Universidad Politécnica
de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Grado en Ingeniería informáticas

Trabajo Fin de Grado

Mejora de un Sistema de Autodocumentación Basado en Comentarios Legibles Mecánicamente y Aserciones.: LPDowner

Autor: Natalia Carpizo González
Tutor(a): Manuel Hermenegildo Salinas

Madrid, Junio,2020

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado
Grado en Ingeniería informáticas

Título: Mejora de un Sistema de Autodocumentación Basado en Comentarios Legibles Mecánicamente y Aserciones.: LPDowner

Junio,2020

Autor: Natalia Carpizo González
Tutor: Manuel Hermenegildo Salinas
Departamento Inteligencia Artificial
ETSI Informáticos
Universidad Politécnica de Madrid

Resumen

En el presente trabajo se describirá el desarrollo seguido para la creación de la herramienta **LPdowner**, encargada de transformar documentación de código entre el formato estilo *markup* y el estilo *markdown*, para el lenguaje de programación Ciao Prolog. La herramienta desarrollada permite tanto traducir entre aserciones de código en estilo *doccomments* como convertir de estilo markup a estilo markdown.

Esta memoria muestra tanto las decisiones tomadas para la implementación como las motivaciones por las que se ha llevado a cabo el trabajo. Finalmente, realizamos un análisis de los resultados obtenidos. A partir de éstos, determinamos por una parte la corrección de la transformación y por otra los aspectos que han funcionado mejor del programa y sus futuras ampliaciones. Al final de la tesis se adjunta también el manual de la herramienta.

Abstract

In this document we describe the steps followed in the development of **LPdowner**, a tool in charge of converting code documentation in *markup*-style format to *markdown*-style format, for the Ciao Prolog programming language. The tool developed allows both translation between assertions in *doccomments* style and converting between *markup* format and *markdown* format.

This report will show both the decisions that guided the implementation and the motivations behind the work completed. Finally, we carry out an analysis of the results obtained. From these, we determine on one hand the correctness of the transformation and on the other the aspects of the program that have worked best, as well as its future extensions. The manual for the tool is attached at the end of the thesis.

Tabla de contenidos

Capítulo 1

Introducción y objetivos

Según aumentamos nuestras habilidades desarrollando código nos damos cuenta de lo importante que es tener una buena explicación del porqué y cómo hacemos un proceso. Ante las múltiples lecturas a las que puede ser sometido un programa necesitamos una descripción, que nos sirva de ayuda para su comprensión. Sin embargo, esta tarea no deja de ser vista como un mero trámite que nos ralentiza de nuestro objetivo aunque nada más lejos de la realidad, ya que tiene múltiples beneficios como comentaremos más adelante. Para facilitar esta tarea surgen herramientas de apoyo denominadas *documentadores automáticos de código*. Aunque existen muchas nosotros vamos a focalizarnos en LPdoc que es en la que se basa nuestro trabajo. LPdoc es un documentador aumático de código dedicado a la programación lógica. Es usado para documentar programas, módulos o librerias desarrolladas en este entorno.

Existen dos principales estilos de escritura de documentación: markup y markdown. Inicialmente LPdoc solo incluía markup pero en la actualidad incorpora ambos estilos, permitíendonos, al contrario de la mayoría de sistemas, incluso combinarlos. Sin embargo, los desarrolladores de Ciao y LPdoc han dedicido unificar toda su documentación de manera que quede toda bajo el mismo “estandar“, siendo el formato elegido markdown. Para ello, se ha hecho necesaria la creación de un programa que permita transformar todo el código disponible (tanto el repositorio principal del lenguaje como cualquier otro programa de usuario) al nuevo formato y, en esto, consistirá el proyecto.
Los objetivos principales de este proyecto serán:

- Desarrollar una herramienta que nos permita pasar del formato nuevo al antiguo. Dicha herramienta debe ser:

  1. Precisa.
  2. Preparada para transformar millones de líneas.
  3. Especializadas en el markup y markdown de Ciao.

- Probar el funcionamiento correcto de dicha herramienta sobre todos los archivos del repositorio, garantizando la equivalencia de la documentación y la no alteración del funcionamiento de los programas.
- Detección y correción de errores en la documentación original.

- Detección y correción de errores en los paquetes *doccomments* y *lpdoc* en caso de que los hubiese.

Este documento seguirá la siguiente estructura:

Estado del arte: En este apartado haremos un breve recorrido por la historia de las principales herramientas que han inspirado este trabajo.

Desarrollo: Descripción de los pasos seguidos para el desarrollo del programa, así como explicación de las decisiones tomadas.

Pruebas: Explicación de los procedimientos seguidos para la elaboración y ejecución de la batería de pruebas.

Resultados y conclusiones: Apartado en el que analizaremos los resultados obtenidos.

1.1. Programación lógica

Para una mejor comprensión de esta memoria haremos un breve resumen sobre la programación lógica, el cual será el lenguaje utilizado para el desarrollo de este trabajo. Existen múltiples lenguajes basado en este tipo de programacón pero nos centraremos concretamente en Prolog, el más utilizado por la comunidad científica y el que usaremos. Prolog (PROgrammation en LOGique) nace en Marsella en los años setenta gracias a los aportes teóricos de R. Kowalski.

A continuación, explicaremos de manera general como funciona Prolog para facilicitar el seguimiento de esta memoria. Prolog está compuesto por lo siguientes términos:

variables: empiezan por mayúscula o _ y están compuestas por números, letras o _ *(Animal1,_Alumno)*. Se utilizan para escribir hechos y reglas generales. Diferenciamos dos tipos según su valor:

- **Variable libre:** aún no ha sido instanciada. Cuando usamos el término instanciar hacemos referencia a que la variable representa un valor.
- **Variable ligada:** tiene valor, es decir, se ha unificado. El valor de una variables no cambia una vez se determina pero, sin embargo, podemos hallar todos sus posibles valores mediante un proceso que explicaremos más adelante llamado *backtracking.*

Aquellas variables que empiezan por _ son denominadas anónimas y las usamos cuando no nos interesa su valor. Para conseguir que una variable libre pase a ser ligada debemos realizar una unificación con una constante. La unificación se basa en dar valor a una variable de modo que en todas sus apariciones dentro de la misma regla el resultado se verifique. Como hemos menciado anteriormente, una vez que una variable toma una valor este no cambia, por eso, no debemos confundir unificación con lo que en la programación imperativa denominamos asignación.

Constantes: Distiguimos dos tipos:

Atomo: empiezan por minúscula. Hace referencia a un dato *(ana, casa_1)*

Números: numeros enteros y reales *(3.14, 2)*.

Predicados: esta compuesto por un átomo denominado functor y un conjunto de argumentos. Se escribe el functor y entre parentesis los argumentos separados por comas. El número de argumentos que tiene un predicado es denominado Aridad. Estor argumentos pueden ser variables o constantes. La aridad puede ser cero, es decir, no tener argumentos. El functor suele representar la relación existente entre los argumentos. Pueden ser representados mediante una lista siendo el primer elementos el functor y el resto los argumentos *(madre(Lucía,Juan) , amigos(Pepe,Sara))*.

Lista: es una coleccion ordenada de términos. Se escribe con dos corchetes y los elementos se separan por comas *([casa, amigos(Pepe,Sara),X])* pudiendo existir listas vacias haber listas vacias [ ].

- **String:** son un caso especial de lista. Se denotan entre " y su contenido se equivale a lista de caracteres o códigos numéricos que se corresponden con cada caracter *("hola" equivaldría a [h,o,l,a] o [104,111,108,97])*.

Una vez tenemos claros los conceptos básicos sobre los elementos que componen Prolog pasaremos a hablar sobre sus relaciones.

Reglas: Están basadas en las cláusulas de Horn. Nos ayuda a representar una relación que en lenguajes natural podríamos traducir como "SI pasa esto ENTONCES pasa esto otro". El cuerpo está compuesto por otros predicados separados por , o ; . En este caso, la coma representa la conjunción y el punto y coma la disyunción. Para que la cabeza sea cierta el cuerpo debe serlo. De esta manera, podemos deducir nuevos predicados en base a otros que tenemos en el cuerpo.

Hechas: son cláusulas sin cuerpo por lo que siempre son ciertas.

Si siguiésemos el orden normal de ejecución de otros programas nos basariamos en un orden secuencial, pero Prolog incluye un mecanismo propio denominado **backtracking**. En la primera ejecución de una regla se aplicará un orden secuencial. Sin embargo, en caso de que un predicado tenga varias soluciones podemos volver hacía atrás y comprobarlas para continuar otra vez de ese punto hacia adelante. Esto nos permite sacar todas las soluciones posibles que se pueden obtener.

Finalmente, para ayudarnos con el manejo del mecanismo anterior contamos con un predicado sin argumentos denomiando **corte** y representado por !. Cuando se ejecuta este predicado, conseguimos que todas las variables que se encuentran dentro de esa regla y delante de este simbolo que ya hayan obtenido un valor, no podrán tomar otros valores al volver a ejecutarse, es decir, que si tenían varias soluciones se omitirán.

1.1.1. Ejemplo

Tenemos el siguiente programa compuesto por una base de hechos con los predicados hace_deberes y estudia y una regla cuya cabeza es `aprueba/2`. El cuerpo de esta regla, está formado por la conjunción de los dos predicados anteriores.

```
1 %  Predicado hace_deberes/2 equivale a aprueba(X,Y)
2 %  este predicado respresenta X hace los deberes de Y
3     hace_deberes(dani,lengua).
4     hace_deberes(javi,matematicas).
```

```
5     hace_deberes(alberto,matematicas).
6
7  %  Predicado estudia/2 equivale a estudia(X,Y)
8  %  este predicado respresenta X estudia Y
9     estudia(javi,matematicas).
10    estudia(dani,quimica).
11    estudia(alberto,matematicas).
12
13 %  Predicado aprueba/2
14    aprueba(X,Y) :-
15        hace_deberes(X,Y),
16        estudia(X,Y).
```

Partiendo de dicho programa podríamos hacer las siguientes consultas:

```
1 ?- aprueba(Persona,Asignatura).
```

Esta consulta nos devolvería `Persona=javi` y `Asignatura=matematicas`. Esto se debe a que la primera unificación de constantes que satisface los dos predicados del cuerpo de la regla son `hace_deberes( javi, matematicas)` y `estudia( javi, matematicas)`. Sin embargo, al ejecutarse de manera secuencial la primera comprobación se haría con `Persona=dani y Asignatura=lengua` pero como el predicado `estudia(javi,lengua)` no existe este falla y vuelve para atrás probando con la segunda solución posible para `hace_deberes`. Con este ejemplo, podemos ver tanto el funcionamiento de backtracking como el de consulta. Otra cosa más que podemos hacer, es después de recibir la primera solución es poner **;**, de esta manera el programa se volverá a ejecutar buscando otra unificación que satisfaga la regla. Esto, nos sirve para obtener todos los resultados posibles que puede tener una consulta.

Gracias a estos métodos también podemos determinar si un hecho es cierto o no, por ejemplo, haciendo la consulta:

```
1 ?- aprueba(dani,quimica).
```

Nos devolvería *no* ya que como hemos visto anteriormente aunque el primer predicado se cumple el segundo no lo hace.

Otra posible consulta sería:

```
1 ?- estudia(X,matem ticas).
```

Esta consulta nos retornaría como primera respuesta `X=javi` y si introducimos ; `X=alberto`.

Por último, vamos a ver un ejemplo de como funciona el corte cambiando nuestra regla:

```
1 aprueba(X,Y) :-
2     hace_deberes(X,Y),
3     estudia(X,Y),!.
```

Al añadir `!` lo que conseguimos es que en la primera consulta (`?- aprueba(Persona, Asignatura).`) después del obtener el primer resultado no nos devolverá más. Esto se

debe a que cuando intenta hacer backtracking todas las variables que ya han tomado valor y se encuentran delante del corte, en este caso `X=javi` y `Y=matematicas`, no cambian.

Capítulo 2

Estado del arte

2.1. Documentación automática de código

Documentar el código se ha convertido en una gran necesidad. No debemos considerarlo un mero adorno para que nuestro código quede más vistoso, sino una parte más del proceso de elaboración. En un programa no es solo importante que funcione, sino que además lo haga de la manera más eficiente posible. Todo programa con un cierto grado de extensión es probable que sufra modificaciones, ya sea porque se quede desactualizado o simplemente porque se encuentren disversos *bugs*. En cualquiera de estos casos, siempre que se sufra un cambio para poder realizarlo, el programa debe ser entendible y estar bien explicado. Para documentar programas podemos hacerlo de diversas maneras, ya sea mediante comentarios en el propio código o mediante manuales. De esta necesidad surgen los documentadores de textos automáticos, es decir, herramientas que nos faciliten esta tarea. Gracias a ellos, podemos generar manuales actualizados y completos, que nos permitan una mayor comprensión del código.

2.1.1. Literate programming

A principio de los 80 **Knuth** planteó una nueva metodología de programación como alternativa a la usada hasta el momento (*programación estructurada*), a la que denominó **Literate programming**[1] [2].

Literate programming propone cambiar el modo de pensar usado hasta la época, guiado por "explicar a los ordenadores lo que queremos que hagan por explicar a los humanos lo que queremos que haga el ordenador". Esta idea quiere decir que se comienza a usar la lógica humana en vez de seguir la de los compiladores para la realización de los programas. En un primer momento, se eligió como herramienta para la documentación TEX (Knuth fue su inventor) y para el código Pascal. Knuth decidió denominar a este lenguaje y sus programas asociados como sistemas WEB. Un programa WEB debe describir el flujo de pensamientos del autor, así como las abstracciones que utilice para su realización intercalándolos con macros y snippets. Las macros están compuestas por un metalenguaje similar al de los algoritmos en pseudocódigo. Para separar el texto de macros y código se utilizó markup. El archivo será pasado por un procesador que seguirá dos rutinas:

1. **Weave:** genera el fichero que recoge la documentación asociada al programa y facilita la información de su mantenimiento.
2. **Tangle:** produce el programa ejecutable.

Escribir código implica una gran cantidad de decisiones tomadas que, normalmente, no quedan reflejadas.

Literate programming nos permite solucionar esto al documentar cada resolución y ligarla a un trozo de código. Si estas notas luego se desarrollan de manera correcta se pueden convertir en la propia documentación, esto se denomina self-documentation, que fijará las bases de una de las siguientes metodologías que comentaremos. Estas anotaciones nos permiten además no olvidar ningún detalle de los pasos ejecutados. Como hemos ido viendo en las lineas anteriores, el uso de Literate Programming tiene numerosas ventajas. Los programadores se ven obligados a escribir la línea de pensamientos que siguen para la elaboración del programa haciendo más fácil darse cuenta de cuando estan aplicando un procedimiento érroneo. Además, nos permite retomar el programa en el punto exacto donde lo dejamos, así como facilitar la tarea a otros programadores que quieran comprender su funcionamiento. El uso de un metalenguaje permite aumentar la cantidad de conceptos que se pueden comprender.

2.1.2. Emacs

Emacs[3] es conocida por ser la primera herramienta autodocumentada, esto no quiere decir que la documentación se cree sola de manera autómata. Cuando hablamos de autodocumentación nos referimos a que se muestra su propia documentación. Los procedimientos contienen un string de documentación que responde a la principal pregunta que puede tener un usuario sobre un comando "¿Qué hace?". Para poder realizar esa pregunta deberíamos saber que existe dicho comando pero, ¿qué pasa si lo que queremos es un comando que haga una acción que necesitamos?. En este caso, la auto documentación también nos ayuda. Los nombres de las funciones suelen ser auto descriptivos, por lo que si buscamos una acción que queramos realizar, se nos listarán todos los comandos que la contengan en su nombre, así como aquellos en los que aparezca en su string de documentación.

En el ejemplo siguiente vemos la definición de la función `hello-world`. Si consultaramos la información de dicha función obtendríamos: **Greet the world. This function implements the canonical example program.**

```
(defun hello-world nil
  "Greet the world. This function implements
  the canonical example program."
      (interactive)
      (message "Hello world!"))
```

2.1.3. LPdoc

LPdoc[4] es un genereador automático de código utilizado en (C)LPSystem. Ha sido elaborado para el sistema CIAO pero puede ser utilizado por casi todos los (ISO-)Prolog. De todas las herramientas mencionadas, esta es la que más nos interesa, ya que es con la que trabajaremos a lo largo de este proyecto. Al igual que pasa en Emacs, LPdoc entiende todo lo que comprendería un compilador. Es muy útil para la

documentación de librerias pero también puede ser usado incluso para documentar una aplicación entera. LPdoc recibe información de ficheros de usuarios y del sistema que han sido especificados en un fichero Settings, dando como salida la creación de manuales o readmes files entre otros. La salida es generada en HTML o texinfo y, en base a ellos, es capaz de generar los manuales en otros muchos lenguajes.

La generación de manuales puede hacerse de dos maneras mediante la terminal o, si disponemos del sistema ciao instalado en emacs, desde el propio editor. Si usamos la terminal únicamente deberemos escribir:

```
1 lpdoc [Options] Input
```

El archivo de entrada será un fichero de configuración de la documentación o un módulo. Entre las opciones que podemos introducir para la creación de la documentación, están el formato en el que queremos la salida o las opciones de generación. Si no introducimos nada, la opción de formato por defecto será HTML. Para la generación de un manual podríamos escribir, por ejemplo:

```
1 lpdoc -t pdf file.pl
```

De esta manera, obtendríamos la documentación de 'file.pl' en PDF. El comando lpdoc también puede servirnos para abrir nuestra documentación.

```
1 lpdoc --view file.pl
```

La documentación puede ser ampliada incluyendo junto al programa assertions y machine-readable comments. Estas deben ser escritas en el Ciao System assertions language. Debemos incluir una librería compatible que se encargará de hacer que los sistemas de programación lógica ignore estas aserciones y comentarios, permitiendo tratar el documento. Las aserciones, se encargan de proporcionarnos de los predicados, propiedades, módulos... Por ejemplo, si tenemos la siguiente aserción:
`:- pred length(L,N) : list * var =>list * integer.`
Nos permite indicar cual es el uso del predicado length, en este caso, nos informa de que L en la entrada será una lista igual que en su salida, sin embargo, N pasará de ser una variable libre a un entero. Esto nos permite docuementar los usos que tiene predicado.

