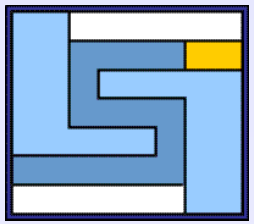
UNIVERSIDAD DE SEVILLA

E. T. S. INGENIERÍA INFORMÁTICA

LENGUAJES Y SISTEMAS INFORMÁTICOS



PRÁCTICAS DE LABORATORIO
ANÁLISIS SINTÁCTICO (1)

LENGUAJES FORMALES Y AUTÓMATAS

CURSO 2006/2007

¿Qué es el análisis sintáctico y qué es la herramienta CUP?

El **análisis sintáctico** consiste en identificar en un texto aquellas cadenas que se ajustan a (encajan en) unos conjuntos de reglas sintácticas, las cuales constituyen una **gramática**.

CUP (**C**onstructor of **U**seful **P**arsers) es una herramienta que, a partir de la especificación de una gramática, es capaz de generar automáticamente un analizador sintáctico (o *parser*) en Java para dicha gramática.

Un ejemplo

Especificación del analizador sintáctico (entrada para CUP)

```
terminal NUM, IDENT;
non terminal balanceados;

balanceados ::= NUM balanceados IDENT
        |
        ;
```

Especificación del analizador léxico (entrada para JFlex)

```
import java_cup.runtime.*;

%%
%class Lexer
%unicode
%cup
Blanco = [ \n\r\t\f]
Numero = 0 | [1-9][0-9]*
Identificador = [a-zA-Z][a-zA-Z0-9]*
%%

{Blanco}+   {;}
{Numero} {return new Symbol (sym.NUM);}
{Identificador}  {return new Symbol (sym.IDENT);}
.       {System.out.println ("Error léxico");}
```

Clase Java principal

```
import java.io.FileReader;
public class Principal
{
 public static void main (String [] args)
 {
   if (args.length == 0)
       System.out.println ("Hay que especificar un
       fichero de entrada.");
   else try{
            FileReader f= new FileReader (args[0]);
            Lexer anLexico= new Lexer (f);
            Parser anSintactico= new Parser (anLexico);
            Object resultado= anSintactico.parse().value;
         }catch (Exception e){
                 System.out.println ("Excepción");}
 }
}
```

Este analizador reconoce entradas tales como:

51 20 1 pera coco piña

25 108 azul rojo

12 delfín

Pero no:

51 20 pera

Estructura de una especificación para CUP

Zona de declaraciones *import* y *package*

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

Zona de componentes de código de usuario:
Permite que el usuario pueda personalizar el código del *parser* generado (p.e. cambiar el cuerpo de algunos métodos o incluir otros nuevos).

```
// Esta zona se estudiará más adelante.
```

Zona de declaración de símbolos terminales y no terminales

```
terminal NUM, IDENT;
non terminal balanceados;
```

Zona de declaraciones de precedencia y asociatividad: ayudan a resolver el análisis sintáctico en caso de que tengamos una gramática ambigua.

```
/* Cero o más sentencias precedence (se estudiarán más
adelante. */
```

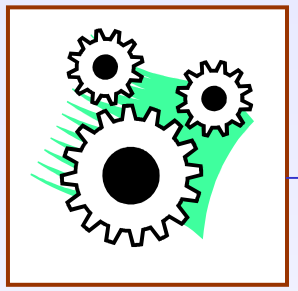
Zona de la gramática:
Contiene un conjunto de reglas (también llamadas *producciones*). La parte izquierda de cada regla debe ser un símbolo no terminal. La parte derecha de cada regla puede contener tanto terminales como no terminales, y puede tener asociada una acción (escrita en Java y enmarcada entre **{:** y **:}**). El símbolo inicial se considera que es el no terminal de la parte izquierda de la primera regla, a menos que se indique uno explícitamente con la construcción **start with** *no-terminal;* en la zona de terminales y no terminales

```
balanceados ::= NUM balanceados IDENT
              |
              ;
```

¿Cómo se ejecuta CUP?

Ejecución de CUP:

```
java java_cup.Main AnalizadorSintactico.cup
```



Produce

AnalizadorSintactico
...
parse () ...

Fichero: `Main.class`

Fichero: `AnalizadorSintactico.java`

sym
Declaración de atributos constantes para los tokens.

