UNIVERSIDADE DA CORUÑA

**Departamento de Computación**

# COMPAS (COMpiler for PArsing Schemata): Manual Resumido de Usuario

CARLOS GÓMEZ RODRÍGUEZ MIGUEL A. ALONSO

6 de diciembre de 2008

# Índice

# 1. Introducción

COMPAS (COMpiler for PArsing Schemata) es un sistema que permite obtener automáticamente implementaciones eficientes de algoritmos de análisis sintáctico, a partir de especificaciones formales en forma de *esquemas de análisis sintáctico* [6].

La intención de este manual de usuario es proporcionar una guía de uso del sistema, incluyendo toda la información necesaria para utilizarlo para transformar esquemas de análisis sintáctico en implementaciones de los algoritmos de análisis correspondientes, así como para ejecutarlos sobre oraciones y gramáticas concretas. Este manual está dirigido al uso práctico del software, y no a la teoría subyacente. Se presupone, pues, que el usuario que quiera diseñar e implementar analizadores con COMPAS debe estar previamente familiarizado con la teoría del análisis sintáctico en general, y de los esquemas de análisis sintáctico en particular. Como referencia básica sobre este formalismo, se recomienda consultar [6]. Para una descripción de alto nivel del sistema COMPAS que incluye una introducción a los esquemas de análisis sintáctico, véase [3].

La web de COMPAS, donde se puede consultar información actualizada sobre el sistema, es [2].

# 2. Condiciones de uso



# 3. Requisitos del sistema

El sistema COMPAS está escrito en lenguaje Java, y por lo tanto se puede ejecutar en cualquier sistema para el que haya disponible una máquina virtual Java (JVM) estándar. En la fecha de escritura de este manual, estos sistemas incluyen Windows (98, 2000, XP, Vista), Linux, Mac OS X y Solaris.

Los requisitos para la ejecución del sistema son los siguientes:

- Debe estar instalado un entorno de ejecución Java (JRE) que implemente la versión 1.4 o superior de las APIs de *Java 2 Standard Edition* (J2SE) [4]. Al funcionar sobre la máquina virtual Java, el sistema podrá ejecutarse en cualquier plataforma que disponga de dicho JRE.

- El compilador no tiene requisitos significativos de memoria y CPU, por lo que puede funcionar en cualquier sistema capaz de ejecutar el mencionado JRE. Los requisitos de memoria y CPU del código generado serán diferentes para cada algoritmo y variarán según el tipo de gramáticas y de oraciones que se analicen.
- Es necesario un compilador de lenguaje Java para compilar las implementaciones generadas y obtener versiones ejecutables.
- Aunque no es estrictamente necesario, es recomendable para mayor comodidad tener instalado el sistema de construcción Apache Ant [1], que permitirá compilar con mayor facilidad el código generado.

## 4. Compilación de un esquema

El compilador de esquemas de análisis sintáctico viene precompilado en esta distribución, en el directorio `build/` (el código fuente completo también se incluye en la distribución, en el directorio `src/`).

Con el propósito de facilitar el uso del compilador de esquemas, se han proporcionado dos scripts que lo ejecutan bajo sistemas Windows y UNIX: `generate.bat` y `generate.sh`. En ambos casos, para ejecutar el compilador debemos situarnos primero en el directorio `build/`, que contiene estos scripts. Para que los scripts funcionen, el directorio que contiene los ejecutables de la máquina virtual Java (JRE) debe formar parte de la variable de entorno PATH del sistema. Normalmente, las instalaciones de Java se encargan automáticamente de incluir la máquina virtual en el PATH.

La sintaxis para ejecutar el script bajo Windows es:

```
generate <schemaFile>[-o <outputDir>] [-g <gClass>]
```

y, en sistemas Unix, es:

```
./generate.sh <schemaFile>[-o <outputDir>] [-g <gClass>],
```

donde:

- El parámetro obligatorio `schemaFile` representa la ruta (absoluta o relativa) al fichero de esquema de análisis sintáctico que se quiere compilar. Se proporcionan varios esquemas de ejemplo en el directorio `schemata/`.
- El parámetro opcional `outputDir` representa la ruta (absoluta o relativa) al directorio en el que se escribirá el código generado y todos los ficheros asociados necesarios para compilarlo y ejecutarlo. En el caso de que no se especifique este último parámetro, su valor por defecto será la ruta relativa `generated/`.
- El parámetro opcional `gClass` es el nombre cualificado (es decir, incluyendo paquete) de la clase de gramáticas que se usará para los ficheros de esquema que no especifiquen una. Por defecto, este parámetro toma el valor `grammar.ContextFreeGrammar`, correspondiendo a una clase para gramáticas independientes del contexto.