Por otro lado, tenemos los comentarios mecánicamente legibles, los cuales nos permiten aportar documentación extra a las aserciones. Estos pueden incluir comandos que nos permitan dar formato al texto, hablaremos con más profundidad sobre ellos en el apartado 2.1.8. Distinguimos dos tipos:

Aserciones tipo doc/2: Este estilo de aserciones están compuestas por dos parámetros. El primero se encargará de indicarnos el tipo de comentario del segundo argumento. Estas aserciones siguen las estructura:
`:-doc(CommentType,Commment).`
CommentType y Comment son variables que deberán ser sustituidas por valores como, por ejemplo:
`:-doc(author,Jesús Muelles).`
Al dar estos valores estamos indicando que el fichero ha sido desarrollado por Jesús Muelles y, al generar la documentación, esto saldrá reflejado en ella.

Comentario de aserciones: Las aserciones pueden incluir comentarios que nos permiten reflejar más información de la dada. Si tomamos de referencia el ejemplo usado previamente y le añadimos un comentario:

```
:- pred length(L,N) : list * var =>list * integer
# "@var{N} nos devolverá el número de elementos de la lista".
```

A los datos obtenidos previamente sobre los valores de entrada y salida se le adjuntará una información extra sobre la variable. Esta aparecerá junto al resto de documentación del predicado.

Finalmente, contamos con un tercer tipo de comentarios legibles mecánicamente, que son los de tipo doccomment. Este tipo nos permite escribir en otro formato los dos tipos anteriores. Para poder hacer uso de ello deberemos importar el paquete doccomments. Todos las posibilidades que se incluyen en los dos anteriores tienen su equivalencia en dicho formato. Por ejemplo, los dos casos anteriores se escribirían:

```
%! @author Jesús Muelles

:- pred length(L,N) : list * var => list * integer.
%< "@var{N} nos devolverá el número de elementos de la lista"
```

La equivalencia entre los comentarios de las aserciones y doccomments aún se encuentra en desarrollo, pero es importante conocer su existencia para el desarrollo de esta memoria.

Podemos escribir manuales más complejos utilizando un archivo de configuración como puede ser SETTINGS.pl. Indicaremos las características deseadas permitiéndonos separar por capítulos la documentación adaptándolo a nuestras necesidades. Una de sus funcionalidades,la cual usaremos en esta memoria, nos permite modificar el comienzo de página de la documentación generada. Esto, nos facilita su inclusión en otros PDF siguiendo la documentación original. En nuestro caso lo utilizaremos para adjuntar a la memoria la documentación de los predicados creados. De esta manera, podemos tener tanto un código bien documentado como su respectivo manual, lo cual era el objetivo principal de los autodocumentadores.

2.1.4. Javadoc, más versiones

En la actualidad, existen múltiples herramientas que nos permiten generar documentación de manera automática. Dependiendo del lenguaje en el que estemos programando, será más recomendable usar una u otra. Una de las más utilizadas a día de hoy es 'Javadoc', que ha sido establecida como el estándar para documentar clases Java. Otros lenguajes utilizados para este fin son PHPdoc o Doxygen.

2.1.5. Markup vs markdown

Con la evolución de los lenguajes de programación surgen nuevos lenguajes orientados al procesado de textos. Hay dos grandes vertientes:[5]

1. **Markup:** para su elaboración usan etiquetas que se encargan de dar formato estructural o de presentación al texto.
2. **markdown:** para la identificación no utilizan etiquetas sino símbolos. Esto facilita su lectura y comprensión.

2.1.6. Sistemas Clásicos.

2.1.6.1. Markup

Como hemos comentado anteriormente markup [6][7]es un lenguaje basado en marcas. Actualmente, es muy utilizado en internet, ya que HTML (lenguaje estándar para el desarrollo web) hace uso de él. Esto ha hecho que cobre una gran importancia por el incipiente auge de internet. También, podemos encontrarlo en procesadores de texto profesionales como es LaTeX. Pese a que visualmente pueda parecerlo no debemos confundirlo con un lenguaje de programación, ya que únicamente se encargan de su estructura y semántica. En un documento con sintaxis de markup, podemos diferenciar dos elementos claves: por un lado tenemos, la información propiamente dicha y, por otro, las etiquetas markup. Podemos separar dos tipos de etiquetado markup principales:

Procesamiento: define como se crea visualiza o procesa un documento siendo el usuario participe de ello y pudiendo modificarlo. A partir de cada etiqueta, se realiza un procesado (TeX).

Descriptivo: determinan las propiedades de los elementos y su estructura, es decir, describe fragmentos de texto (LaTeX, HTML).

Podemos hacer una tercera categoría en la que incidiremos más adelante:

Presentational: usado por los antiguos editores de texto que hacen uso de **WYSIWYG**. Su principal característica, es que son invisibles al autor, es decir, no pueden ser modificados.

A continuación, podemos apreciar un ejemplo de markup. A la izquierda tenemos un editor escrito en markup sin procesar. En él podemos apreciar como cuando queremos dar algún tipo de decoración o estructurar al texto lo hacemos mediantes marcas. Por ejemplo, nos podemos fijar en el comando `\textbf{}`, que se encarga de poner la palabra en negrita. Ahora, si nos fijamos en la parte de la derecha de la imagen, podemos ver cual será el resultado tras procesar las marcas que teníamos.

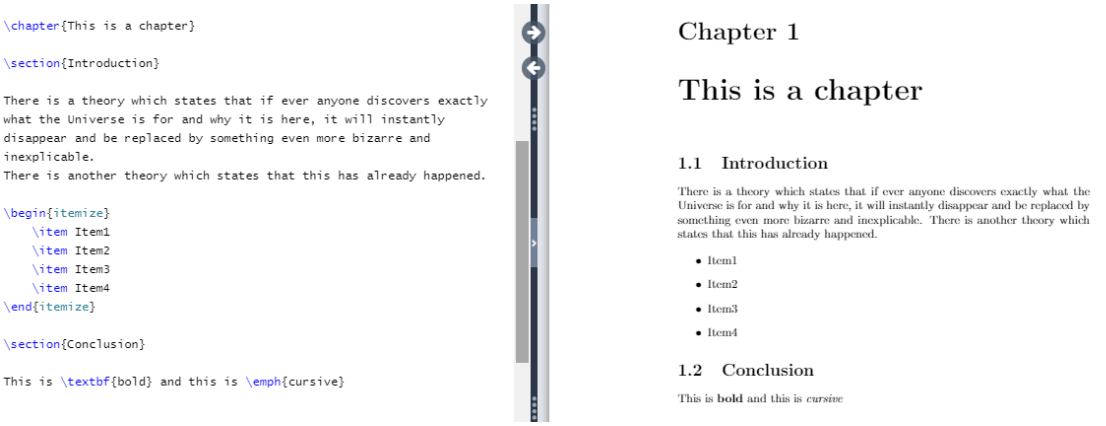

Figura 2.1: Comparativa markup procesado y sin procesar

2.1.6.2. WYSIWYG

What **Y**ou **S**ee **I**s **W**hat **Y**ou **G**et (WYSIWYG) permite ver al usuario mediante una interfaz, el resultado que se obtendrá. Esto quiere decir, que si ponemos, por ejemplo, un trozo de texto en negrita este aparecerá en negrita en el resultado. Nace como una iniciativa para ayudar al usuario a mejorar su experiencia, debido al gran número de editores en markup existentes. Al haber tantos editores, los autores se veían obligados a recordar un montón de comandos, que variaba según cual programa se usara. WYSIWYG permite al usuario olvidarse de las etiquetas y cambiarlo por algo más intuitivo. La aparición de este tipo de procesadores, supuso una revolución en un punto en el que los ordenadores personales empezaban a incrementar sus ventas. El uso de este tipo de editores permitía ver al usuarios como afectaba al documento un cambio de manera de inmediata, siendo el formato visto en pantalla, el mismo que se obtendrá al imprimirlo. Esto permitió, que la presentación formase parte de la propia escritura en lugar de ser una característica añadida.

Sin embargo, pese a lo que indica su nombre, este tipo de editores a veces no da el efecto deseado y lo que muestra no es lo que obtendrás sino algo similar. Este tipo de procesadores de texto son apropiados cuando queremos editar textos cortos o de ámbito personal, pero se quedan limitados cuando lo usamos de manera más profesional.

2.1.7. Markdown

Es un lenguaje de marcado ligero (tipo de formateo de texto que ocupa de poco espacio y fácil de usar) [8]creado por John Gruber. Este tipo de formato permite una mayor legiblilidad y facilidad de escritura y lectura. Esta inspirado en las convenciones de formateo de texto plano de los correos electrónicos. Puede ser abierto por bloc de notas, aunque también hay editores de texto especializados, que incorporan una interfaz que permite ver el resultado como en una herramietna WYSIWYG. Gracias a esto, permite compatibilidad con todos los dispositivos, ya que siempre dispondrá de una herramienta que permita su modificación. La idea de markdown consiste en identificar las estrucuturas lógicas de un documento, como puede ser un título o una sección y asociarlas a unos caracteres, de tal manera que al compilarlo consigamos un documento con el formato que hemos deseado. Otro de los objetivos era la conversión de documentos escritos en markdown a HTML, para aumentar la velocidad respecto a si usasemos direcamente HTML. En la actualidad, markdown esta tomando gran peso en distintos ámbitos por su comodidad y su uso en foros, en Github o para la toma rápida de apuntes.

Para entenderlo mejor tenemos la siguiente imagen. En la parte de la izquierda tenemos un texto escrito en markdown sin procesar. Podemos ver que en este caso, como ya habiamos comentado, hemos cambiado los comandos por símbolos. Por ejemplo en lugar de `\textbf{}` tenemos la palabra entre . Al igual que en la imagen usada como ejemplo en markup, tenemos en la parte izquierda el texto markdown, el cual a simple vista no parece más que un simple texto plano. Sin embargo, una vez se procesan los símbolos claves que acompañan a las palabras, podemos ver como se obtiene un resultado (parte derecha), mucho más elaborado de lo que parecía a primera vista. Como podemos apreciar este texto en markdown es mucho más legible que el del markup. Utilizando distintos estilos de editores podemos obtener resultados parecidos, pero cada lenguaje tiene sus propias limitaciones.

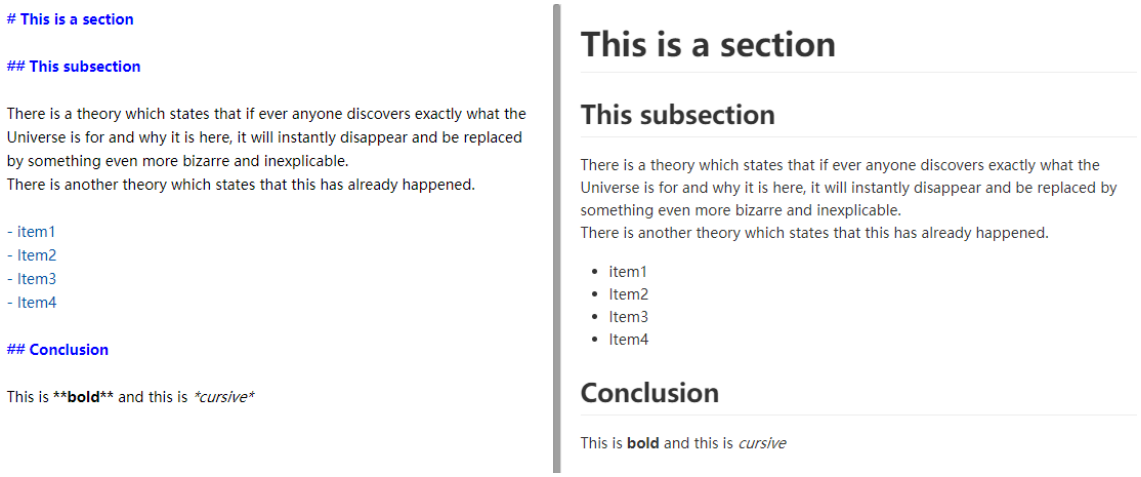

Figura 2.2: Comparativa markdown procesado y sin procesar

2.1.8. MU y MD de lpdoc

LPdoc[9] como hemos hablado anteriormente incluye tanto formato markdown como markup. La principal ventaja, es que podemos combinar ambas de manera que usemos siempre la opción más sencilla para nosotros. No todos los comandos tienen su equivalente en markdown. Es importante que entendamos cual es la transformación que sufre cada comando, debido a que en eso consiste gran parte de este proyecto. A continuación tenemos una tabla comparativa para poder apreciarlo mejor. Esta tabla nos será de gran ayuda a lo largo de todo el documento.

Markup	Markdown
Formatting commands	
@begin{ itemize } @item Apple @item Apple @item Apple @end{ itemize}	- Apple - Apple - Apple
@begin{ description } @item{Apple} red @item{Watermelon} green @item{Banana} yellow @end{ description}	- Apple :: red - Watermelon :: green - Banana :: Yellow
@begin{ enumerate } @item{Apple} First @item{Watermelon} Second @item{Banana} Third @end{ enumerate}	-# First -# Second -# Third
	1. First 2. Second 3. Third

Sigue en la página siguiente.

Markup	Markdown
@begin{ cartouche } text @end{ cartouche }	begin{ cartouche } text \end{ cartouche }
@begin{ note } text @end{ note }	\begin{ note } text \end{ note }
@begin{ alert } text @end{ alert }	\begin{ alert } text \end{ alert }
@begin{ verbatim } text @end{ verbatim }	``` text ```
@section{text}	# text
@subsection{text}	## text
@subsubsection{text}	### text
@comment{text}	\comment{text}
@footnote{text}	\footnote{text}
@em{text}	*text*
	\em{text}
	text
@bf{text}	**text**
	\bf{text}
	__text__
@tt{text}	`text`
	\tt{text}
@key{text}	\key{text}
@sp{N}	\footnote{N}
@p	\p
@noindent	\noindent
@hfill	\hfill
Indexing commands	
@index{text}	\index{text}
@cindex{text}	\cindex{text}
@pred{text}	\op{text}
@decl{text}	\lib{text}
@var{text}	‘text‘
	\var{text}
@apl{text}	\apl{text}

Sigue en la página siguiente.

Markup	Markdown
@file{text}	\file{text}
Referencing commands	
@cite{keyword}	\cite{keyword}
@ref{section title}	\ref{section title}
@href{URL}	\href{URL}
	[[URL]]
@href{URL}{text}	[text](URL)
	[[URL][text]]
@email{address}	\email{address}
@email{text}{email}	\email{text}{address}
@author{text}	\author{text}
Date and Version	
@today	\today
@version	\version
Inclusion commands	
@include{filename}	\include{filename}
@includecode{filename}	\includecode{filename}
@includefact{filename}	\includefact{filename}
@includedef{filename}	\includedef{filename}
@image{filename}	\image{filename}
@image{epsfile}{width}{height}	\image{epsfile}{width}{height}
Mathematics	
@math{filename}	$filename$
@begin{ displaymath } formula @end{ displaymath }	\begin{ displaymath } formula \end{ displaymath }
@defmathcmd{cmd}{n}{det}	\defmathcmd{cmd}{n}{det}
@defmathcmd{cmd}{det}	\defmathcmd{cmd}{det}

Cuadro 2.1: Relacion Markdown Markup de LPdoc

2.2. Conversion tools

Como hemos visto en los puntos anteriores existen muchas técnicas diferentes para editar textos. En la actualidad no hay un consenso sobre cual es el mejor, siendo esto una opinión subjetiva que depende del fin del documento y de los gustos personales del autor. Ante tanta diversidad surge la necesidad de herramientas, que permitan la conversión de un formato sin pérdidas de información. En la actualidad hay diversas herramientas siendo Pandoc una de las más utilizadas.

2.2.1. Pandoc

Pandoc[10] es un conversor de documentos que puede ser usada como herramienta de escritura. Su misión es traducir de un formato a otro siendo el mejor tipo de archivo soportado el markdown. Sin embargo, admites otros lenguajes como LaTeX o HTML. Sin embargo, esta herramienta no es totalmente completa, ya que a veces no nos proporciona una conversión exacta sino una aproximación. Pandoc también nos resulta útil cuando queremos solventar carencias de un lenguaje mezclándolo con otro que sí lo haga. Una característica interesante es, que para traducir de markdown a pdf utiliza una plantilla LaTeX a la que podemos acceder. Esta plantilla nos es muy útil para cambiar características del aspecto en las que markdown se queda limitado. De esta manera, tenemos las ventajas de markdown (escritura más rápida, texto más legible) con las del markup de LaTeX (gran control sobre su aspecto). Pandoc, también nos puede ser de ayuda cuando queremos aprender un nuevo lenguaje. Por ejemplo, si nosotros ya hemos aprendido todo sobre markdown y queremos adentrarnos en el uso de HTML, podemos usarlo como un "diccionario". Imaginemos que sabemos que la los encabezados en markdown se escriben poniendo un “ delante, si introducimos esto en Pandoc y hacemos la transformación a HTML veremos que obtenemos, que los encabezados se escriben con la etiqueta <h1>. Obviamente, no vamos a poder usar esto como herramienta completa de aprendizaje. Como hemos visto, Pandoc no es un herramienta exacta y no todos los comandos de un lenguaje tienen una correspondecia directa, ya que cada uno tiene sus propias carencias y particularidades.