Fichero: `sym.java`

Compilación del analizador sintáctico:

```
javac AnalizadorSintactico.java
```

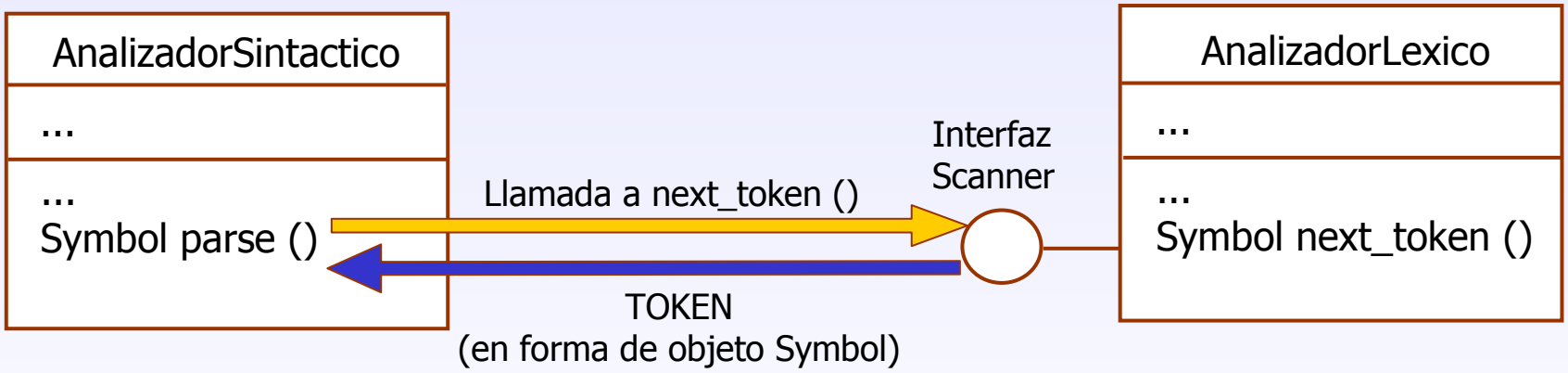
Fichero: `AnalizadorSintactico.class`

Compilación: `javac sym.java`

Fichero: `sym.class`

¿Cómo se comunica el analizador sintáctico con el analizador léxico?

El analizador sintáctico llama al método next_token () del analizador léxico cada vez que necesita que se reconozca el siguiente token de la entrada.



La clase Symbol es también utilizada internamente por el analizador sintáctico para representar tanto los símbolos terminales (*tokens*) como los no terminales.

Patrones sintácticos

Para simplificar el diseño de gramáticas definiremos un conjunto de **patrones sintácticos**, de manera que podamos crear una gramática utilizando dichos patrones como componentes.

Patrones no recursivos:

- Agregado.
- Opción.

Patrones recursivos:

- Lista simple (es decir, no anidada)
- Lista anidada

Patrones no recursivos

Patrón	Esquema de reglas (sigue la notación vista en teoría)	Ejemplo
Agregado	$A \rightarrow X_1 X_2 \ldots X_k$	asig → IDENT ‘=’ expr ‘;’
Opción	$A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_k$	sentencia → asig | condicional | iteracion

(Block.cup) Escriba un analizador sintáctico que reconozca la siguiente gramática:

Begin_Block
<cuerpo>
End_Block

Donde <cuerpo> puede ser una de las siguientes opciones:

a) **Msg** seguido de un numero, dos identificadores y una cadena.
b) **Ack** seguido de un numero y dos identificadores.
c) **Ready** seguido de un identificador o de un número.
d) **Alarm** seguido de un identificador y una cadena.

Patrón recursivo: Lista Simple (1)

Caso	Esquema simplificado (*)	Ejemplo
Sin separadores y sin delimitadores	$A \rightarrow a\ A \mid a$ (*no admite* λ)	numeros → NUM numeros | NUM
	$A \rightarrow a\ A \mid \lambda$ (*sí admite* λ)	texto → CADENA texto | λ

(*) En general, en vez de a podemos tener una secuencia de símbolos terminales y no terminales α (por ejemplo, otro patrón).

(ListaNumeros.cup) Escriba un analizador sintáctico que reconozca listas (posiblemente vacías) formadas por números enteros (sin signo).

(Transmission.cup) Escriba un analizador sintáctico que reconozca registros de transmisiones en un determinado protocolo. Un registro comienza con la palabra clave **Begin_Transmission** y finaliza con **End_Transmission.** Entre ambas aparece una lista no vacía de elementos **Begin_Block ... End_Block** (véase el ejercicio **Block.flex**).

Patrón recursivo: Lista Simple (2)

Existen muchas variantes de la lista, como por ejemplo:

- Con separadores y sin delimitadores: azul / verde / rojo
- Sin separadores y con delimitadores: @ azul verde rojo @
- Con separadores y con delimitadores: @ azul / verde / rojo @

En cada caso se puede admitir la lista vacía o no.

A continuación haremos algunos ejercicios de estos casos.

(ListaMinusculas.cup) Escriba un analizador sintáctico que reconozca cualquier lista (posiblemente vacía) formada por letras minúsculas que pueden estar separadas tanto por el símbolo # como por el -.

(Ticket.cup) Escriba un analizador sintáctico que reconozca cualquier ticket de compra. Un ticket está formado por líneas. Cada línea está formada por el nombre de un producto (entrecomillado) seguido del número de unidades compradas (número natural) y del importe parcial (en euros). Las líneas están separadas por retornos de carro.
Nota: cualquier ticket tiene, al menos, un producto.

(Rutas.cup) Escriba un analizador sintáctico que reconozca nombres de ficheros con sus rutas en MS-DOS. Por ejemplo: C:\ejemplo\leeme.txt . Supóngase que los nombres de directorios y archivos constan de un identificador o de un identificador seguido de un punto y de otro identificador (que hace las veces de extensión).

(LlamadaFuncion.cup) Escriba un analizador sintáctico que reconozca las llamadas simples a funciones en un lenguaje de programación. Una llamada tiene el formato:

identificador (*parámetros*)

donde *parámetros* es una secuencia (posiblemente vacía) de números naturales e identificadores, separados por comas.

Patrón recursivo: Lista Anidada

Una lista anidada es aquella en la que alguno de sus elementos es a su vez una lista. Podemos tener combinaciones de los casos vistos antes.
Por ejemplo:

(rojo [alto bajo] verde)

45 – 12 – 23 – @ a b c @ – 76 – @ x y z @ – 10

(LlamadaFuncion2.cup) Escriba un analizador sintáctico que reconozca las llamadas a funciones en un lenguaje de programación. Una llamada tiene el formato:

identificador (*parámetros*)

donde *parámetros* es una secuencia (posiblemente vacía) de números naturales, identificadores, o llamadas a funciones, separados por comas. Por ejemplo:

```
calculaCoordenadas (3, var1, alfa (1, 2))

calculaCoordenadas (99, 12, calculaCoordenadas (a, b, c))
```