A modo de ejemplo, supongamos que tenemos el esquema que genera árboles sintácticos según el algoritmo de Earley en un fichero `c:\schemata\OptimizedEarleyWithTree.sch`. Aquí se muestra el fichero con todas las definiciones de los elementos que aparecen en el esquema; aunque estas definiciones se podrían omitir si están en el fichero de definiciones globales (`elements.txt`):

```
@begin_elements
element.Symbol:nonGroundFromString:[A-RT-Za-ho-z]
element.RuleWrapper:fromString:[A-Za-z \.]+->[A-Za-z \.]*
element.IntElement:nonGroundIntElement:#[i-n]
element.StringPosition:nonGroundFromString:[i-n]
element.IntElement:groundIntElement:#[0-9]+
element.StringPosition:groundFromString:[0-9]+
element.SumOfIntegersExpression:fromString:#[0-9i-k\+\-#]+
element.SumOfPositionsExpression:fromString:[0-9i-k\+\-]+
element.SymbolSequence:fromString:alpha
element.SymbolSequence:fromString:beta
element.SymbolSequence:fromString:gamma
element.Dot:fromString:\.
element.Symbol:fromString:S
element.SentenceLengthExpression:fromString:length
element.unification.Term:groundFromString:%%empty
element.unification.Term:nonGroundFromString:%arbol1
element.unification.Term:nonGroundFromString:%arbol2
element.unification.Term:nonGroundFromString:%arbolfinal
element.unification.TermConstructorExpression:fromString:
  %%[A-Za-z0-9 \(\)\[\]\-\>\.;%]*
element.unification.TermConstructorExpression:addTreeExpressionFromString:
  %u[A-Za-z0-9 \(\)\[\]\-\>\.;%]*
element.unification.TermConstructorExpression:fromString:%%[A-Za-z0-9\(\);%]*
@end_elements

@step OptEarleyInitter
----------------------------- S -> alpha
[ S -> . alpha , 0 , 0 , %%S ]

@step OptEarleyScanner
[ A -> alpha . a beta , i , j , %arbol1 ]
[ a , j , j+1 ]
---------------------------------
[ A -> alpha a . beta , i , j+1 , %%%arbol1(%%a) ]

@step OptEarleyCompleter
[ A -> alpha . B beta , i , j , %arbol1 ]
[ B -> gamma . , j , k , %arbol2 ]
---------------------------------
[ A -> alpha B . beta , i , k , %%%arbol1(%arbol2) ]

@step OptEarleyPredictor0
```

```
 [ A -> alpha . B beta , i , j , %arbol1 ]
-------------------------------- B ->
 [ B -> . , j , j , %%B ]

 @step OptEarleyPredictor1
 [ A -> alpha . B beta , i , j , %arbol1 ]
-------------------------------- B -> C
 [ B -> . C , j , j , %%B ]

 @step OptEarleyPredictor2
 [ A -> alpha . B beta , i , j , %arbol1 ]
-------------------------------- B -> C D
 [ B -> . C D , j , j , %%B ]

 @step OptEarleyPredictor3
 [ A -> alpha . B beta , i , j , %arbol1 ]
-------------------------------- B -> C D E gamma
 [ B -> . C D E gamma , j , j , %%B ]

 @goal [ S -> alpha . , 0 , length , %arbolfinal ]
```

Para compilar este esquema, ejecutamos:

```
generate.bat c:\schemata\earleyOptTree.sch -o
c:\generatedparsers\earleytree\
```

y obtenemos la siguiente salida:

```
(...)
Generating code for deductive step 1...
...generated [file: c:\generatedparsers\earleytree\src\schema\step\
 SP_OptEarleyInitterStep.java, size: 2392 characters]
Generating code for deductive step 2...
...generated [file: c:\generatedparsers\earleytree\src\schema\step\
 SP_OptEarleyScannerStep.java, size: 10583 characters]
Generating code for deductive step 3...
...generated [file: c:\generatedparsers\earleytree\src\schema\step\
 SP_OptEarleyCompleterStep.java, size: 12806 characters]
Generating code for deductive step 4...
...generated [file: c:\generatedparsers\earleytree\src\schema\step\
SP_OptEarleyPredictor0Step.java, size: 5142 characters]
Generating code for deductive step 5...
...generated [file: c:\generatedparsers\earleytree\src\schema\step\
SP_OptEarleyPredictor1Step.java, size: 5304 characters]
Generating code for deductive step 6...
...generated [file: c:\generatedparsers\earleytree\src\schema\step\
SP_OptEarleyPredictor2Step.java, size: 5466 characters]
Generating code for deductive step 7...
...generated [file: c:\generatedparsers\earleytree\src\schema\step\
SP_OptEarleyPredictor3Step.java, size: 6285 characters]
Generating code for item indexing...
```

```
...generated [file: c:\generatedparsers\earleytree\src\schema\item\
ItemHandler.java, size: 85638 characters]
Generating code for deductive step indexing...
...generated [file: c:\generatedparsers\earleytree\src\schema\step\
StepHandler.java, size: 16099 characters]
Generating code for goal condition 8...
...generated [file: c:\generatedparsers\earleytree\src\schema\step\
SP_Goal1.java, size: 3225 characters]
Generating code for step factory...
...generated [file: c:\generatedparsers\earleytree\src\schema\step\
StepFactory.java, size: 833 characters]
Copying common files...
...copy successfully completed.
```

La salida nos informa de los ficheros que han sido generados, así como de su tamaño. Nótese que no aparecen mencionados todos los ficheros que se obtienen como salida de la compilación; sino sólo aquéllos que se generan específicamente para cada esquema. Otros ficheros, que son genéricos y simplemente se copian, no se mencionan explícitamente en esta salida.

Asimismo, si existe algún error en el fichero de esquema, la salida del compilador proporcionará información detallada acerca del error, útil para corregirlo. Por ejemplo, si nos hubiésemos saltado una coma en el paso deductivo `OptEarleyCompleter`, escribiéndolo de la siguiente manera:

```
@step OptEarleyCompleter
[ A -> alpha . B beta , i , j %arbol1 ]
[ B -> gamma . , j , k , %arbol2 ]
---------------------------------
[ A -> alpha B . beta , i , k , %%%arbol1(%arbol2) ]
```

(donde falta la coma entre j y %arbol1), obtendríamos el siguiente mensaje de error:

```
Schema File: ../schemata/OptimizedEarleyWithTree.sch
parser.sparser.ParseException: La cadena j  %arbol1 no se corresponde con
ninguno de los formatos definidos para elementos.
        at parser.sparser.SchemaParser.Element(SchemaParser.java:470)
        (...)
        at launcher.Main.main(Main.java:94)
```