2.2.2. Programa desarrollado

Siguiendo el flujo de este punto, introduciremos el programa desarrollado en este proyecto. La herramienta programada se encuentra dentro del apartado de herramientas de conversión. Al contrario de otras herramientas creadas hasta la fecha su misión será proporcionarnos una equivalencia exacta entre los comandos markdown y markup propios de LPdoc. Se encargará también de proporcionarnos una traducción exacta entre aserciones y doccomments.

Su desarrollo será llevado a cabo en tres fases, permitiéndonos llevar un mayor control sobre la exactitud de las traducciones. Deberemos probar el programa sobre una extensa batería de archivos compuesta por más de 8000 ficheros. Al realizar una conversión gradual podremos detectar más fácilmente el motivo de posibles fallas del programa y proceder a solucionarlas al tener un mayor control sobre las transformaciones aplicadas.

Capítulo 3

Desarrollo

El desarrollo de este proyecto se llevará a cabo en tres fases. Cada una de esas fases será sometido a sus respectivas pruebas no pasando a la siguiente hasta que hayamos solventado cualquiera problema de la anterior. De esta manera conseguiremos una transformación progresiva de la documentación.

- **Fase 1:** La primera fase consistirá en la transformación del fichero de entrada de markup a markdown.
- **Fase 2:** Convertiremos los principales predicados `doc/2` a doccomments.
- **Fase 3:** Pasaremos los comentarios de las aserciones a doccomments y finalizaremos la conversión de las aserciones `doc/2`

Previamente a explicar en detalle el desarrollo de cada una de las fases pondremos en contexto los cuatros ficheros que formarán nuestro programa junto a un breve resumen de su función.

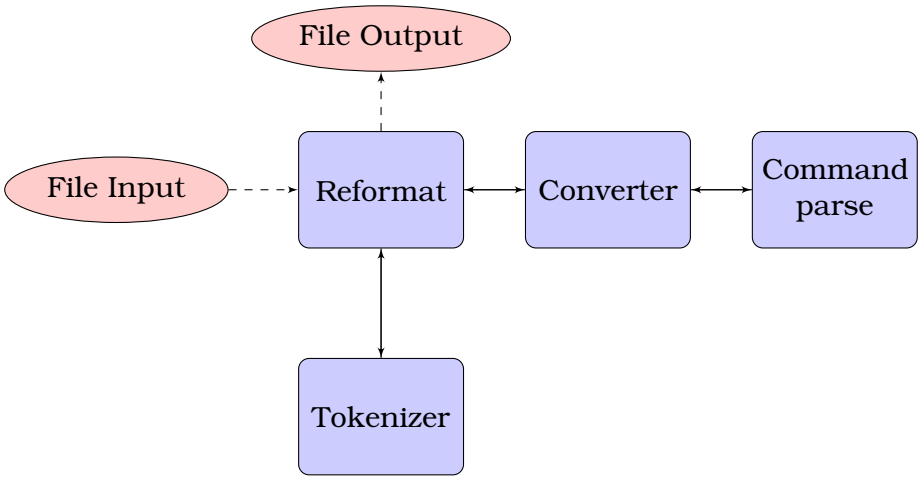

Figura 3.1: Relación entre módulos

Por un lado, tenemos las entradas y salidas del programa:

File Input: Es archivo de entrada. La idea principal de este programa sería que fuese un fichero de extensión .pl o .lpdoc, el cual contuviese documentación escrita

con aserciones `doc/2` y en markup.

File Output: Es el archivo de salida. Este fichero será equivalente al File Input manteniendo su estructura original únicamente modificando las aserciones `doc/2` por doccomments y los comandos markup por markdown.

Por el otro lado, nos encontramos los módulos que conforman el programa en sí:

Tokenizer: Se encarga de separar por tokens un string dado.

Command_parse: Separa un string devolviéndonos el resultado en una lista, donde cada elemento representa cada uno de los comandos que pueden aparecer.

Converter: Su función consiste en transformar, en base a lo obtenido por el fichero `command_parse`, los comandos a markdown.

Reformat: Es el archivo principal, filtra los tokens recibidos, tranforma de `doc/2` a doccomments y realiza la llamada a converter para transformar a markdown.

Los ficheros de entrada pueden contener comandos tanto en markdown como en markup, ya que como hemos comentado previamente, una de las grandes ventajas que nos ofrece LPdoc es la posibilidad de combinarlos. En un primer momento este programa estaba orientado a traducir ficheros totalmente en markup, es decir, en caso de aparecer comando escrito entre símbolos markdown estos se tratarían como caracteres que debemos escapar. Para solucionar este problema, cuando ejecutemos, añadiremos la posibilidad de indicar que es un documento que contiene markdown. De esta manera, en caso de indicarlo antes de comenzar la separación mediante comandos, aplicaremos un cambio de markdown a markup eliminando así cualquier problema que pueda entorpecernos el proceso. En este caso, nuestro diagrama de dependencias añadiría un nuevo módulo (`markdown_translate`) que se encarga de realizar dicha traducción quedando la interacción así.

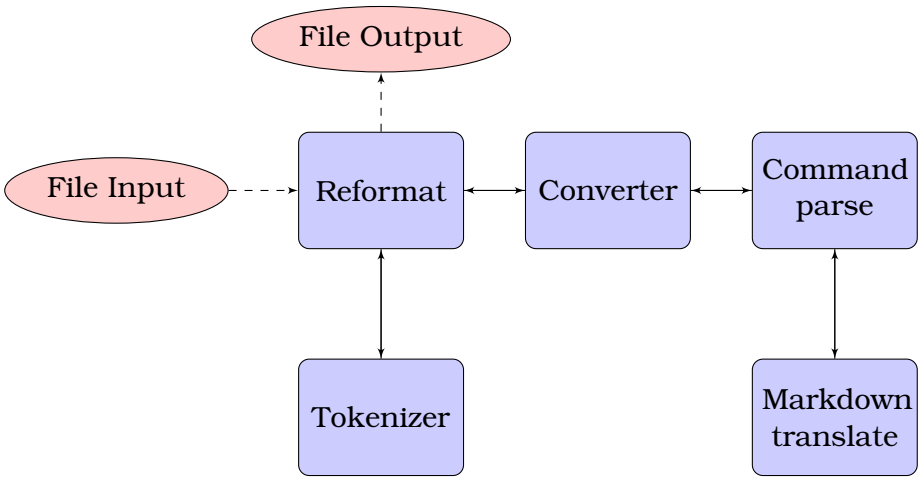

Figura 3.2: Relación entre módulos con traducción a markdown

3.1. Fase 1

El objetivo principal de esta fase será pasar de markup a markdown toda la documentación. Para ello, comenzaremos separando por tokens todo el archivo. En base

a dichos tokens seremos capaz de diferenciar que elementos son los que deben cambiar y cuáles permanecer intactos. En la tabla 3.1 podemos encontrar la relación entre los comandos que tenemos y los que queremos conseguir. Para entender mejor los resultados que buscamos en esta primera fase, vamos a proceder a explicarla con un ejemplo muy sencillo, que nos permitirá ver mejor los cambios a realizar. Imaginemos que, a nuestro programa le llega un fichero con el siguiente contenido:

```
:-use_package(assertions).
:- doc(module,"Esto es un @bf{modulo} @em{prueba}.
Nuestros objetivo son:
@begin{itemize}
@item pasar aserciones doc a markdown
@item traducir comentarios de asercionn a doccomments
@end{itemize}").

:- pred q(A)
    # "Esto es un @tt{comentario}".
q(_).
```

En vista a ese extracto, nosotros deberemos devolver el siguiente resultado:

```
:-use_package(assertions).
:- doc(module,"Esto es un **modulo** *prueba*.
Nuestros objetivo son:
- pasar aserciones doc a markdown
- traducir comentarios de asercionn a doccomments
").

:- pred q(A)
    # "Esto es un `comentario`".
q(_).
```

Como podemos apreciar, lo único modificado ha sido el contenido markup, tanto la estructura como el contenido inicial han sido respetados. En las siguientes secciones analizaremos módulo a módulo los pasos desarrollados para la obtención de los resultados de esta primera etapa.

3.1.1. Tokenizer

Para nuestro proyecto reutilizaremos el módulo `tokenizer` del bundle `ciaofmt` de Ciao al que incorporaremos cambios para adaptarlo a nuestras necesidades. El objetivo principal que queremos conseguir es, que gracias a estos tokens seamos capaces de diferenciar en reformat aquellos que deben ser modificados. Los tokens sigue la siguiente estructura:
`token(TipoToken,ValorToken)`
Nos comunicaremos con este módulo a través del predicado `identify_tokens/2`. Este predicado se encargará de guardar en el primer argumento una lista con los tokens que equivalen al segundo argumento. Por ejemplo, si ejecutamos el siguiente predicado:

```
1 ?- identify_tokens(Tokens, "atomo "Esto es un string ", predicado(A,B).").
```

obtendremos en la variable Token la siguiente lista:

```
Token=[token(atom,"atomo"),
token(spaces," "),
token(string,"""Esto es un string"""),
token(spaces," "),
token(separator,","),
token(spaces," "),
token(openfunc,"predicado("),
token(var,"A"),
token(separator,","),
token(var,"B"),
token(closepar,")"),
token(endclause,".")]
```

Para facilitar tareas posteriores, incorporaremos cuatro nuevos tipos de tokens(**doc, assertion, assertionC y commentA**) que nos servirán para distinguir todas las aserciones de predicado en el documento.Estas aserciones serán precedidas por el símbolo :- y diferemos tres grupos:

- **Sin comentarios:** Agrupa todas las aserciones, excepto las de tipo doc, que no tienen comentarios. Un ejemplo de este tipo puede ser:

  ```
  :- pred p(+A,B) : integer(A) =>  ground(B).
  ```

  el cual cuando apliquemos el predicado `identify_token` será representado por el tipo de token assertion. Concretamente en el caso anterior recibiriramos:

  ```
  token(assertion,":- pred p(+A,B) : integer(A) =>  ground(B).")
  ```

- **Con comentarios** Son aquellas asercion que tengan comentarios. Junto con las aserciones de tipo doc estos serán los tokens que más nos interesan a lo largo de esta memoria. Este tipo de aserciones las dividiremos en dos tokens. Por un lado representaremos la asercion y, por otro, el comentario en si mismo. Para comprender mejor su funcionamiento usaremos la siguiente representación:

  $$\underbrace{\texttt{:- pred p(A,B) : integer(A) =>ground(B) \#}}_{\textbf{assertionC}}\ \underbrace{\texttt{"Esto es un comentario".}}_{\textbf{commentA}}$$

  De este modo, si aplicasemos el predicado `identify_tokens/3` sobre el ejemplo anterior recibiriamos la lista de tokens:

  ```
  [token(assertionC,":- pred p(A,B) : integer(A) =>  ground(B) #"),
  token(commentA," "Esto es un comentario".")]
  ```

- **Doc:** Son aquellas aserciones que, como su nombre indica, son de tipo doc. Sigue la siguiente estructura:

  ```
  :- doc(CommentType, CommentText).
  ```

  Serán representadas por el nuevo token de tipo doc el cual tendrá un formato especial. Como hemos visto la aserción doc está compuesto por dos argumentos. En el módulo reformat será importante tener ambos diferenciados. Para facilitar dicho procesado lo que haremos será dar como valor del token una lista en la que cada elemento será un argumento. Por tanto, si tenemos,por ejemplo, el siguiente caso:

```
:- doc(module, "Esto es un ejemplo").
```

el token que lo representa será:

```
token(doc,[item("module"),item(""Esto es un ejemplo"")])
```

Por regla general, este token estará compuesto únicamente únicamente por dos elementos, sin embargo, hay una excepción. Hay veces que el segundo argumento esta compuesto por dos strings, por ejemplo:

```
:- doc(module, "Esto es un ejemplo" || " De concatenado").
```

Ante esta situación,el token doc diferenciará las dos partes del segundo argumento siendo el resultado:

```
token(doc,[item("module"),item(""Esto es un ejemplo""),
item("" De concatenado"")])
```

Es importante recordar que para que estos token aparezcan, las aserciones deben aparecer al principio del documentos o detrás del fin de una clásula. Esto quiere decir que si, por ejemplo, tenemos el siguiente caso:

```
doccomments( :- doc(_,_)).
```

no obtendríamos el token de tipo doc si no la descomponsición del predicado en tokens, es decir, tendriamos la siguiente lista de token que representan esta cláusula:

```
[token(openfunc,"doccomments("),
token(spaces," "),
token(operator,":-"),
token(spaces," "),
token(openfunc,"doc("),
token(var,"_"),
token(separator,","),
token(openfunc,"doc("),
token(closepar,")"),
token(closepar,")"),
token(endclause,".")]
```

3.1.2. Reformat

Será nuestro módulo principal, su predicado main será el encargado de iniciar la ejecución del programa. Dicho predicado recibirá el path del fichero de origen y del fichero destino así como las opciones que queremos aplicar. Estas opciones irán aumentando según se vaya ampliando el programa, por tanto, para ver todas las posibles opciones de las que disponemos consultaremos el capítulo **??**

El primer paso será conseguir cargar el fichero de origen en un string que no permita manejarlo, esto se hace gracias al predicado propio de Ciao `file_to_string`. Para ampliar el uso de la herramienta daremos también la posibilidad de cargar por la entrada de la terminal el string a traducir. Una vez tenemos el string que queremos convertir llamamos al predicado `identify_tokens/3` del módulo `tokenizer` para obtener los tokes asociados. La misión principal de este módulo será filtar dichos tokens de manera que diferenciemos aquellos que deben ser modificados y los que no. En esta primera fase nos centraremos en dos tokens ya conocidos los de tipo `doc` y `commentA`.

Como ya sabemos, estos tokens contienen los string con comandos markup, es decir, esos que queremos traducir. Comenzaremos hablando del tratamiento que sufriran los primeros. Para poder entender mejor los pasos seguidos usaremos como ejemplo el siguiente token doc:

```
token(doc,[item("module"),item(""Esto es un @bf{ejemplo}"")])
```

que es la representación de:

```
:- doc(module,"Esto es un @bf{ejemplo}" ).
```

En esta primera fase, distinguiremos entre aquello predicados doc cuyo comentario asociado sea un docstring y los que no, es decir, aquellos que sean un átomo,term...El primer tipo lo identificaremos como `docT` y el resto como `doc`. Para ello, contamos con el predicado `commandDoc/4`. En esta primera fase el tercer argumento de esta función no nos interesará. Por tanto, siguiendo nuestro ejemplo la llamada a realizar será:

```
commandDoc("module",Group,_,phase1)
```

A continuación, procederemos a tratar el segundo argumento mediante el predicado `translateComment/4`. Si estamos ante un predicado de grupo `docT` deberemos traducirlo llamando al predicado `converter/5`. Una vez tenemos el predicado traducido deberemos juntar los dos argumentos en una aserción doc obteniendo:

```
:- doc(module,"Esto es un **ejemplo**" ).
```

En caso de que el segundo argumento sea una concatenación de strings se procederá a llamar a `converter/5` con cada uno de los string. Finalmente, se juntaría también en una aserción de tipo doc.

Si hubiesemos recibido un token doc del otro grupo al tratarse no tratarse de un docstring no será necesario llamar a `converter/5`. Por lo que únicamente formariamos la aserción doc con los argumentos correspondientes.

Por otro lado, en caso de notificar que estamos ante un documento LPdoc el proceso se simplifica. Prescindiremos del paso en el que debiamos tokenizar todo el documetno ya que en este caso todo el texto debe ser transformado. Por tanto, una vez tenemos el documento en un string llamaremos a converter indicando que estamos ante tipo `lpdoc` y el resultado devuelto será escrito en fichero de salida.

3.1.3. Command_parser

Tomaremos de base el módulo `autodoc_parse` de LPdoc. Su predicado principal, `parse_docstring/3`, nos permite separar un string dado, en una lista de elementos diferenciados según pertenezcan a un comando o sean únicamente un string. El primer predicado nos indicará si estamos ante un string que contiene markdown o si únicamente está compuesto de markup. Si nos encontramos ante el primer caso, deberemos hacer uso del predicado `markdown_transalate/2` del módulo LPdoc. Este predicado se encargará de traducir cualquier comando escrito en markdown a markup unficando de esta manera nuestro string.

Como podemos ver en la figura, el objetivo siempre es separar los comandos en base a un string escrito en markup, ya sea porque hagamos una traducción previa a markup o porque lo fuese desde un principio.

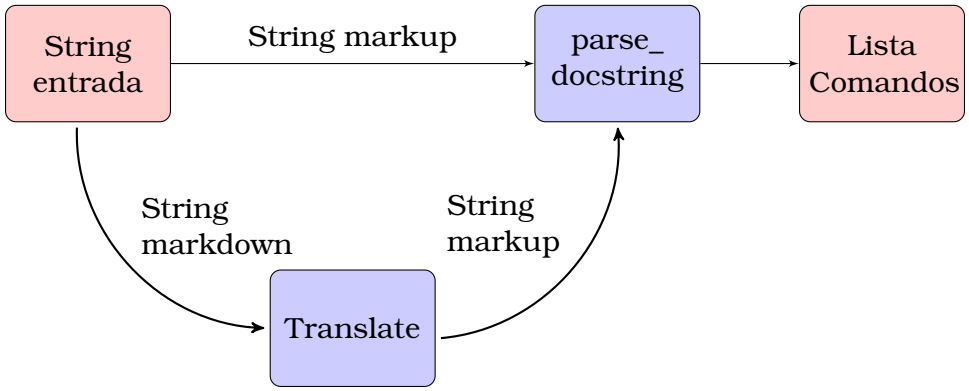

Figura 3.3: Opciones parse_docstring

Para separar el texto se hará uso de un parse, que se encargue de diferenciar cuando estamos ante un string normal y cuando nos encontramos ante un comando. Al funcionamiento original añadiremos modificaciones, que nos son necesarias para mantener la estructura original del documento. Es importante recordar que este predicado añade los elementos escapando su contenido, es decir, si tuviésemos un string, que contiene en su interior _, en el elemento aparecerá _. En base a esto, deberemos ser nosotros, cuando tratemos estos tokens los que nos encarguemos de escapar los caracteres necesarios.

