁTICA EXPRESIONES CON PRIORIDAD

```
%token NUMERO, MAS, POR, '(', ')'  
%left MAS  
%left POR  
%start S /* simbolo axioma sentencial */  
  
%%  
S: expr {printf("resultado: %d\n", $$); }  
  
expr:  expr MAS expr  
      | expr POR expr  
      | '(' expr ')'  
      | NUMERO  
  
%%
```




ARBOLES SINTÁCTICOS

DEFINICIÓN DE LA GRAMÁTICA

Terminales

No Terminales

terminal SALTO;

terminal String ES, ENG;

terminal String TEXTO;

terminal LANG;

non terminal Traductor S;

non terminal Idioma idioma;

non terminal Texto texto;

non terminal Sentencias sentencias;

non terminal Sentencia sentencia;

Gramática

$S ::= \text{sentencias} \mid \text{lambda}$

$\text{sentencias} ::= \text{sentencias}$
 $\text{sentencia} \mid \text{sentencia}$

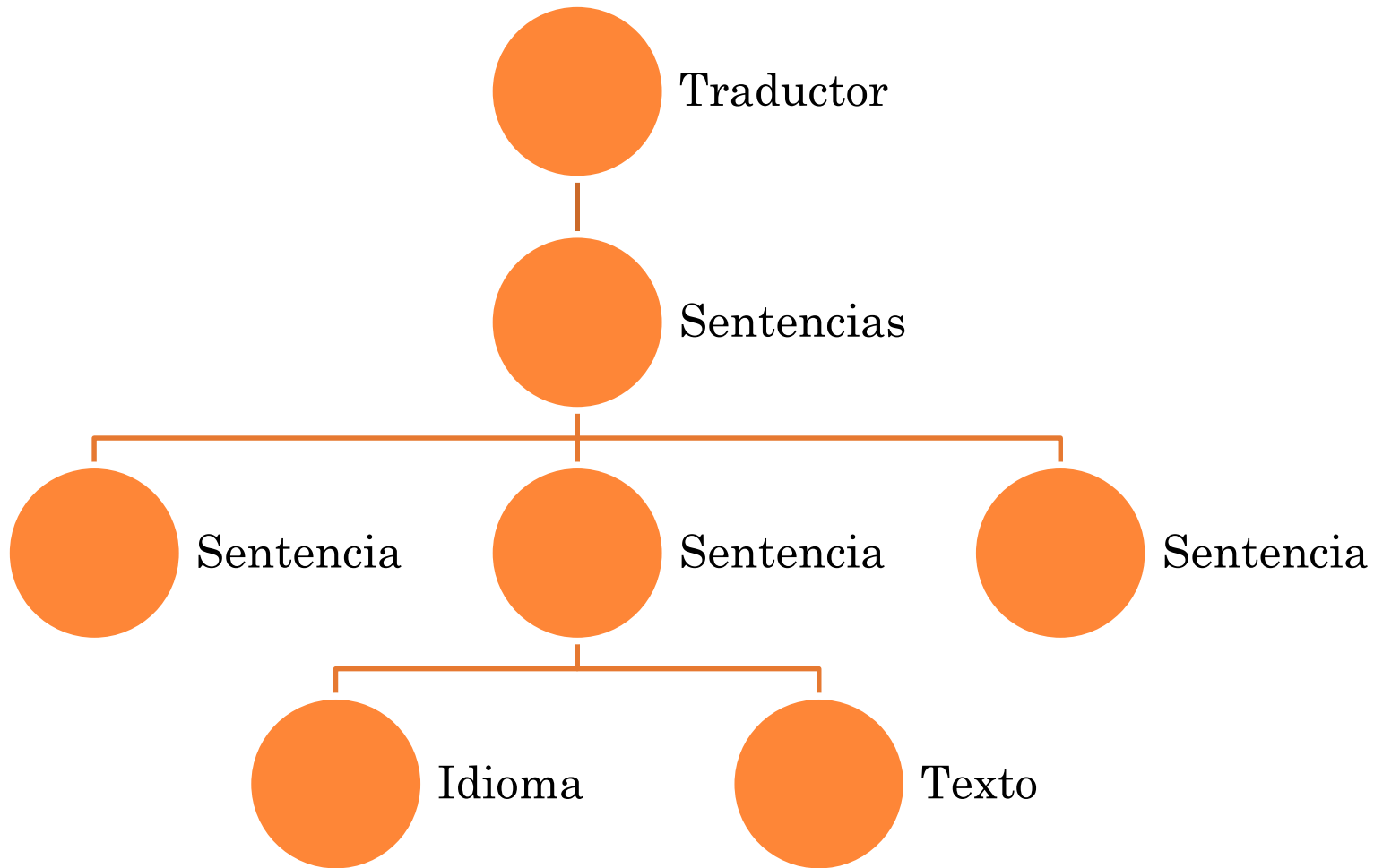
$\text{sentencia} ::= \text{LANG idioma}$
 texto SALTO

$\text{texto} ::= \text{TEXTO} \mid \text{texto}$
 TEXTO

$\text{idioma} ::= \text{ES} \mid \text{ENG}$



AST: EJEMPLO



CREACIÓN CLASE NODOAST

- Objeto padre que contendrá la información básica de cada uno de los nodos de nuestro árbol.
- La información básica en nuestro caso es un Symbol del paquete CUP.
- Además guardamos sus antecesor y precedente para mejorar la muestra de errores.
- Todos nuestros nodos (Traductor, Sentencias, etc.) deberán heredar de este nodo padre.



CREACIÓN CLASE NODOAST

```
private static final long serialVersionUID = -2300536436799840750L;  
private Symbol sym_;
```

```
public NodoAST(Symbol s){  
    sym_ = s;  
}
```

```
public Symbol getSymbol() {  
    return sym_;  
}
```

```
public String toString(){  
    if(sym_ != null){  
        return "[" + sym_.left + ":" + sym_.right + "];"  
    }else{  
        return "[desconocida]";  
    }  
}
```



VARIABLE RESULT

- La variable RESULT es usada por CUP para devolver el valor al padre.
 - Básicamente RESULT devuelve el valor asignado al no terminal del lado derecho. Este valor debe ser el mismo tipo de dato que el no terminal.
- Al acabar de parsear todo nuestro programa nuestro parser tendrá dentro de la propiedad value el último valor pasado a la variable RESULT.
 - `Symbol parse_tree = null;`
 - `Traductor ast = (Traductor) (parse_tree.value);`



CAMBIOS EN EL ARCHIVO CUP

- Importamos el paquete donde hayamos creado nuestros nodos AST.
 - `import ast.*;`
- Cambios en el Main:
 - `Traductor ast = (Traductor) (parse_tree.value);`
 - `System.out.println (ast.toString());`
- Tipamos nuestros terminales y no terminales.
 - `Terminal String ES, ENG, TEXTO;`
 - `non terminal Traductor S;`
 - `non terminal Idioma idioma;`
 - `non terminal Texto texto;`