Que, como vemos, nos apunta a la parte del esquema que hemos escrito mal. Si dejáramos sin escribir el corchete que cierra el ítem
`[ A -> alpha . B beta , i , j , %arbol1 ]`, el error sería:

```
Schema File: ../schemata/OptimizedEarleyWithTree.sch
parser.sparser.ParseException: Encountered "[" at line 38, column 1.
Was expecting one of:
    "]" ...
    "," ...
```

```
        at parser.sparser.SchemaParser.generateParseException
        (SchemaParser.java:826)
        (...)
        at launcher.Main.main(Main.java:94)
```

Diciéndonos explícitamente que encontró un carácter `[` donde esperaba encontrar un `]` (fin de ítem) o una coma (continuación de ítem), e indicándonos en qué línea se encuentra el error. Todos los errores de formato que podamos cometer en el esquema de entrada producen mensajes de alguno de estos dos tipos al ejecutar el compilador de esquemas.

## 5. Compilación del código generado

El contenido del directorio `c:\generatedparsers\earleytree\` tras la compilación anterior consiste en un subdirectorio `src`, que contiene el código Java de la implementación completa y autocontenida del algoritmo de Earley que se ha generado, y un fichero `build.xml` que sirve para compilar mediante el sistema de construcción `ant`.

Para compilar el código Java generado, simplemente tenemos que situarnos en el directorio `c:\generatedparsers\earleytree\` y escribir:

```
ant
```

Obteniendo una salida como ésta:

```
Buildfile: build.xml

init:
    [mkdir] Created dir: C:\generatedparsers\earleytree\build

compile:
    [javac] Compiling 45 source files to C:\generatedparsers\earleytree\build

dist:
    [mkdir] Created dir: C:\generatedparsers\earleytree\dist
      [jar] Building jar: C:\generatedparsers\earleytree\dist\
            MyProject-20050619.jar

BUILD SUCCESSFUL
Total time: 5 seconds
```

En el caso de no tener instalado el sistema de compilación ant (que se puede descargar gratuitamente de `http://ant.apache.org/`, el código generado también puede compilarse a mano mediante cualquier compilador de Java.

# 6. Ejecución del código generado

La compilación del código generado mediante la herramienta `ant` genera dos directorios más dentro de `C:\generatedparsers\earleytree\`, además del `src` que ya teníamos: el directorio `build`, que contiene los "bytecodes" de las clases Java que implementan el algoritmo de Earley, y el directorio `dist`, que contiene las mismas clases empaquetadas en un fichero `.jar` (Java Archive).

Para ejecutar el código generado, COMPAS proporciona dos interfaces distintos: un interfaz gráfico y uno basado en línea de comandos. El interfaz gráfico es el más cómodo y sencillo de usar, y por lo tanto adecuado para hacer pruebas y ejecuciones puntuales de algoritmos. El interfaz de línea de comandos proporciona funcionalidad más avanzada, como analizar muchas oraciones de un mismo fichero con un solo comando, o incluso ejecutar miniprogramas (scripts) que pueden contener distintas órdenes de análisis sintáctico referidas a diferentes ficheros. Por lo tanto, el interfaz de línea de comandos es el más completo, proporcionando funcionalidad útil para tareas de investigación como experimentos en los que haya que analizar un gran número de oraciones.

A continuación veremos cómo podemos ejecutar el algoritmo que hemos generado a partir del esquema Earley, tanto mediante el interfaz gráfico como con la línea de comandos.

## 6.1. Ejecución mediante interfaz gráfico

Para ejecutar el interfaz gráfico que nos permitirá probar el algoritmo de Earley, hacemos

```
cd c:\generatedparsers\earleytree\build
```

y a continuación:

```
java test.ParserInterface
```

Con lo cual se nos abrirá una sencilla ventana como la de la figura 1, permitiendo ejecutar el analizador sintáctico generado.

Nótese que el interfaz de usuario es intencionadamente sencillo, pues su finalidad es simplemente la de facilitar al diseñador de algoritmos de análisis sintáctico la prueba y prototipado de diferentes esquemas. En un sistema de procesamiento del lenguaje natural real, el análisis sintáctico se ejecuta en "background", entregando su salida a (o interaccionando con) un analizador semántico, que nos proveería del significado de las frases a analizar y que a su vez entregaría su salida a (o interaccionaría con) otros módulos de análisis del lenguaje. Para este tipo de interacciones, o para experimentos complejos que requieran el análisis de grandes cantidades de oraciones, es más adecuado el interfaz basado en línea de comandos que se describe en la siguiente subsección.

Lo primero que debemos hacer para utilizar el analizador es cargar una gramática. Para ello, podemos teclear directamente la ruta al fichero de gramática en el campo de texto "Gramática" y a continuación pulsar "Cargar". Otra alternativa, más cómoda en la mayoría de los casos, es utilizar el botón "Examinar...", que abrirá el diálogo que se muestra en la figura 2, permitiendo seleccionar el fichero de gramática. Una vez