En el predicado original, no sé notificaba cuando había un salto de linea sólo cuando se producía una ruptura de párrafo, ya sea mediante comandos, o mediante lineas en blanco. Para poder conseguir que cuando hagamos la traducción mantengamos los saltos de línea originales, es necesario añadir un nuevo elemento que los diferencie y modificar el parse. Anteriormente, cuando nos encontrábamos ante el final de una linea se trataba como si fuese un espacio normal (`string_esc(" ")`). Ahora, lo que haremos será generar el elemento `break([])`. En el código original se encontraban los siguientes predicados(las lineas han sido numerada para una mejor explicación del código):

```
1  pick_space(noverb, R, R0) --> !,
2      ( pick_blank(Blank) ->
3          { R = [string_esc('' '')|R1],
4          { Blank = paragraph} ->
5          R1 = [p('''')|R0]
6          ; Blank = normal -> R1 = R0}
7      ; { R = R0 }
8      ).
9
10 pick_blank(Blank) -->
11     ( spaces_or_tabs, newline ->
12         spaces_or_tabs,
13         ( newline ->
14             spaces_or_tabs,
15             % Two newlines generates a paragraph break
16             { Blank = paragraph }
17         ; % One newline is just a blank
18           { Blank = normal }
```

```
19          )
20      ; space_or_tab ->
21          spaces_or_tabs,
22         % Several blanks are just collapsed
23          { Blank = normal }
24      ).
```

El predicado pick_spaces se encarga de realizar el tratamiento adecuado dependiendo del tipo de espacio ante el que nos encontremos. El tipo de espacio nos lo determinará el predicado pick_black. Como podemos comprobar en la linea 18 cuando únicamente había una linea el tipo dado en Blank, se marcaba como normal y, entonces, en pick_blank se añadía un espacio (por la línea 3) en lugar de una línea en blanco. Para solucionarlo la línea 23 se ha modificado por:

```
1 { Blank = line }
```

y añadiríamos la siguiente condición entre las líneas 4-5, que se encarga de añadir de un token break("") cuando el tipo de Blank sea line:

```
1 ; Blank = line -> R1  = [break(""),R0]
```

De esta manera, tenemos creado un nuevo token que represente los saltos de línea.

Otro punto a tratar será la expansión de algunos comandos. Como vimos en la tabla 3.1, existe un tipo de comando, que se encarga de incluir otros fichero o documentación al fichero actual. El parse_docstring original lo que hacía era expandir esos comandos, es decir, si recibía @includeinformación.lpdoc, lo que nos devolvería sería un token con el contenido de información .lpdoc. Sin embargo, nosotros no queremos que esto suceda así, debido a que únicamente queremos realizar modificaciones enfocadas en el estilo de los comandos y no en el contenido. Por tanto, no querremos expandirlo, sino indicar el tipo de comando que es y el contenido del mismo, no del fichero. Para ellos, cuando detectemos un comando de este tipo lo que haremos será modificar el token, de tal modo que lo que hagamos será devolver un token incl_command(Type,Content). La variable Type la sustituiremos según proceda indicando el comando de inclusión del que se trata y la variable Content nos indicará el fichero o predicado que queremos incluir. Para comprender el funcionamiento tras los cambios usaremos de ejemplo el proceso seguido ante el comando @include{}. Anteriormente teníamos:

```
1  handle_incl_command(include(FileS), DocSt, Verb, RContent) :- !,
2      atom_codes(RelFile, FileS),
3      handle_incl_file(include, RelFile, DocSt, Verb, RContent).
4
5  handle_incl_file(Mode, RelFile, DocSt, Verb, RContent) :-
6      ( ( Mode = includeverbatim, % TODO: remove, add includecode
           instead?
7          error_protect(find_doc_source(RelFile, File), fail)
8        ; error_protect(find_file(RelFile, File), fail)
9        ),
10       read_file(File, Content) ->
11         autodoc_message(verbose, "-> Including file ~w in
              documentation string", [File]),
12         incl_verb(DocSt, Verb, Verb2),
```

```
13          parse_docstring__1(DocSt, Verb2, Content, RContent),
14          autodoc_message(verbose, "Done including file ~w in
               documentation string", [File])
15      ; RContent = err(parse_error(cannot_read, [RelFile]))
16      ).
```

Como vemos, mediante el predicado `handle_incl_file` se procesaba el argumento recibido en el comando `include` para conseguir cargar al programa su contenido. Por ello, el funcionamiento de dicho predicado se queda obsoleto en su función dentro de nuestra herramienta. Por tanto, tendremos que eliminarlo y modificar el predicado encargado de manejar el comando `@include{}`, es decir, `handel_incl_command/4`.

```
1 handle_incl_command(include(FileS), _DocSt, _Verb, RContent) :- !,
2     RContent=incl_command(include,FileS).
```

Se ha creado un token común a todos los comandos de inclusión:

`incl_command(Type,Content)`

El primer argumentos nos marca el tipo de comando ante el que estamos (`@include{}`, `@includecode{}...`) y el segundo será el propio contenido del comando. Por ejemplo, ante el comando:

`@include{file.pl}`

recibiremos:

`incl_command(include,"file.pl")`

Este proceso será extrapolable al resto de comandos de inclusión, puesto que nos permite unificar bajo un mismo token todos los comandos de inclusión.

Debido a que el `parse_docstring` original está enfocado a la generación de documentación, los comentarios son omitidos. No obstante, como nosotros queremos preservar todos los comandos, deberemos generar un nuevo token para este tipo de comandos, ya que de otra manera estaríamos perdiendo información. En el código original, cuando nos encontrábamos con el predicado `@comment{Text}` generábamos el token `comment(Text)`, donde ambas variables Text tienen el mismo valor. El problema viene cuando procesamos el token para añadirlo a lista final se incluía el siguiente fragmento:

```
1 (Token = comment(_) ->
2 State = ignoreblank,Rs=R0;
```

Este bloque de código forma parte de una condicional, que indicaba que, en caso de estar ante un token de tipo comment lo ignorásemos y pasáramos a la siguiente iteración, es decir, analizar el resto del string. Si queremos conseguir que este token no se omita, deberemos eliminar esta condicional. Similar a este punto, encontramos uno de nuestros objetivos, la eliminación de los comandos `@noindent`. El uso de este comando ha quedado obsoleto, por lo que al realizar la conversión, se ha decidido eliminar todas sus apariciones. Para ello, se ha usado el mismo procedimiento que se usaba para obviar `@comment`, añadiendo en el mismo lugar donde los ignorábamos:

```
1 (Token = noindent(_) ->
2 State = ignoreblank,Rs=R0;
```

Otro de los grandes inconvenientes que tiene parse_docstring, es cuando nos encontramos ante comandos que no entiende. Debemos recordar que, un comando puede ser escrito empezado por @ o \. En el uso original, cuando nos encontramos ante una palabra que empiece de dicha forma, lo asociaremos a que es un comando y, en caso de no ser unos de los comandos admitidos, se mostrará un warning por pantalla y omitirá. Sin embargo, en nuestro caso no podemos permitir que se omita nada. Tenemos que tener presente que, solo queremos hacer un cambio de formato, es decir, si algo no se entiende no se omite, simplemente se tiene que dejar como está. Por tanto, hemos modificado el parse para cambiar su funcionamiento inicial. De este modo, en caso de detectar un posible comando, se procederá a la comprobación habitual para conocer si estamos ante uno conocido y en caso de no ser así, lo trataremos como un string normal. Para ello, modificaremos el predicado que se encarga de generar el token de error ante la aparición de un comando desconocido. El código original del predicado get_token enfocado a los comandos y de handle_command era:

```
get_token(DocSt, Verb, Token) -->
    % Parse a command
    [C], { cmdchar(C) },
    command_body(Struct),
    !,
    { handle_command(Struct, DocSt, Verb, Token) }.

handle_command(Command, DocSt, Verb, NewCommand):-
    functor(Command, Cmd, A),
    functor(BT, Cmd, A),
    ( cmd_type(BT) ->
        Command =.. [_|Xs],
        BT =.. [_|Ts],
        parse_cmd_args(Ts, Xs, DocSt, Ys),
        B1 =.. [Cmd|Ys],
        handle_incl_command(B1, DocSt, Verb, NewCommand)
    ; Command =.. [CommandName,Body],
      NewCommand = err(parse_error(unrecognizedcmd, [CommandName,
          Body]))
    ),
    !.

```

y el modificado:

```
get_token(DocSt, Verb, Token,S,S0):-
    % Parse a command
    S=[C|S1],  cmdchar(C) ,
    command_body(Struct,S1,S0),!,
    (handle_command(Struct, DocSt, Verb, Token)
    ;append(CommandF,S0,S),
     Token=string_esc(CommandF)).

handle_command(Command, DocSt, Verb, NewCommand):-
    functor(Command, Cmd, A),
    functor(BT, Cmd, A),
```

```
12      ( cmd_type(BT) ->
13          Command =.. [_|Xs],
14          BT =.. [_|Ts],
15          parse_cmd_args(Ts, Xs, DocSt, Ys),
16          B1 =.. [Cmd|Ys],
17          handle_incl_command(B1, DocSt, Verb, NewCommand)
18      ; fail
19      ),
20      !.
```

El cambio introducido en `handle_command` provoca que, cuando detecta un comando que no reconoce el sistema, en lugar de inicializar el token con error simplemente fuerce el fallo del predicado. Por otro lado, el predicado `get_token` se ha cambiado de tipo **D**efinite **C**lause **G**rammar (el operador `->`) a predicado normal (`:-`). Esto nos permitirá que, en caso de fallo de `handle_commmand`, el cual se produce cuando el comando no es uno de los válidos en LPdoc, añadamos dicho comando como si de un string normal se tratase. Un ejemplo de su uso es la aparición del comando `\ldots` que, aunque no es un comando propio de LPdoc si que tiene un uso en la generación de la documentación. Si con la implementación anterior nos hubiese llegado como comando a analizar:

`\ldots`

el programa hubiese devuelto una excepción. Sin embargo, ahora nos devolverá:

`string_esc("\ldots")`

Este módulo no será modificado en el resto de las fases.

3.1.4. Converter

Este módulo es el encargado de realizar la conversión de markdown a markup. Recibimos un String y una variable que, nos indica si este contiene comandos en markdown. Estas variables se las pasaremos al `predicado_docstring/3` que se encarga de devolvernos una lista con el string separado según pertenezca a un comando o a un string normal. Por ejemplo si ejecutamos:

```
1 parse_docstring(_,''Esto es un string @bf{Esto es un comando @href{www.ciao-lang.com}}'',X).
```

Nos devolverá:

```
X=[string_esc("Esto es un string"),
bf([string_esc("Esto es un comando"),href("www.ciao-lang.com")])]
```

De este predicado hemos hablado con más profundidad en el 3.1.3, pero para poder entender su funcionamiento general comentaremos un par de cosas previas. Lo primero es mencionar el uso del primer argumento, en nuestro caso hemos metido una variable anónima porque no nos importa su valor. Sin embargo, si estuviésemos ante un string con markdown deberíamos introducir la constante `markdown`. Finalmente, es importante mencionar que hay comandos que permiten anidación como bf y otros que no como href. Aquellos que si lo permiten, devuelven una lista con comando como valor. En el caso de bf: `[string_esc("Esto es un comando"), href("www.ciao-lang.com")]` y los que tienen directamente el string como valor en

`href ["www.ciao-lang.com"]`. Cuando nos encontremos con el primer caso, deberemos empezar un nueva recursividad del predicado `converter` con la nueva lista de comandos. Para entenderlo mejor lo ejemplificaremos con pseudocódigo en el que todas las variables, que por el momento no nos interesan, las escribiremos como variables anónimas. Es importante recordar que un String se equivale a una lista. A continuación, explicaremos los pasos seguidos cuando estamos ante un comando que dentro tiene una lista de comandos. Primero, mostraremos un pseudocódigo que nos permite hacernos una idea general delos pasos seguidos en `converter_` y, posteriormente, haremos hincapie en cada uno de ello:

```
converter_([bf(ComandosBf)|Comandos],ListAux,ListFinal,_,_,_,_,_,_) :-
1. converter_(ComandosBf,[],ListFinal1,_,_,_,_,_,_) ,
2. Transformamos a MD el comando bf con el contenido de ListFinal1
   y lo guardamos en  ListMD,
3. Añadimos a ListAux  la nueva lista ListMD y lo guardamos NewAuxList
4. converter_(Comandos,NewAuxList,ListFinal,_,_,_,_,_,_).
```

Pasos explicados del algoritmo seguido(podemos ver la correspondencia entre el texto de arriba y el de abajo fijandonos en su numeración):

0. Nos llega como cabeza de la lista que estamos analizando el comando bf cuyo valor es una lista de comandos, por ejemplo:

   ```
   bf([string_esc(''Esto es un comando''),href(''www.ciao-lang.com'')])
   ```

1. llamaremos a `converter_` metiendole como lista a convertir `[string_esc("Esto es un comando"),href("www.ciao-lang.com")]`

2. El paso anterior nos devolverá en una variables el resultado en markdown de la lista `[string_esc("Esto es un comando"),href("www.ciao-lang.com")]`.

3. Añadimos este resultado al string final que devolveremos con la transformación a markdown.

4. seguimos analizando el resto de comando de la lista.

Por otro lado, cuando el comando que nos llega no tiene otros comandos internos no deberemos realizar el paso 2, es decir, el procedimiento sería:

```
converter_([bf(ComandosBf)|Comandos],ListAux,ListFinal,_,_,_,_,_,_) :-
1. converter_(ComandosBf,[],ListFinal1,_,_,_,_,_,_) ,
2. Transformamos a MD el comando bf con el contenido de ListFinal1
   y lo guardamos en  ListMD,
3. Añadimos a ListAux  la nueva lista ListMD y lo guardamos NewAuxList
4. converter_(Comandos,NewAuxList,ListFinal,_,_,_,_,_,_).
```

Como vimos en la tabla 3.1 hay muchos comando que no tienen equivalente en markdown. Estos comandos deberemos dejarlos con su valor markup siguiendo el procedimiento anterior. A continuación, explicaremos los que si van a sufrir variaciones. Para poder explicarlo deberemos fijarnos en los elementos que puede devolver `parse_docstring/3` debido a que son los que tenemos que analizar.

- **section_env([level(X)],_,ListaContenido,_)** Este elemento hace referencia a los comandos **section, subsection y subsubsection**. De sus cuatro parámetros solo nos importarán el primero y el tercero. El primer argumento nos indicará el

tipo sección en el que estamos de tal modo que, si la X vale 2 será section, 3 subsection y 4 subsubsection. En markdown este tipo de comandos se sustituyen por almohadillas. Usaremos el valor de la X como punto de referencia para saber cuantas debemos poner, siempre será una menos que el valor de X. Por tanto, si disponemos del comando:

```
@subsection{Esto es una subsección.}
```

`parse_docstring/3` no devolverá:

```
section_env([level(3)],_,[string_esc("Esto es una subsección")],_)
```

Basándonos en el procedemimiento explicado obtendremos:

```
## Esto es una subsección
```

- **env_(verbatim,Contenido** la variable `Contenido` es una lista que representa el texto entrecerrado por los comandos **@begin{verbatim} @end{verbatim}**. Normalmente, `Contenido` es una lista formada únicamente por el elemento `string_verb`, el cual es el elemento asociado a los verbatim. Sin embargo, esta lista también puede puede incluir caracteres especiales como \. Es importante destacar que, cuando estemos dentro de un verbatim no deberemos escapar los caracteres. Por lo que cuando realicemos la llamada de `converter` con Contenido como lista principal, deberemos activar un flag que nos indique que, no debemos escapar ninguno de los caracteres especiales (@,/,* ...). Este tipo de comando se traduce entre-cerrándolo en triples comillas. Así pues, ante el siguiente texto en markdown:

  ```
  @begin{verbatim}
  Texto de prueba con @@ escapada
  @end{verbatim}
  ```

  se equivaldría a:

  ```
  env_(verbatim,[string_verb("
  Texto de prueba con"),@,string_verb(" escapada
  ")])
  ```

  Es importante notar que se notifican los saltos de línea, lo cual nos facilita el trabajo a la hora de entrecerrarlo entre comillas puesto que el cambio de línea viene ímplicto. Por conseguiente, el resultado final será:

  ````
  ```
  Texto de prueba con @ escapada
  ```
  ````

  Como podemos ver, la @ ha dejado de estar escapada ya que dentro de un verbatim(bloque de código) de markdown no es necesario. Otro punto a destacar es que las tres comillas de inicio deben estar alineadas con las tres comillas finales.