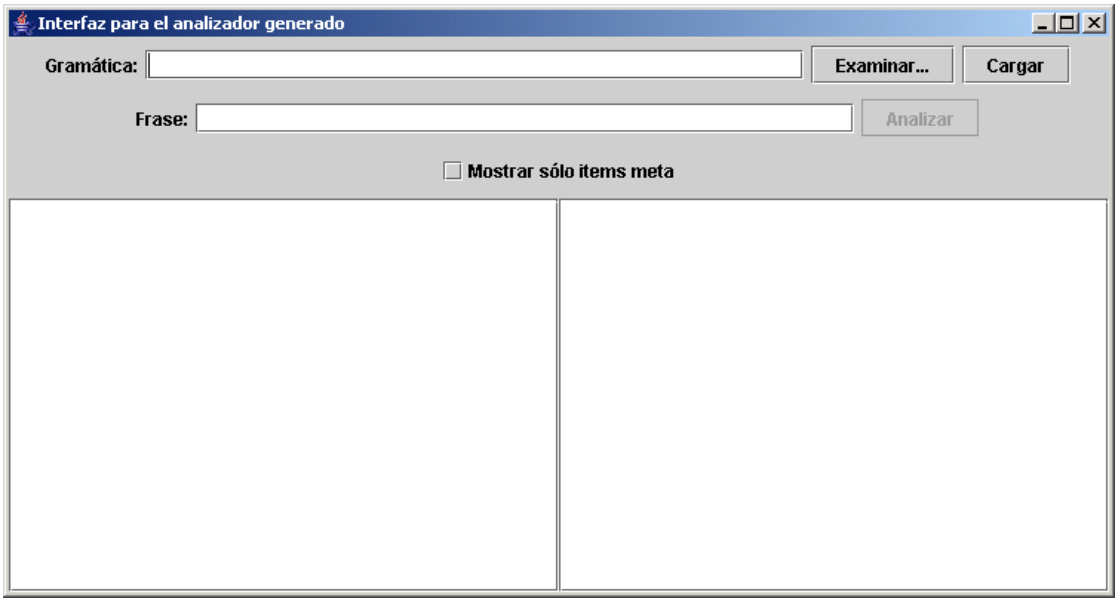


Figura 1: Interfaz del analizador generado.

seleccionado un fichero, su ruta aparecerá en el campo "Gramática" y podremos pulsar "Cargar" directamente para leer la gramática.

En el directorio `grammars` de la distribución se incluyen diversas gramáticas que se pueden utilizar para las pruebas. Por ejemplo, una gramática independiente del contexto muy sencilla es `TelescopioGrammar`, una gramática para probar con la oración ambigua "Juan vio a un hombre con un telescopio".

En la figura 3 se muestra la ventana del programa después de cargar esta pequeña gramática de ejemplo. En el área de texto que ocupa la parte inferior izquierda de la ventana aparece información de estado y mediciones relacionadas con la carga de la gramática. La función de esta área de texto siempre va a ser mostrar información de este tipo, mientras que la que ocupa la parte inferior derecha de la ventana mostrará los resultados de los análisis sintácticos, es decir, los ítems obtenidos.

La información que se muestra al cargar la gramática es la siguiente:

- `Preparando algoritmo para gramática...`: este mensaje se muestra cuando el fichero de gramática ha sido leído con éxito, y se va a utilizar la gramática para instanciar los pasos deductivos.
- `18 pasos deductivos instanciados en 250 ms`: se muestra tras instanciar los pasos deductivos, indicando cuántas instancias se han creado y el tiempo invertido en la operación.
- `8 ítems computados en 90 ms`: se muestra tras la llamada que se hace a la máquina deductiva para calcular todos los ítems posibles antes de analizar frases.

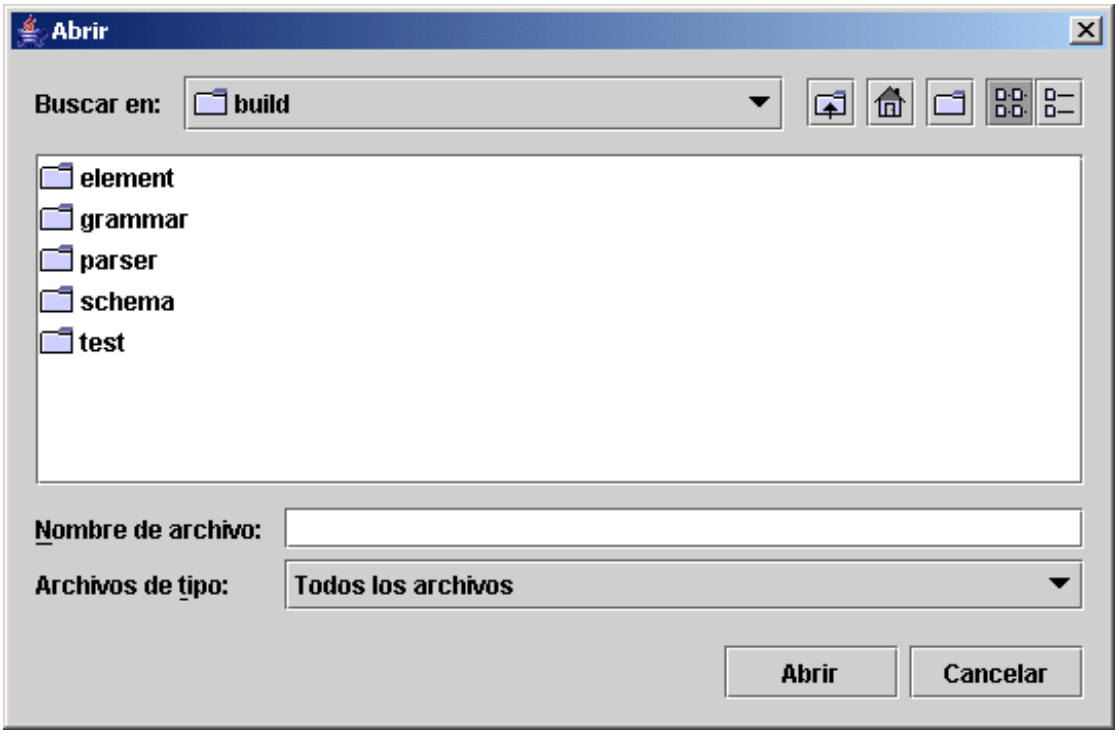


Figura 2: Diálogo para seleccionar un fichero de gramática.

En este caso, se precalculan ocho ítems, correspondientes a la ejecución de los pasos INITTER y PREDICTOR del algoritmo de Earley.

- `Gramática preparada.`: indica que el proceso de carga de la gramática ha terminado con éxito, y que ya se pueden analizar frases con ella. Nótese que, en el caso de gramáticas grandes (como la gramática del corpus Susanne, incluida en la distribución) utilizadas con algoritmos donde se precalculen muchos ítems (como el algoritmo Left-Corner), este mensaje y el anterior pueden tardar segundos en aparecer.

Una vez terminada de cargar la gramática, ya podemos utilizar el algoritmo de análisis sintáctico para analizar frases. Para ello, basta con teclear la frase que se desea analizar en el campo "Frase:" y pulsar "Analizar". Los ítems obtenidos como resultado se mostrarán en el área de texto de la parte inferior derecha de la ventana. Antes de pulsar "Analizar" puede ser interesante cambiar la opción de la casilla "Mostrar sólo ítems meta". Si esta casilla no está marcada, el área de texto mostrará todos los ítems que se obtengan en el proceso deductivo de análisis sintáctico; mientras que si está marcada sólo aparecerán los ítems finales, es decir, los que cumplan las condiciones de meta (`@goal`), que representan análisis sintácticos completos para la oración de entrada.

Si analizamos la frase "Juan vio a un hombre con un telescopio" con la casilla "Mostrar sólo ítems meta", la ventana nos queda tal y como se muestra en la figura 4.

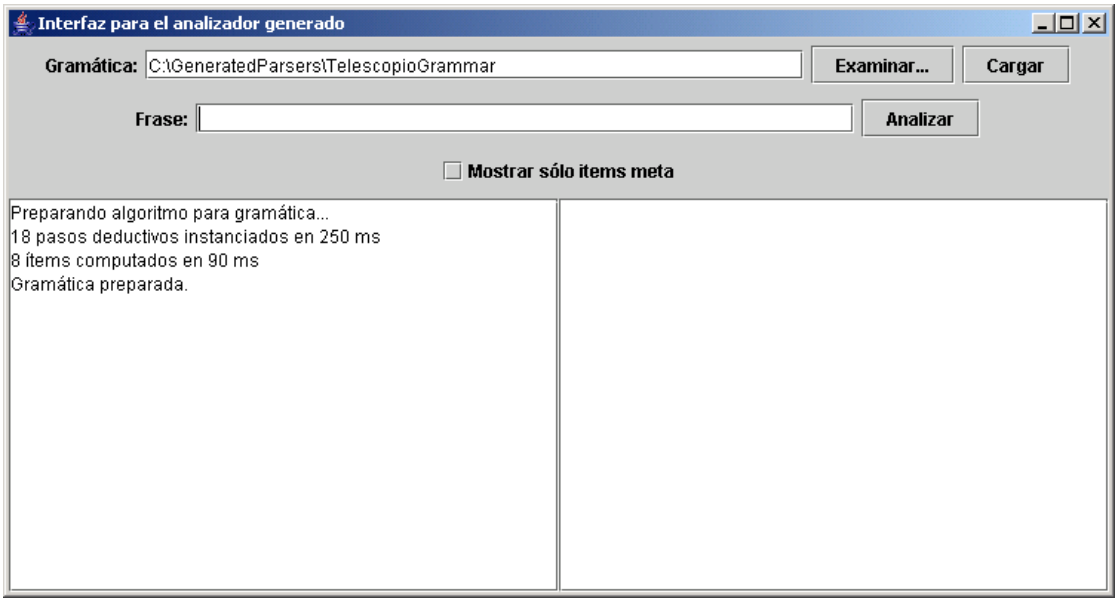


Figura 3: Estado de la interfaz tras cargar la gramática.

La información de estado que aparece en el área de texto izquierda es la siguiente:

- `ItemHandler restaurado en 0 ms`: este mensaje indica que la clase encargada de almacenar ítems (`ItemHandler`) ha sido restaurada al estado en que estaba inmediatamente después de cargar la gramática. Esta restauración de estado se hace siempre antes de analizar una frase, pues por un lado no nos interesa que los ítems que se calcularon en análisis anteriores sigan en el conjunto de ítems (pudiendo producir resultados incorrectos); pero por otro lado sí nos interesa que estén inicialmente presentes los ítems que se precalculan al cargar la gramática, sin tener que calcularlos otra vez. De ahí que almacenemos estos ítems en memoria, recuperándolos rápidamente antes de cada análisis.

- `82 ítems computados en 10 ms`: Este mensaje se muestra una vez terminado el análisis, cosa que puede llevar un tiempo significativo (dependiendo, obviamente, de la gramática, frase y algoritmo utilizados). Informa del tiempo de ejecución del algoritmo y del número total de ítems generado. Este cómputo de ítems incluye también los que se calculan al cargar la gramática (y que han sido simplemente copiados).