- **math(ListaMath),em(ListaEm),bf(ListaBf)** representa los comandos **@math{}, @em{}, @bf{}**. Estos comandos tienen la particularidad en markup de permitir la anidación, sin embargo, en markdown no lo está. Por consiguiente, cuando estemos ante uno de estos comandos nace la necesidad de tener dos nuevos flags:

- **FlagIn** que nos indica que estamos dentro de uno de los comandos. Cuando activamos este flag estamos informando que no podemos escribir comandos anidados en markdown. Con él, solucionamos este tipo de conflicto:

  ```
  @pred{prueba @bf{comando}}
  ```

  Siguiendo la lógica de traducción usada en lo pasos anteriores deberíamos convertirlo a:

  ```
  @pred{prueba **comando**}
  ```

  Sin embargo, como hemos introducido en este punto la anidación en markdown no esta permitida. Con la nueva heurística lo que haremos antes de convertir `bf` a markdown será mirar si estamos dentro de un comando. Esta comprobación la haremos mediante el `FlagIn`, el cual en caso de estar dentro de un comando estará activado. De manera que al estar activo, como pasa en este ejemplo, dejaremos el comando en markup, es decir, devolveremos:

  ```
  @pred{prueba @bf{comando}}
  ```

- **FlagOut** nos indica que hemos encontrado un comando anidado. Todos los comandos permitidos en el sistema deberán comprobar previo a su traducción si estamos dentro del primer flag y en caso de estarlo activar este último. Este flag solucionaría el siguiente problema:

  ```
  @bf{Esto es @em{prueba}}
  ```

  Por un lado, gracias a `FlagIn` nos aseguraremos que el comando `@em{}` no sea escrito en markdown dado que se encuentra dentro de otro comando. Por otra parte, cuando analicemos el comando `@em{}` miraremos si `FlagIn` está activo, que en este caso lo estará, para activar `FlagOut`. De esta manera, podremos notificar al comando `bf` que contiene comandos en su interior y, por tanto, no debe ser traducido a markdown dando como resultado:

  ```
  @bf{Esto es @em{prueba}}
  ```

  en lugar de

  ```
  @bf{Esto es @em{prueba}}
  ```

Finalmente, podemos ver su uso completo analizando los pasos a seguir para truducir:

```
@bf{Esto esta en engrita @em{y en cursiva''}}
```

Nos llega a `converter_` tras aplicar `parse_docstring` el elemento:

```
bf([string_esc("Esto esta en negrita"),
em([string_esc("y en cursiva")])])
```

Para conseguir transformar este comando deberemos analizar antes su lista de comandos, es decir, llamar a `converter_` con `string_esc(”Esto esta en negrita”), em([string_esc(”y en cursiva”)])]`. En esta llamada deberemos activar el primer flag para indicar que estamos dentro de uno los comandos especiales. Gracias a la activación de este flag, cuando lleguemos a analizar `string_esc( em([string_esc(”y en cursiva”)])]`, sabremos que estamos ante un comando

y, por consiguiente, no debemos traducirlo a markdown si no dejarlo en markup. A su vez, activaremos `flagOut` para informar de que hemos encontrado un comando dentro del comando `@bf{}` y que, por tanto, no deberemos traducirlo a markdown. Finalmente, en caso de encontrarnos ante estos comandos sin ningún tipo de anidación obtendremos lo equivalente en markdown siendo estos:

- **@em{Texto em}** en markdown *Texto em*
- **@bf{Texto bf}** en markdown **Texto bf**
- **@math{Texto math}** en markdown $Texto math$

- **tt(ListaTt), var(ListaVar):** como su nombre indica representa a los comandos **@var{} y @tt{}**. Este tipo de comando comparte las particularidades de los anteriores añadiendo además una nueva. Al igual que pasaba en verbatim, dentro de estos no deberemos escapar los caracteres que haya dentro. Ambos comandos se traducirán de la misma manera, es decir, `@var{Text}` y `@tt{Text}` equivaldrán a 'Text'.
- **href(URL,Text):** hace referencia al comando **@href{URL}{Text}**. En este caso estaremos ante dos componentes que forman el href. Por un lado, tenemos URL el cual estará formado unicamente por un string, por lo que no deberemos aplicar de nuevo `converter_` sobre él, pero si escaparlo. Text, por otro lado, es una lista de comandos así que deberemos aplicar nuestro algoritmo para traducir a markdown la lista Text. La particularidad de este comando radica en que sus argumentos en markdown cambian de orden, es decir, ante el comando:

  ```
  href(URL,TextList)
  ```

  obtendremos:

  ```
  [TextList](URL)
  ```

- **env_(itemize,ListaItems):** hace referencia a **@begin{itemize} @item @end{itemize}:** Dentro de los tres tipos de lista este es el que tiene la transformación más sencilla y nos servirá como punto de referencia para explicar las otras dos. Tomaremos como punto de partida el siguiente ejemplo:

  ```
  @begin{itemize}
  @item Hola
  @end{itemize}",X).
  ```

  Al introducirlo en parse_docstring obtendremos:

  ```
  X = [env_(itemize,[string_esc(" "),break([]),item([]),
  string_esc(" "),string_esc("Hola"),break([])])
  ```

  Lo primero que debemos hacer será eliminar los saltos de línea o de párrafo que, haya al principio y al final de itemize. Esto lo haremos debido a que, cuando realizamos la transformación tanto la línea de `@begin{itemize}` como `@end{itemize}` desaparecerán. Si eliminásramos solo el contenido y no los saltos de línea nos quedarían dos líneas en blanco innecesarias. Todos los elementos de una lista markdown comienzan con la misma tabulación, indentando dos espacios a la derecha cada vez que creemos un nueva lista o sublista. Visto con un ejemplo, si tenemos en markdown:

```
- Elemento 1
- Elemento 2
  - Elemento 3
```

tendremos una lista con Elemento 1 y Elemento 2 y una sublista con Elemento 3. Cada línea de un ítem comenzará con un espacio respecto al guión que marca el comienzo del ítem, es decir, si tenemos un ítem con dos líneas lo representaríamos:

```
- Elemento 1 Linea 1
  Elemento 1 Linea 2
```

Con ello, aparece la necesidad de tener un flag que nos indique la tabulación que debemos tener en cada momento, ya que no será la misma cuando estemos dentro de una lista que fuera. Cada vez que nos encontremos ante una lista deberemos aumentar en dos unidades la tabulación anterior. De esta manera controlaremos la indentación necesaria en cada momento.
El otro rasgo a tratar sobre las lista es la aparición de los elementos ítem. En este tipo de listas cuando nos aparezca este tipo de elementos deberemos incluir en nuestro string de elementos markdown un guión y un espacio, ya que es su equivalente en markdown.

- **env_(description,ListaItems):** símboliza las listas **@begin{description} @item{} @end{description}** Para comprender mejor este tipo de listas partiremos del siguiente ejemplo:

  ```
  @begin{description}
  @item{Prueba} Descripción
  @end{description}
  ```

  lo cual se equivale a:

  ```
  env_(description,[string_esc(" "),break([]),
  item([string_esc("Prueba")]), string_esc(" "),
  string_esc("Descripción"), break([])])
  ```

  Como podemos comprobar, los ítem están enfocados en realizar definiciones sobre un término dado. Al traducirlo a markdown el contenido del comando ítem debe ser seguido de dos puntos que indican el principio de la definición, es decir, respecto al caso anterior debemos conseguir:

  ```
  - Prueba :: Descripción
  ```

  Para ello, será importante notificar a los ítem cuando nos encontramos ante una lista description y no ante una itemize o enumerate, para lo que contamos con un flag que nos permite diferenciarlo. Los ítem de tipo description nos llegan con una lista con el contenido de término al que queremos agregar una descripción. Como con todas las listas en markdown diferenciaremos cada elemento mediante un guión. Una vez hayamos transformado el contenido del ítem a markdown deberemos añadir los dos punto característicos de este tipo de listas. Finalmente, procederemos a seguir transformando el resto de elementos.

- **env_(enumerate,ListaItems):** representa los comandos **@begin{enumerate} @item @end{enumerate}** Dentro de este tipo de lista deberemos diferenciar dos casos:

las listas autonumeradas y las que numeramos nosotros. Como en el caso anterior, el funcionamiento general de la traducción funcionará igual que las itemize diferenciándose exclusivamente en el tratamiento propio de los ítem. Empezaremos explicando las primeras con el siguiente caso:

```
@begin{enumerate}
@item First
@end{enumerate}
```

Esto equivaldría a:

```
[env_(enumerate,[string_esc(" "),break([]),item([]),
string_esc(" "),string_esc("First"),break([])])]
```

Las listas numeradas automáticamente no sufren apenas modificaciones respecto a las itemizer. El único cambio que deberemos hacer será añadir # tras el guión que, escribimos siempre que comienza un ítem. De este modo obtendremos el resultado:

```
-# First
```

Para la segunda opción usaremos el ejemplo:

```
@begin{enumerate}
@item{1} First
@end{enumerate
```

que se transforma tras aplicar parse_docstring en:

```
env_(enumerate,[string_esc(" "),break([]),item([string_esc("1")]),
string_esc(" "),string_esc("First")
```

Este tipo de lista numerada sirve para que seamos nosotros los que marquemos la numeración de cada ítem. Para traducirlo a markdown, únicamente deberemos tomar como referencia el número que aparezca en el ítem, para saber cual será la numeración deseada y añadirle un punto detrás. De este modo indicamos que queremos una lista numerada por nosotros siendo la traducción del caso de ejemplo:

```
1. First
```

Obviamente, todas los tipos de lista citadas pueden combinarse entre ellos. Este es uno de lo mejores ejemplos de mejora de markdown respecto markup, debido al aumento de legibilidad de los términos y su mayor

- **break([]),p([])** representa los saltos de línea y de párrafo respectivamente. En esta primera fase únicamente deberán añadir un salto de línea en el primer caso o dos si estamos ante el segundo.
- **string_esc(ListaString)** este es elemento que representa todo aquello que no es una comando. Lo único que deberemos hacer con la variable `ListaString` será escapar los caracteres, ya que, como hemos visto en el módulo `command_parser`, los caracteres llegarán sin escapar. Al estar dentro de un string, para escapar deberemos utilizar `\\`. Los caracteres a escapar serán aquellos que tengan un significado en markdown, es decir, `*,_,$,"` y `‘`. Si llegásemos a encontrar alguno de estos símbolos deberemos devolver `\*,\_,\$,` y `\‘`.

- **string_verb(ListaVerb):** Como hemos comentado anteriormente, este tipo de elemento representa el contenido del interior de un verbatim. Este texto no debe ser escapado, ya que al estar dentro de un verbatim no es necesario, por lo que no deberemos modificarlo. Sin embargo, es importante destacar de cara a las futuras fases que, en caso de que aparezca un salto de línea, no vendrá dado con el token break([]) si no que aparecerá tal cual.

3.2. Fase 2

El próposito de la fase 2 es añadir al funcionamiento de la fase 1, la transformación de las aserciones `doc/2` a doccomments. Esto quiere decir que, cuando aplicamos la fase 2 estamos aplicando la fase1+fase2.

```
:-doc(title,"Titulo del manual").
```

a esto:

```
%! @title Titulo del manual
```

Es importante destacar, que en esta fase se excluirán aquellos `doc/2` cuyo primer argumento se trate del nombre de un predicado. A partir de ahora, denominaremos al primer argumento `CommentType` y al segundo `CommentText`. En la siguiente figura clasificaremos todos los valores posibles que puede tomar `CommentType`:

Como podemos ver en la figura 3.4 tenemos dos grandes grupos principales. Por un lado, tenemos los comandos que no deben ser traducidos a estilo `doccomment`, ya que no tienen equivalente funcionando en `doccomments` (a excepción de `PredicateName`, que si tiene pero lo hará en la fase 3) pero sí serán traducidos a markdown. Por otro lado, tenemos los comandos, que si serán traducidos pero a su vez esta subdivididos en tres grupos. Estos subgrupos se diferencian en la forma en la que deberemos traducir cada uno.

- Grupo 1: Lo diferente de este grupo es que cuando nos encontramos ante un predicado mutilínea, todas las líneas exceptuando la primera deberán estar indentadas tres espacios respecto al `CommentType`. Por ejemplo, si tenemos:
  ```
  :-doc(title,"Title Line 1
  Title Line 2").
  ```
  su equivalente será:
  ```
  %! \title Title Line 1
  %     Title Line 2
  ```
- Grupo 2: La peculiaridad de este grupo es que la primera línea será escrita debajo del `CommmentType` y no seguido de este. Si escribimos la siguiente aserción:
  ```
  :-doc(module,"Module Line 1
  Module Line 2").
  ```
  en `doccomments` se representará:
  ```
  %! \module
  %  Module Line 1
  %  Module Line 2
  ```
- Group 3: Este grupo no tiene ninguna particularidad, ya que esta compuesta por el `CommentType` y un término o lista de términos. En estilo `doc\2` recibiremos:
  ```
  :-doc(hide,p\3).
  ```

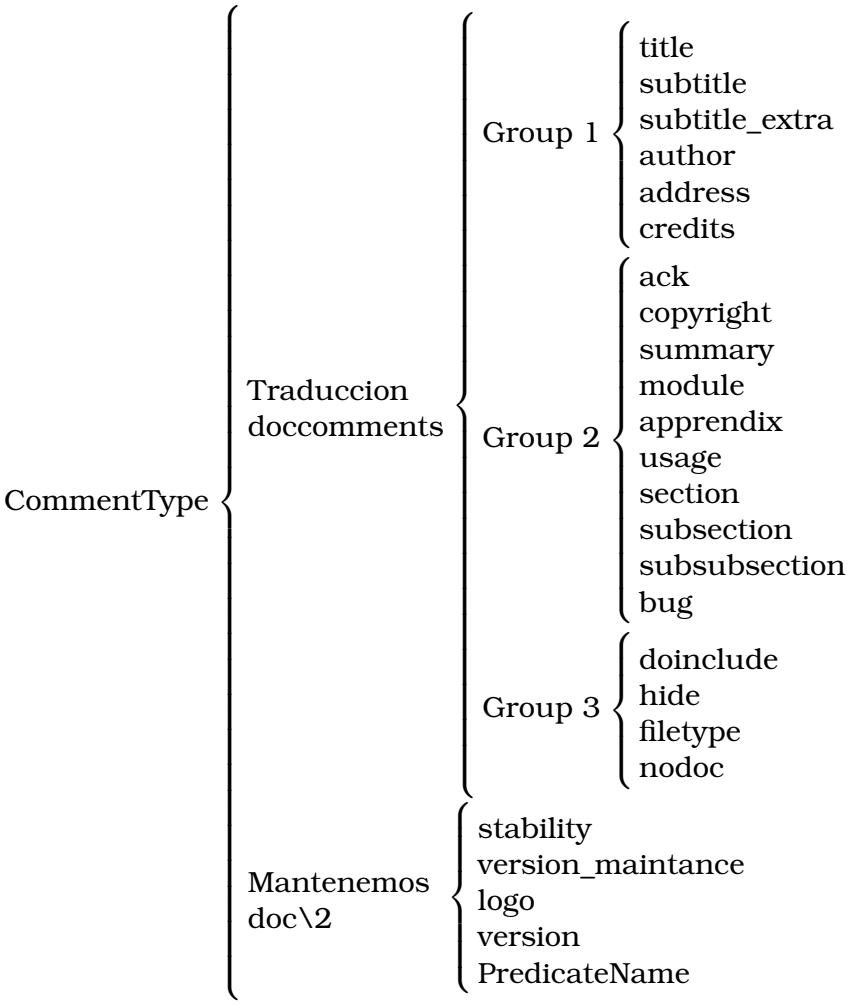

Figura 3.4: Clasifiación de los tipos de comentarios

por lo que se traducirá como:
`%! \hide p\3`

Aunque hay más opciones de traducción para cada uno de los `CommentTypes`, se ha adaptado este traductor para la necesidades del grupo Ciao. En base a estas agrupaciones, podremos realizar esta fase del proyecto.

Esta traducción conllevará realizar más cambios en el fichero original. Primero, debemos tener en cuenta, que para que podamos usar el formato `doccomments` el paquete `doccommentsa` tiene que ser importado. Otro factor a contemplar es que en caso de existir `doc\2` seguido de un comentario, estos deben separarse, ya que si no el comentario pasará a ser parte de la documentación, es decir, en caso de tener:

```
:-doc(title,"Prueba").
% Comentario prueba
```

deberemos traducirlo:

```
%! \title Prueba

% Comentario prueba
```

y no:

```
%! \title Prueba
% Comentario prueba
```

3.2.1. Tokenizer

A los cambios implementados incorporaremos una última modificación. En esta fase ha nacido la necesidad de añadir el paquete doccomments en caso de que no este importado. Este paquete puede ser includo de dos maneras:

- **:- use_package(doccomments).**
- **:- module(...,...,[doccomments]).**

En base a estas opciones crearemos dos tokens. El primero hará referencia a la importación mediante `use_package` y el segundo en module. Para el primer caso, deberemos mirar, que cuando nos encontremos con un `use_package` el paquete que importa sea `doccomments`. Por tanto, si recibimos:

```
:- use_package(doccomments).
```

devolveremos:

```
token(doccomments,":- use_package(doccomments).")
```

Sin embargo, si nos llega:

```
:- use_package(assertions).
```

el resultado será:

```
token(operator,":-"),
token(openfunc,"use_package("),
token(atom,"doccomments"),
token(closepar,")"),
token(endclause,".")
```

Sin embargo, en el caso de la segunda opción, no será necesario que la aserción module incluya en su interior `doccomments`. Por consiguiente, ante el caso siguiente:

```
:- module(_,[],[doccomments]).
```

recibiremos el token:

```
token(module,":- module(_,[],[doccomments]).")
```

Del mismo modo que lo haremos con:

```
:- module(Ejemplo,[p/3],[p/2,p1]).
```

Obteniendo:

```
token(module,":- module(Ejemplo,[p/3],[p/2,p1]).")
```

3.2.2. Reformat

En esta fase, Reformat.pl toma un papel más importante, ya no se encarga solo de llamar a los predicados `identify_tokens/3` y `converter/10`. Su nueva misión será transformar las aserciones `doc/2` a `doccomments`. Dividiremos esto en dos tareas:

1. **Transformar a doccomments**
2. **Añadir paquete doccomments**

Empezaremos con la primera tarea. Para poder llevarla a cabo deberemos recordar el funcionamiento explicado en el capítulo 3.1.2. Como ya sabemos, gracias a univ obtenemos un lista, en la que tenemos los argumentos del predicado dos separados en una lista. Para poder seguir la explicación usaremos el mismo ejemplo que en el capitulo 3.1.2:

```
1 ?.- doc(module,"Esto es un @bf{ejemplo}")=..X.
2 X=[doc,module,"Esto es un @bf{ejemplo}"].
```

Si bien en la fase uno nos centramos en el segundo argumento, en esta segunda fase el protagonista será el primero. Una vez tenemos la lista devuelta por univ, deberemos comprobar a que grupo de los distinguidos en la figura 3.3.1 pertenece. Para ello, contamos con el predicado `commandDoc/4`, que se encarga de asociar cada tipo de comentario con el grupo en base a la fase en la que estamos. Las asociaciones serán:

- **Group 1:** comT.
- **Group 2:** comT1.
- **Group 3:** com.
- **Group 4:** doc.
- **Group 5:** docT.