En el área de texto de la derecha se nos muestran los ítems finales obtenidos de la ejecución del algoritmo, que en este caso contienen los dos árboles sintácticos válidos que el algoritmo de Earley obtiene para la frase de entrada:

```
Ítems finales:
[[S,NP,VP,.],0,8,S(NP(N(Juan)))(VP(V(vio))(PP(Prep(a))(NP(Det(un))
```

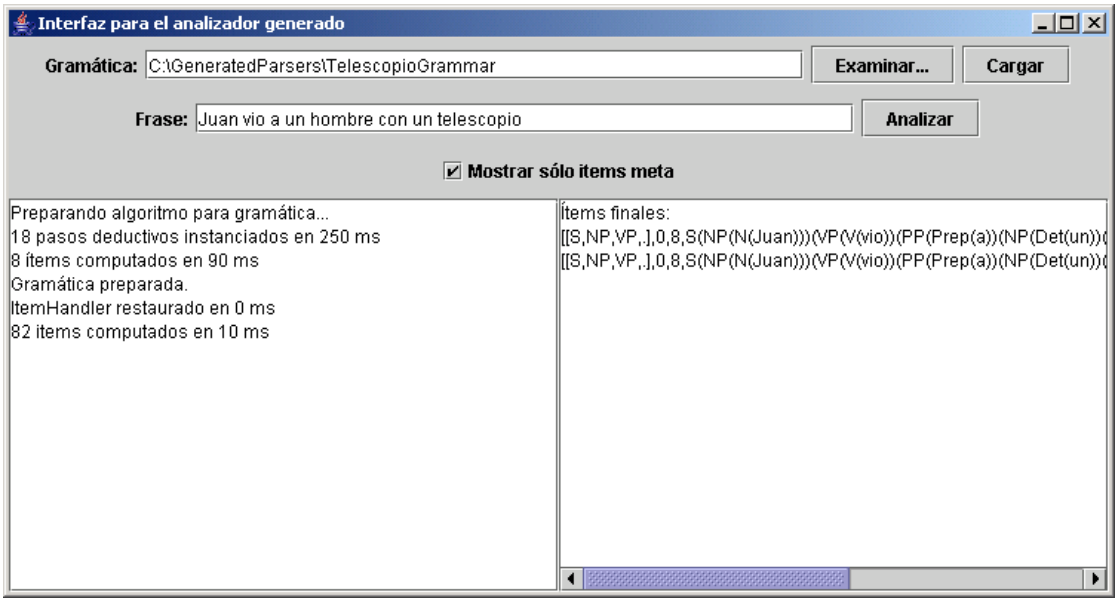


Figura 4: Estado de la interfaz tras analizar una frase.

```
  (N(hombre))))(PP(Prep(con))(NP(Det(un))(N(telescopio)))))]
[[S,NP,VP,.],0,8,S(NP(N(Juan)))(VP(V(vio))(PP(Prep(a))(NP(Det(un))
  (N(hombre))(PP(Prep(con))(NP(Det(un))(N(telescopio)))))))]
```

Si no hubiésemos marcado "Mostrar sólo ítems meta", habríamos obtenido una lista completa de los 82 ítems presentes en el `ItemHandler` tras la ejecución del algoritmo:

```
Ítems obtenidos:
[[S,.,NP,VP],0,0,S]
[[NP,.,N],0,0,NP]
(...)
[[VP,V,PP,.,PP],1,8,VP(V(vio))(PP(Prep(a))(NP(Det(un))(N(hombre))
  (PP(Prep(con))(NP(Det(un))(N(telescopio))))))]
[[S,NP,VP,.],0,8,S(NP(N(Juan)))(VP(V(vio))(PP(Prep(a))(NP(Det(un))
  (N(hombre))(PP(Prep(con))(NP(Det(un))(N(telescopio)))))))
```

La lista de ítems aparece en el orden en que han sido generados; pero esto no implica necesariamente que los ítems meta aparezcan al final (de hecho, en este caso concreto el último ítem es final pero el penúltimo no, y el otro ítem final aparece más arriba en la lista). En situaciones en las que sólo interesen los ítems meta, por lo tanto, será interesante marcar la opción "Mostrar sólo ítems meta" para no tener que buscarlos entre el resto de ítems generados.

## 6.2. Ejecución mediante línea de comandos

Si en lugar de utilizar el interfaz gráfico queremos hacer uso del interfaz de línea de comandos para ejecutar nuestro algoritmo de análisis, tendremos que hacer

```
cd c:\generatedparsers\earleytree\build
```

y a continuación:

```
java test.ParserConsole
```

Con lo cual se abrirá una línea de comandos, en la que podemos teclear distintas órdenes para utilizar nuestro analizador. Si introducimos `HELP`, obtendremos una lista de los comandos disponibles:

```
Valid commands:
LOAD <path-to-grammar-file>
PARSE [-fsn] <sentence>
EXEC <script-file-to-execute>
DUMP [-fn] <path-to-file-to-dump-results-to>
PARSEFILE <file-to-parse> <file-to-dump-results-to>
EXIT
```

La función de los comandos es la siguiente:

- `LOAD <fichero gramática>`: Carga la gramática del fichero dado, del mismo modo que se podía hacer con el campo de texto "gramática" en el interfaz gráfico. La consola nos proporcionará la misma información sobre el progreso de la carga de la gramática que aparecía en la parte izquierda de dicho interfaz.
- `PARSE [-fsn] <oración>`: Analiza sintácticamente la oración dada. Si no se especifica ninguna opción en el campo `[-fsn]`, el programa mostrará la lista de todos los ítems generados en el análisis. Si se especifica alguna de las opciones, el comportamiento es el siguiente:
  - Opción `-f`: se muestran sólo los ítems meta.
  - Opción `-n`: no se muestran ítems, simplemente se dan estadísticas sobre el número de ítems generados en el análisis y el tiempo invertido en él.
  - Opción `-s`: opción silenciosa, no se muestra nada por la consola. Nótese que los resultados se guardan (aunque no se muestren) y pueden volcarse a un fichero más adelante.
- `DUMP [-fs] <fichero>`: Vuelca los resultados de la última oración analizada a un fichero. Siempre se escriben en dicho fichero resultados estadísticos sobre tiempo de ejecución y número de ítems generados. Adicionalmente, y de forma análoga al comando `PARSE`:
  - Si no se especifica opción `-f` ni `-s`: se vuelca la lista completa de ítems generados en el análisis.
  - Opción `-n`: no se vuelca la lista de ítems.

- Opción `-f`: se vuelcan sólo los ítems finales.

Los ficheros generados con el comando `DUMP` tienen el siguiente aspecto:

```
BEGINNING OF TEST RESULTS
Test executed at: Fri Dec 05 17:11:18 GMT 2008
Grammar: /home/c/cg/cg204/compas/grammars/TelescopioGrammar
Sentence: el telescopio es bonito
Grammar init time (ms): 9
Parsing time (ms): 1
Grammar init items: 8
Total items: 12
Items generated during parsing time: 4
Item list:
[[S,.,NP,VP],0,0,S]
(...)
[bonito,3,4]
End of item list.
END OF TEST RESULTS
```

Donde se muestran la fecha y la hora a la que se ejecutó el análisis, la gramática y la frase utilizadas, los tiempos de inicialización de la gramática y de análisis (todos ellos mostrados en milisegundos), y la cantidad de ítems generados. Las líneas `BEGINNING OF TEST RESULTS` y `END OF TEST RESULTS` sirven para diferenciar fácilmente los resultados de un experimento y el siguiente si se vuelcan todos al mismo fichero, especialmente si los datos se van a leer con alguna herramienta para su tratamiento automático.

- `PARSEFILE <entrada><salida>`: Analiza línea por línea las frases contenidas en el fichero `<entrada>`, y vuelca los resultados al fichero `<salida>`.
- `EXEC <script>`: Ejecuta un fichero de script, que es un fichero que contiene una lista de órdenes válidas para el interfaz de línea de comandos de las implementaciones generadas por COMPAS. Es decir, una lista de órdenes `LOAD`, `PARSE`, `PARSEFILE`, etc. Este comando se puede utilizar para llevar a cabo experimentos sobre multitud de ficheros y gramáticas, volcando los resultados a ficheros, y pudiendo repetir dichos experimentos cuando se desee o con distintos analizadores (ya que todos los analizadores generados por COMPAS tienen la misma interfaz de línea de comandos).
- `EXIT`: Comando utilizado para salir de la interfaz de línea de comandos del analizador.

# 7. Ejemplos proporcionados con la distribución

La distribución de COMPAS incluye diferentes ejemplos de esquemas de análisis y de gramáticas, que se pueden utilizar para probar el sistema o como punto de partida para la elaboración de esquemas propios. Son los siguientes:

### Gramáticas:

- SusanneGrammar: Gramática completa del inglés extraída del corpus Susanne [5].
- SusanneChomsky: La misma gramática, en forma normal de Chomsky, para su uso en analizadores de tipo CYK.
- SusanneTestSetTerminals.txt: Conjunto de oraciones de prueba, generadas automáticamente para la gramática Susanne (aviso: esto NO es una gramática, aunque por conveniencia se incluya en el directorio correspondiente a gramáticas).
- TelescopioGrammar: Gramática muy simple para la oración ambigua "Juan vio a un hombre con un telescopio", que se puede ver en el ejemplo anterior.
- TelescopioGrammarChomsky: La misma gramática, transformada a forma normal de Chomsky.
- Parentesis: Gramática que define el lenguaje de las cadenas de paréntesis equilibradas o lenguaje de Dyck (donde los paréntesis se representan mediante las palabras L y R).
- ParenChomsky: La misma gramática, en forma normal de Chomsky.
- nvgrammar: Pequeña gramática "de juguete" que sirve para analizar oraciones de la forma nombre+verbo.
- toytag/trees.xml: Una gramática de adjunción de árboles sencilla, con frases nominales y verbales.
- testtag/: Gramáticas de adjunción de árboles automáticamente generadas, que pueden ser utilizadas para probar el rendimiento de los analizadores para este formalismo.

### Esquemas de análisis:

- CYKRecognizer.sch: Un analizador CYK para gramáticas independientes del contexto en forma normal de Chomsky.
- CYKVariant.sch: Una manera diferente de expresar el mismo analizador CYK, denotando las reglas de la gramática mediante ítems.

- CYKAnyGrammarRecognizer.sch: Lo mismo que CYKRecognizer; pero este esquema hace referencia a una clase de gramáticas que se encarga de transformar la gramática de entrada a la forma normal de Chomsky automáticamente, si es que no cumple ya las restricciones de esta forma normal.
- CYKWithTree.sch: Analizador CYK que genera árboles sintácticos para las oraciones.
- SimpleEarleyRecognizer.sch: Un reconocedor de tipo Earley, tal como aparece descrito en el libro de esquemas de análisis sintáctico de Sikkel [6].
- OptimizedEarleyRecognizer.sch: Una versión del reconocedor Earley que incorpora una optimización mediante técnicas de "unrolling", para que la indexación de ítems resulte un poco más rápida.
- OptimiedEarleyWithTree.sch: Un analizador estilo Earley que genera árboles sintácticos para las oraciones.
- LCStandard.sch: Un analizador de tipo left-corner, éste es el analizador llamado "LC" descrito en [6]. En esta implementación, se utilizan ítems para representar las relaciones left-corner usadas por el algoritmo.
- LCSimplifiedItems.sch: Un analizador left-corner optimiado, éste es el esquema que se llama "sLC" en [6].
- LCWithPredicates.sch: Una implementación alternativa del analizador left-corner donde se usan predicados, en lugar de ítems, para representar las relaciones left-corner.
- TopDown.sch: Un analizador descendente simple e ineficiente.
- EarleyNVPforXTAG.sch: Un analizador para gramáticas de adjunción de árboles, adaptado para su uso con la gramática inglesa XTAG, incluyendo unificación de estructuras de rasgos y varias características específicas para esta gramática.
- Lyon.sch: Analizador sintáctico con corrección de errores de Lyon, que usa una cola de prioridad como agenda.