Según el grupo ante el que estemos deberemos aplicar un tratamiento distinto:

comT: Este grupo deberá ser transformado a `doccomments` y markdown. Por consiguiente, lo primero será traducir a `doccomments` el primero gracias al predicado `commandDoc/4`. Una vez hemos conseguido esto, deberemos traducir el segundo argumento a markdown y `doccomments` mediante el predicado `converter/5`. En este caso, en lugar de meter como tipo string meteremos `comT`. De esta manera, informamos a `converter/5`, que el argumento debe ser traducido en modo `doccomments`. Esto implicará que el carácter " no deberá ser escapado ya que no estamos dentro de un string. Una vez, tenemos ambos predicados en `doccomments` y en markdown los añadiremos al resultado final.

comT1: El procedimiento a seguir será el mismo que el utilizado en el grupo anterior. Únicamente cambiará que cuando llamemos a `converter` introduciremos el tipo `comT1` en lugar de `com`.

com: Este grupo representa aquellas aserciones `doc` que queremos traducir `doccomments` pero que su segundo elemento no contendrá nunca comandos en markdown. Esto sucede debido a que el segundo argumento siempre será un término o lista de ellos. Por tanto, para llevar a cabo dicha transformación, lo que deberemos hacer será añadir a la transformación devuelta por `commandDoc`, el término o lista de términos del segundo argumento sin necesidad de pasar por el predicado `converter`.

doc: En este caso, las aserciones de `doc` no sufren ningún tipo de cambio no se pasa a `doccomments` y tampoco contienen comandos en markdown. Por tanto,

deberemos añadir al contenido final el contenido tal cual del token. Por ejemplo, si tenemos el token:

```
token(doc,":- doc(stability,alpha).")
```

en el documento final aparecerá:

```
:- doc(stability,alpha).
```

docT: Cuando estamos ante elemento perteneciente a este grupo significa, que deberemos mantener la estructura `doc` pero traducir el segundo argumento a markdown en caso de ser posible. Por consiguiente, este grupo seguirá el mismo procedimiento que realizábamos en `reformat` durante la fase 1.

Los puntos anteriores nos resumen los procedimientos seguidos para realizar las traducciones pero esto tiene más implicaciones.

Cuando traducimos una aserción `doc` a `doccomments` podemos encontrarnos un problema con los comentarios. Si nos encontramos, por ejemplo, el siguiente caso:

```
:- doc(module,"This is an example").
% Comment
```

al traducir la aserción, el comentario pasará a formar parte de la documentación, ya que tendríamos el siguiente resultado:

```
%! @module
%  This is an example
% Comment
```

Para evitar este problema, lo que haremos será activar un flag (`FlagSpaces`) que se encarga de controlar esta situación. Cuando tenemos este flag activado si al tratar el siguiente token, que será uno de tipo `spaces`, lo que haremos será comprobar si el consecutivo es un comentario. En ese caso deberemos añadir los saltos de línea necesarios. Para ello, tendremos que comprobar cuantos saltos de línea contiene el token `spaces`. En caso de incluir menos de dos, deberemos ser nosotros los que los añadamos. De tal modo que ante el caso del ejemplo anterior obtengamos el siguiente resultado:

```
%! @module
%  This is an example

% Comment
```

De esta manera, el problema queda solventado. Esta solución sirve también para el siguiente problema:

```
:- doc(module,"This is an example"). % Comment
```

ya que con la resolución aportada seguiríamos obteniendo:

```
%! @module
%  This is an example

% Comment
```

En caso de que no haya conflicto puesto que haya más de dos saltos de línea, se mantendrá el número original de estos. Por ejemplo, ante la siguiente entrada:

```
:- doc(module,"This is an example").


% Comment
```

al no haber problema al traducir, obtendremos los saltos de línea sin modificaciones:

```
%! @module
%  This is an example


% Comment
```

Relacionado con el caso anterior tenemos otro factor a tener en cuenta. Si nos encontramos varias aserciones del grupo 2 deberemos agruparlas, es decir, si disponemos del siguiente caso:

```
:- doc(title, "Example title" ).
:- doc(subtitle, "Example subtitle").

:- doc(author, "Juan Pedro Gómez").
```

Para poder resolver este problema volveremos a utilizar el `flagSpace`. Cuando lo activemos para los casos generales usaremos el valor `yes`, y si quien lo activa es una aserción `doc` del grupo le daremos el valor `comT`. De esta manera, cuando estemos ante token de tipo `spaces` con el flag activado, lo primero que haremos será comprobar si el siguiente token es un comentario. Si esto falla, y el `flagSpaces` es de tipo `comT`, deberemos comprobar si el siguiente token es de tipo `doc`. En caso de serlo, si también es del segundo grupo, tendremos que unirlos. Para ello, a cada salto de línea que hubiese en el token de tipo `spaces` le añadiremos `%`. Finalmente, también contemplaremos que al unir dichas aserciones solo será necesario que la primera comience por `%!`. Todas las demás deberán hacerlo únicamente con `%`. En base a esto, el resultado obtenido será:

```
%! @title Example title
%  @subtitle Example subtitle
%
%  @author Juan Pedro Gómez
```

Cuando traducimos a `doccomments` para que funcione de manera correcta será necesario, en caso de que no exista ya, añadir el paquete `doccomments`. Dividiremos este objetivo en tres tareas:

1. **Notificar si necesitamos el paquete `doccomments`.**
2. **Ver si paquete `doccomments` está importado**. Dividiremos este punto en dos subtareas:
   a) Mirar si existe el token module y si importar el paquete `doccomments`.
   b) Comprobar si se encuentra el token `doccomment`.

3. **Importa el paquete `doccomments`**. Este paso se realizará únicamente en caso de que la primera sea cierta y en la segunda no encuentre el paquete.

Para la primera tarea lo que haremos será activar el flag `FlagDoccAdd`, en caso de que realicemos una traducción a `doccomments`. Para las otras dos contamos con dos flags, `FlagModule` y `FlagDoccomments`. El primer flag se encargará de notificarnos la existencia de la declaración module y el segundo la importación del paquete `doccomments`. Una vez recibimos la lista de tokens, lo primero que haremos será comprobar si existe el token module, el cual deberá aparecer al comienzo del documento. Si existe, activaremos el `FlagModule` y si, además, importa el paquete `doccomments` activará el `FlagDoccomments`. Otra manera de activar el `Flagdoccomments` será si durante el análisis del resto de tokens, aparece el token `doccomments`. Finalmente, el predicado `checkDoccomments` se encargará de añadir los cambios pertinentes según el valor de los tokens. En la tabla siguiente describiremos todas las posibilidades siguiendo las siguiente nomenclatura: **X** representará que el flag esta desactivado, ✓ si el flag esta activado y **-** si es indiferente el valor del flags en ese caso.

FlagDoccAdd	FlagDoccomments	FlagModule	Resultado
X	-	-	FlagDoccAdd esta desactivado por lo que no necesitamos importar el paquete doccomments. No hacemos modificaciones
-	✓	-	El paquete doccomments está ya importado. No hacemos modificaciones
✓	X	✓	Importaremos el paquete dentro de la declaración module
✓	X	X	Añadimos el paquete mediante :-use_package(doccomments).

Cuadro 3.1: Aparición del predicado doccomments

Para entender mejor este último razonamiento, usaremos dos ejemplos para las dos últimas filas, ya que las dos primeras no sufren variaciones. En el siguiente caso que representa `FlagDoccAdd`(✓), `FlagDoccomments`(X) y `FlagModule`(✓):

```
:- module(ejemplo,[],[assertions].
:- doc(module, "@bf{Test}").
```

devolveremos:

```
:- module(ejemplo,[],[assertions,doccomments].
%! @module
%  **Test**
```

Sin embargo, si estamos ante el caso:

```
:- doc(module, "@bf{Test}").
```

el resultado será:

```
:- use_package(doccomments).
```

```
%! @module
%  **Test**
```

3.2.3. Converter

La conversión a doccomments implicará varios cambios en este módulo. Por un lado, si estamos dentro de un elemento doccomments, ya sea de tipo comT o comT1, los saltos de línea que incluyan, deberán ser sucedidos por el símbolo % acompañado de la tabulación correspondiente. Para el tipo comT, la tabulación base será 5 y para comT1 2. Esto es debido a que la alineación en la que queremos mostrar depende del grupo en el que estemos. Recordemos que si estamos ante una aserción doc/2 de tipo comT de más de una línea como, por ejemplo:

```
:- doc(title, "Este título
tiene dos líneas").
```

deberemos devolver en la traducción:

```
%! @title Este título
%     tiene dos líneas
```

Mientas que en los de tipo comT1 :

```
%! @title Este título
%     tiene dos líneas
```

el resultado será:

```
%! @module
%  Este módulo
%  tiene dos líneas
```

En este apartado deberemos tener en cuenta el siguiente caso:

```
:- doc(module, " Esta frase empieza por espacio
y tiene dos lineas").
```

Siguiendo la estructura explicada hasta ahora el resultado devuelto sería:

```
%! @module
%   Este modulo empieza por espacio
%  y nos puede dar problemas
```

Sin embargo, esto al generar la documentación no nos generará lo que deseamos. Tras finalizar la primera sentencia imprimiría un salto de línea en lugar de escribir el texto seguido. Esto quiero decir que obtendremos:

```
Esta frase empieza por espacio
y nos puede dar problemas
```

en lugar de:

```
Esta frase empieza por espacio
y nos puede dar problemas
```

Esto sucede debido a que la primera línea indica la indentación que debe seguir el resto. Para solucionar esto lo que haremos será mirar si el string contiene espacios

en blanco antes de que comience el texto. En caso de encontrarlos, los eliminaremos de esta forma el ejemplo usado quedaría:

```
%! @module
%  Este modulo empieza por espacio
%  y nos puede dar problemas
```

De esta forma, resolvemos este inconveniente.

3.3. Fase 3

En esta fase tendremos dos objetivos. El primero será traducir los predicados `doc\2` cuyo primer argumento sea el nombre de un predicado. Esta traducción será diferente de las realizadas hasta ahora. Por ejemplo, si disponemos del siguiente predicado:

```
:- doc(p/2,"Esto es un ejemplo").
```

deberemos transformarlo con la siguiente estructura:

```
%! p/2:
%  Esto es un ejemplo
```

De esta manera, finalmente, tendremos todos los `doc\2` que tienen equivalente `doccomments` traducidos.

El otro objetivo de esta fase se enfoca en traducir a `doccomments` los comentarios de las aserciones. Esta meta forma parte de un prototipo experimental que se quiere desarrollar para facilitar la escritura de dichos comentarios. Para apoyar en el futuro esta tarea, dejaremos implementada la conversión aunque en la actualidad todavía este en desarrollo. Por lo tanto, ante la siguiente aserción:

```
:- regtype bar(X) # "@var{X} is an acceptable kind of bar.".
```

deberemos obtener:

```
:- regtype bar(X). %< @var{X} is an acceptable kind of bar.
```

Al igual que pasaba cuando traducíamos a `doccomments` las aserciones de tipo `doc`, en caso de haber un comentario seguido al comentario de la aserción deberemos separarlo. Todos los cambios necesarios para esta fase serán llevado a cabo únicamente en el módulo Reformat.

3.3.1. Reformat

Empezaremos explicando la transformación a `doccomments` de los comentarios de las aserciones. Como podemos observar, ahora este tipo de comentarios forman parte del grupo2. Esto supone que cuando llamemos a converter deberemos indicar que estamos ante un tipo `comT1` con las implicaciones que hemos explicado en el punto 3.2.3. Esta conversión afectará a los dos tokens que componen una aserción con comentarios, es decir, `AssertionC` y`CommentA`. Para facilitar la comprensión, utilizaremos el siguiente ejemplo para explicar los pasos seguidos. Partimos de la siguiente aserción con comentarios:

```
 :- pred p(A,B) : integer(A) =>ground(B) # "Esto es un @bf{comentario}".
```

que dividido en tokens equivale a:

```
[token(assertionC,":- pred p(A,B) : integer(A) =>  ground(B) #"),
token(commentA," "Esto es un @bf{comentario}".")]
```

Al detectar el tokens `assertionC` en phase3 deberemos cambiar # por . para indicar el fin de la aserción. Por otro lado, a `commentA` le quitaremos el . final. Finalmente, al contenido de `commentA` traducido por `converter` deberemos añadirle al principio `%<`para indicar que se trata del comentario de una aserción. Por tanto, el resultado final será:

```
 :- pred p(A,B) : integer(A) =>ground(B). %< Esto es un **comentario**
```

En caso de que el comentario tuviese más de una línea seguirá la estructura del grupo 3 siendo, por ejemplo:

```
 :- pred p(A,B) : integer(A) =>ground(B). %< Esto es un **comentario**
 %  Esta es la segunda línea
```

Nuestro segundo objetivo será convertir las aserciones `doc/2` cuyo primer argumento sea un predicado a `doccomments`. El encargado de hacer esto será el predicado `commmandDoc/4`. Cuando detecte que esta ante este tipo de caso, lo que hará será añadir tras el nombre del predicado `:\n%` y al inicio `%!` . De esta manera, tenemos en `doccomments` el nombre del predicado, a continuación, se traducirá el segundo argumento llamando al predicado `converter` con tipo `comT`. De esta manera, si nos fijamos por ejemplo en:

```
:- doc(p(a,b), "Comentario sobre predicado
con líneas").
```

El predicado `commandDoc/4` nos devolverá:

```
%! p(a,b):
%
```

y `converter`:

```
Comentarios sobre predicado
%  con dos líneas
```

Por lo que si unimos ambos tendremos el resultado buscado;

```
%! p(a,b):
%  Comentarios sobre predicado
%  con dos líneas
```

En vista a estos cambios, la estructura de la clasificación de los `CommentType` usada en la figura 3.4 sufrirá dos modificaciones que mostraremos en la figura 3.5.

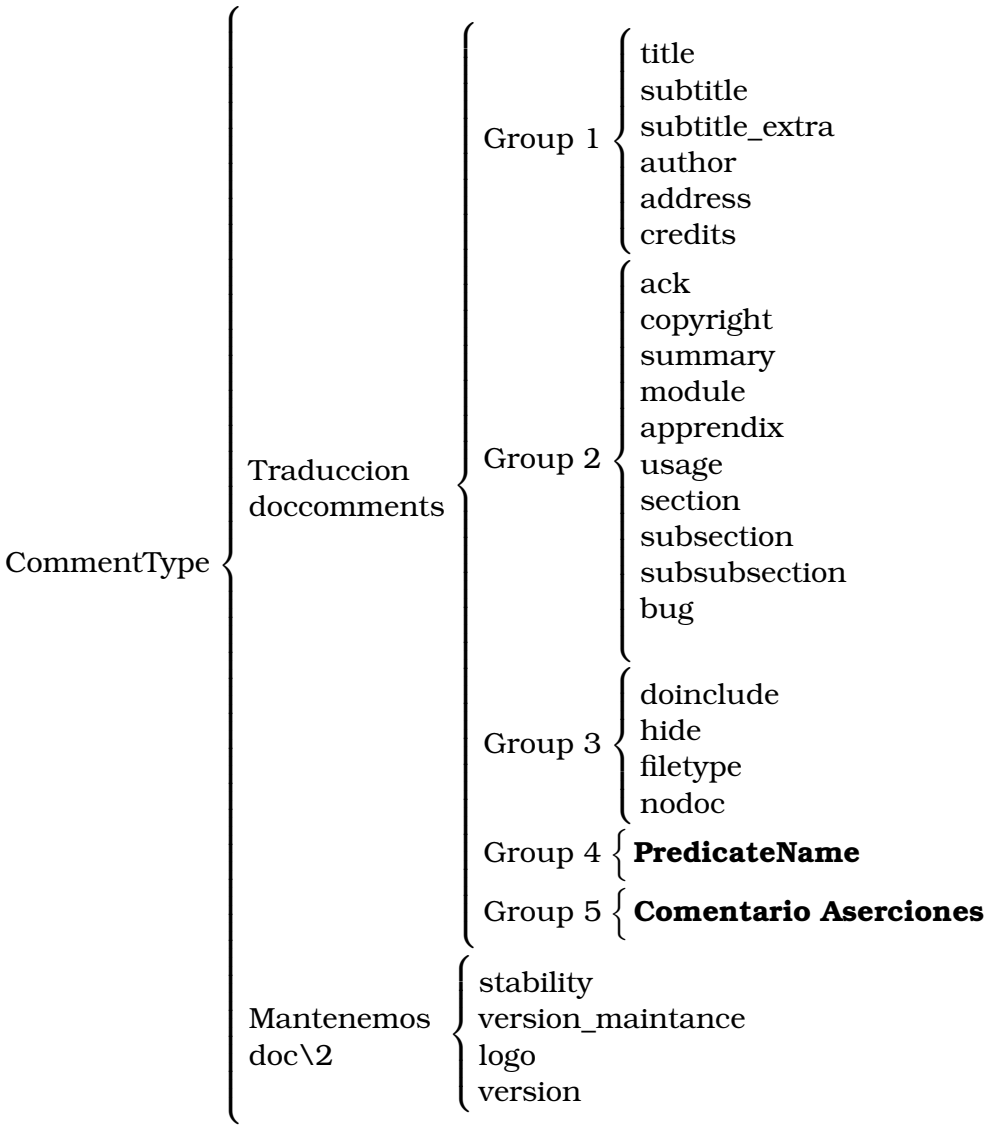

Figura 3.5: Clasifiación de los tipos de comentarios en fase 3

Capítulo 4

Pruebas

Para comprobar el correcto funcionamiento del programa deberemos realizar las pertinentes pruebas. Nuestro objetivo principal es poder ejecutar el programa a lo largo de todos los fichero que componen el repositorio Ciao. Al tratarse tantos programas es muy importante asegurarnos que la aplicación de nuestro conversor no altere el funcionamiento normal de los ficheros ni cambie la salida de la documentación generada.Para conseguir el sistema de pruebas deseado deberemos realizar los siguiente pasos:

1. Instalación de un entorno Ciao. Deberemos ejecutar el comando `lpdoc` sobre todos los ficheros que componen el sistema para obtener todos los manuales. También ejecutaremos ciao build para obtener todos los ficheros compilados.

2. Instalación de un nuevo entorno Ciao. Es importante tener dos entornos instalados. Si solo disponemos de uno surge conflicto cuando generamos la documentación debido a que los `usages` de los predicados salen repetidos.

3. Ejecutaremos nuestro programa sobre todos los archivos.

4. Una vez tenemos todos los ficheros en el nuevos formato procederemos a generar su docuementación con LPdoc así como a compilarlos.

5. Comprobaremos que tanto la documentación generado como los archivos compilados nos hayan sufrido modificaciones.

Al tratarse de un repositorio tan amplio no es posible mostrar todos los ficheros (+8000) sobre los que se han realizados pruebas. Sin embargo, contamos con un fichero test que usaremos para demostrar el correcto funcionamiento del programa. También nos sirve como punto de referencia para comprobar que al realizar modifcaciones no hemos roto nada del funcionamiento. Al igual que hemos hecho con el repositorio original someteremos este fichero a las 3 fases para detectar cualquier tipo de error. Este fichero esta compuesto por los casos más complejos que se han encontrado a lo largo de las pruebas.

4.1. Fichero test

```
:- module(fichero_prueba,[],[assertions]).

:- doc(title,"Lpdowner").
:- doc(subtitle,"Lpdowner test").
:- doc(subtitle_extra,"Lpdowner test: @em{Check} if
 Lpdowner work well").

:- doc(author,"@bf{Will Smith}").

:- doc(address, "@tt{clip@@dia.fi.upm.es}"
||"@tt{http://www.clip.dia.fi.upm.es/}").
:- doc(credits, "The Ciao Development Team").

:- doc(copyright, "
Copyright @copyright{} 1997-2002 The Clip Group.

@include{FreeDocLicense.lpdoc}
").

:- doc(ack, "Some parts of the documentation are taken from the
   documentation of GNU's @apl{gmake}."). % Comment
:- doc(stability, devel). % Comment

:- doc(summary, "
   @em{portability}: compiled to bytecode it runs without need for
   recompilation on any platform where Ciao is supported. Second,
   @em{improved programming capabilities}.").
% Comment

:- doc(usage, "This module is automatically imported by the
   @lib{assertions} package. It can be imported explicitly with the
   @tt{:- use_module(engine(basic_props))} directive.").
:- doc(module, "@section{Simple mark-up}

Text can be @em{emphasized} or @bf{bold}.

@section{Paragraphs}

Begin of
a paragraph.
This paragraph ends here.

Begin of a new paragraph.
This paragraph ends here.

The last new paragraph. This paragraph ends here.
```

```
@section{Lists}

@subsection{Simple lists}

Here is a list:
@begin{itemize}
@item with some item
@item other item
@item more items
@end{itemize}

@subsection{Nested lists}

A list with nested items:
@begin{itemize}
@item item 1
@begin{itemize}
@item subitem 1-1
@item subitem 1-2
@begin{itemize}
@item subitem 1-2-1
@end{itemize}
@item subitem 1-3
@end{itemize}
@item item 2
@begin{itemize}
@item subitem 2-1
@end{itemize}
@item item 3
@end{itemize}

@subsection{Enumerated lists}

Enumerated list with automatic numbering:
@begin{enumerate}
@item First
@item Second
@item Third
@end{enumerate}

@noindent Enumerated list with explicit numbering:
@begin{enumerate}
@item{1} First
@item{2} Second
@item{3} Third
@end{enumerate}

@noindent Enumerated list with non-consecutive explicit numbering:
@begin{enumerate}
```

```
@item{2} First
@item{4} Second
@item{6} Third
@end{enumerate}

@begin{alert}
  The following only works in some backends.
@end{alert}

@noindent Enumerated list mixing all above:
@begin{enumerate}
@item{2} First
@item{4} Second
@item Third
@item Fourth
@item{10} Fifth
@item Sixth
@end{enumerate}

@subsection{Description lists}

Description lists:
@begin{description}
@item{opt} First
@item{foo} Second
@item{bar} Third
@end{description}

@subsection{Description lists (more complex cases)}

Description lists with richer items:
@begin{description}
@item{@tt{atom}} an atom.
@item{@pred{append/3}} a predicate or functor name.
@item{@tt{f(X0,...,Xn)}} some term with variables @var{X0}, ..., @var{Xn}.
@item{@var{X}} a variable.
@item{@math{x^2}} some math.
@item{@math{\bigwedge_j f_j(x_0, \ldots, x_n)}} some complex math.
@end{description}

@section{Sections and subsections}
Text for the section.

@subsection{Subsection}
Text for the subsection.

@subsubsection{Subsubsection}
Text for the subsubsection.
```

```
@section{Links}

A link to @href{https://ciao-lang.org} showing its URL.

A link to @href{https://ciao-lang.org}{Ciao} hiding its URL.

A link to @href{https://ciao-lang.org}{The @bf{Ciao} System} hiding its
URL with a complex string.

@section{Anchors, labels, references, bibliographical citations}

A reference to the first section in this document @ref{Paragraphs}.

@begin{alert}
  Implement symbolic labels. Fix the @tt{texinfo} backend by resolving
  them to the section title.
@end{alert}

@section{Other elements}

We will not include lightweight mark-up syntax for anything else
not described in this document (e.g., images).

@section{Syntax for code}

@subsection{Code spans}

This is a predicate name @pred{append/3}, a variable name @var{X}, an
atom name @tt{foo}, a quoted atom name @tt{'foo'}.

@subsection{Blocks of code}

Text that is 4-char indented is recognized as code:

@begin{verbatim}
list([]).
list([X|Xs]) :- list(Xs)
@end{verbatim}

Code itself can have comments:

@begin{verbatim}
% definition for lists
list([]). % see append/3
list([X|Xs]) :- list(Xs)
@end{verbatim}

Code itself can have documentation comments:
```

```
@begin{verbatim}
%! definition for lists
list([]). %< see ‘append/3‘
list([X|Xs]) :- list(Xs)
@end{verbatim}

@subsection{Blocks of other code}

This is a piece of C code:

@begin{verbatim}
#include <stdio.h>
int main(void) { return 0; }
@end{verbatim}").

:- doc(p/3,"A @bf{general comment} on the predicate." ).
%% Documenting some typical usages of the predicate
:- pred p/3
    : int * int * var
       => int * int * list
    + (iso,not_fails)
    # "This @em{mode} is nice.".
```

4.2. Fase1

4.2.1. Fichero de salida

```
:- module(fichero_prueba,[],[assertions]).

:- doc(title,"Lpdowner").
:- doc(subtitle,"Lpdowner test").
:- doc(subtitle_extra,"Lpdowner test: *Check* if
 Lpdowner work well").

:- doc(author,"**Will Smith**").

:- doc(address,"‘clip@dia.fi.upm.es‘ "
||"‘http://www.clip.dia.fi.upm.es/‘").
:- doc(credits,"The Ciao Development Team").

:- doc(copyright,"
 Copyright @copyright 1997-2002 The Clip Group.

 @include{FreeDocLicense.lpdoc}
 ").

:- doc(ack,"Some parts of the documentation are taken from the
 documentation of GNU’s @apl{gmake}."). % Comment
:- doc(stability,devel). % Comment
```

```
:- doc(summary,"
 *portability*: compiled to bytecode it runs without need for
 recompilation on any platform where Ciao is supported. Second,
 *improved programming capabilities*.").
% Comment

:- doc(usage,"This module is automatically imported by the
 @lib{assertions} package. It can be imported explicitly with the
 `:- use_module(engine(basic_props))` directive.").
:- doc(module,"# Simple mark-up

 Text can be *emphasized* or **bold**.

 # Paragraphs

 Begin of
 a paragraph.
 This paragraph ends here.

 Begin of a new paragraph.
 This paragraph ends here.

 The last new paragraph. This paragraph ends here.

 # Lists

 ## Simple lists

 Here is a list:
 - with some item
 - other item
 - more items

 ## Nested lists

 A list with nested items:
 - item 1
   - subitem 1-1
   - subitem 1-2
     - subitem 1-2-1
   - subitem 1-3
 - item 2
   - subitem 2-1
 - item 3

 ## Enumerated lists

 Enumerated list with automatic numbering:
```

```
-# First
-# Second
-# Third

Enumerated list with explicit numbering:
1. First
2. Second
3. Third

Enumerated list with non-consecutive explicit numbering:
2. First
4. Second
6. Third

@begin{alert}
The following only works in some backends.
@end{alert}

Enumerated list mixing all above:
2. First
4. Second
-# Third
-# Fourth
10. Fifth
-# Sixth

## Description lists

Description lists:
- opt ::  First
- foo ::  Second
- bar ::  Third

## Description lists (more complex cases)

Description lists with richer items:
- ‘atom‘ ::  an atom.
- @pred{append/3} ::  a predicate or functor name.
- ‘f(X0,...,Xn)‘ ::  some term with variables ‘X0‘, ..., ‘Xn‘.
- ‘X‘ ::  a variable.
- $x^2$ ::  some math.
- $\bigwedge_j f_j(x_0, \ldots, x_n)$ ::  some complex math.

# Sections and subsections
Text for the section.

## Subsection
Text for the subsection.
```

```
### Subsubsection
Text for the subsubsection.

# Links

A link to @href{https://ciao-lang.org} showing its URL.

A link to [Ciao](https://ciao-lang.org) hiding its URL.

A link to [The **Ciao** System](https://ciao-lang.org) hiding its
URL with a complex string.

# Anchors, labels, references, bibliographical citations

A reference to the first section in this document @ref{Paragraphs}.

@begin{alert}
Implement symbolic labels. Fix the ‘texinfo‘ backend by resolving
them to the section title.
@end{alert}

# Other elements

We will not include lightweight mark-up syntax for anything else
not described in this document (e.g., images).

# Syntax for code

## Code spans

This is a predicate name @pred{append/3}, a variable name ‘X‘, an
atom name ‘foo‘, a quoted atom name ‘’foo’‘.

## Blocks of code

Text that is 4-char indented is recognized as code:

‘‘‘
list([]).
list([X|Xs]) :- list(Xs)
‘‘‘

Code itself can have comments:

‘‘‘
% definition for lists
list([]). % see append/3
list([X|Xs]) :- list(Xs)
‘‘‘
```

```
 Code itself can have documentation comments:

 ```
 %! definition for lists
 list([]). %< see `append/3`
 list([X|Xs]) :- list(Xs)
 ```

 ## Blocks of other code

 This is a piece of C code:

 ```
 #include <stdio.h>
 int main(void) { return 0; }
 ```").

:- doc(p/3,"A **general comment** on the predicate.").
%% Documenting some typical usages of the predicate
:- pred p/3
    : int * int * var
       => int * int * list
    + (iso,not_fails)
    # "This *mode* is nice.".
```

4.3. Fase2

4.3.1. Fichero de salida

```
:- module(fichero_prueba,[],[assertions,doccomments]).

%! @title Lpdowner
%  @subtitle Lpdowner test
%  @subtitle_extra Lpdowner test: *Check* if
%    Lpdowner work well
%
%  @author **Will Smith**
%
%  @address `clip@dia.fi.upm.es` `http://www.clip.dia.fi.upm.es/`
%  @credits The Ciao Development Team

%! @copyright
%
%  Copyright @copyright 1997-2002 The Clip Group.
%
%  @include{FreeDocLicense.lpdoc}
%
```

```
%! @ack
%  Some parts of the documentation are taken from the
%  documentation of GNU's @apl{gmake}.

% Comment
:- doc(stability,devel). % Comment

%! @summary
%
%  *portability*: compiled to bytecode it runs without need for
%  recompilation on any platform where Ciao is supported. Second,
%  *improved programming capabilities*.

% Comment

%! @usage
%  This module is automatically imported by the
%  @lib{assertions} package. It can be imported explicitly with the
%  `:- use_module(engine(basic_props))` directive.

%! @module
%  # Simple mark-up
%
%  Text can be *emphasized* or **bold**.
%
%  # Paragraphs
%
%  Begin of
%  a paragraph.
%  This paragraph ends here.
%
%  Begin of a new paragraph.
%  This paragraph ends here.
%
%  The last new paragraph. This paragraph ends here.
%
%  # Lists
%
%  ## Simple lists
%
%  Here is a list:
%  - with some item
%  - other item
%  - more items
%
%  ## Nested lists
%
%  A list with nested items:
%  - item 1
```

```
%    - subitem 1-1
%    - subitem 1-2
%      - subitem 1-2-1
%    - subitem 1-3
%  - item 2
%    - subitem 2-1
%  - item 3
%
%  ## Enumerated lists
%
%  Enumerated list with automatic numbering:
%  -# First
%  -# Second
%  -# Third
%
%  Enumerated list with explicit numbering:
%  1. First
%  2. Second
%  3. Third
%
%  Enumerated list with non-consecutive explicit numbering:
%  2. First
%  4. Second
%  6. Third
%
%  @begin{alert}
%  The following only works in some backends.
%  @end{alert}
%
%  Enumerated list mixing all above:
%  2. First
%  4. Second
%  -# Third
%  -# Fourth
%  10. Fifth
%  -# Sixth
%
%  ## Description lists
%
%  Description lists:
%  - opt ::  First
%  - foo ::  Second
%  - bar ::  Third
%
%  ## Description lists (more complex cases)
%
%  Description lists with richer items:
%  - `atom` ::  an atom.
%  - @pred{append/3} ::  a predicate or functor name.
```

```
%  - `f(X0,...,Xn)` ::  some term with variables `X0`, ..., `Xn`.
%  - `X` ::  a variable.
%  - $x^2$ ::  some math.
%  - $\bigwedge_j f_j(x_0, \ldots, x_n)$ ::  some complex math.
%
%  # Sections and subsections
%  Text for the section.
%
%  ## Subsection
%  Text for the subsection.
%
%  ### Subsubsection
%  Text for the subsubsection.
%
%  # Links
%
%  A link to @href{https://ciao-lang.org} showing its URL.
%
%  A link to [Ciao](https://ciao-lang.org) hiding its URL.
%
%  A link to [The **Ciao** System](https://ciao-lang.org) hiding its
%  URL with a complex string.
%
%  # Anchors, labels, references, bibliographical citations
%
%  A reference to the first section in this document @ref{Paragraphs}.
%
%  @begin{alert}
%  Implement symbolic labels. Fix the `texinfo` backend by resolving
%  them to the section title.
%  @end{alert}
%
%  # Other elements
%
%  We will not include lightweight mark-up syntax for anything else
%  not described in this document (e.g., images).
%
%  # Syntax for code
%
%  ## Code spans
%
%  This is a predicate name @pred{append/3}, a variable name `X`, an
%  atom name `foo`, a quoted atom name ``foo''.
%
%  ## Blocks of code
%
%  Text that is 4-char indented is recognized as code:
%
%  ```
```

```
%  list([]).
%  list([X|Xs]) :- list(Xs)
%  ```
%
%  Code itself can have comments:
%
%  ```
%  % definition for lists
%  list([]). % see append/3
%  list([X|Xs]) :- list(Xs)
%  ```
%
%  Code itself can have documentation comments:
%
%  ```
%  %! definition for lists
%  list([]). %< see `append/3`
%  list([X|Xs]) :- list(Xs)
%  ```
%
%  ## Blocks of other code
%
%  This is a piece of C code:
%
%  ```
%  #include <stdio.h>
%  int main(void) { return 0; }
%  ```

:- doc(p/3,"A **general comment** on the predicate.").
%% Documenting some typical usages of the predicate
:- pred p/3
    : int * int * var
       => int * int * list
    + (iso,not_fails)
    # "This *mode* is nice.".
```