# 8. Construyendo nuevos esquemas de análisis sintáctico

Hasta el momento, se ha mostrado el uso del sistema COMPAS mediante la compilación de los esquemas de ejemplo que se incluyen en su distribución. En esta sección se describe cómo crear un fichero de esquema compilable mediante COMPAS.

La notación que COMPAS utiliza para describir los esquemas es muy sencilla, coincidiendo prácticamente con la notación formal con la que se suelen definir en la teoría. Concretamente, la gramática EBNF que todo fichero de esquema debe seguir se muestra en la figura 8 (donde se ha utilizado la notación estándar que representa los constructos

```
Schema ::= [ ElementDefinitionList ] [ OptionList ]
    { StepName StepDescription } { @goal GoalDescription }
ElementDefinitionList ::=
    @begin_elements { ElementDefinition } @end_elements
ElementDefinition ::= element_definition
OptionList ::= { @begin_options Option @end_options }
Option ::= @option key value
StepName ::= @step ID
StepDescription ::= Antecedent Separator Conditions Consequent
GoalDescription ::= Antecedent
Antecedent ::= { ItemDescription }
Separator ::= { "-" }
Consequent ::= ItemDescription
ItemDescription ::= "[" ElementList "]"
ElementList ::= [ ElementWrapper { , ElementWrapper } ]
ElementWrapper ::= Element
Conditions ::= ElementList
Element ::= element
```

Figura 5: EBNF grammar for parsing schema files.

opcionales entre corchetes [ ], los que se repiten cero o más veces entre llaves { }, y el texto literal mediante letra negrita o entrecomillado).

Como se puede ver, existen dos símbolos en la gramática ("element" y "element_definition") que no están definidos. Esto se debe a que su definición podrá variar dependiendo de los elementos notacionales definidos por el usuario. Cuando el compilador de esquemas encuentre una cadena cualquiera sin espacios ni comas (y que no pueda ser confundida con otros elementos del fichero) en un lugar donde pueda ir una "element_definition", la interpretará como una expresión regular que define el conjunto de cadenas asociado a un tipo de elemento que puede aparecer en los esquemas. Cuando encuentre una cadena así donde pueda ir un "element", usará las expresiones regulares que aparezcan en esas definiciones de elemento (además de las que se incluyan en el fichero de configuración `elements.txt` del sistema) para decidir a qué clase de elemento se refiere la cadena.

Así, la estructura general del fichero de esquema es la siguiente:

- Una sección opcional, enmarcada entre los delimitadores `@begin_elements` y `@end_elements`, donde se llevan a cabo definiciones de elementos. Estas definiciones son asociaciones de expresiones regulares con clases y métodos Java representando elementos que puedan aparecer en los esquemas (bien elementos definidos por el usuario, o predefinidos de los que se incluyen con COMPAS).

- Una sección opcional, enmarcada entre los delimitadores `@begin_options` y `@end_options`, que se utilia para parametrizar el analizador sintáctico. Por defecto, las opciones se pueden utilizar para parametrizar el tipo de agenda o de máqui-

na deductiva que se utiliza en el código generado (por ejemplo, podemos poner `@option agendaClass agenda.PriorityQueueAgenda` para utilizar una cola de prioridad como agenda). El contenido de las líneas de opción también es accesible mediante una sencilla API desde el código generado, de manera que las clases definidas por el usuario pueden ser parametrizadas con estas opciones.

- Una serie de pasos deductivos, expresados en el formato

```
@step NombreDelPasoDeductivo
[ Antecedente1 ]
[ Antecedente2 ]
  ...
[ AntecedenteN ]
---------------- Condiciones Laterales
Consecuente
```

- Una serie de metas o ítems finales, expresados en el formato

```
@goal [ ÍtemFinal ]
```

El sistema compilará a código Java cualquier fichero de esquema que se ajuste a este formato. Nótese que las definiciones de elementos de los ficheros de esquema pueden hacer referencia no sólo a clases incluidas en la distribución de COMPAS; sino también a clases Java definidas por el propio usuario, proporcionando un mecanismo de extensibilidad que permite que cualquier tipo de objeto pueda aparecer en un esquema.

## Agradecimientos

Parcialmente financiado por: MEC y FEDER (TIN2004-07246-C03, HUM2007-66607-C04), Xunta de Galicia (PGIDIT07SIN005206PR, PGIDIT05PXIC-10501PN, PGIDIT05PXIC30501PN, INCITE08PXIB302179PR, INCITE08E1R104022ES, Rede Galega de Procesamento da Linguaxe e Recuperación de Información), Axudas para a Consolidación e Estruturación de Unidades de Investigación (Xunta de Galicia)

## Referencias

[1] Apache Ant. `http://ant.apache.org/`.

[2] COMPAS (COMpiler for PArsing Schemata) website. `http://www.grupocole.org/software/COMPAS/`.

[3] Carlos Gómez-Rodríguez, Jesús Vilares, and Miguel A. Alonso. A compiler for parsing schemata. *Software: Practice and Experience*, 2008. (DOI 10.1002/spe.904).

[4] Java Runtime Environment. `http://www.java.com/`.

[5] G. Sampson. The Susanne corpus, release 3, 1994.

[6] Klaas Sikkel. *Parsing Schemata — A Framework for Specification and Analysis of Parsing Algorithms*. Texts in Theoretical Computer Science — An EATCS Series. Springer-Verlag, Berlin/Heidelberg/New York, 1997.