4.4. Fase3

4.4.1. Fichero de salida

```
:- module(fichero_prueba,[],[assertions,doccomments]).

%! @title Lpdowner
%  @subtitle Lpdowner test
%  @subtitle_extra Lpdowner test: *Check* if
%    Lpdowner work well
%
%  @author **Will Smith**
```

```
%
%  @address ‘clip@dia.fi.upm.es‘ ‘http://www.clip.dia.fi.upm.es/‘
%  @credits The Ciao Development Team

%! @copyright
%
%  Copyright @copyright 1997-2002 The Clip Group.
%
%  @include{FreeDocLicense.lpdoc}
%

%! @ack
%  Some parts of the documentation are taken from the
%  documentation of GNU’s @apl{gmake}.

% Comment
:- doc(stability,devel). % Comment

%! @summary
%
%  *portability*: compiled to bytecode it runs without need for
%  recompilation on any platform where Ciao is supported. Second,
%  *improved programming capabilities*.

% Comment

%! @usage
%  This module is automatically imported by the
%  @lib{assertions} package. It can be imported explicitly with the
%  ‘:- use_module(engine(basic_props))‘ directive.

%! @module
%  # Simple mark-up
%
%  Text can be *emphasized* or **bold**.
%
%  # Paragraphs
%
%  Begin of
%  a paragraph.
%  This paragraph ends here.
%
%  Begin of a new paragraph.
%  This paragraph ends here.
%
%  The last new paragraph. This paragraph ends here.
%
%  # Lists
%
```

```
%  ## Simple lists
%
%  Here is a list:
%  - with some item
%  - other item
%  - more items
%
%  ## Nested lists
%
%  A list with nested items:
%  - item 1
%    - subitem 1-1
%    - subitem 1-2
%      - subitem 1-2-1
%    - subitem 1-3
%  - item 2
%    - subitem 2-1
%  - item 3
%
%  ## Enumerated lists
%
%  Enumerated list with automatic numbering:
%  -# First
%  -# Second
%  -# Third
%
%  Enumerated list with explicit numbering:
%  1. First
%  2. Second
%  3. Third
%
%  Enumerated list with non-consecutive explicit numbering:
%  2. First
%  4. Second
%  6. Third
%
%  @begin{alert}
%  The following only works in some backends.
%  @end{alert}
%
%  Enumerated list mixing all above:
%  2. First
%  4. Second
%  -# Third
%  -# Fourth
%  10. Fifth
%  -# Sixth
%
%  ## Description lists
```

```
%
%  Description lists:
%  - opt ::  First
%  - foo ::  Second
%  - bar ::  Third
%
%  ## Description lists (more complex cases)
%
%  Description lists with richer items:
%  - ‘atom‘ ::  an atom.
%  - @pred{append/3} ::  a predicate or functor name.
%  - ‘f(X0,...,Xn)‘ ::  some term with variables ‘X0‘, ..., ‘Xn‘.
%  - ‘X‘ ::  a variable.
%  - $x^2$ ::  some math.
%  - $\bigwedge_j f_j(x_0, \ldots, x_n)$ ::  some complex math.
%
%  # Sections and subsections
%  Text for the section.
%
%  ## Subsection
%  Text for the subsection.
%
%  ### Subsubsection
%  Text for the subsubsection.
%
%  # Links
%
%  A link to @href{https://ciao-lang.org} showing its URL.
%
%  A link to [Ciao](https://ciao-lang.org) hiding its URL.
%
%  A link to [The **Ciao** System](https://ciao-lang.org) hiding its
%  URL with a complex string.
%
%  # Anchors, labels, references, bibliographical citations
%
%  A reference to the first section in this document @ref{Paragraphs}.
%
%  @begin{alert}
%  Implement symbolic labels. Fix the ‘texinfo‘ backend by resolving
%  them to the section title.
%  @end{alert}
%
%  # Other elements
%
%  We will not include lightweight mark-up syntax for anything else
%  not described in this document (e.g., images).
%
%  # Syntax for code
```

```
%
%  ## Code spans
%
%  This is a predicate name @pred{append/3}, a variable name `X`, an
%  atom name `foo`, a quoted atom name ``foo``.
%
%  ## Blocks of code
%
%  Text that is 4-char indented is recognized as code:
%
%  ```
%  list([]).
%  list([X|Xs]) :- list(Xs)
%  ```
%
%  Code itself can have comments:
%
%  ```
%  % definition for lists
%  list([]). % see append/3
%  list([X|Xs]) :- list(Xs)
%  ```
%
%  Code itself can have documentation comments:
%
%  ```
%  %! definition for lists
%  list([]). %< see `append/3`
%  list([X|Xs]) :- list(Xs)
%  ```
%
%  ## Blocks of other code
%
%  This is a piece of C code:
%
%  ```
%  #include <stdio.h>
%  int main(void) { return 0; }
%  ```

%! p/3:
%  A **general comment** on the predicate.

%% Documenting some typical usages of the predicate
:- pred p/3
    : int * int * var
       => int * int * list
    + (iso,not_fails)
    . %< This *mode* is nice.
```

Capítulo 5

Resultados y conclusiones

Tras la realización de las prueba podemos observar el correcto funcionamiento del programa. Dividir la implementación en tres fases nos has permitido localizar de manera más sencilla cualquier tipo de error además de obtener un programa más versátil que se adapte en cada momento a nuestras necesidades. Al contar con una batería de pruebas tan extensa, hemos podido probar todo tipo de casos que quizás con un sistemas propio no nos hubiésemos imaginado. Uno de los principales problemas encontrados ha sido la propia gran diversidad permitida por LPdoc. Al tratarse de un herramienta que tiene que analizar documentación escritas por otras personas nos hemos encontrado con muchos casos donde los propios documentos estaban mal escrito. Parte de esta trabajo, aunque únicamente quede reflejado de manera superficial ha sido detectar los errores de sintaxis en la documentación original y corregirlo para que al aplicar el programa no sufra ningún fallo. Por este motivo, cuando se realicen futuras ejecuciones del programa es recomendable mirar la salida obtenida por si acaso nos encontramos con alguno de estos casos.

Otro problema encontrado y subsanado ha sido la incorporación de todos los comandos a `doccomments`. Al realizar las pruebas, pudimos comprobar que cuando teníamos, por ejemplo:

```
%! @ack
%  Esto es un ejemplo
```

pese a estar bien escrito no quedaba reflejado en la documentación. Esto era debido a que en la implementación original de `doccomments` no estaban incluidos los tipos de comentarios `ack`, `usage`, `appendix` entre otros. Finalmente, también nos ha servido para encontrar una fallo en la sintaxis markdown de los comandos. Anteriormente, si teníamos,por ejemplo, la palabra:

predicado_con_barrabaja

el resultado obtenido sería:

predicado*con*barrabaja

En la actualidad esto ya no sucede así. Los símbolos identificados como markdown únicamente actuarán como tal si son empiece o final de una palabra o varias y no si están en su interior. Esto quiere decía que la palabra anterior ahora aparecerá como:

predicado_con_barrabaja

Dentro del mismo tema, también fue localizado un bug que provocaba que los comandos en markdown no permitiesen saltos de líneas en su interior.

La idea final para este proyecto será incorporarlo como un bundle (paquete de distribución de software) del sistema Ciao Prolog. De esta manera, quedará visible para cualquier usuario que quiera hacer ejercicio de sus funcionalidades.

Como comentario final, destacar que el manual incluido en el anexo ha sido escrito en markup y después procesado mediante LPdowner (herramienta implementada) y LPdoc. De esta manera, tenemos otra vez constancia del buen funcionamiento y de sus utilidades.

5.1. Trabajo futuro

Como ya hemos comentado anteriormente, tratamos con temas que aún se encuentran en una fase experimental de desarrollo. Esto puede provocar que en un futuro nuestra solución pueda ser sometida a cambio. Otro punto a tratar podría ser la incorporación de nuevos comandos que queramos que tengan su equivalencia en markdown. Cabe recordar la existencia de comandos que tienen múltiples traducciones y, en un primer momento, se ha elegido las más conveniente para las necesidades actuales pero estas pueden cambiar. Los desarrolladores de Ciao Prolog tienen pensado incorporar nuevas mejoras en el sistema de LPdoc y más concretamente en su paquete `doccomments` que es con el que hemos trabajado nosotros. Dichas actualizaciones pueden crear nuevas necesidades y, por lo tanto, nuevos cambios en traducción.

De manera general podemos ver que el futuro de este herramienta en cuando a cambios o modificaciones dependerá mucho de las necesidades que se vayan creando. Si el día de mañana decidimos cambiar como escribimos un comando nos será muy útil ya que con un pequeño cambio tendremos toda la documentación en el nuevo formato. Otra funcionalidad muy ligada a la actual que se podría añadir es la de embellecer el código, es decir, conseguir una estructura más ordenada de los comentarios de documentación. Esto puede incluir reajustar la longitud de las líneas a una medida deseada, poner toda la documentación en un parte específica del documento u otros requisitos que vayan apareciendo con el tiempo.

Otra posible salida que puede tener el sistema desarrollado en vista a un futuro más lejano es la traducción a otro nuevo lenguaje. Sabemos que en la informática es un sector que avanza muy rápido por lo que no es difícil imaginar la aparición de un nuevo formato que acapare toda nuestra atención. En caso de ocurrir esto, puede sernos de gran ayuda utilizar la implementación de este programa para crear un nuevo traductor que se adapte a esa nueva situación o, simplemente, añadir dicha funcionalidad al actual proyecto.

De cara a usos futuros, más allá del utilizado hasta ahora basado en conseguir toda la documentación actual en un mismo estilo. Deberá ser la de conseguir que la documentación escrita posteriormente también tenga toda el mismo formato. Si algo nos facilita LPdoc es la gran ventaja de poder escribir mezclando los estilos que más nos gusten o a los que más acostumbrados estamos. Con la el programa desarrollado se permitirá que cada uno de los autores pueda escribir de la manera que le sea más cómodo él pero consiguiendo que el repositorio mantenga un línea homogénea.

Bibliografía

[1] Donald Ervin Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[2] Eric W Van Ammers and Mark R Kramer. The clip style of literate programming. *submitted for publication*, 1993.

[3] Richard M Stallman. Emacs the extensible, customizable self-documenting display editor. In *Proceedings of the ACM SIGPLAN SIGOA symposium on Text manipulation*, pages 147–156, 1981.

[4] Manuel Hermenegildo and José Francisco Morales. The lpdoc documentation generator. Technical report, Technical report, School of Computer Science, Technical University of Madrid . . . , 1996.

[5] Jesús Tramullas. Desarrollos en elaboración de documentación técnica: lenguajes de marcado ligero. *Anuario ThinkEPI*, 13.

[6] A Fernández Valmayor, A Navarro, Baltasar Fernández-Manjón, and José Luis Sierra. Lenguajes de programación, lenguajes de marcado y modelos hipermedia: una visión interesada de la evolución de los lenguajes informáticos. *Estudios de lingüística del español*, 24, 2006.

[7] James H. Coombs, Allen H. Renear, and Steven J. DeRose. Markup systems and the future of scholarly text processing. *Commun. ACM*, 30(11):933–947, November 1987.

[8] Markdow.

[9] Ciao-lang.

[10] Massimiliano Dominici. An overview of pandoc. *TUGboat*, 35(1):44–50, 2014.

Anexo

LPDowner

Natalia Carpizo

Table of Contents

1 Summary

An automatic markup-to-markdown translator for the LPdoc documentation language.

2 Introduction

LPDowner is a tool for automatic translation between markup and markdown as well as from assertions to doccomments. Its implementation is based on three phases.

- First: LPDowner converts markup commands to markdown. Por example, if we have the following input:

```
:- doc(module,"This is an @bf{example}").
:- doc(p/2,"This is a nice @em{predicate}").
:- pred length(L,N) : list * var => list*integer
   # "Computes the length of @var{L} ".
```

  the result will be:

```
:- doc(module,"This is an **example**").
:- doc(p/2,"This is a nice *predicate*").
:- pred length(L,N) : list * var => list*integer
   # "Computes the length of `L` ".
```

- Second: Tool translates all doc assertion that have an equivalent in doccomments except those whose first argument is a predicate. Add package doccoments if it is necessary. Phase2= Phase2+Phase1. If we use the example above, the result will be:

```
:- use_package(doccomments).
%! @module
%  This is an **example**
:- doc(p/2,"This is a nice *predicate*").
:- pred length(L,N) : list * var => list*integer
   # "Computes the length of `L` ".
```

- Third: It converts assertion comments and doc assertion whose first argument is a predicate to doccomments. Phase3 =Phase2 +Phase3. If we use previous input, output will be:

```
:- use_package(doccomments).
%! @module
%  This is an **example**
%! p/2:
%  This is a nice *predicate*
:- pred length(L,N) : list * var => list*integer.
   %< Computes the length of `L`
```

The program is executed with the predicate `main(Args)`: `Args` is a list with the necessary arguments to execute LPDowner.

- `Args` = [-h],[–helps]: shows help manual.
- `Args` = [ ]: Input will be translated as a .pl file. Input is the standard input and output is the standard output. Markdown is not allowed in the input. It uses phase 2.
- `Args` = [Source]: Input will be translated as a .pl file. Source is the path of the input file. out is the standard output Markdown is not allowed in the input. It uses phase 2.
- `Args` = [Source,SourceO]: Input will be translated as a .pl file. Source is the path of the input file. SourceO is the path of the output file. Markdown is not allowed in the input. It uses phase 2.
- `Args` = [Source,SourceO,lpdoc,MarkDownOpt]: Input will be translated as a .lpdoc file. `Source` is the path of the input file. `SourceO` is the path of the output file. Markdown is allowed if `MarkDownOpt` is ground.
- `Args` = [Source,SourceO,pl,MarkDownOpt,Phase]: Input will be translated as a .pl file. `Source` is the path of the input file. `SourceO` is the path of the output file. Markdown is allowed if `MarkDownOpt` is ground. `Phase` is the type of transalation used(Phase1,Phase2,Phase3).

3 conversor (library)

Translate a given string from markdown to markup.

3.1 Usage and interface

- **Library usage:**
  `:- use_module(/home/natalia/ciao/lpdowner/src/conversor.pl).`
- **Exports:**
  - *Predicates:*
    `converter/4`, `dappend/3`.

3.2 Documentation on exports

converter/4: PREDICATE

`converter(TextMarkUp,TextoMarkDown,MDOpt,FAssertion)`

`TextMarkUp`:: string in markup.

`TextMardown`
: `TextMarkUp` in markdown.

`MDOpt`
: it is ground if `TextMarkUp` contains markdown

`FAssertion`
: Type of comment (doc assertion, assertion comment, lpdoc...) to which TextMarkUp belongs.

‘

dappend/3: PREDICATE

Join two differential list

3.3 Documentation on internals

listToDl/3: PREDICATE

No further documentation available for this predicate.

3.4 Documentation on imports

This module has the following direct dependencies:

- *Application modules:*
  `parseCommand`.
- *System library modules:*
  `lists`, `llists`, `messages`, `write`, `markdown_translate`, `hiordlib`, `stream_utils`.

– *Internal (engine) modules:*
  `term_basic`, `arithmetic`, `atomic_basic`, `basiccontrol`, `exceptions`, `term_compare`, `term_typing`, `debugger_support`, `basic_props`, `runtime_control`, `stream_basic`.
– *Packages:*
  `prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `dcg`, `fsyntax`.

4 parseCommand (library)

Author(s): Manuel Hermenegildo, Jose F. Morales, Natalia Carpizo.

Parser for the lpdoc mark-up language (`docstring/1`). Separate a given string into a list of elements according to the command to which it belongs.

4.1 Usage and interface

- **Library usage:**
  `:- use_module(/home/natalia/ciao/lpdowner/src/parseCommand.pl).`
- **Exports:**
  - *Predicates:*
    `parse_docstring/3`, `parse_docstring0/3`.

4.2 Documentation on exports

parse_docstring/3: PREDICATE

No further documentation available for this predicate.

parse_docstring0/3: PREDICATE

No further documentation available for this predicate.

4.3 Documentation on imports

This module has the following direct dependencies:

- *System library modules:*
  `write`, `lists`, `errhandle`, `markdown_translate`, `autodoc_state`, `autodoc_aux`, `autodoc_errors`, `autodoc_index`, `autodoc_doctree`, `autodoc_settings`, `autodoc_filesystem`, `autodoc_messages`, `comments`, `assrt_lib`, `pretty_print`, `vndict`, `io_port_reify`, `port_reify`.
- *Internal (engine) modules:*
  `term_basic`, `arithmetic`, `atomic_basic`, `basiccontrol`, `exceptions`, `term_compare`, `term_typing`, `debugger_support`, `basic_props`, `hiord_rt`, `runtime_control`.
- *Packages:*
  `prelude`, `initial`, `condcomp`, `dcg`, `assertions`, `assertions/assertions_basic`, `regtypes`, `basicmodes`.

5 tokens (library)

This module separates a given string into the tokens that form it.To achieve this,the program reads the string letter to letter until it one of the posibble tokens is obtained.

5.1 Usage and interface

- **Library usage:**
  `:- use_module(/home/natalia/ciao/lpdowner/src/tokens.pl).`
- **Exports:**
  - *Predicates:*
    `identify_tokens/2`.

5.2 Documentation on exports

identify_tokens/2: PREDICATE

`identify_tokens(Tokens,String0)`

Identifies the tokens that make up `String0` and stores them in `Tokens`.

5.3 Documentation on internals

get_token/5: PREDICATE

`get_token(Token,FlagDoc,FlagDocF,String0,String)`

Read `String0` until it finds the first token.

`FlagDoc` it can take three values:

0	Token can be an assertion (doc,module...)
1	Token is an assertion comment.
2	Token can take any value except those related to assertions.

`FlagDocF` its value will be the value of FlagDoc in the next iteration.

`String0` String to read.

`String` String0 without token value.

`Token` First token in String0. It is a term with the format *token(TokenType,TokenValue)*. TokenType and TokenValue will be instantiated in this predicate.

flag_doc/3: PREDICATE

`flag_doc(TokenType,OldFlag,NewFlag)`

It will be give value to NewFlag based on TokenType. The possible values are:

- 0: TokenType ends a clause.
- 1: TokenType is `assertionC`, in other words, if next token will be `commentA`.
- 2: TokenType is inside a clause.

If TokenType is `spaces`,`commentn` or `comment1`, the value of NewFlag will be OldFlag's value.

5.4 Documentation on imports

This module has the following direct dependencies:

– *System library modules:*

`lists`, `llists`, `messages`, `hiordlib`, `stream_utils`.

– *Internal (engine) modules:*

`term_basic`, `arithmetic`, `atomic_basic`, `basiccontrol`, `exceptions`, `term_compare`, `term_typing`, `debugger_support`, `basic_props`, `runtime_control`, `stream_basic`.

– *Packages:*

`prelude`, `initial`, `condcomp`, `assertions`, `assertions/assertions_basic`, `dcg`, `fsyntax`.

References

(this section is empty)

Library/Module Index

Predicate Index

Property Index

(Index is nonexistent)

Regular Type Index

(Index is nonexistent)

Declaration Index

(Index is nonexistent)

Concept Index

(Index is nonexistent)

Author Index

Global Index

This is a global index containing pointers to places where concepts, predicates, modes, properties, types, applications, authors, etc., are referred to in the text of the document.

S

T

V

W